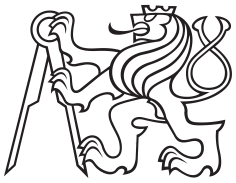
Diplomová práca



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačov

Framework pro testování výkonnosti instance Redmine aplikace s nainstalovanými rozšířeními

Bc. Martin Krupa

Školiteľ: Ing. Karel Frajták, Ph.D.
Odbor: Otvorená informatika
Zameranie: Softwarové inžinierstvo
Január 2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Krupa** Jméno: **Martin** Osobní číslo: **434972**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Framework pro testování výkonnosti instance Redmine aplikace s nainstalovanými rozšířeními

Název diplomové práce anglicky:

Performance testing framework for Redmine instance

Pokyny pro vypracování:

Navrhněnete a naimplementujte framework pro testování výkonnosti instance Redmine aplikace s nainstalovanými rozšířeními. Framework po nakonfigurování provede testování aplikace pod simulovanou zátěží a poskytne informace o výsledcích testů obsluze a interpretuje výsledky testování.
Konfigurace umožní nastavení parametrů testů, jako je například rozložení zátěže v čase, nebo počet současně přistupujících uživatelů. Dalšími parametry jsou očekávané hodnoty sledovaných metrik, např. čas odpovědi (response time). Obsluha také určí testovaný scénař pro testovaní.
Samotné testování proběhne v izolovaném prostředí v dockeru, kde budou k dispozici jak kontejnery samotné testované aplikace, tak i nástroje pro sběr dat z těchto kontejnerů (stav procesorů, paměti...) a komunikace (telemetrie) mezi nimi.
Nástroj bude zaznamenávat využití zdrojů (paměť, procesor, apod.) aplikace samotné. Profiling aplikace poslouží k identifikaci problematických částí aplikace.
Po skončení testů budou sesbíraná data zanalyzována a bude navrhnuto rešení pro případnou optimalizaci a vyjmenována úzká místa v aplikaci.
Framework navrhněte tak, aby bylo možné ho použít v CI pipeline.
Součástí práce by měla být rešerše stávajících řešení a diskuze možného rozšíření i na jiné aplikace psané v jiných programovacích jazycích.

Seznam doporučené literatury:

Molyneaux, Ian. The art of application performance testing: from strategy to tools. " O'Reilly Media, Inc.", 2014.
Srivastava, Nishi, Ujjwal Kumar, and Pawan Singh. "Software and performance testing tools." Journal of Informatics Electrical and Electronics Engineering (JIEEE) 2.1 (2021): 1-12.
Magill, John R., et al. "Establishing age-and sex-specific norms for pediatric return-to-sports physical performance testing." Orthopaedic Journal of Sports Medicine 9.8 (2021): 23259671211023101.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Karel Frajták, Ph.D. laboratoř inteligentního testování systémů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **14.09.2022** Termín odevzdání diplomové práce: **10.01.2023**

Platnost zadání diplomové práce: **19.02.2024**

Ing. Karel Frajták, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

Poďakovanie

Ďakujem svojmu školiteľovi pánovi Ing. Karlovi Frajtákovi, Ph.D. za odborné vedenie, cenné rady a pomoc pri vypracovaní diplomovej práce.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval samostatne, a že som uviedol všetku použitú literatúru.

V Prahe, 10. januára 2023

Abstrakt

Diplomová práca sa zaoberá frameworkom pre výkonnostné testovanie Redmine aplikácie. Cieľom bolo navrhnúť a naimplementovať framework, ktorý bude vykonávať výkonnostné testovanie Redmine aplikácie. Framework je implementovaný ako Redmine plugin. Poskytuje rozhranie určené na konfiguráciu výkonnostných testov. Pre vykonávanie výkonnostného testovania bol použitý nástroj k6. Po vykonaní testov framework spracuje výstupy a interpretuje ich užívateľovi.

Naimplementovaný framework bol použitý na testovanie Redmine aplikácie. Na základe výstupov bol vykonaný profiling miest, ktoré boli vo výstupoch testovania identifikované ako problematické. V závere práce sú výstupy profilingu vyhodnotené.

Kľúčové slová: výkonnostné testovanie, framework, Redmine, redmine rozšírenie, k6, Ruby, Ruby on Rails

Školiteľ: Ing. Karel Frajták, Ph.D.

Abstract

The diploma thesis deal with performance testing framework for Redmine application. The goal was to design and implement framework that will be able to perform performance testing of Redmine application. Implementation is written as Redmine plugin. Framework provides interface for configuration of performance tests. Implementation is using k6 load testing tool to run performance tests. After tests are finished, framework processes results and interprets them to the user.

Implemented framework was used for testing of Redmine application. Identified problematic output data by testing was profiled. Results of profiling are discussed in thesis conclusion.

Keywords: performance testing, framework, Redmine, redmine plugin, k6, Ruby, Ruby on Rails

Title translation: Performance testing framework for Redmine instance

Obsah

Obrázky

Kapitola 1

Úvod

Redmine je aplikácia na projektové riadenie, ktorá poskytuje nástroje pri spravovaní úloh a projektov. Kód aplikácie je voľne dostupný a k jeho vývoju môže hypoteticky prispieť ktokoľvek. Spôsob, akým je aplikácia napísaná, taktiež dovoľuje dopĺňať do Redminu vlastné rozšírenia.

To môže mať značný vplyv na výkon aplikácie. Vzniká tu potencionálne riziko, že sa medzi zdrojové kódy dostane neoptimalizovaný kód, alebo nekvalitné rozšírenie.

V prípade, že užívateľ siahol po možnosti takéhoto voľne dostupného nástroja, môže mať v úmysle používať aj vlastné technické vybavenie. V takom prípade potrebuje overiť, že je jeho zariadenie dostatočne výkonné na prevádzku takejto aplikácie pri predpokladanej záťaži.

Cieľom diplomovej práce bude z toho dôvodu navrhnúť a naimplementovať framework, ktorý bude schopný vykonať výkonnostné testovanie nad Redmine aplikáciou za účelom odhalenia neoptimalizovaného kódu, alebo overenia, že technické vybavenie je dostatočné na prevádzku aplikácie.

Kapitola 2

Testovanie výkonu aplikácie

Testovanie softwaru je proces, pomocou ktorého overujeme a vyhodnocujeme do akej miery splnil softwarový produkt, alebo aplikácia zadané požiadavky.[7] Testovanie softwaru môžeme vykonávať dvoma spôsobmi, manuálne, alebo automatizovane.[27] V procese softwarového testovania overujeme dva typy požiadaviek:

- Funkčné požiadavky, ktoré definujú akú funkcionalitu má aplikácia poskytovať.
- Nefunkčné požiadavky, ktoré nám definujú kvalitatívne vlastnosti (quality attributes) aplikácie.

Kvalitatívna vlastnosť je merateľná vlastnosť aplikácie, ktorá nám popisuje v akej miere spĺňa aplikácia zadané požiadavky.[22] Pri meraní kvalitatívnych vlastností využívame scenáre kvalitatívnych vlastností (quality attribute scenarios). Scenáre kvalitatívnych vlastností môžeme definovať ako špecifické požiadavky kvalitatívnych vlastností. Všeobecne sa takýto scenár skladá zo šiestich častí (prevzaté z [22]):

- *zdroj podnetu* (*Source of stimulus*),
- *podnet* (*Stimulus*),
- *prostredie* (*Environment*),

- *artefakt* (*Artifact*),
- *odpoveď* (*Response*),
- *meranie odpovede* (*Response measure*).

Medzi kvalitatívne vlastnosti patrí aj výkon (performance) aplikácie a patrí medzi 5 základných vlastností kvality označovaných FURPS[3]:

- funkčnosť (**F**unkcionality),
- použiteľnosť (**U**sability),
- spoľahlivosť (**R**eliability),
- výkon (**P**erformance),
- udržovateľnosť (**S**upportability).

Testovanie výkonu aplikácie je proces, pri ktorom monitorujeme chovanie aplikácie pod záťažou. Pomocou softwarových nástrojov simulujeme záťaž na aplikáciu tak, aby zodpovedala predpokladanej návštevnosti.[20]

2.1 Výhody testovania výkonu aplikácie

Testovanie výkonu nám pomáha vylepšiť celkovú výkonnosť aplikácie. Identifikuje slabú škálovateľnosť, prípadne môže pomôcť odhaliť rôzne problémy. Napríklad prítomnosť úzkeho miesta v aplikácii (bottlenecku).[28]

Bottleneck je miesto v aplikácii, ktoré má limitovanú kapacitu a obmedzuje priechodnosť aplikáciou.[28] Jednoduchý názorný príklad bottlenecku z reálneho života je zúženie jazdných pruhov na diaľnici.

Slabá škálovateľnosť môže oslabiť výkonnosť aplikácie, čo môže viesť k výskytu chýb v aplikácii, oneskoreniu spracovania requestov, alebo k situáciám, ktoré nazývame únik pamäte (memory leak).[28]

Únik pamäte je situácia, kedy nedochádza k uvoľneniu nepotrebnej operačnej pamäte. Napríklad, keď už nevyužívaný objekt ostáva alokovaný v pamäti.

Tento problém vedie k postupnému spomaľovaniu aplikácie, pretože pamäťové zdroje aplikácie sa zmenšujú s každým výskytom. V prípade úplného vyčerpania zdrojov nastáva zrútenie aplikácie.

2.2 Performance Test

Test výkonnosti, tiež nazývaný performance test, meria stabilitu, výkonnosť, škálovateľnosť a priechodnosť webovej aplikácie. Účelom výkonnostných testov je zabezpečiť, že finálna verzia aplikácie spĺňa predpoklady bezproblémovej prevádzky pri väčšom množstve užívateľov.[20]

2.3 Smoke Test

Smoke test je prvým krokom pri testovaní výkonu aplikácie. Prakticky sa jedná o bežný test priechodnosti testovacím scenárom pri minimálnej záťaži.[20]

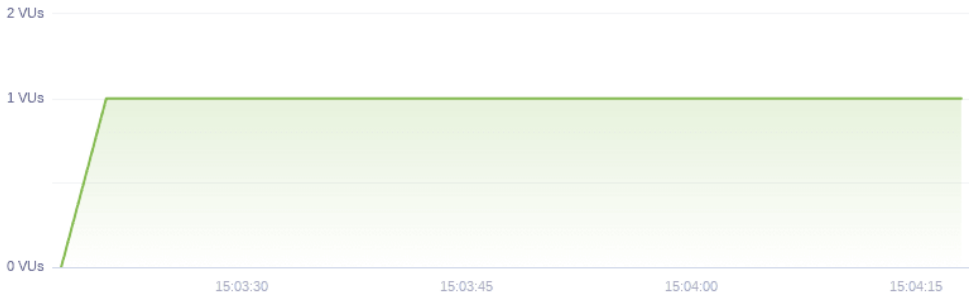
Smoke testy majú zvyčajne dva účely[17]:

- Overiť, že pri priechode testovacím scenárom nenastane aplikačná chyba.
- Overiť schopnosť systému zvládnuť testovací scenár pri minimálnej zátaži.

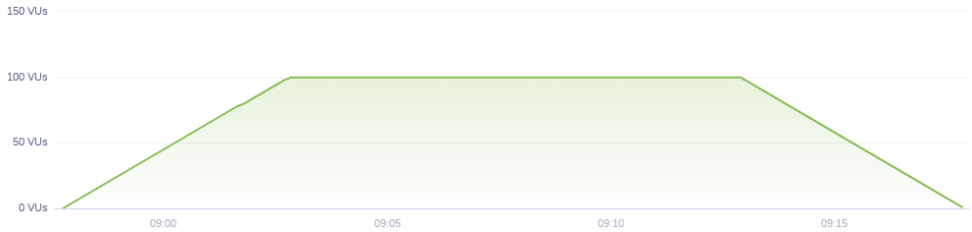
Smoke test by mal byť spúšťaný pred každým záťažovým testom, aby mohla byť overená schopnosť systému zvládnuť minimálne nároky (obrázok 2.1).[17] Ak nastane zlyhanie, nie je potrebné pokračovať v ďalšom testovaní a je nutné najskôr opraviť nájdené chyby.[20]

2.4 Load Test

Load testy vyhodnocujú schopnosť aplikácie fungovať pri zvýšenej záťaži.[20]



Obrázok 2.1: Ukážka rozloženia záťaže smoke testu[17]



Obrázok 2.2: Ukážka rozloženia záťaže load testu[11]

Záťaž sa simuluje pomocou väčšieho počtu virtuálnych užívateľov, ktorí vykonávajú HTTP requesty podľa testovacieho scenára. Load testy sledujú, ako tieto akcie ovplyvňujú chovanie aplikácie a dobu odozvy jednotlivých requestov.[20] Ukážka rozloženia záťaže je zobrazená na obrázku 2.2.

Load testy by sme mali spúšťať za účelom[11]:

- Posúdiť aktuálny výkon aplikácie pri očakávanej priemernej záťaži a záťaži očakávanej v špičke.
- Uistiť sa, že systém spĺňa stanovené výkonnostné požiadavky aj po vykonaní zmien v kóde, alebo v štruktúre aplikácie.

2.5 Stress Test

Stress testy sú podobné Load testom. Na rozdiel od Load testov, sa Stress testy zameriavajú na chovanie aplikácie pri extrémnej záťaži, ktorá siaha za hranice predpokladanej bežnej prevádzky.[28] Rozloženie záťaže v stress teste je zobrazené na obrázku 2.3.

Obrázok 2.3: Ukážka rozloženia záťaže stress testu[19]

Obyčajne spúšťame Stress test aby sme zistili[19]:

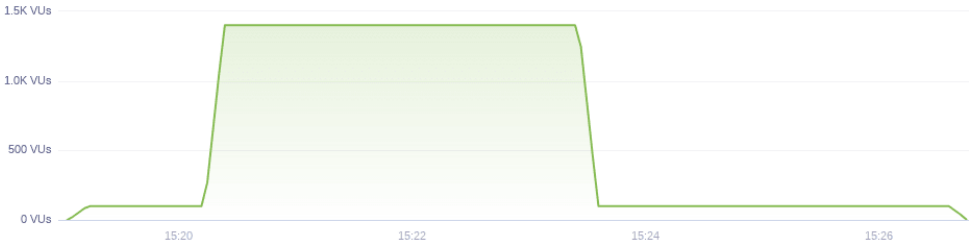
- Ako systém reaguje pri extrémnej záťaži.
- Aká je maximálna kapacita užívateľov, alebo maximálna kapacita priechodnosti aplikácie.
- Bod, pri ktorom nastane zlyhanie systému.
- Schopnosť systému zotaviť sa z extrémnej záťaže bez manuálneho zásahu.

2.5.1 Spike Test

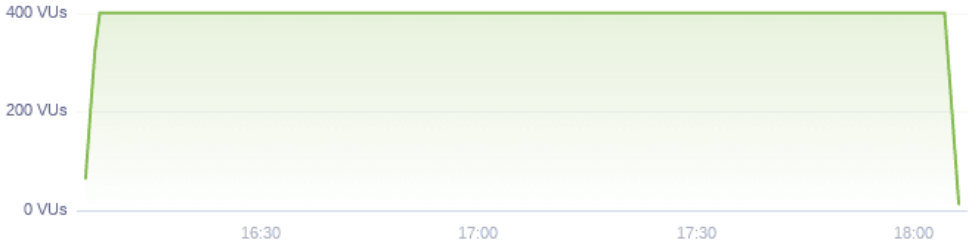
Spike test je špecifický typ Stress testu. Rovnako sa zameriava na chovanie aplikácie pri extrémnej záťaži, ale v prípade Spike testu je extrémna záťaž nadobudnutá vo veľmi krátkom čase.[19] Obrázok 2.4 obsahuje ukážku Rozloženie záťaže pre spike test.

Úlohou Spike testuje je určiť[19]:

- Ako bude systém reagovať na náhly nárast záťaže na extrémne hodnoty.
- Schopnosť systému zotaviť sa po znížení záťaže na priemer.



Obrázok 2.4: Ukážka rozloženia záťaže spike testu[19]



Obrázok 2.5: Ukážka rozloženia záťaže soak testu[18]

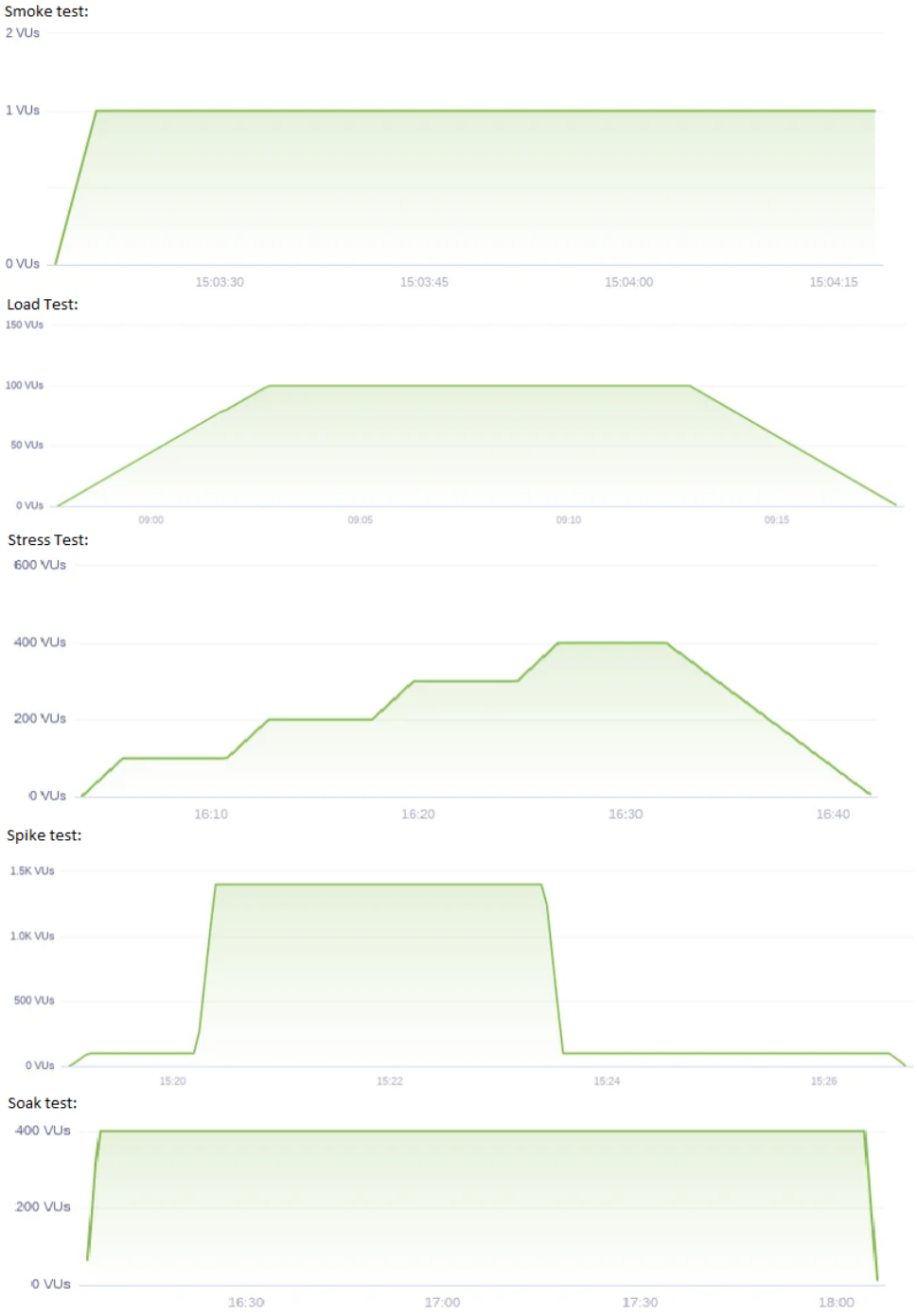
2.6 Soak Test

Soak testy sa zameriavajú na výkonnosť aplikácie počas dlhšieho časového obdobia. Pomocou Soak testov dokážeme odhaliť problémy súvisiace so spoľahlivosťou a výkonom aplikácie, ktorá dlhodobo pracuje pod záťažou.[18] Záťaž počas soak testu je zobrazená na obrázku 2.5.

Úlohou soak testov je[18]:

- Overiť, že systém neobsahuje chyby, ktoré by spôsobili zlyhanie alebo reštart počas dlhšieho časového obdobia.
- Overiť, že ak dôjde k reštartu aplikácie, nedôjde k strate requestov.
- Overiť, že nenastane vyčerpanie pridelených zdrojov.
- Overiť, že nenastane zlyhanie komunikácie s externými službami po určitom počte requestov.

Každý spomenutý typ testu sa zameriava na špeciálnu situáciu simulovania záťaže. Všetky však majú spoločný cieľ: objaviť potencionálne riziká v aplikácii pri simulovanej prevádzke.[20] Pre porovnanie sú grafy všetkých typov testov znázornené na spoločnom obrázku 2.6.



Obrázok 2.6: Grafické porovnanie typov testov[18]

Kapitola 3

Analýza

Úlohou práce je navrhnúť a implementovať framework pre testovanie výkonnosti Redmine aplikácie. Pred vypracovaním úlohy je potrebné zanalyzovať zadanie, vyvodiť vlastnosti navrhovaného frameworku a určiť, aké nástroje je možné pri práci použiť.

Zo zadania vyplýva niekoľko funkcionalít, ktoré by mal framework obsahovať:

- Konfigurátor testov, ktorý umožní užívateľovi nastavenie parametrov testov, sledované metriky a testovacie scenáre, ktoré budú definovať priechod aplikáciou.
- Mechanizmus pre spúšťanie a vykonanie nakonfigurovaného testu.
- Interpretáciu výsledkov a výstupov vykonaného testu.
- Možnosť profilingu aplikácie v prípade neuspokojivých výsledkov testu.

Z uvedeného zoznamu sa dá vidieť, že základom bude definovanie mechanizmu vykonávania testov. Od toho sa následne bude odvíjať:

- Návrh a implementácia konfigurátoru testov.
- Návrh a implementáciu mechanizmu pre spúšťanie testov.

- Interpretáciu výsledkov a výstupov.

Pri návrhu by bolo možné využiť existujúce nástroje na otestovanie výkonnosti aplikácií, a namiesto náročnej implementácie integrovať vhodný nástroj do frameworku. Tým by sa definoval mechanizmus vykonávania testov. Zároveň by sa definovala aj forma vstupov a výstupov, ktorým by sa mohol prispôsobiť konfigurátor testov a interpretácia výsledkov.

Potreba profilingu bude závislá od interpretácie výsledkov a zvoleného profilingového nástroja.

Profiling je spôsob dynamickej programovej analýzy, ktorá prebieha počas behu programu a analyzuje jeho chovanie, prácu s pridelenými zdrojmi a čas, za ktorý sa beh programu vykoná.

Pre návrh profilingu je potrebné zanalyzovať existujúce profilingové nástroje a nájsť vhodný nástroj, ktorý bude na tieto účely profilingu použitý.

3.1 Nástroje na testovanie záťaže aplikácie

Nástroje na výkonnostné testovanie nám pomáhajú určiť rýchlosť, efektivitu, spoľahlivosť a škálovateľnosť aplikácie, na ktorú sú použité.[25]

Takéto nástroje sa zväčša zameriavajú na dotazovanie zvolenej aplikácie pomocou veľkého množstva paralelne fungujúcich virtuálnych užívateľov. Pri dotazovaní merajú čas načítania stránky, čas odozvy, prípadne ďalšie užitočné metriky.

3.1.1 Apache JMeter

Apache JMeter™ je open-source software napísaný v jazyku Java a je určený pre testovanie výkonu webových aplikácií, alebo API. Simuluje záťaž na server, či skupinu serverov, sieť, alebo objekt tak, aby otestoval a analyzoval celkový výkon pri rôznych typoch záťaže.[1] Podrobnejšie informácie sú dostupné na `https://jmeter.apache.org/index.html`.

3.1.2 Gatling

Gatling je nástroj na testovanie záťaže webových aplikácií. Obsahuje vlastný webový zapisovač, pomocou ktorého je možné generovať scenáre ich priechodom v aplikácii. Testy je možné písať v jednom z troch jazykov: Java, Scala, Kotlin. Podporuje rôzne nástroje ako: Maven, Gradle, Sbt.[4] Podrobnejšie informácie sú dostupné na `https://gatling.io`.

3.1.3 K6.io

Grafana k6 je open-source nástroj na výkonnostné testovanie. Poskytuje rozhranie pre prácu v príkazovom riadku a prístupné skriptovacie API. Skripty sú písané v jazyku Javascript a sú jednoducho konfigurovateľné s podporou lokálnych aj vzdialených modulov.[9] Podrobnejšia dokumentácia je dostupná na `https://k6.io/docs`.

3.1.4 Artillery

Artillery je moderný nastroj na jednoduché výkonnostné testovanie. Je navrhnutý na testovanie back-endových systémov nezávisle na tom, v akom jazyku sú napísané, alebo aký protokol používajú. Pomocou Artillery je možné otestovať API, alebo web aplikácie spolu s ich webovým rozhraním.[21] Podrobnejšia dokumentácia je dostupná na `https://www.artillery.io`.

3.1.5 Použitý testovací nástroj

Z vymenovaných testovacích nástrojov bude použitý nástroj k6.io. Pri výbere sa zohľadnili možnosti integrácie a použitia daných nástrojov v rámci Ruby kódu.

Nástroj k6 poskytuje *Command Line Interface*, skrátene CLI, pomocou ktorého je možné spustiť testovacie skripty. Spustené skripty môžu byť z lokálneho, ale aj zo vzdialeného zdroja. CLI umožňuje definovať parametre testu pri volaní príkazu, prípadne je možné definovať parametre v samotnom skripte.

Pre použitie nástroja k6 je potrebná inštalácia na zariadení, kde sa budú púšťať testovacie skripty.

Nástroj k6 je schopný spustiť rôzne typy testov. Napríklad *smoke testy*, *load testy*, *stress testy*, *spike testy*, alebo *soak testy*.

Test je reprezentovaný k6 skriptom. Typ testu je definovaný podľa konfigurácie počtu virtuálnych užívateľov počas priebehu testu. Takáto konfigurácia môže mať viac foriem.

Prvá forma obsahuje dve hodnoty. Jednu označujeme *vus* a určuje konštantnú hodnotu počtu virtuálnych užívateľov počas trvania testu. Druhú hodnotu označujeme *duration* a udáva nám dĺžku trvania testu. Listing 3.1 zobrazuje ukážku konfigurácie jedeného užívateľa počas 30 sekúnd.

Listing 3.1: Ukážka definície konštantného počtu virtuálnych užívateľov

```
export const options = {
    vus: 1, // 1 user for 30 secs
    duration: 30s
}
```

Druhá forma je pole s názvom *stages*. Elementy v tomto poli definujú počet virtuálnych užívateľov počas časovej etapy testu. Každý element obsahuje dve hodnoty. Hodnotu *duration*, ktorá určuje trvanie etapy. A hodnotu *target*, ktorou definujeme počet virtuálnych užívateľov počas časovej etapy. Ak mala predchádzajúca časová etapa menšiu hodnotu *target*, tak dochádza k rovnomernému nárastu počtu virtuálnych užívateľov počas celej etapy. Ak bola predchádzajúca hodnota *target* vyššia, dochádza k rovnomernému poklesu počtu virtuálnych užívateľov. Takúto formu zápisu konfigurácie využívame, keď chceme dosiahnuť nárast, prípadne pokles záťaže za určitú časovú etapu, alebo keď chceme dosiahnuť nárast, alebo pokles záťaže v pravidelných intervaloch.

Listing 3.2: Ukážka definície časových intervalov s rôznym počtom virtuálnych užívateľov

```
export const options = {
    stages: [
        { duration: "2m", target: 50 }, // rise up users from 1
            to 50 during 2 minutes
        { duration: "11m", target: 50 }, // keep 50 users for
           11 minutes
        { duration: "2m", target: 0 }, // drop users from 50 to
            0 during 2 minutes
    ]
}
```

Listing 3.2 je ukážkou druhej formy konfigurácie. Obsahuje tri časové etapy. Prvá etapa trvá dve minúty a dochádza počas nej k rovnomernému nárastu počtu virtuálnych užívateľov z 1 na 50. Druhá etapa trvá 11 minút a cieľový počet virtuálnych užívateľov je rovnaký ako v predchádzajúcej etape (50). Tento počet sa nezmení a drží sa na rovnakej hodnote počas celého trvania etapy. Posledná etapa trvá dve minúty a cieľový počet užívateľov je 0. Tu dôjde k rovnomernému zníženiu z hodnoty predchádzajúcej etapy (50) na 0.

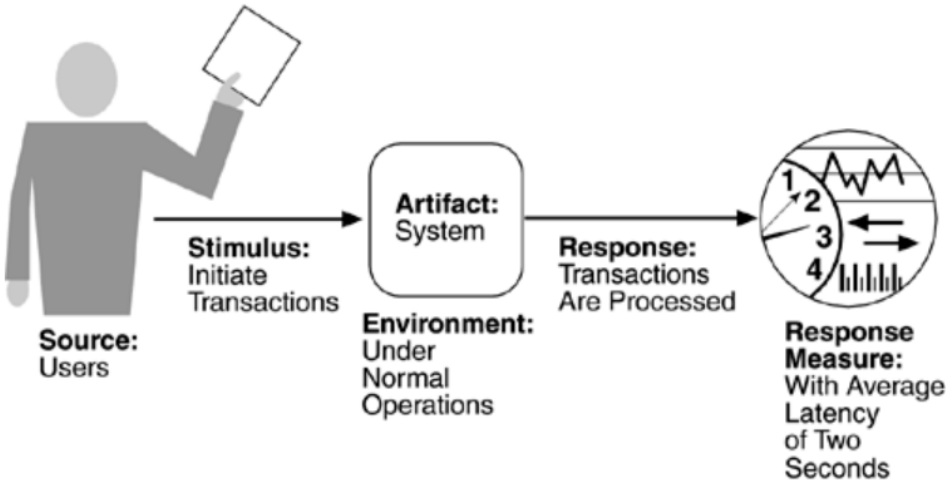
K6 skripty musia byť napísané v jazyku Javascript. Skripty sa dotazujú na webovú aplikáciu pomocou HTTP requestov. Samotný skript sa môže skladať z viacerých častí:

- Setup, kde je možné dopredu definovať rôzne vlastnosti.
- Options, ktoré obsahujú konfiguráciu testu.
- Testovaciu funkciu, ktorá definuje scenár testu.
- Funkciu na správu výstupu.

Kritéria úspešnosti testu, z hľadiska výkonnosti, sa definujú v konfigurácii testu ako prahové hodnoty (*thresholds*). Tieto kritéria sa skladajú z prahovej hodnoty (*threshold*) pre konkrétnu metriku, ktorá je súčasťou kvalitatívnej vlastnosti výkonu aplikácie (performance).

Testovací skript môžeme teda označiť ako scenár výkonu aplikácie (obrázok 3.1). Konkrétne časti, z ktorých sa scenár skladá, vypadajú nasledovne:

- *Zdroj podnetu* – nástroj k6 a sada virtuálnych definovaných užívateľov v konfigurácii skriptu.
- *Podnet* – requesty na aplikáciu definované v testovacej funkcii skriptu.
- *Prostredie* – stav, v ktorom je aplikácia, na ktorú smerujú requesty.
- *Artefakt* – aplikácia, na ktorú smerujú requesty.
- *Odpoveď* – spracovanie prichádzajúcich requestov aplikáciou a jej odpoveď na jednotlivé requesty.
- *Meranie odpovede* – metriky, ktoré meria nástroj k6 počas príjmania odpovedí na odoslané dotazy.



Obrázok 3.1: Ilustrácia scenáru výkonu aplikácie[22]

Thresholds je JavaScriptový objekt, kde kľúč je zvolená metrika, na ktorú bude *threshold* naviazaný a hodnota je pole, kde definujeme kritéria *thresholdu* vo forme reťazca. Pre jednu metriku môžeme definovať viac *thresholdov*. Listing 3.3 obsahuje ukážku, ako by *thresholdy* mohli vypadať. V tomto prípade je definované, že počet neúspešných requestov musí byť menší ako 1% a 99% všetkých requestov musí trvať menej, než jednu sekundu.

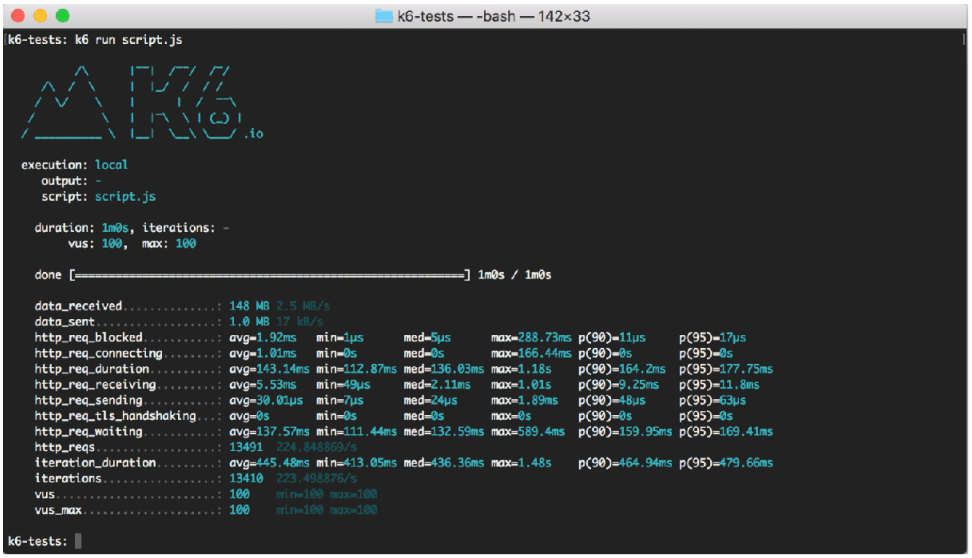
Pre overenie, že jednotlivé kroky testovacieho scenára prebehli úspešne, sa dá využiť funkcia kontroly (*check*), ktorú poskytuje balíček k6.

Listing 3.3: Ukážka použitia *thresholds* pri konfigurácii testu

```
export const options = {
    thresholds: {
        http_req_failed: ["rate < 0.01"], // count of erros
            should be less than 1%
        http_req_duration: ["p(99) < 1000"] // 99% requests
            should be faster than 1000ms
    }
}
```

Predvoleným výstupom testu je výpis do konzoly v textovom formáte. K6 podporuje mnoho externých formátov výstupu, medzi ktorými sú napríklad *Json* a *Csv*.

Podrobná dokumentácia je dostupná na webovej adrese `https://k6.io/docs`.



Obrázok 3.2: Príklad k6 výstupu v textovom formáte v konzole[16]

Nástroj k6 vymedzuje formát vstupu a spôsob spúšťania testov, na základe čoho sa prispôsobí návrh a implementácia častí frameworku.

3.2 Nástroje na interpretáciu výsledkov testov

Nástroj k6 disponuje množstvom formátov pre výstup. Predvoleným výstupom testu je výpis do konzoly v textovom formáte. Tento formát zobrazuje základné metriky a ich agregované dáta (obrázok 3.2).[16]

Nástroj k6 podporuje základné dátové štruktúry ako sú Json a CSV, prípadne aj dátové úložiská ako Prometheus a InfluxDB.[16] Okrem toho podporuje aj rôzne platformy pre interpretáciu výstupov.

3.2.1 K6 cloud

Platforma k6 cloud je SaaS priamo vytvorená pre prácu s nástrojom k6. Poskytuje jednoduché rozhranie pre vizualizáciu a prácu s výstupmi, ktoré nástroj k6 produkuje.[10]

Umožňuje spúšťanie testov v rámci cloudu, prípadne aj nahranie výstupov

Obrázok 3.3: Príklad k6 výstupu vo formáte Json[8]

z lokálne spustených testov.[2]

K6 cloud predstavuje najjednoduchšiu cestu ako vizualizovať výsledky, avšak jedná sa o platenú platformu.

3.2.2 Grafana

Grafana Cloud je open-source platforma pre pozorovanie importovaných dát.[5] V kombinácii s nástrojom Prometheus je možné vizualizovať výstup k6 testov v Grafana Cloude.[6]

Nástroj poskytuje nastaviteľné rozhranie dashboardov, pomocou ktorých si užívateľ nakonfiguruje, čo chce sledovať a na čo sa v rámci nahraných výstupov zamerať.

3.2.3 Zvolený spôsob interpretácie

Menované platformy sú vyhovujúcim spôsobom vizualizácie výsledkov, keď je možné používať dodatočné platformy. V tomto prípade je však potrebné vizualizovať výsledky v rámci frameworku. Integrácia týchto platforiem by bola príliš komplikovaná.

Pre frameworku bude teda vhodnejšie, keď bude obsahovať implementáciu vlastnej vizualizácie. Tá bude založená na výstupoch vo formáte Json (obrázok 3.3).

3.3 Nástroje na profiling testovanej aplikácie

V prípade, že výsledky vykonaných testov nesplnia nastavené požiadavky, alebo sa v niektorých momentoch priblížia hranici neúspechu, bude potrebné zistiť, prečo sa tak stalo. Dôvodov môže byt niekoľko.

Jedným z nich môže byť výkon stroja, na ktorom je aplikácia nasadená. V takom prípade bolo použitie výkonnostného testu opodstatnené a stroj nemá dostatočný výkon na prevádzku aplikácie.

Ďalším dôvodom by mohla byť implementácia samotnej aplikácie. Aby sme mohli neúspech testu pripísať výkonu stroja, je potrebné tento dôvod preveriť a vylúčiť. Pre tento účel by mal byť využitý profiling aplikácie.

Aby bolo možné vykonať profiling aplikácie, je na to potrebný profilingový nástroj. Ten pomáha identifikovať, aké časti aplikácie sú najviac využívané a kde aplikácia pri práci trávi najviac času.[13]

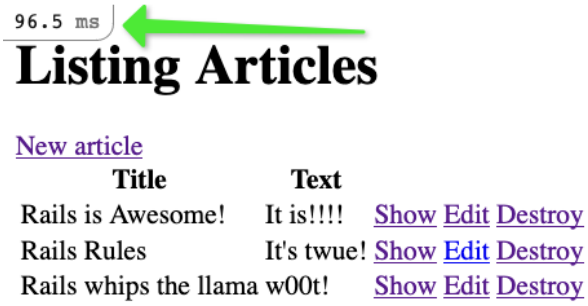
Existuje niekoľko nástrojov pre profiling Ruby on Rails aplikácie. Najpopulárnejšie z nich sú:

- Rack mini profiler,
- Stackprof,
- Ruby Spy.

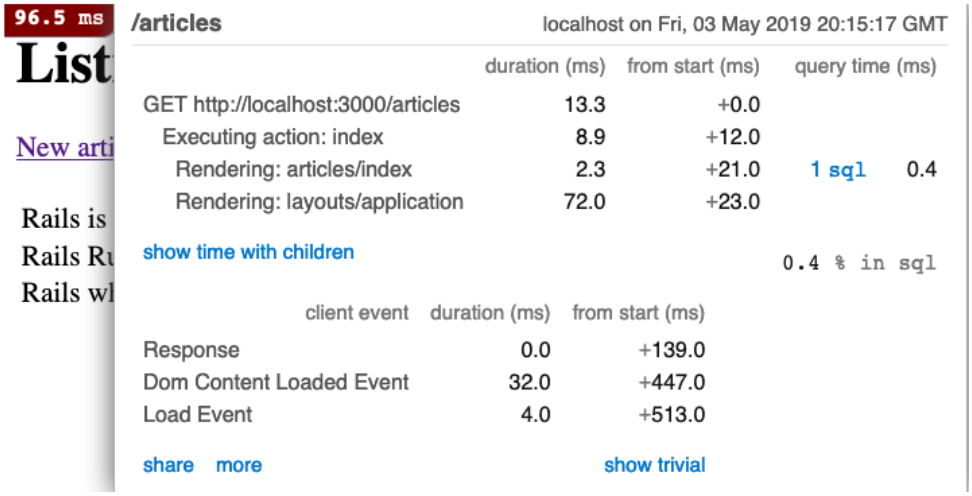
Z vymenovaných nástrojov bude pre vykonanie profilingu použitý Rack mini profiler. Výber najviac ovplyvnili funkcie nástroja, možnosť použitia v rôznych prostrediach a spôsob, akým interpretuje výsledky.

3.3.1 Rack mini profiler

Rack mini profiler je sada nástrojov na meranie výkonu a profiling aplikácie. Obsahuje aj detailný rozboru SQL dotazov a response time servera.[23]



Obrázok 3.4: Ukážka Rack Mini Profiler indikatoru[24]



Obrázok 3.5: Ukážka Rack Mini Profiler rozbaleného indikátoru[24]

Je navrhnutý tak, aby ho bolo možné spustiť nielen vo vývojom prostredí, ale aj v produkčnom.[23] Rack mini profiler sa zameriava na analýzu jedného requestu. Pridáva indikátor do rohu stránky (obrázok 3.4), ktorý zobrazuje dobu načítania.[24] Indikátor je možné rozbaliť a zobraziť tak detailnejší popis analýzy (obrázok 3.5) častí requestu a ich trvania.[24] Súčasťou detailného zobrazenia sú aj odkazy, ktoré sú farebne odlíšené. Vedľa každej časti requestu môžeme nájsť takýto odkaz na detailný popis SQL dotazov (obrázok 3.5), ktoré boli počas daného requestu vykonané.

Okrem indikátoru môžeme s profilerom pracovať aj pomocou URL, a to tak, že na koniec adresy pridáme parameter.

Prvým parametrom, s ktorým je možné pracovať je *pp=backtrace*. Pomocou tohto parametru môžeme meniť *backtrace*pre jednotlivé SQL dotazy zobrazené v indikátore. Tento parameter má tri variácie (Listing 3.4): *pp=full-backtrace*, *pp=normal-backtrace* a *pp=no-backtrace*, pričom *pp=no-backtrace* je predvolená.[24] Zmena *backtracu* trvá dovtedy, kým sa užívateľ nerozhodne

Obrázok 3.6: Ukážka časti zobrazenia *pp=env*[24]

použiť inú variáciu.

Listing 3.4: Ukážka použitia parametru *pp=backtrace*

```
http://localhost:3000/articles?pp=full-backtrace
http://localhost:3000/articles?pp=no-backtrace
http://localhost:3000/articles?pp=normal-backtrace
```

S parametrom *pp=env* (Listing 3.5) sa zobrazí informácia o aplikačnom prostredí (obrázok 3.6).[24]

Listing 3.5: Ukážka použitia parametru *pp=env*

```
http://localhost:3000/articles?pp=env
```

S parametrom *pp=analyze-memory* (Listing 3.6) sa zobrazí stránka s popisom využitia pamäte aplikácie spolu s rozborom dátových typov.[24]

Listing 3.6: Ukážka použitia parametru *pp=analyze-memory*

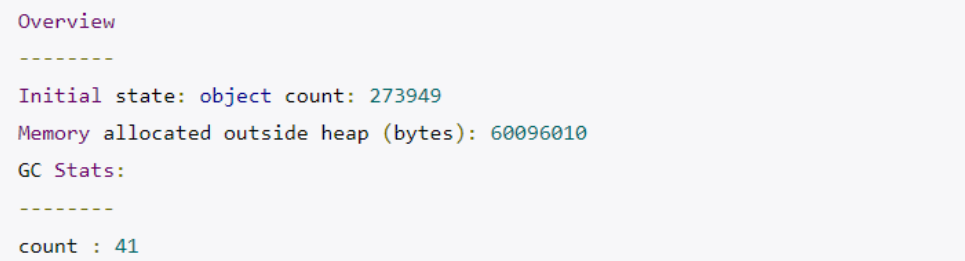
```
http://localhost:3000/articles?pp=analyze-memory
```

Parameter *pp=profile-gc* (Listing 3.7) zobrazí stránku o stave *garbage collectoru* (obrázok 3.7).[24]

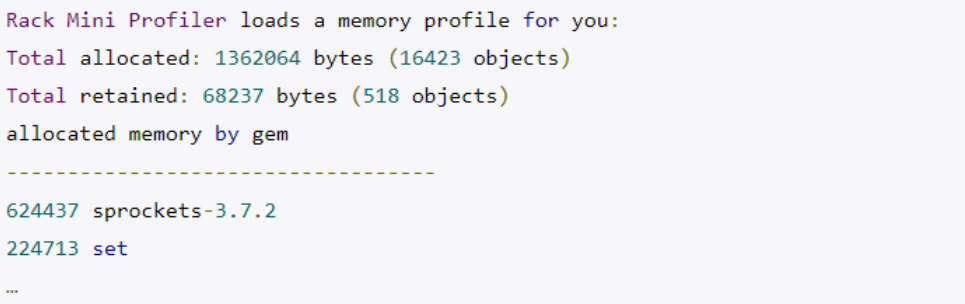
Listing 3.7: Ukážka použitia parametru *pp=profiler-gc*

```
http://localhost:3000/articles?pp=profiler-gc
```

Gem Rack mini Profiler je možné kombinovať s ďalšími gemami, čo rozširuje jeho funkcionalitu. Gemy, s ktorými je možné ho kombinovať sú *stackprof*, *memory_profiler*, *flamegraph*.[23]



Obrázok 3.7: Ukážka časti zobrazenia *pp=profiler-gc*[24]



Obrázok 3.8: Ukážka časti zobrazenia *pp=profile-memory*[24]

V kombinácii s gemom *memory_profiler* a parametrom *pp=profile-memory* (Listing 3.8) stránka poskytuje pokročilejší rozbor využitia pamäte (obrázok 3.8).[24]

Listing 3.8: Ukážka použitia parametru *pp=profile-memory*

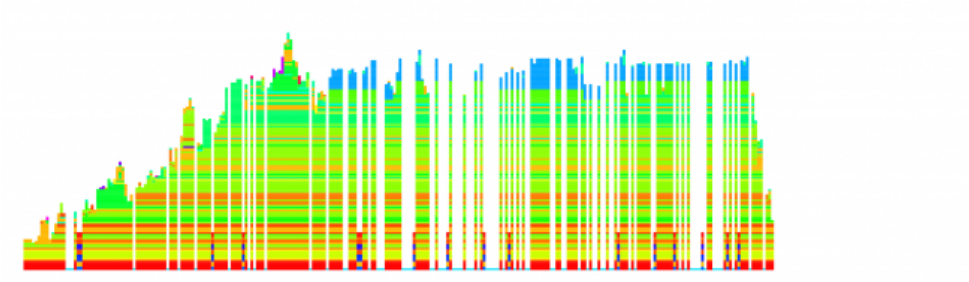
```
http://localhost:3000/articles?pp=profile-memory
```

Po pridaní gemov *stackprof* a *flamegraph* je možné využiť ďalšiu funkcionalitu - flamegraf (obrázok 3.9). Parameter pre zobrazenie flamegrafu je *pp=flamegraph* (Listing 3.9).[24] Flamegraf je interaktívny a reaguje na kliknutie myšou, alebo udalosť *mouseover*. Každý riadok zodpovedá jednej funkcii. Informácie sú rozložené nasledovne:[24]

- Na vertikálnej ose je hĺbka zásobníku.[24]
- Na horizontálnej ose sú volania radené podľa času volania, alebo podľa času trvania.

Listing 3.9: Ukážka použitia parametru *pp=flamegraph*

```
http://localhost:3000/articles?pp=flamegraph
```



Obrázok 3.9: Ukážka zobrazenia *pp=flamegraph*[24]

Z funkcionality nástroja Rack mini profiler vyplýva, že pri použití vo frameworku bude prebiehať profiling manuálne, pre jednotlivé kroky testovacieho scenára. Tejto skutočnosti sa bude musieť prispôsobiť návrh interpretácie výsledkov tak, aby vizualizácia výsledkov na prvý pohlaď indikovala najpomalšie requesty.

Kapitola 4

Návrh

V rámci analýzy sa podarilo definovať základy frameworku. Určilo sa, ako bude prebiehať výkonnostné testovanie. Z toho vyplynul tvar vstupu a výstupu testovania, ako aj spôsob spúšťania testov. V prípade vizualizácie výsledkov sa nepoužije existujúci nástroj, ale framework bude obsahovať vlastnú implementáciu. Spolu s tým sa zvolil aj nástroj, ktorý sa využije pri profilingu aplikácie.

Spolu s návrhom frameworku je vhodné definovať, nad ktorými vlastnosťami Redminu sa bude vykonávať výkonnostné testovanie.

4.1 Redmine

Redmine je open-source aplikácia pre projektové riadenie. Implementácia je napísaná v jazyku Ruby a využíva framework Ruby on Rails.[14]

Redmine obsahuje rozsiahle nástroje pre správu projektov a úloh. Pri práci s úlohami poskytuje možnosť emailových notifikácií, históriu zmien, správu príloh, či dokumentov a vykazovanie času.[15]

Podrobnejšie informácie o Redmine sú dostupné na webovej adrese `https://www.redmine.org/`

4.2 Základné testovacie scenáre

Pri návrhu základných scenárov sa brala do úvahy povaha Redmine aplikácie. Základnými objektami aplikácie sú projekty a úlohy, preto by práca s nimi mala byť súčasťou testovacích scenárov. Okrem týchto objektov by scenáre mali obsahovať aj ďalšie často používané objekty.

Nasledujúce scenáre sú pomerne jednoduché a zväčša sa zameriavajú na prácu s jedným typom objektu. Okrem toho je však potrebné otestovať aj situáciu, kedy simulujeme záťaž nielen v rozmedzí práce s jedným typom objektu. Preto je posledný scenár trochu komplikovanejší a je kombináciou všetkých predchádzajúcich.

4.2.1 Prihlásenie do aplikácie

Názov: Prihlásenie do aplikácie.
Popis: Užívateľ sa prihlási do aplikácie.
Predpoklad: Užívateľ musí byť zaregistrovaný v aplikácii.
Kroky testu:

1. Užívateľ sa prihlási do aplikácie.
2. Užívateľ sa odhlási z aplikácie.

Očakávaný výsledok: Úspešné prihlásenie užívateľa do aplikácie.

4.2.2 Zobrazenie projektu

Názov: Zobrazenie projektu.
Popis: Neprihlásený návštevník si zobrazí projekt.
Predpoklad: Existencia zobrazeného projektu v aplikácii.
Kroky testu:

1. Návštevník si zobrazí všetky projekty.
2. Vyberie si projekt, ktorý chce zobraziť.
3. Klikne na náhľad projektu.

Očakávaný výsledok: Úspešné zobrazenie zoznamu projektov, úspešné zobrazenie náhľadu vybraného projektu.

4.2.3 Zobrazenie úlohy

Názov: Zobrazenie úlohy.
Popis: Prihlásený užívateľ si zobrazí vybranú úlohu.
Predpoklad: Užívateľ musí byť zaregistrovaný v aplikácii. Existencia zobrazenej úlohy v aplikácii.
Kroky testu:

1. Užívateľ sa prihlási do aplikácie.
2. Zobrazí si všetky úlohy.
3. Vyberie si úlohu, ktorú chce zobraziť.
4. Klikne na náhľad úlohy.
5. Užívateľ sa odhlási z aplikácie.

Očakávaný výsledok: Úspešné prihlásenie užívateľa do aplikácie. Úspešné zobrazenie zoznamu úloh, úspešné zobrazenie náhľadu vybranej úlohy.

4.2.4 Vykázanie odpracovaného času na úlohe

Názov: Vykázanie odpracovaného času na úlohe.
Popis: Prihlásený užívateľ si zobrazí vybranú úlohu a vykáže čas odpracovaný na úlohe.
Predpoklad: Užívateľ musí byť zaregistrovaný v aplikácii. Existencia zobrazenej úlohy v aplikácii.
Kroky testu:

1. Užívateľ sa prihlási do aplikácie.
2. Zobrazí si všetky úlohy.
3. Vyberie si úlohu, na ktorej chce vykázať čas.
4. Klikne na náhľad úlohy.
5. Klikne na tlačidlo *Pridať čas*.
6. Vyplní a odošle formulár.
7. Užívateľ sa odhlási z aplikácie.

Očakávaný výsledok: Úspešné prihlásenie užívateľa do aplikácie. Úspešné zobrazenie zoznamu úloh, úspešné zobrazenie náhľadu vybranej úlohy a úspešné zobrazenie formulára na vykázanie času. Úspešné odoslanie formulára.

4.2.5 Aktualizácia stavu úlohy

Názov: Aktualizácia stavu úlohy.
Popis: Prihlásený užívateľ si zobrazí vybranú úlohu a aktualizuje jej stav.

Predpoklad: Užívateľ musí byť zaregistrovaný v aplikácii. Existencia zobrazenej úlohy v aplikácii.
Kroky testu:

1. Užívateľ sa prihlási do aplikácie.
2. Zobrazí si všetky úlohy.
3. Vyberie si úlohu, ktorú chce upraviť.
4. Klikne na náhľad úlohy.
5. Klikne na tlačidlo *Upraviť*.
6. Vyplní pole *Hotovo* na požadovanú hodnotu a odošle formulár.
7. Užívateľ sa odhlási z aplikácie.

Očakávaný výsledok: Úspešné prihlásenie užívateľa do aplikácie. Úspešné zobrazenie zoznamu úloh, úspešné zobrazenie náhľadu vybranej úlohy a úspešné zobrazenie editačného formulára. Úspešné odoslanie formulára.

4.2.6 Priechod širšou funkcionalitou Redminu

Názov: Priechod širšou funkcionalitou Redminu.
Popis: Užívateľ sa prihlási do aplikácie. Zobrazí si zoznam všetkých úloh a vytvorí novú úlohu pomocou formulára. Znova si zobrazí zoznam úloh. Vyberie si úlohu a zobrazí si jej náhľad. Potom zmení pomocou formulára atribút „Hotovo“ a úlohu sa pokúsi aktualizovať. Po tomto kroku si užívateľ zobrazí náhľad projektu prostredníctvom zoznam projektov. Následne si po tretí krát zobrazí zoznam úloh a vyberie úlohu, na ktorej chce vykázať čas. Pomocou formulára vytvorí nový výkaz o odpracovanom čase. Na koniec si užívateľ zobrazí zoznam vykázaného času, vyberie si výkaz a pokúsi sa tomuto výkazu zmeniť odpracovaný čas.
Predpoklad: Užívateľ musí byť zaregistrovaný v aplikácii.
Kroky testu:

1. Užívateľ sa prihlási do aplikácie.
2. Zobrazí si všetky úlohy.
3. Klikne na tatíčko nová úloha a zobrazí si formulár na vytvorenie úlohy.
4. Vyplní formulár a vytvorí úlohu.
5. Zobrazí si všetky úlohy.
6. Zobrazí si náhľad vybranej úlohy.
7. Klikne na tlatíčko upraviť.
8. Zmení hodnotu vlastnosti „Hotovo“ a aktualizuje úlohu.
9. Užívateľ sa pomocou tlačidla vo vrchnom menu presunie na zoznam projektov.
10. Zobrazí si náhľad zvoleného projektu.
11. Užívateľ si znova zobrazí všetky úlohy.
12. Vyberie si úlohu a znova zobrazí jej náhľad.
13. Užívateľ klikne na tlačidlo „pridať čas“.

14. Vyplní formulár a vytvorí výkaz o odpracovanom čase.
15. Zobrazí si všetky vykázane časy v tabe „Strávený čas“.
16. Vyberie si jeden výkaz a tomu vo formulári upraví odpracovaný čas.
17. Aktualizuje výkaz.
18. Užívateľ sa odhlási z aplikácie.

Očakávaný výsledok: Užívateľ sa úspešne prihlási do aplikácie. Úspešné zobrazenie zoznamu, zobrazenie náhľadu, vytvorenie a úprava menovaných objektov.

4.3 Návrh frameworku

Návrh frameworku vychádza z analýzy, kde boli definované základné vlastnosti, ktoré bude obsahovať. Podľa základných vlastností je framework rozdelený na viacero častí, z čoho každá časť reprezentuje riešenie jednej vlastnosti.

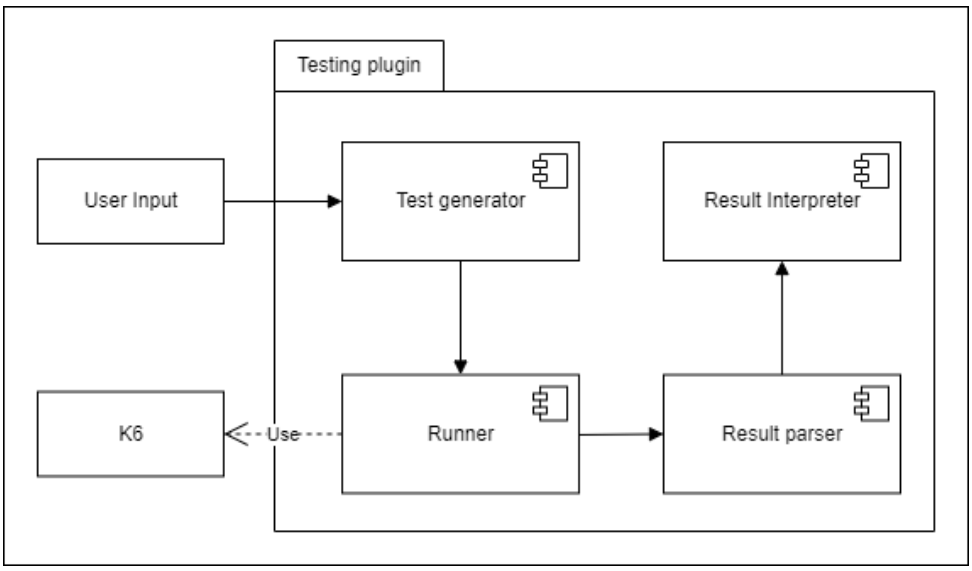
Konfiguráciu testov bude mať na starosti časť *Test generátor*. Na výkonnostné testovanie sa použije existujúci nástroj k6. Preto chovanie samotného testovania nie je nutné navrhnúť a implementovať. Avšak bude potrebné vytvoriť spúšťací mechanizmus pre tento nástroj. Táto časť frameworku bude *Runner*. O spracovanie výstupov z nástroja k6 sa postará *Result parser*. Spracované výstupy si následne prevezme *Result interpreter* a zobrazí ich užívateľovi.

Návrh vizualizuje diagram komponent na obrázku 4.1

4.3.1 Test generátor

Test generátor je časť frameworku, ktorá bude mať na starosti tvorbu testovacích skriptov pre nástroj k6. Testovacie skripty bude vytvárať z konfigurácie poskytnutej užívateľom. Je možné povedať, že bude tvoriť užívateľský vstup do celého procesu testovania.

Testovací skript sa skladá z viacerých častí, ktoré sú vymenované v sekcii 3.1.5. Časti, bez ktorých sa skript nezaobíde, sú *Options* a *Testovacia funkcia*. Ukážka testovacieho scriptu je zobrazená na listingu 4.1.



Obrázok 4.1: Diagram komponent návrhu frameworku

Listing 4.1: Ukážka testovacieho skriptu s *options* a *testovaciou funkciou*

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export const options = {
    vus: 1,
    duration: "1m",
    thresholds: {
        http_req_failed: ["rate < 0.01"],
        http_req_duration: ["p(99) < 1000"]
    }
};

export default () => {
    let response;

    response = http.get("https://localhost:3000/projects");
    check(response, {'page loaded successfully': (r) => r.
        status === 200});

    sleep(1)
};
```

Options je objekt, ktorý obsahuje vlastnosti testu. Môže sa v ňom definovať trvanie testu, záťaž na aplikáciu a podmienky úspešnosti.

Testovacia funkcia je JavaScriptová funkcia, ktorá obsahuje testovací scenár v podobe dotazov na endpointy testovanej aplikácie. Súčasťou môže byť aj overenie odpovede na dotázané endpointy. Toto overenie sa volá *Check* a nemá vplyv na úspešnosť testu, ak tak sám užívateľ nenastaví v objekte *Options*. Nesplnený *Check* indikuje, že daný dotaz neprebehol podľa očakávania, čo môže mať vplyv na výstup testu.

Test generátor poskytne užívateľovi rozhranie, pomocou ktorého si bude môcť obe spomenuté časti nakonfigurovať a vytvárať tak dynamické testovacie skripty.

4.3.2 Runner

Runner ponesie zodpovednosť za spúšťanie testov a prácu s nástrojom k6. Od *Test generátoru* si prevezme skript, ktorý následne predá nástroju k6 a spustí testovanie.

Po tom, čo k6 vykoná testovanie, si *Runner* prevezme výstupy testu a uloží ich. Potom zavolá *Result parser* a predá mu výstupy na ďalšie spracovanie.

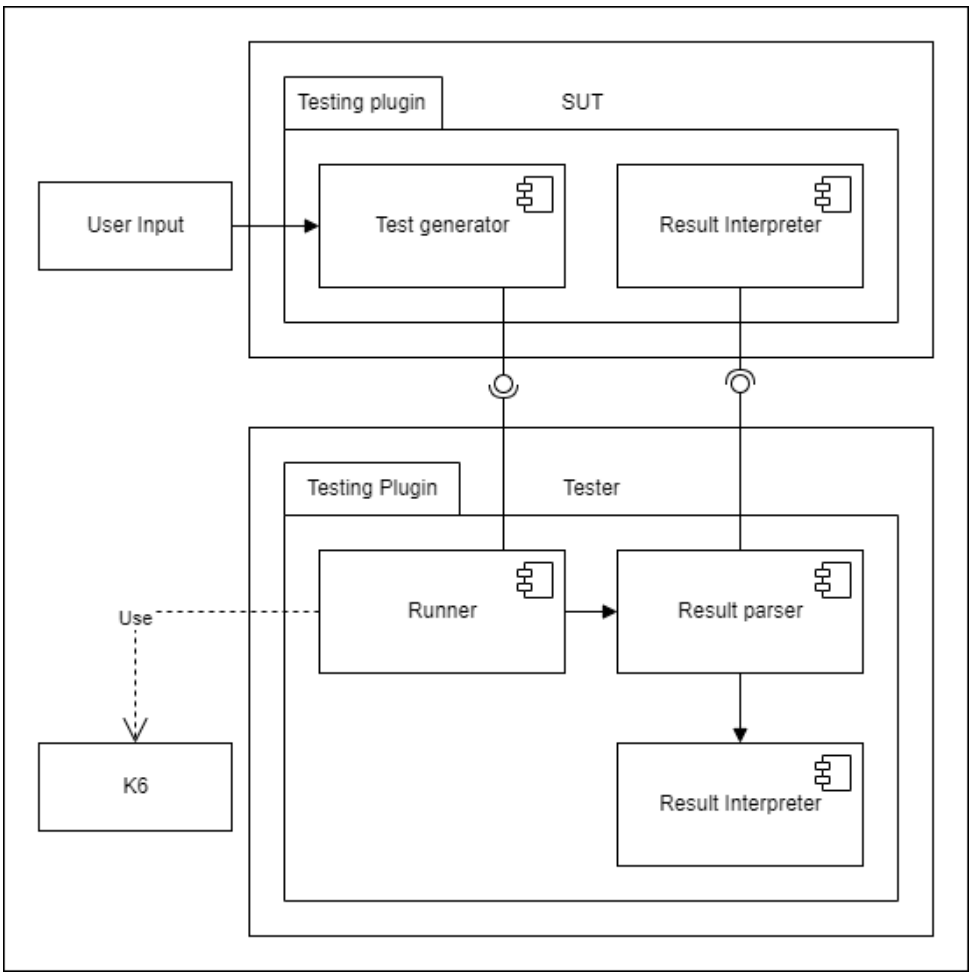
4.3.3 Oddelenie testovaného prostredia

Bolo by vhodné, aby testované prostredie bolo oddelené od prostredia, v ktorom sa bude spúšťať a bude prebiehať výkonnostné testovanie. Je to z dôvodu povahy niektorých typov testov, napríklad *Stress testov* (sekcia 2.5), ktorými ide o snahu tlačiť systém na hranicu zlyhania.

V takomto prípade by mohlo dôjsť k situácii, že aplikácia, ktorej súčasťou je framework, spustí testovanie, ako dôsledok testu spadne a nedôjde k interpretácii výsledkov. *Test generátor* aj *Runner* by sa preto mal prispôsobiť viacerým možnostiam, ako sa takejto situácii vyhnúť.

Jedným spôsobom je, že nakonfigurovaný skript sa bude dotazovať na iné prostredie ako to, v ktorom bol konfigurovaný.

Ďalším spôsobom bude existencia párového prostredia, ktoré poslúži ako externý *Runner*, teda Tester. Aplikácia by požiadala Testera o spustenie



Obrázok 4.2: Diagram komponent návrhu izolácie testovaného prostredia

testu s nakonfigurovaným skriptom. Po skončení by sa výsledky spracovali na strane Testera a následne predali späť aplikácii. V prípade, že by predsa len došlo k zlyhaniu systému ako dôsledok testu, si Tester uloží výsledky a bude ich schopný vizualizovať vo svojom prostredí. Takýto návrh je vizualizovaný pomocou diagramu komponent na obrázoku 4.2.

4.3.4 Result parser

Úlohou *Result parseru* bude spracovať výstupy z nástroja k6, ktoré poskytne *Runner*. Predvolený textový výstup k6, ktorý je znázornený na obrázku 3.2, poskytuje z obecného hľadiska užitočné informácie, ale pri určovaní problémových miest v aplikácii sú nedostatočné. Informácie vo formáte Json sú detailnejšie a preto aj vhodnejšie.

Z toho dôvodu bude *Result parser* prijímať formát JSON ako vstup. Ten sa následne preiteruje a zozbierjú sa potrebné dáta. Zameria sa hlavne na dáta ako sú:

- Počet virtuálnych užívateľov počas testu.
- Počet splnených a nesplnených kontrol (*Check*).
- Miera neúspešných requestov.
- Dáta o každej prahovej hodnote (*Threshold*), ktorú užívateľ nakonfiguruje, a teda aj rozhodnutie, či bola prahová hodnota splnená.

Okrem spomenutých dát, budú potrebné aj informácie o trvaní jednotlivých requestov. Na základe týchto informácií sa určia requesty, na ktoré je potrebné sa zamerať s profilingom. Ako námet poslúži textový výstup. Pre každý dotázaný endpoint treba zozbierať a agregovať dáta, ktoré by obsahovali:

- priemernú hodnotu trvania requestu,
- minimálnu hodnotu trvania requestu,
- maximálnu hodnotu trvania requestu,
- priemernú hodnotu trvania requestu pre percentil 90,
- priemernú hodnotu trvania requestu pre percentil 95,
- priemernú hodnotu pre akýkoľvek iný percentil, ktorý si užívateľ zadá do *Thresholdov*.

Výsledné spracované dáta sa uložia do Json súboru spolu s ostatnými výstupmi. Bude vhodné, ak pôvodné dáta ostanú nedotknuté a k dispozícii pre prípadnú detailnejšiu analýzu.

4.3.5 Result interpreter

Spracované dáta si prevezme *Result interpreter* a postará sa, aby boli zobrazené užívateľovi. Súčasťou tejto časti frameworku budú teda triedy, ktorých inštancie reprezentujú spracované dáta, spolu s pôvodnými výstupmi, ktoré

vráti nástroj k6. Interpretácia bude implementovaná tak, aby obsah súborov bolo možné zobraziť vo webovom prostredí.

Okrem toho by mali byť všetky tieto súbory vo webovom prostredí zobrazené ako zoznam, ktorým budú môcť užívatelia navigovať a mať tak k dispozícii aj predchádzajúce výsledky, nie len tie aktuálne.

4.4 Interpretácia výsledkov

Zozbierané a spracované výsledky budú interpretované tak, aby užívateľovi zobrazili, ako prebiehal test a miesta, kde dochádzalo k spomaleniu aplikácie.

Hodnoty, ktoré by bolo vhodné zobraziť sú *Thresholdy*, *Checky* a s nimi aj patričné indikátory, či boli splnené alebo nie. To dá užívateľovi na prvý pohľad najavo, či bol test úspešný, alebo nie, a či dotázané endpointy vrátili očakávaný response.

Na zobrazenie informácií o priebehu testu budú použité grafy. Tie budú znázorňovať vývoj hodnôt jednotlivých zozbieraných metrík počas celého trvania testu. Pomocou grafov bude možné znázorniť aj počet virtuálnych užívateľov alebo počet neúspešných requestov.

Agregované dáta o trvaní jednotlivých requestov by bolo dobré zobraziť formou, pri ktorej bude možné jednoducho porovnávať hodnoty rovnakého typu pre jednotlivé requesty. Pre tento účel bude použitá tabuľka, pričom riadky budú reprezentovať jednotlivé requesty a stĺpce budú zoskupovať hodnoty jedného agregovaného typu.

Porovnávanie hodnôt v jednotlivých stĺpcoch sa môže využiť pri rozhodovaní, kde sa bude vykonávať profiling v prípade, že by test nebol úspešný a nesplnil všetky požadované *Thresholdy*.

4.5 Zapojenie do Redmine aplikácie

Aplikácia Redmine má stále aktívny vývoj, pričom dochádza k vydaniu novej verzie v pravidelných, niekoľko mesačných intervaloch. Zásah do Redmine implementácie by mohol skomplikovať, v krajnom prípade aj znemožniť, prechod aplikácie na novú verziu.

Z tohto dôvodu nie je správne riešenie pridať navrhnutý framework priamo do implementácie. Redmine však ponúka spôsob, akým sú riešené vlastné rozšírenia a zásahy do implementácie. Jedná sa o implementáciu Redmine pluginu.

Správny spôsob zapojenia navrhnutého frameworku do Redmine aplikácie je implementácia vo forme Redmine pluginu.

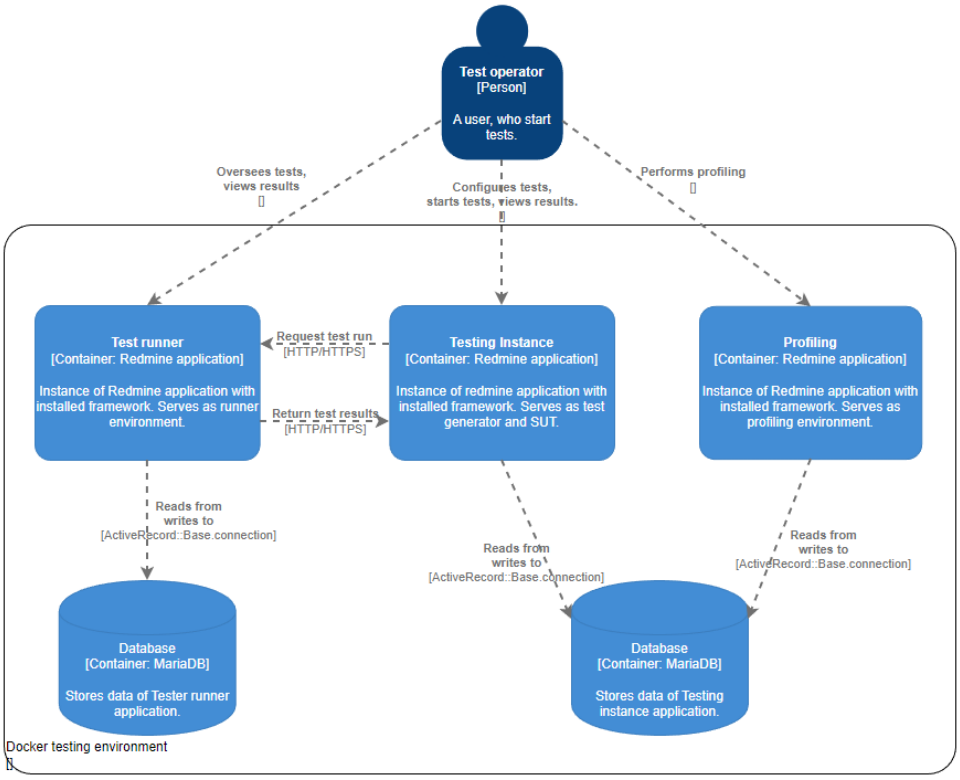
4.6 Prostredie v dockeri

Jeden z bodov zadania naznačuje použitie prostredia Dockeru pre využitie navrhovaného frameworku na otestovanie aplikácie. Jednou z výhod použitia prostredia Dockeru je izolácia prostredia v rámci kontajnerov.[26] Prostredie v kontajneri je izolované od svojho okolia, pokiaľ nemá kontajner definovanú nejakú závislosť. Kontajnerov môžeme v rámci prostredia definovať niekoľko. Docker ponúka jednoduchú konfiguráciu kontajnerov, ktorá je definovaná v súbore.[26] Ten sa môže kedykoľvek spustiť a vytvoriť tak novú verziu prostredia. Použitím Dockeru sa tiež vylučuje problém so spustením aplikácií na rôznych platformách. Okrem toho, Docker poskytuje aj jednoduché zapojenie do CI pipeliny.[26]

Z toho dôvodu je potrebné navrhnúť, ako vytvoriť prostredie, v ktorom bude možné rozbehnúť Redmine aplikáciu spolu s integráciou navrhovaného pluginu.

Obrázok 4.3 vizualizuje návrh takéhoto prostredia. Skladá sa z piatich kontajnerov. Dva z nich obsahujú image databázy. Zvyšné tri obsahujú image aplikácie Redminu s integrovaným pluginom.

Pre oddelenie testovaného prostredia od prostredia, v ktorom prebiehajú



Obrázok 4.3: Návrh prostredia v dockeri

testy, bude existovať párová Redmine aplikácia, ktorá poslúži ako Tester, respektíve *Runner* a *Result parser*. Tester bude viazaný s jedným databázovým kontajnerom a bude viditeľný pre ostatné aplikácie na verejnom porte.

Ďalším kontajnerom s imagom Redmine aplikácie bude testovaná inštancia. Toto testované prostredie bude tiež naviazané na vlastný kontajner s databázou. Testovaná inštancia zastupuje *Test generátor* a *Result interpreter*. Z tohto prostredia sa budú inicializovať requesty na prostredie Testera.

Posledným kontajnerom s imagom Redmine aplikácie bude prostredie pre profiling. Toto prostredie nemá vlastnú databázu, ale využíva databázu testovaného prostredia. Je to z dôvodu, že počas behu testu nástroj k6 pracuje s reálnym prostredím, a teda objekty, ktoré sa počas testu vytvoria, v prostredí aj zostanú. Profiling teda môže pracovať s naplneným prostredím a nie je nutné tento proces opakovať.

4.7 Prostredie profilingu

Profiling bude prebiehať v osobitnom prostredí, aby bol oddelený proces profilingu od procesu výkonnostného testovania.

Jedným dôvodom tohto rozhodnutia je, že profiling aplikácie môže využívať zdroje prostredia. To by mohlo mať nežiadúci vplyv na výstupy testov.

Pod pojmom osobitné prostredie sa nemyslí len samostatný kontajner a image aplikácie v dockeri. Pre profiling sa definuje aj vlastné *Rails* prostredie určené špeciálne pre profiling. V rámci tohto prostredia sa definujú profilingové knižnice (gemy) a potrebné premenné, ktoré sa počas profilingu využíjú.

Pomocou dockerovej konfigurácie sa docieli, aby image aplikácie spúšťal definované profilingové prostredie a držal aplikáciu pripravenú na použitie.

Kapitola 5

Implementácia frameworku

Podľa návrhu framework obsahuje implementáciu všetkých navrhovaných komponentov. Súčasťou je aj implementácia prostredia dockeru a implementácia potrebná pre profiling.

Chovanie frameworku je znázornené sekvenčným diagramom (obrázok 5.1) a diagramom aktivít (obrázok 5.2)

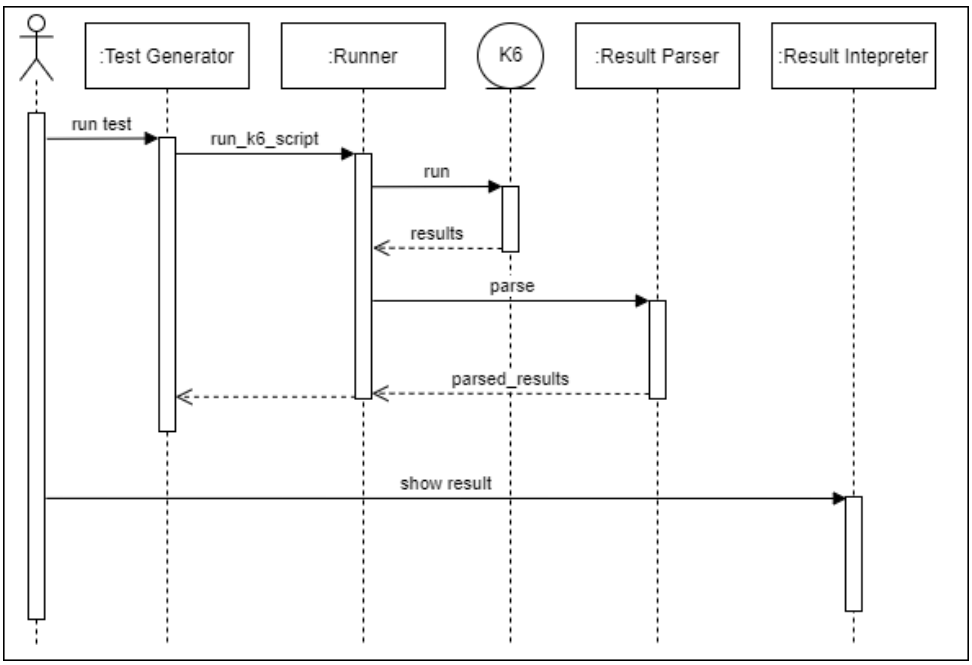
Framework sa integruje do Redmine aplikácie ako plugin. Rozširuje administráciu a administračné menu aplikácie o nový endpoint, ktorý dovoľuje administrátorovi aplikácie prístup ku implementovaným objektom a rozhraniu.

Všetky definície pluginu sú obsiahnuté v súbore *init.rb*. Nové endpointy sú definované v súbore *config/routes.rb*. Rozšírenie databázy o nové tabuľky a počiatočné dáta pluginu je definované v migračných skriptoch v zložke *db/migrate*.

Implementácia tried je sústredená do zložky */app* a */lib*. Zložka */patches* obsahuje modifikácie pôvodnej implementácie redminu.

Testy implementácie sa nachádzajú v zložke */test/spec*. Pri testovaní sa využíva gem, ktorý sa volá *Rspec*.

Štruktúra implementovaného pluginu je na obrázku 5.3.



Obrázok 5.1: Sekvenčný diagram procesu testovania

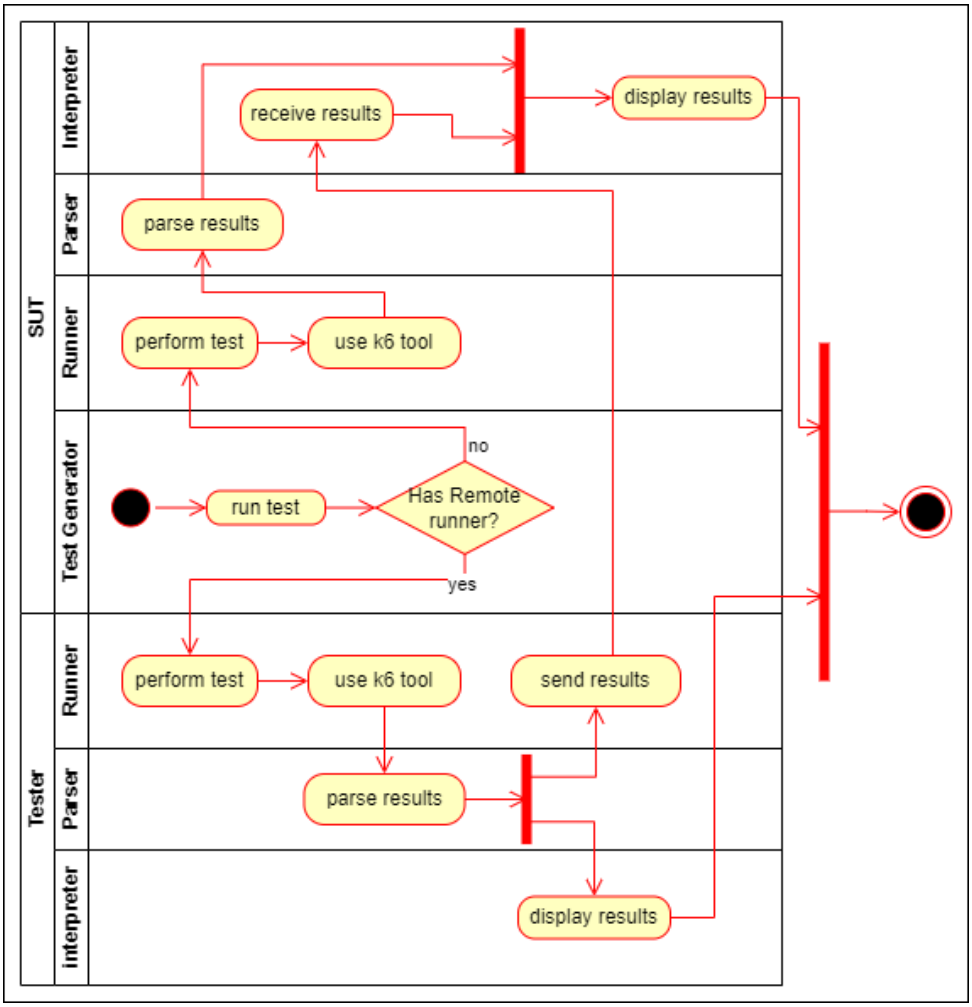
5.1 Test generátor

Test generátor sa skladá zo štyroch tried *K6TestConfiguration*, *K6TestScenario*, *K6TestType*, *K6TestScript*. Tie reprezentujú celý proces konfigurácie testov užívateľom. Objekty, respektíve inštancie týchto tried sa ukladajú do databáze. Návrh tried je postavený tak, aby boli konfigurácie znovu použiteľné a užívateľ nemusel pre každý test konfigurácie opakovať. Na obrázku 5.4 je zobrazený class diagram modelov.

Trieda *K6TestConfiguration* definuje vlastnosti pre triedy *K6TestScenario* a *K6TestType*. V jej implementácii sa využíva princíp *Single-Table inheritance*.

Single-Table inheritance, alebo skrátene STI, je princíp, pri ktorom viacej tried dedí od jednej triedy. Táto trieda má definovanú tabuľku v databáze, ktorú zdieľajú všetky dediace triedy. To znamená že tieto triedy disponujú rovnakými vlastnosťami. Tabuľka musí mať definovaný jeden stĺpec, ktorý určuje príslušnosť k dediacej triede.

Prístup STI sa zvolil preto, že triedy *K6TestScenario* a *K6TestType* zdieľajú rovnaké vlastnosti, ale z pohľadu väzieb je potrebné ich rozlíšiť.



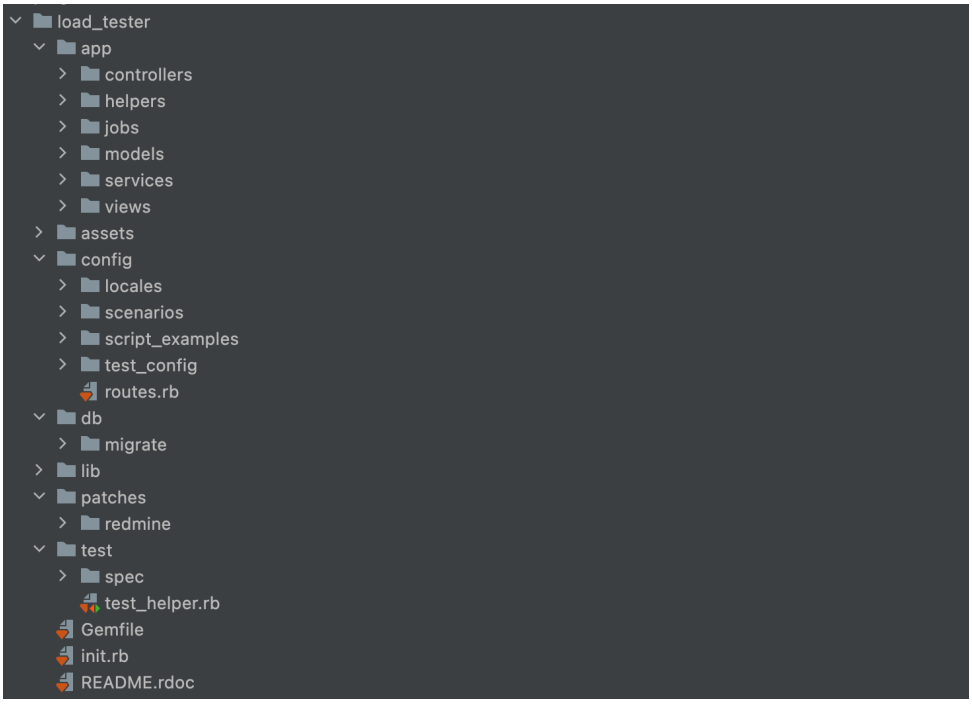
Obrázok 5.2: Diagram aktivít procesu testovania

Triedy *K6TestScenario* a *K6TestType* reprezentujú časti skriptu, ktoré užívateľ konfiguruje. Obidve disponujú atribútom *k6_script_configuration*, ktorý je typu Json. *K6TestScenario* v tomto atribúte obsahuje kroky scenára a *K6TestType* nastavenia testu.

Obidve triedy sú pod správou vlastného kontroleru, ktorý poskytuje užívateľské rozhranie pre vytvorenie, editáciu a zmazanie inštancie.

Trieda *K6TestScript* spája všetky potrebné časti dohromady a vytvára tak test. Každá inštancia má práve jeden scenár a práve jednu konfiguráciu testu (typ), preto sa jedná o agregačnú väzbu. Toto je dôvod, prečo je potrebné od seba odlíšiť triedy *K6TestScenario* a *K6TestType*.

K6TestScript spravuje *K6TestScriptController*. Obsahuje užívateľské ro-



Obrázok 5.3: Štruktúra implementovaného pluginu

zhranie pre vytvorenie, editáciu a zmazanie inštancie. Okrem toho ponúka aj rozhranie pre runner, pomocou ktorého predá testovací skript a možnosť spustenia runneru. Kontroler tiež využíva pomocné metódy obsiahnuté v module *K6TestScriptsHelper*.

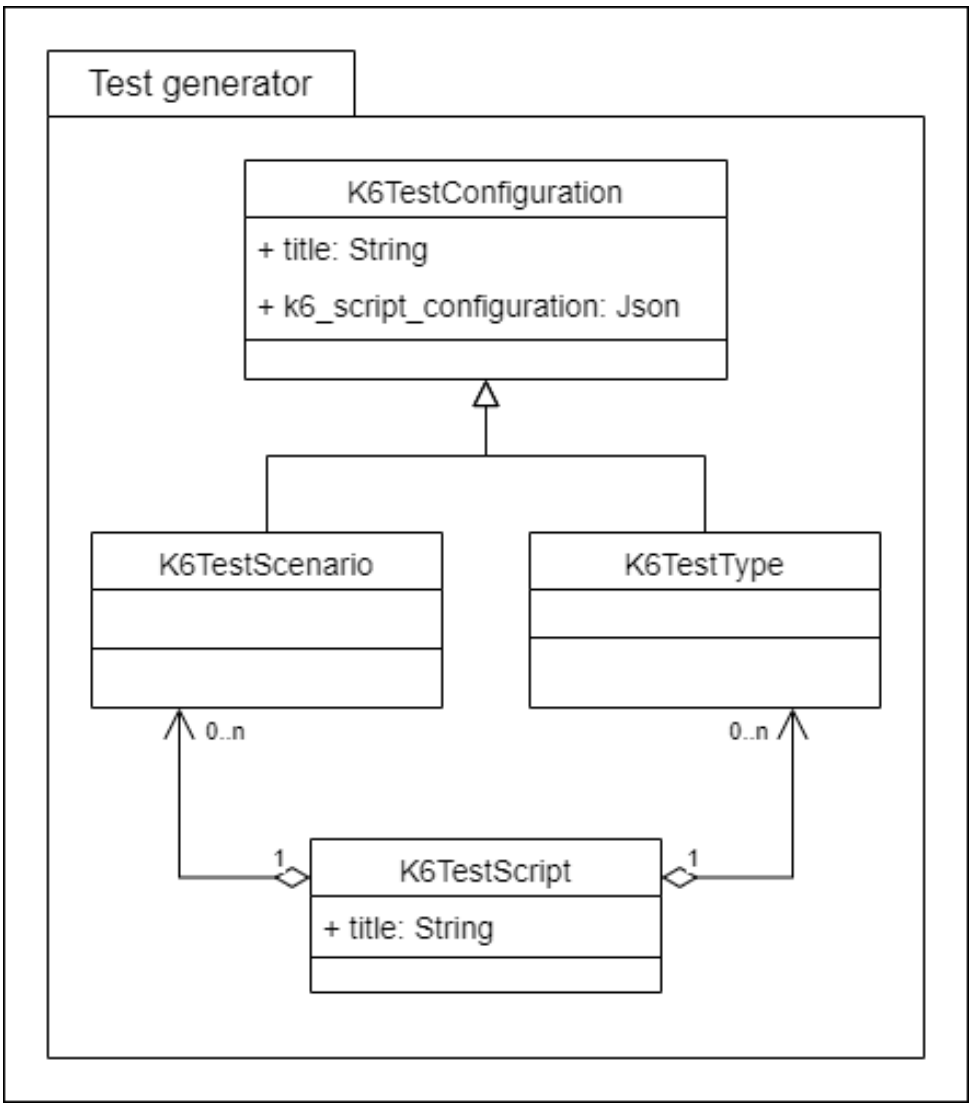
5.1.1 Rozhranie pre runner

Rozhranie pre runner je implementované ako sada troch endpointov pod správou *K6TestScriptControlleru*.

Prvý endpoint je zobrazenie testovacieho skriptu vo formáte javascriptu. Je to zobrazenie inštancie triedy *K6TestScript*, ktorá je zostavená z konfigurácie testu a scenára.

Využitím endpointu pre zostavenie testovacieho skriptu sa vytvorila možnosť predania skriptu externému runneru.

Druhý a tretí endpoint slúži k spusteniu testov. Pomocou druhého endpointu sa spustí jeden test a pomocou tretieho endpointu sa spustia všetky testy



Obrázok 5.4: Test generátor class diagram

v určenom poradí. V oboch prípadoch sa zavolá *runner*. Pokiaľ sa využíva externý *runner*, tak sa vytvorí request, ktorý sa dotazuje na vystavený endpoint tohto *runneru*.

5.2 Runner

Runner je implementovaný pomocou tried *K6RunnerJobBase*, *K6LocalRunnerJob*, *K6RemoteRunnerJob*, ktoré dedia od ActiveJob::Base. Trieda ActiveJob::Base je súčasťou frameworku ActiveJob a Ruby on Rails.

ActiveJob môžeme definovať ako background job, prípadne background task, ktorý je možné volať príkazmi *peform_now* a *perform_later*.

Tento prístup bol zvolený z dôvodu, že niektoré testy, ako napríklad *Soak testy*, môžu bežať hodiny. Je preto potrebné spúšťať testy asynchronne.

K6RunnerJobBase je trieda, ktorá združuje logiku pre *K6LocalRunnerJob* a *K6RemoteRunnerJob*. Obsahuje definície všetkých spoločných metód a atribútov. Nemá implementovanú metódu *perform*.

K6LocalRunnerJob dedí od *K6RunnerJobBase* a reprezentuje lokálny *Runner* aplikácie. Je priamo volaná *K6TestScriptControllerom*. Trieda má definovanú funkciu *perform*, v ktorej sa nachádza logika spúšťania testov.

K6RemoteRunnerJob rovnako dedí od *K6RunnerJobBase* a reprezentuje externý *Runner* aplikácie. Triedu volá *K6RunnerController*, ktorý poskytuje endpoint pre prijímanie requestov od *K6TestScriptControlleru* cudzej aplikácie. Trieda má rovnako definovanú funkciu *perform*, v ktorej sa nachádza logika spúšťania testov. Je však rozšírená o zostavenie a odoslanie odpovede aplikácii, ktorá požiadala o test.

Funkcia *perform* volá k6 CLI rozhranie pomocou *backticku*. *Backtick* (listing 5.1) je jeden zo spôsobov, ako je možné v Ruby kóde používať Shell príkazy. Základný príkaz k6 CLI rozhrania je *k6 run *názov_scriptu**. Príkaz je možné rozšíriť o definíciu formátu výstupov, prípadne o prístupy, ak sa použije špeciálny typ výstupu, ako je *Prometheus*. Implementácia frameworku zahŕňa možnosť použitia *Promethea* a *Grafany*.

Listing 5.1: Backtick s interpoláciou stringu

```
# example of running local script with k6
result = ‘k6 run k6_test_script.js‘

# example of running k6 script with URL source
script_url = https://example_application_domain.com/
               k6_test_scripts/14/k6_test_script.js
result = ‘k6 run #{script_url}‘
```

Príkaz *k6 run* môže ako argument brať cestu k lokálnemu skriptu, alebo URL. Tu sa využije vystavený endpoint pre zobrazenie testovacieho skriptu. Má to však jeden háčik. Príkaz *k6 run* príjma iba URL s protokolom HTTPS. V prípade lokálneho vývojového prostredia, pred spustením testu, sa testovací skript uloží do súboru pomocou funkcie *wget*. Pri spustení príkazu *k6 run* sa

použije cesta k tomuto súboru ako argument, a po skončení testu je súbor so skriptom zmazaný.

Po skončení testov sa výstup ukladá do zložky */public/k6_run_results*. Ak zložka *k6_run_results* neexistuje, pred prvým testom sa vytvorí. Aktuálna implementácia ukladá textový formát výstupu spolu s formátom Json.

Po uložení výstupov sa zavolá *Result parser*, aby výstupný Json súbor spracoval a poskytol dáta potrebné pri vizualizácii výsledkov.

Súčasťou runneru je aj model *K6TestingClient*. Ten zastáva aplikácie, ktoré môžu využívať danú aplikáciu ako externý *Runner*. *Runner* musí poznať aplikáciu, ktorá žiada o spustenie testu. V opačnom prípade je request zamietnutý. Inštancia triedy sa ukladá do databázy a obsahuje informácie ako je názov aplikácie, URL aplikácie a unikátny kľúč aplikácie pre prístup k endpointu, ktorý zobrazuje testovacie skripty. Súčasne sa vytvára klientska zložka v */public/k6_run_results* pre zjednodušenie organizácie výstupov.

Triedu *K6TestingClient* má na starosť jej vlastný kontroler s rozhraním pre správu jej inštancii.

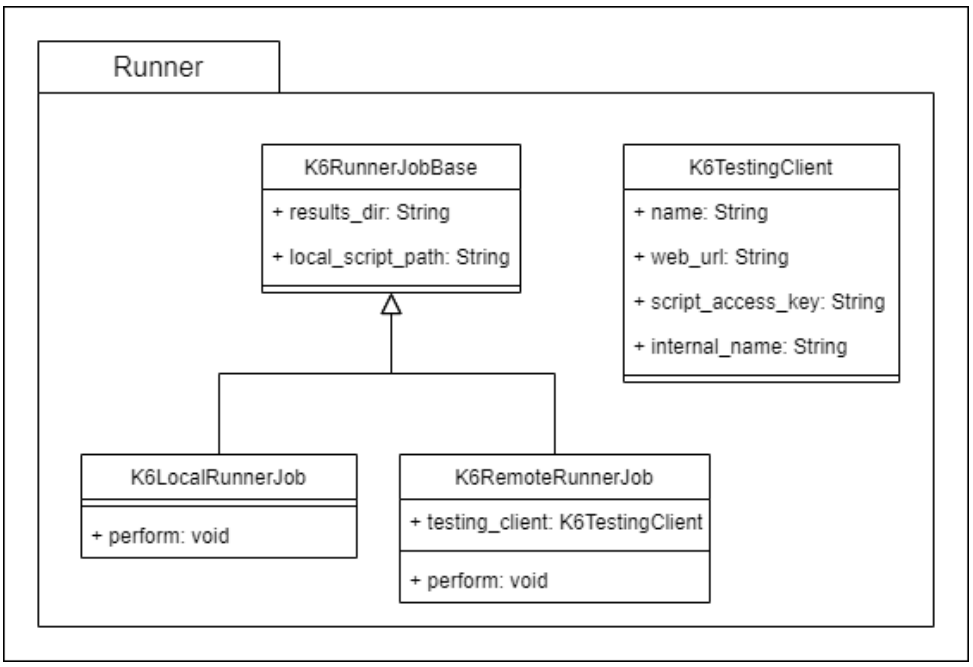
V prípade vystaveného endpointu *K6RunnerControllerom* pre ostatné aplikácie dochádza k overenie unikátneho kľúča vygenerovaného aplikáciou externého *Runneru*.

Implementáciu *Runneru* znázorňuje class diagram 5.5.

5.3 Triedy K6Settings a K6Threshold

Menované triedy nepatria do žiadnej konkrétnej časti frameworku, no pritom sú používané viacerými z nich.

K6Settings je teda trieda, ktorá obsahuje nastavenia pluginu. Nastavujú sa v nej premenné pre použitie externého *Runneru*. Napríklad jeho access_key, či URL. Môžeme tu nastaviť api_key v prípade, že sa skripty budú dotazovať na Redmine API a je potrebná autentizácia. V prípade, že *Runner* využíva ako output *Promethea*, zadávajú sa tu prístupy.

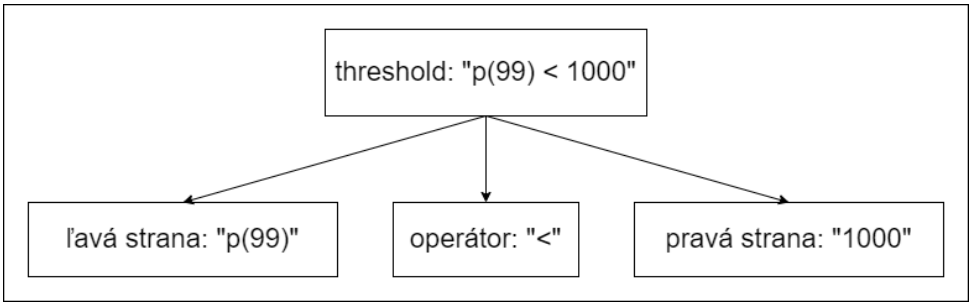


Obrázok 5.5: Runner class diagram

Trieda je implementovaná ako model a vlastnosti ukladá do databázy. Využíva návrhový vzor *Singleton*, a teda v celom frameworku má len jednu inštanciu. Táto inštancia je vytvorená pri prvom volaní. Volanie tejto inštancie je implementované cez triedne metódy, aby sa predišlo inicializácii duplicitných inštancií.

Triedu spravuje *K6SettingsController*, ktorý poskytuje iba rozhranie pre úpravu inštancie. Užívateľ je pri prvom prístupe do pluginu presmerovaný na formulár *K6Settings*, kde vyplní potrebné nastavenia na fungovanie pluginu.

Trieda *K6Threshold* nie je modelom a nevyužíva databázu. Je to len pomocná trieda, ktorá reprezentuje thresholdy definované užívateľom v rámci testovacieho skriptu. Pri inicializácii prijíma, ako argument, textovú formu thresholdu, ktorú rozdelí na 3 časti: ľavú stranu, pravú stranu a operátor (obrázok 5.6). S inštanciami triedy *K6Threshold* pracujú časti frameworku *Result parser* a *Result interpreter*.



Obrázok 5.6: Rozdelenie textovej formy thresholdu na 3 časti

5.4 Result parser

Časť frameworku *Result parser* (obrázok 5.7) sa skladá zo sady servisných tried a je rozdelený na 3 časti:

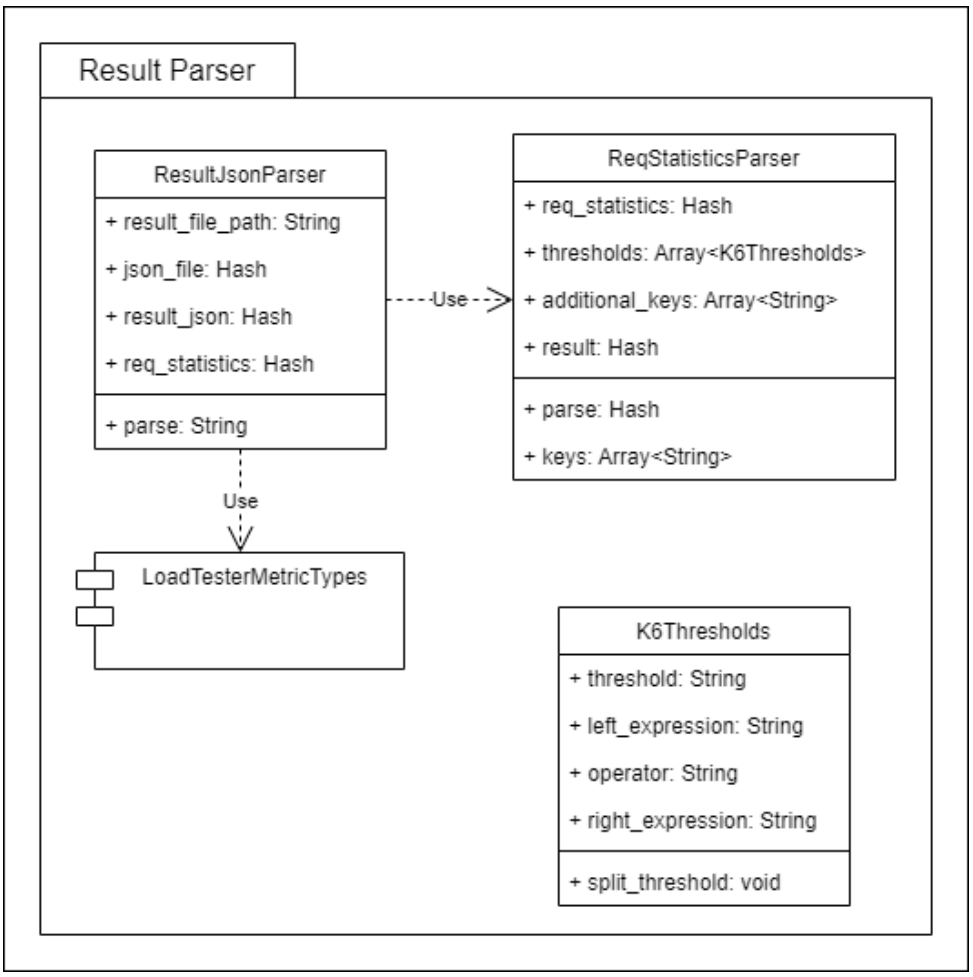
- trieda *ResultJsonParser*,
- trieda *ReqStatisticsParser*,
- modul *LoadTesterMetricTypes*.

Vstupným bodom do procesu spracovania výstupov je *ResultJsonParser*. Inicializačným argumentom je cesta k výstupom testov vo formáte Json. Trieda má implementovanú public metódu *parse*, v ktorej prebieha celý proces.

Obsahom výstupného súboru z testov, vo formáte Json, je veľké množstvo Json objektov oddelených znakom nového riadku. Tieto Json objekty sú vlastne každá metrika a každý údaj danej metriky cez všetky dotázané requesty počas priebehu testu.

Začiatkom procesu je iterácia cez toto obrovské množstvo objektov. Vyberajú sa z nich metriky spolu s údajmi, na ktoré je naviazaný nejaký threshold. Okrem týchto metrík sa vyberajú aj metriky, popisujúce počet virtuálnych užívateľov v priebehu testu a miera splnených checkov.

Počas tejto iterácie sa súčasne zaznemnávajú aj informácie o trvaní každého requestu do osobitného objektu.



Obrázok 5.7: Result parser class diagram

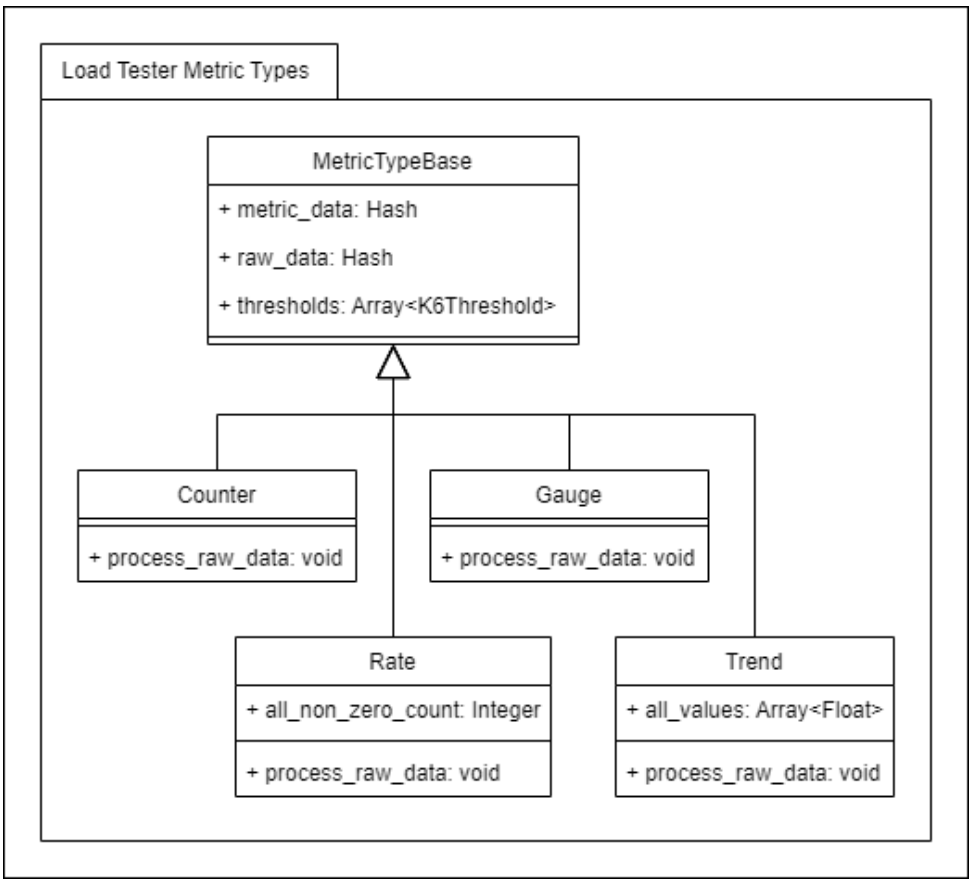
Po iterácii je použitý modul *LoadTesterMetricTypes* pre spracovanie dát vybraných metrík.

Ďalším krokom je spracovanie údajov o trvaní každého requestu. To má na starosti *ReqStatisticsParser*.

Posledným krokom procesu je uloženie výsledkov do nového súboru. Návratová hodnota funkcie *parse* je cesta k tomuto novo vytvorenému súboru.

Modul *LoadTesterMetricTypes* obsahuje 5 tried: *MetricTypeBase*, *Counter*, *Gauge*, *Rate*, *Trend*. Class diagram modulu je znázornený na obrázku 5.8.

Trieda *MetricTypeBase* je rodičovská trieda, ktorá obsahuje zdieľanú implementáciu, vrátane inicializácie a spracovania vstupu.



Obrázok 5.8: Class diagram modulu *LoadTesterMetricTypes*

Vstupom sú zozbierané dáta konkrétnej metriky. Dáta sú zoskupované podľa času, v ktorom boli zozbierané a následne sú spracované na základe typu metriky. Každá metrika má vlastné agregované dáta, na ktoré sa zameriava. Nástroj k6 pracuje so štyrmi typmi metrík:

- Counter,
- Gauge,
- Rate,
- Trend.

Counter je metrika, ktorá postupne sčíta pridané hodnoty.[12] Trieda *Counter* prejde všetky hodnoty, spočíta ich počet a celkovú sumu, ktorú predstavujú dohromady.

Gauge je metrika, ktorá ukladá minimálnu pridanú hodnotu, maximálnu pridanú hodnotu a posledné pridané hodnoty.[12] Trieda *Gauge* prejde všetky hodnoty zoskupené podľa času a ponechá si len poslednú. Okrem toho nájde a uloží minimálnu a maximálnu hodnotu zo všetkých.

Rate je metrika, ktorá zisťuje percento nenulových hodnôt.[12] Trieda *Rate* zozbiera z dát celkový počet všetkých namerných hodnôt, celkový počet všetkých nenulových hodnôt a počet všetkých nenulových hodnôt pre každý časový moment.

Trend je metrika, ktorá počíta štatistické hodnoty nad všetkými pridanými hodnotami.[12] Trieda *Trend* vypočíta priemernú hodnotu, priemernú minimálnu hodnotu a priemernú maximálnu hodnotu pre každý časový moment. Okrem toho zozbiera všetky hodnoty, aby z nich bolo možné vpočítať percentil. Pre výpočet percentilu sa používa gem *descriptive_statistics*.

Všetky triedy, ktoré sú súčasťou modulu *LoadTesterMetricTypes*, nemajú konkrétny výstup, ale modifikujú vstupný objekt. Ten je vo vlastníctve inštancie triedy *ResultJsonParser*, ktorá triedu na spracovanie metriky zavolala.

Poslednou triedou je *ReqStatisticsParser*. Jej úlohou je spracovať individuálne zozbierané dáta o trvaní jednotlivých dotazov. Dotazy sa delia na základe prideleného názvu v podobe tagu a type metódy. Pre každý dotaz sa následne vypočítajú agregované hodnoty ako je celkový počet, minimum, maximu, priemerná hodnota, percentil 90 a percentil 95. Okrem toho sa medzi tieto hodnoty pridá akýkoľvek percentil, ak je súčasťou thresholdov. Rovnako ako v prípade trendu, aj tu je pre výpočet percentilu využitý gem *descriptive_statistics*.

Výstupom triedy sú spracované dáta, ktoré si následne *ResultJsonParser* uloží do výstupného objektu. Trieda tiež poskytuje public metódu, ktorá vráti zoznam názvov všetkých vypočítaných agregovaných hodnôt.

5.5 Result interpreter

Poslednou časťou frameworku je *Result interpreter*. Jeho implementácia obsahuje dve triedy *K6Result* a *K6ParsedResult*. Ich úlohou je reprezentácia výstupov testu v rámci užívateľského webového rozhrania. Class diagram je zobrazený na obrázku 5.9.

K6Result je jednoduchá trieda, ktorá zastupuje akýkoľvek súbor v zložke */public/k6_run_results*. Inicializačný argument je cesta od rootu aplikácie k súboru, ktorá je pri inicializácii rozobratá a rozdelená na vlastnosti triedy. Tie popisujú čas spustenia testu, názov skriptu a prípadne rodičovskú zložku, ak nejakú má. Trieda poskytuje adresu detailného náhľadu.

V prípade, že sa jedná o výstup vygenerovaný nástrojom K6, je tento súbor zobrazený vo webovom prostredí taký, ako je. Ak ide o výstup vytvorený časťou frameworku *Result parserom*, tak je zobrazený náhľad parsovaného súboru. Pre tento endpoint je súbor zastúpený triedou *K6ParsedResult*.

K6ParsedResult je teda trieda, ktorá pri detailnom zobrazení zastupuje súbor vytvorený *Result parserom*. Trieda očakáva, akú štruktúru má tento súbor a načíta si z neho všetky potrebné informácie do svojich atribútov.

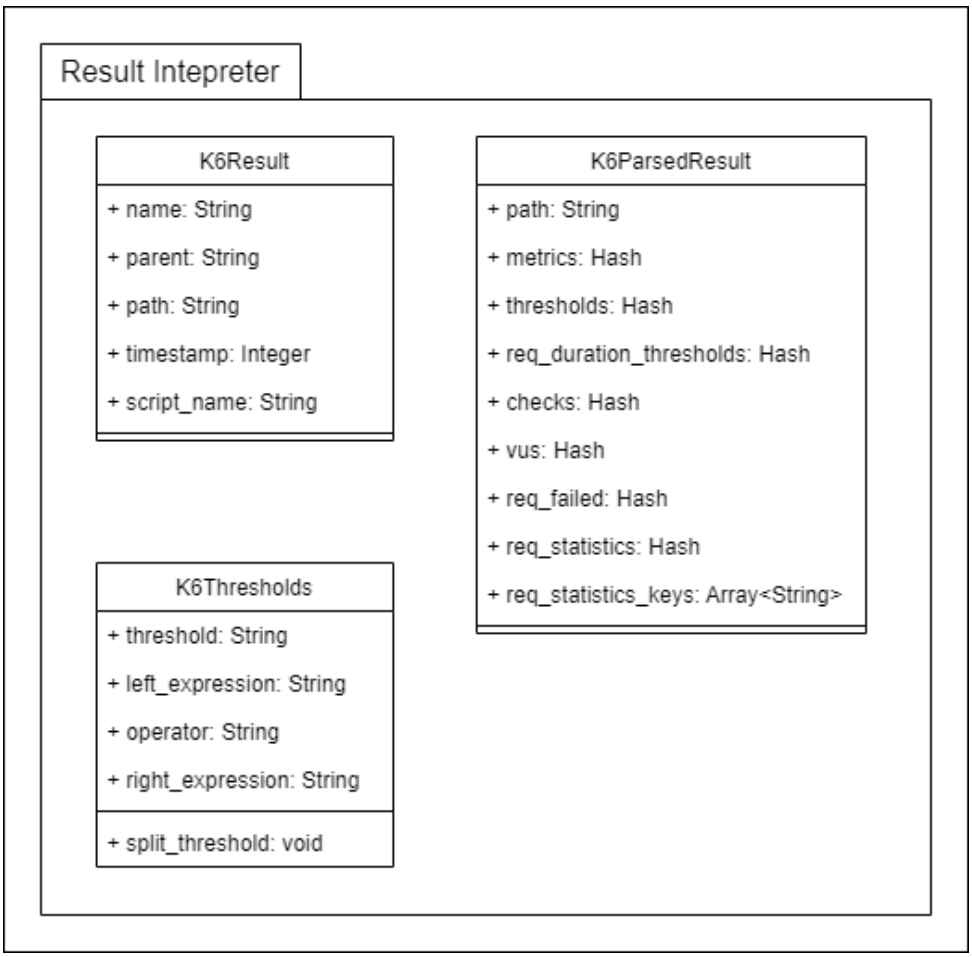
Obidve triedy spravuje *K6ResultsController*, ktorý ponúka rozhranie pre zoznam existujúcich výstupov a ich detailný náhľad. Okrem toho ponúka ešte ďalšie dva endpointy. Jeden slúži ako príjemca výstupov odoslaných externým runnerom. Druhý spúšťa funkciu vygenerovania databázového dumpu aplikácie.

5.5.1 Vizualizácia výstupov

Výsledky testu sú zobrazené na jednej stránke, ktorá je implementovaná pomocou parciálnych ERB šablón. Do nich sú dopĺňané dáta inštanciou triedy *K6ParsedResult*.

Navrchu stránky, ako aj súčasťou URL, je názov súboru. Hneď prvá informácia, ktorú môže užívateľ vidieť na stránke, sú zobrazené thresholdy a checky testu. (obrázok 5.10). Vedľa nich sú indikátory, ktoré jasne naznačujú či boli podmienky testu splnené. U checkov je naviac zobrazené, koľko z nich bolo úspešných.

Následne pokračuje vizualizácia priebehu testu pomocou viacerých grafov. Pre vizualizáciu grafov poslúži gem *Chartkick* (`https://github.com/ankane/chartkick`). Prvý graf (obrázok 5.10) znázorňuje prístup užívateľov počas testu. Pod grafom sa nachádzajú hodnoty minima a maxima počas celého priebehu testu.



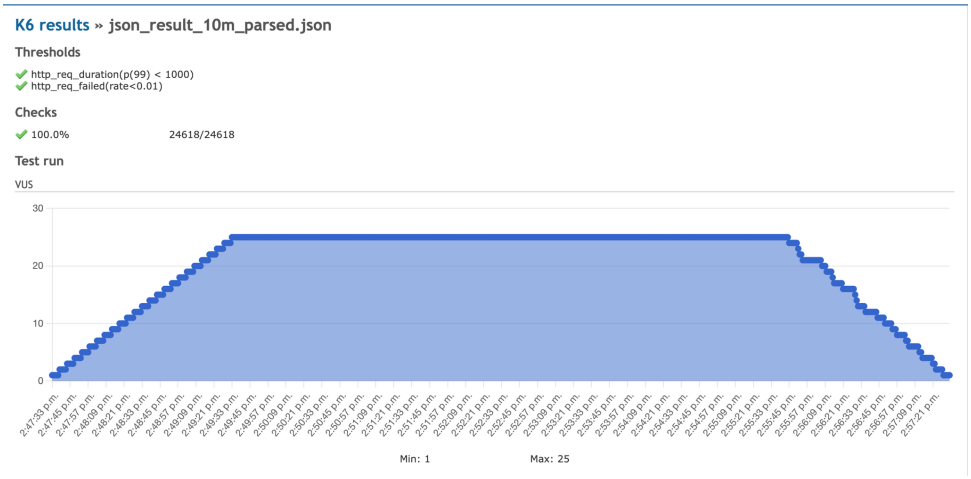
Obrázok 5.9: Result interpreter class diagram

Druhý graf (obrázok 5.11) zobrazuje počet neúspešných requestov v daný časový moment počas priebehu testu. Pod grafom sa nachádza celková miera neúspešnosti requestov.

Všetky nasledujúce grafy popisujú chovanie metriky v priebeh testu na základe zvolených thresholdov. Posledný graf (obrázok 5.11) teda reprezentuje threshold pre dĺžku trvania requestu.

Obsah a typ grafov sa líši na základe typu metriky. Graf pre užívateľov je typu *Gauge*. Graf pre mieru nesplnených requestov je typu *Rate*. Graf pre dĺžku trvania requestu je typu *Trend*.

Poslednú časť zobrazenia tvorí tabuľka (obrázok 5.12) so štatistikou doby trvania pre jednotlivé requesty. Requesty sa delia nielen podľa názvu, prípadne URL, ale aj podľa metódy. Hodnoty v tabuľke sú farebne odlíšené. Z obecného



Obrázok 5.10: Zobrazenie výslekov - Thresholdy, Checky a užívatelia

hľadiska sa vyznačia všetky hodnoty pod 1 sekundu ako zelené, všetky hodnoty medzi 1 a 2 sekundami ako oranžové a všetky hodnoty nad 2 sekundy ako červené.

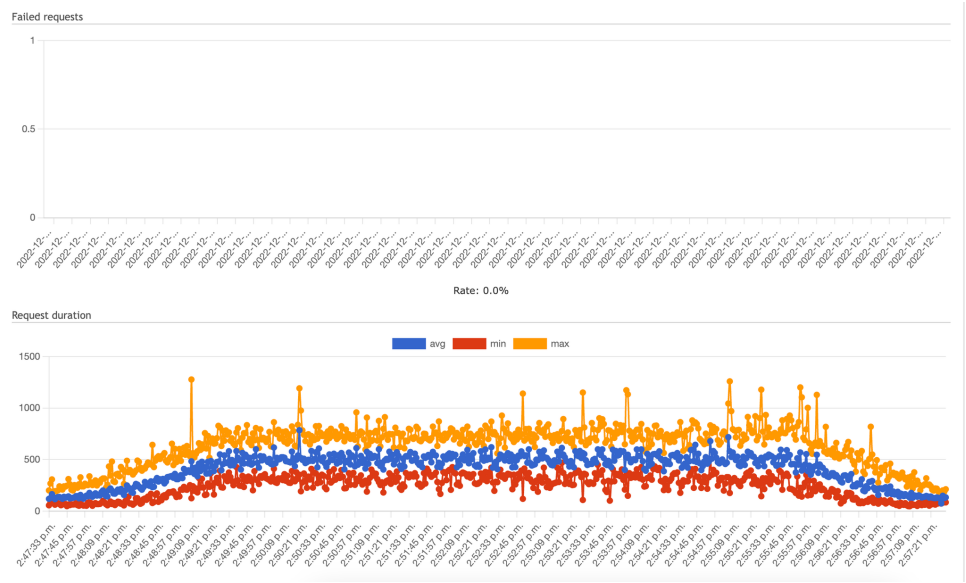
V tabuľke sa môžu objaviť dva requesty s rovnakým menom a rozdielnou metódou. V takom prípade ide o presmerovanie po vykonaní requestu s metódou POST.

Ak sú hodnoty v niektorom stĺpci súčasťou thresholdu, tak sú rozsahy tomu prispôsobené: zelené 0-80% z hodnoty thresholdu, oranžové 80-100% z hodnoty thresholdu a červené 100% a viac z hodnoty thresholdu. Červené requesty nesplnili threshold.

5.6 Prostredie dockeru

Implementácia dockeru sa skladá zo súborov:

- *docker-compose.yml*,
- *Dockerfile*,
- *docker-entrypoint.sh*.



Obrázok 5.11: Zobrazenie výslekov - graf miery neúspešných requestov a graf doby trvania requestov

Request duration statistics

Request name	Method	Count	Avg	Min	Max	p(90)	p(95)	p(99)
login	GET	2238	406.66	47.81	1133.27	603.27	639.78	723.44
login	POST	1119	325.47	54.52	730.15	469.87	497.61	545.22
issues	GET	3357	542.2	146.77	1259.05	724.12	771.19	907.03
new_issue	GET	1119	386.03	83.57	1069.69	531.24	601.82	708.25
create_issue	POST	1119	355.74	77.45	965.37	504.94	536.11	595.83
create_issue	GET	1119	478.5	123.93	1149.12	645.37	685.01	800.55
issue	GET	2238	474.91	117.49	1068.83	643.32	676.08	772.71
edit_issue	GET	1119	417.13	93.68	922.96	598.22	636.17	691.51
update_issue	POST	1119	392.46	73.01	773.92	561.24	593.43	695.56
update_issue	GET	745	472.49	129.19	998.62	642.21	677.35	785.04
projects	GET	1119	366.04	72.65	891.26	512.57	561.17	632.72
project	GET	1119	369.4	77.65	901.37	523.56	551.71	649.45
new_time_entry	GET	1119	362.84	76.38	747.61	508.47	541.85	616.08
create_time_entry	POST	1119	341.81	65.15	914.44	486.77	520.45	585.79
create_time_entry	GET	1119	545.1	158.2	1224.77	720.85	766.94	912.7
time_entries	GET	1119	535.12	149.56	1092.68	716.47	763.88	851.64
edit_time_entry	GET	1119	364.41	82.62	958.81	504.03	535.05	611.47
update_time_entry	POST	1119	337.13	64.35	705.85	485.51	511.64	576.86
update_time_entry	GET	1119	552.72	158.02	1276.92	728.31	760.23	863.86

Obrázok 5.12: Zobrazenie výslekov - tabuľka štatistiky trvania requestov

V rámci súboru *docker-compose.yml* sú definované všetky komponenty, ich image a všetky premenné prostredia, tak ako je to navrhnuté na obrázku 4.3.

Image Redmine aplikácie s integrovaným frameworkom používa na inicializáciu prostredia implementovaný *Dockerfile*. Ten obsahuje image s linuxovým prostredím a nainštalovaným Ruby, z ktorého vychádza. Je to *ruby:3.1.2-slim-bullseye*. Následne inštalujú základné závislosti spolu s MariaDb klientom.

Ďalšie v poradí sú definované všetky potrebné premenné prostredia a *WORKDIR*, do ktorého je nakopírovaný celý obsah zložky s Redmine aplikáciou.

Potom prebieha inštalácia nástroja k6 do linuxového prostredia. Pri inštalácii je potrebné brať do úvahy platformu, na ktorej docker pobeží. Napríklad

pre MacOS s procesorom M1 nieje možné použiť bežný inštalačný príkaz dostupný na adrese `https://k6.io/docs/getting-started/installation/`. Z tohto dôvodu sú binárne súbory nástroja k6 pre verziu 0.41.0 linux-arm64 uložené v zložke */bin*. Tie sú nakopírované do prostredia, namiesto inštalácie.

Po inštalácii nástroja k6 sa vykoná príkaz *bundle install* a vystaví port, na ktorom bude aplikácia dostupná pre vonkajší svet.

Nakoniec sa definuje súbor *docker-entrypoint.sh* ako použitý entrypoint a zavolá funkciu na spustenie, ktorá je v tomto súbore obsiahnutá.

Súbor *docker-entrypoint.sh* obsahuje popis funkcie na spustenie aplikácie a cyklus, ktorý s vykonaním funkcie čaká, až sa inicializuje kontajner a image s priradenou databázou.

Spúšťacia funkcia obsahuje:

- Príkaz na import databázy z dumpu uloženého v projekte.
- Príkaz na migráciu databázových tabuliek, pre prípad, že by dump bol zastaralý.
- Príkaz na migráciu databázových tabuliek v pluginoch, pre prípad, že by dump bol zastaralý.
- Príkaz na spustenie webového servera s aplikáciou.

5.7 Profiling

Pre profiling bolo definované nové prostredie v zdrojovom kóde s názvom *profiling.rb*. Ten sa nachádza v zložke *config/environments/* a vychádzo zo zdrojového súboru pre prostredie development.

Po vytvorení nového prostredia bolo nutné upraviť *Gemfile.rb* nachádzajúci sa v roote pluginu. Tam bolo potrebné pridať gemy pre profiling. Tie sú definované tak, že sa k aplikácii pridajú, len ak aplikácia beží v prostredí s názvom *profiling*. Gem *descriptive_statistics* bol z prostredia odstránený, nakoľko nedokáže fungovať s *RackMiniProfilerom*.

Po úprave suboru *Gemfile.rb* bolo nutné vytvoriť *mini-profiler.rb* v zložke *config/initializers/*. Týmto spôsobom sa nástroj inicializuje v rámci projektu. Táto inicializácia je podmienená, že prebehne výhradne v prostredí s názvom *profiling*.

Gem *RackMiniProfiler* je natívne navrhnutý tak, aby pracoval v development prostredí. Vzhľadom na to, že bolo inicializované nové prostredie, tak sa musí upraviť aj *ApplicationController*, aby s týmto nástrojom v novom prostredí počítal.

V rámci situácie, že framework je implementovaný ako plugin, sa zasahuje do implementácie samotnej aplikácie len minimálne. Takáto úprava bola dosiahnutá pomocou patchu *ApplicationControllerPatch*, ktorý sa nachádza v zložke pluginu */patches/redmine/controllers/*. Rovnako je podmienené, aby sa kód vykonal len v prípade *profiling* prostredia.

Aby mohlo byť prostredie nasadené do docker kontajneru, potrebovalo aplikačnú premennnú *secret_key_base*. Ten sa vygeneroval do súboru *config/initializers/secret_token.rb*, pomocou príkazu *rake generate_secret_token.rb*.

Súbor *config/database.yml*, ktorý obsahuje definované databázové údaje pre každé prostredie, bol rozšírený o nastavenie pre prostredie *profiling*.

5.8 Použitie pluginu v ďalších aplikáciách

Jedna z vlastností *K6Settings* dovoľuje zadať URL, ktorá bude prednostne použitá v testovacom skripte, takže je možné otestovať akúkoľvek aplikáciu.

Napriek tomu, že je framework implementovaný ako Redmine plugin, hlavná funkcionalita frameworku nie je zásadne závislá na implementácii Redmine aplikácie. To otvára možnosť upraviť Redmine plugin na *Rails Engine*, ktorý sa využije podobným spôsobom v Rails aplikácii, alebo na samostatnú aplikáciu, ktorá bude slúžiť na testovanie ostatných aplikácií.

Pokiaľ ide o využitie návrhu v inom programovacom jazyku, tak jediná podmienka je, že daný jazyk musí byť schopný spúšťať príkazy v príkazovom riadku, aby mohol používať nástroj k6. V opačnom prípade musí byť použitý iný nástroj na vykonávanie záťažových testov.

Kapitola 6

Otestovanie Redmine aplikácie v izolovanom prostredí dockeru

Po návrhu frameworku a implementácii pluginu je potrebné otestovať jeho funkcionalitu na Redmine aplikácii. To vyplýva aj zo zadania, ktoré stanovuje, že testovanie by malo prebehnúť v izolovanom prostredí dockeru.

6.1 Testovacie prostredie

Počas implementácie boli vytvorené potrebné súbory pre inicializáciu vyhovujúceho prostredia. Toto prostredie sa teda skladá z viacerých kontajnerov.

Prvým kontajnerom, ktorý je prístupný, je *testing_instance*. Ten reprezentuje inštanciu Redmine aplikácie, ktorej súčasťou je implementovaný plugin, a poslúži v úlohe testovaného prostredia, tiež označovaného ako SUT, aj ako konfigurátor a spúšťač testovacích skriptov. *Testing_instance* má definovaný vlastný databázový kontajner s imagom MariaDB databázy. Aplikácia *testing_instance* je pre verejné prostredie vystavená na porte 3000.

Druhým kontajnerom je *runner*. Rovnako obsahuje inštanciu Redmine aplikácie s integrovaným pluginom. Úlohou tohto kontajnera bude zastupovať prostredie, v ktorom prebehne testovanie a následne spracovanie výsledkov. Výsledky budú po testovaní dostupné aj v aplikácii *testing_instancie* aj v aplikácii *runneru*. *Runner* má vlastný databázový kontajner s imagom

MariaDB databázy. Aplikácia *runner* je pre verejné prostredie vystavená na porte 3001.

Posledným kontajnerom je *profiling*. Ako naznačuje jeho názov, jedná sa o prostredie, v ktorom po skončení testov prebehne profiling. Tiež obsahuje inštanciu Redmine aplikácie s integrovaným pluginom, avšak aplikácia beží vo vlastnom *Rails* prostredí. Kontajner *profiling* nemá vlastný kontajner s databázou, je však napojený na kontajner a databázu určenú pre *testing_instance*. Aplikácia *profiling* je pre verejné prostredie vystavená na porte 3002.

6.2 Priebeh testovania

Pred spustením samotného testovania boli vytvorené potrebné objekty, ktoré definujú testovacie skripty.

Nakonfigurované boli dva typy testov. *Smoke test* (Listing 6.1), ktorý pobeží jednu minútu a obsahuje jedného virtuálneho užívateľa. Druhý typ, *Load test* (Listing 6.2), pobeží 15 minút, je rozdelený na tri etapy (stage) a celkovo dosiahne 15 užívateľov. Obidva typy majú definované rovnaké prahové hodnoty úspešnosti (*Thresholdy*): percentil 99 doby trvania requestov musí byť menší ako 1 sekunda a miera neúspešných requestov musí byť menšia ako 1%.

Listing 6.1: konfigurácia *smoke testu*

```
export const options = {
    vus: 1,
    duration: "1m",
    thresholds: {
        http_req_failed: ["rate < 0.01"],
        http_req_duration: ["p(99) < 1000"]
    }
}
```

Pre testovanie boli využité všetky testovacie scenáre popísané v sekcii 4.2. Ich konfiguráciu nebolo nutné vytvárať, nakoľko majú implementovanú dátovú migráciu, ktorá vytvorí zastupujúce objekty pri prvom spustení aplikácie, ak ich databáza neobsahuje.

Z popísaných konfigurácii boli zostavené testovacie skripty. Celkovo bolo

teda vytvorených 12 testovacích skriptov. Tie boli následne spustené.

Listing 6.2: konfigurácia *load testu*

```
export const options = {
    stages: [
        { duration: "2m", target: 15 },
        { duration: "11m", target: 15 },
        { duration: "2m", target: 0 }
    ],
    thresholds: {
        http_req_failed: ["rate < 0.01"],
        http_req_duration: ["p(99) < 1000"]
    }
}
```

6.2.1 Smoke testy

Ako prvé boli spustené testy typu *smoke test* pre každý scenár. Všetky testy prebehli bez neúspešných requestov a splnili všetky *Checky* a *Thresholdy*. Týmto spôsobom bolo overené, že prostredie zvládne minimálnu záťaž. Priemerné hodnoty z testovania sú zobrazené na obrázku 6.2.

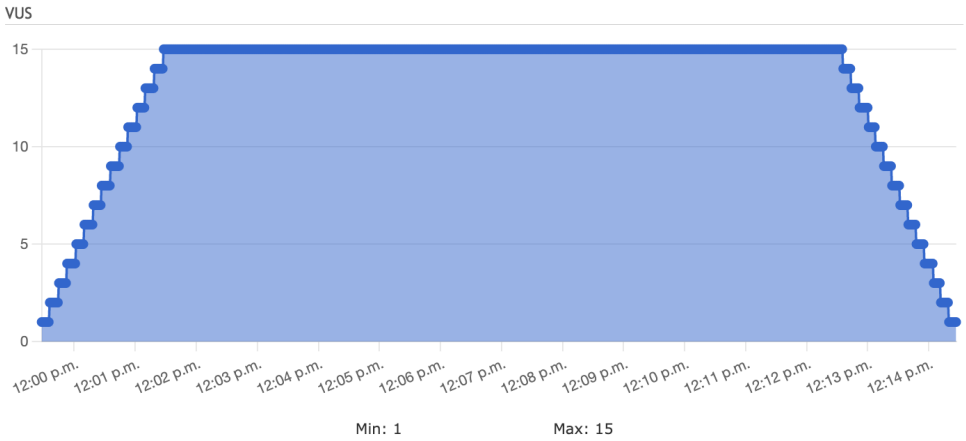
6.2.2 Load testy

Záťažové testovanie poskytlo odlišné výsledky. Došlo ku značnému nárastu trvania requestov. Záťaž počas testovania je zobrazená na obrázku 6.1.

Počas testov došlo k viacerým neúspešným requestom v každom teste. Neúspešné requesty mali však za následok nesplnenie *checkov* a zhoršenie štatistiky trvania requestov.

Napriek tomu bol *Threshold* pre neúspešné requesty splnený. *Threshold* trvania requestov mal úspešnosť 50%.

Zhrnutie výstupov je zobrazené v tabuľkách na obrázkoch 6.2 a 6.3. Z tabuľky 6.2 je možné vidieť, že pri niektorých testoch došlo k extrémnemu nárastu hodnôt. V prípade maxima to bolo spôsobené neúspešnými requestami.



Obrázok 6.1: Simulovaná záťaž počas *load testov*

V prípade nárastu priemernej hodnoty a percentilu trvania requestu došlo k najväčšiemu nárastu pri testovacích scenároch *Vykázanie odpracovaného času*, *Aktualizácia stavu úlohy*, *Priechod širšou funkcionalitou Redminu*. Zároveň sa jedná o všetky testy, ktoré nesplnili kritéria trvania requestov. Pri týchto scenároch bude dobré prejsť štatistiku trvania requestov, aby sa určili miesta na profiling.

Pre tento účel bol vybraný posledný scenár. Z pohľadu stanovených kritérií priniesol najhorší výsledok. Na obrázku 6.4 je vidieť vývoj neúspešných requestov a na obrázku 6.5 sú agregované hodnoty trvania requestov. Jeho štatistika je zobrazená na obrázku 6.6. Podľa nej je možné určiť miesta vhodné na profiling: formulár prihlásenia, úprava úlohy, formulár úpravy vykázaného času, úprava vykázaneho času.

V stĺpeci *Count* sú viditeľné rozdiely. Väčšina sa pohybuje okolo počtu 800. Výrazné rozdiely sú vidieť pri requeste *issues*, *issue* a *update_issue*. V prvých dvoch prípadoch je to z dôvodu, že daný krok je v scenári zahrnutý viac krát. V prípade *update_issue* to bude potrebné preveriť.

6.3 Profiling aplikácie

Pri analýze jednotlivých endpointov boli použité flamegrafy. Pre analýzu boli vybrané endpointy, pri ktorých boli namerané zvýšené hodnoty priemernej doby trvania, alebo precentilu. Všetky endpointy boli analyzované v dvoch situáciách:

Test		agregované hodnoty (v ms)					
nazov	typ	avg	min	max	p(90)	p(95)	p(99)
Prihlásenie do aplikácie	smoke	24.1	5.91	124.98	49.5	52.25	61.41
	load	76.53	0	2860	191.89	254.37	357.83
	nárast	318%	0%	2288%	388%	487%	583%
Zobrazenie projektu	smoke	52.03	22.39	164.57	79.12	81.23	122.25
	load	88.66	0	960.32	148.99	176.2	248.38
	nárast	170%	0%	584%	188%	217%	203%
Zobrazenie úlohy	smoke	33.56	6.24	198.27	69.76	72.29	109.46
	load	277.59	0	60000	418.28	504.45	839.23
	nárast	827%	0%	30262%	600%	698%	767%
Vykázanie odpracovaného času na úlohe	smoke	34.39	6.17	121.49	69.95	74.2	109.64
	load	358.36	0	14520	806.05	2010	3027.38
	nárast	1042%	0%	11952%	1152%	2709%	2761%
Aktualizácia stavu úlohy	smoke	40.65	6.33	281.32	73.15	79.3	136.79
	load	392.96	0	4830	1230	2050	2929
	nárast	967%	0%	1717%	1681%	2585%	2141%
Priechod širšou funkcionalitou Redminu	smoke	56.47	7.42	417.76	105.96	122.34	167.51
	load	508.19	0	6930	492.07	3680	5743.12
	nárast	900%	0%	1659%	464%	3008%	3429%

Obrázok 6.2: Porovnanie agregovaných hodnôt medzi typmi testov pre každý scenár

- počas minimálnej záťaže,
- počas navýšenej záťaže.

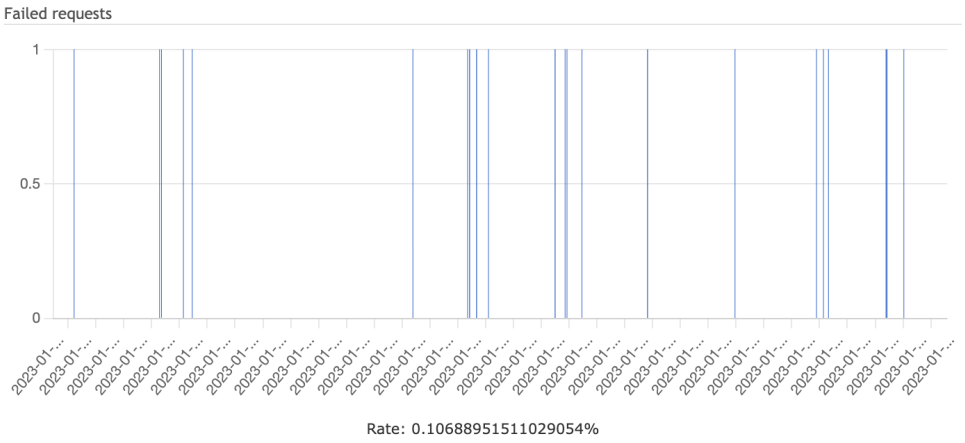
Pri vizualizácii času vykonania requestu pomocou flamegrafu, tvoria základnú vrstvu dva bloky. Jedným je vykonanie akcie. Druhým je *garbage collection*. *Garbage collection* je funkcia na vyčistenie nepoužívanej pamäte, ktorú vykonáva *Garbage collector*.

Práca bloku *garbage collection* pri minimálnej záťaži predstavovala priemerne 2-5 % z celkového času vykonania requestu. Pri zvýšenej záťaži dochádzalo k nárastu práce bloku *garbage collection* priemerne na 40%. V niektorých prípadoch sa stalo, že *garbage collection* presiahol hranicu 50% a pracoval viac ako samotné vykonanie akcie.

Blok pre vykonanie akcie začína inicializáciou vlákna webovým serverom *Puma*. To následne volá potrebné moduly frameworku *Ruby on Rails*, ako napríklad *ActiveSupport* a ich metódy. Pred každou akciou dochádza k overeniu sedenia (session), čo však ani pri zvýšenej záťaži nejavilo známky, že by sa jednalo o bottleneck. Následne je volaný blok kontrolerovej akcie.

Test	neúspešné requesty	neúspešné *Checky*	nesplnené *Thresholdy*	splnené *Thresholdy*	počet vykonaných requestov
Prihlásenie do aplikácie	2	2	-	2	48092
Zobrazenie projektu	2	4	-	2	19862
Zobrazenie úlohy	51	54	-	2	29085
Vykázanie odpracovaného času na úlohe	8	8	1	1	26177
Aktualizácia stavu úlohy	15	15	1	1	23517
Priechod širšou funkcionalitou Redminu	22	22	1	1	20582

Obrázok 6.3: Zhrnutie úspešnosti jednotlivých scenárov počas load testov

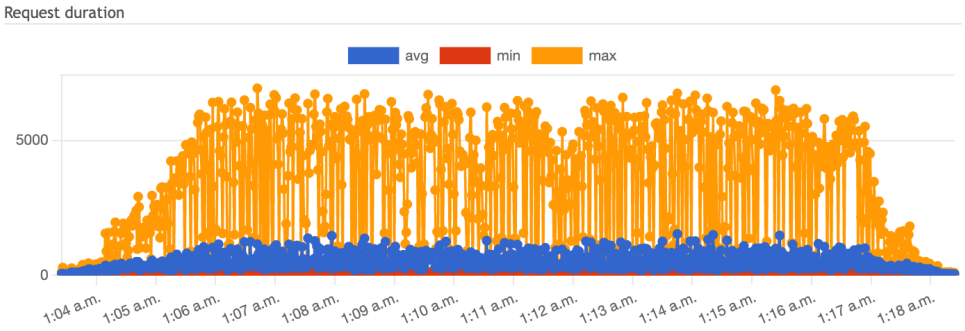


Obrázok 6.4: Vývoj neúspešných requestov počas záťažového testu scenára *Priechod širšou funkcionalitou Redminu*

Analýza flamegrafu akcie úpravy úlohy

Endpoint je zobrazený v tabuľke na obrázku 6.6 ako *update_issue* s metódou *POST*. Pred vykonaním blok kontrolerovej akcie došlo k načítaniu objektu. Blok kontrolerovej akcie je v tomto prípade *IssuesController#update*.

Pod minimálnou záťažou strávil request pri vykonaní v kóde 53ms. V bloku *IssuesController#update* strávil 75% celkového času. Tento blok sa následne delí na 2 menšie bloky:



Obrázok 6.5: Agregované hodnoty trvania requestov počas záťažového testu scenára *Priechod širšou funkcionalitou Redminu*

Request duration statistics

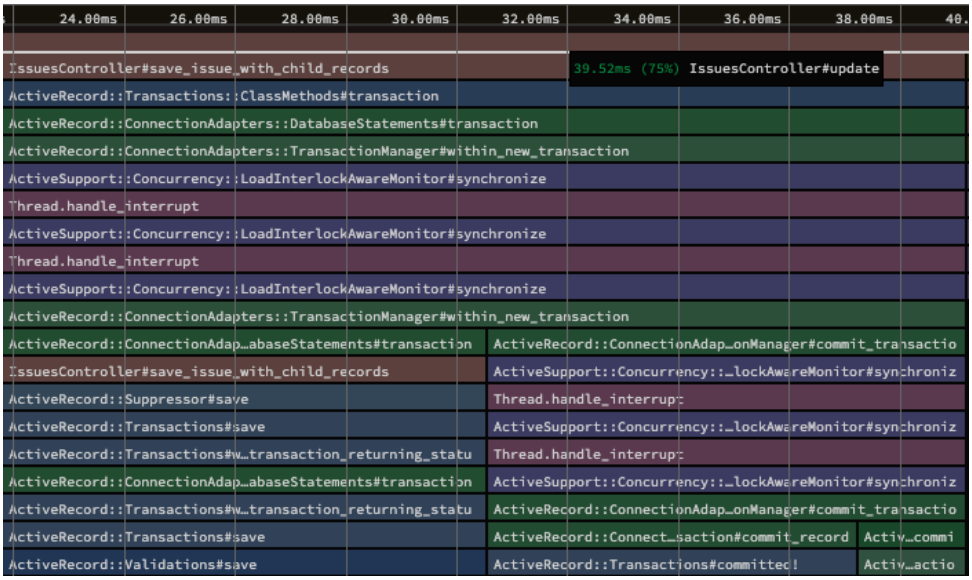
Request name	Method	Count	Avg	Min	Max	p(90)	p(95)	p(99)
login form	GET	804	628.54	0	4164.44	1532.9	1765.97	3300.29
login	POST	803	71.39	9.42	221.41	100.9	113.72	153.48
login	GET	803	181.57	41.6	335.86	226.15	240.41	275.42
issues	GET	2409	207.86	44.53	1089.56	256.73	278.56	309.69
new_issue	GET	803	227.26	39.78	401.37	291.89	314.17	346.66
create_issue	POST	803	173.23	26.1	346.9	221.31	241.39	282.1
create_issue	GET	803	407.66	80.02	642.29	495.48	523.87	568.76
issue	GET	1606	389.12	61.67	904.98	476.49	497.81	534.89
edit_issue	GET	803	268.51	44.35	433.04	330.29	354.62	385.88
update_issue	POST	803	3860.71	0	6930.54	5917.22	6200.43	6668.76
update_issue	GET	524	453.61	80.61	6096.2	510.27	537.78	629.65
projects	GET	803	187.45	19.75	5776.98	180.51	213.89	2911.88
project	GET	803	190.44	27.17	1598.54	236.88	248.31	286.99
new_time_entry	GET	803	177.65	24.68	325.33	224.61	237.86	288.42
create_time_entry	POST	803	111.82	14.08	234.8	151.63	166.89	202.69
create_time_entry	GET	803	257.24	49.39	551.45	320.19	352.15	391.14
time_entries	GET	803	239.53	46.49	419.12	304.35	326.81	363.29
edit_time_entry	GET	803	2435.32	0	6575.2	5422.84	5854.35	6343.65
update_time_entry	POST	800	1372.03	0	6587.52	5148.97	5597.67	6252.63
update_time_entry	GET	797	354.92	51.59	6451.53	316.32	355.35	5054.59
logout form	GET	800	71.72	9.22	3787.96	89.23	102.56	141.72
logout	POST	800	44.97	6.17	1217.73	64.7	79.5	115.72
logout	GET	800	67.7	9.71	426.29	98.5	118.27	151.21

Obrázok 6.6: Štatistika trvania requestov počas záťažového testu scenára *Priechod širšou funkcionalitou Redminu*

- úprava načítaného objektu,
- uloženie upraveného objektu.

Blok úprava načítaného objektu (obrázok 6.8) ďalej pracuje na úrovni kódu s načítanou inštanciou a všetkými asociovanými objektami, pričom dôjde k úprave vlastnosti podľa dát z odoslaného formulára. Blok je následne delený na početné malé bloky. Táto časť nevykazuje známky, že by sa jednalo o bottleneck.

Blok uloženie upraveného objektu (obrázok 6.7) drží otvorenú transakciu do databázy počas celého trvania zapísania objektu. Neukladá sa len samotný



Obrázok 6.7: Blok uloženie upraveného objektu úlohy

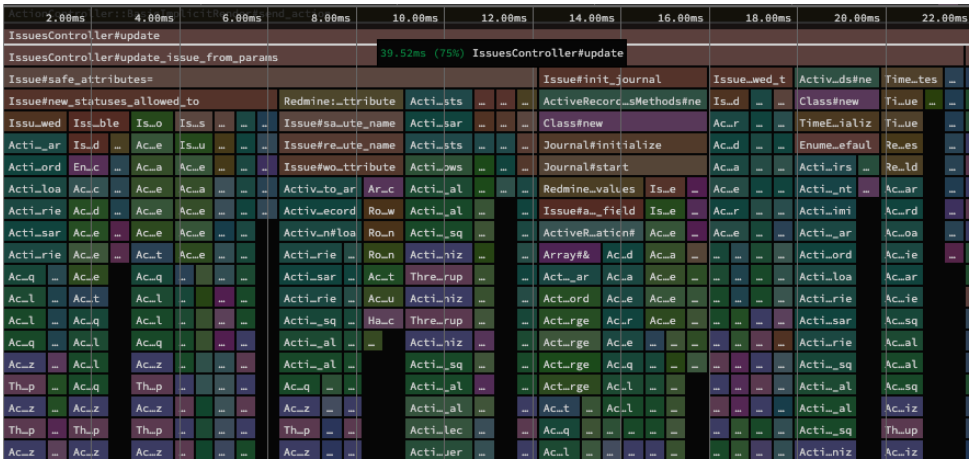
objekt, ale aj asociované objekty. Celkovo tento blok vo flamegrafe zaberá značné miesto, pričom dochádza počas neho k pripojeniu do databázy.

Skutočnosť, že blok drží otvorenú databázovú transakciu je dôvod, prečo je rozdielny počet requestov pri *update_issue* s metódou *POST* a s metódou *GET*. Potvrdilo sa to pri analýze počas záťaže.

Počas navýšenej záťaže nastali dve situácie. Pri jednej došlo k úprave objektu a tento request strávil v kóde 112,25ms. V bloku *IssuesController#update* strávil 46% celkového času. Došlo teda k dlhšiemu vykonaniu akcie, ale nejde o značný rozdiel.

Pri druhej situácii nedošlo k úprave objektu. Dôvodom je, že daná úloha bola v tom momente upravovaná iným vláknom a došlo k návratu na editačný formulár. Request strávil v kóde 333ms. Značná časť z tohto času je pripísaná práci bloku *garbage collection*, čo tvorí 58% z celkového času. Blok *IssuesController#update* sa však delí na 3 bloky, pričom pribudol blok *render*. Tento blok zaberá väčšinu práce bloku *IssuesController#update* a dochádza tu k renderovaniu formulára na *backende*.

Celkovo najdlhšie strávený čas pri záťaži predstavuje *garbage collection* a renderovanie formulára pri neúspešnom uložení.



Obrázok 6.8: Blok úprava načítaného objektu úlohy

Analýza flamegrafu akcie zobrazenia formulára pre úpravu vykázaného času

Endpoint je zobrazený v tabuľke na obrázku 6.6 ako *edit_time_entry*. Pred vykonaním blok kontrolerovej akcie došlo k načítaniu objektu, ktorý bol použitý na vyplnenie vstupných polí formulára aktuálnymi informáciami. Blok kontrolerovej akcie je v tomto prípade priamo *ActionController::Instrumentation#render*.

Pri minimalnej záťaži strávil request 62,16ms vykonávaním kódu. V bloku *render* strávil 62% z celkového času, čo je 38.25ms.

Počas navýšenej záťaže request vykonával kód 110ms. V bloku *render* strávil 47% z celkového času, čo je 51.91ms.

Navýšenie záťaže nepreukázalo veľký časový rozdiel pri načítaní objektu. Došlo však k nárastu času, ktorý request strávil renderovaním formulára pre úpravu. To môže predstavovať problém, ktorý zapríčinil navýšené hodnoty v tabuľke. Okrem toho sa zvýšil aj čas, počas ktorého pracoval *garbage collection* podobne ako pri úprave úlohy, nie však natoľko, aby pracoval dlhšie než vykonanie akcie.

Obrázok 6.9: Blok uloženie upraveného objektu vykázaného času

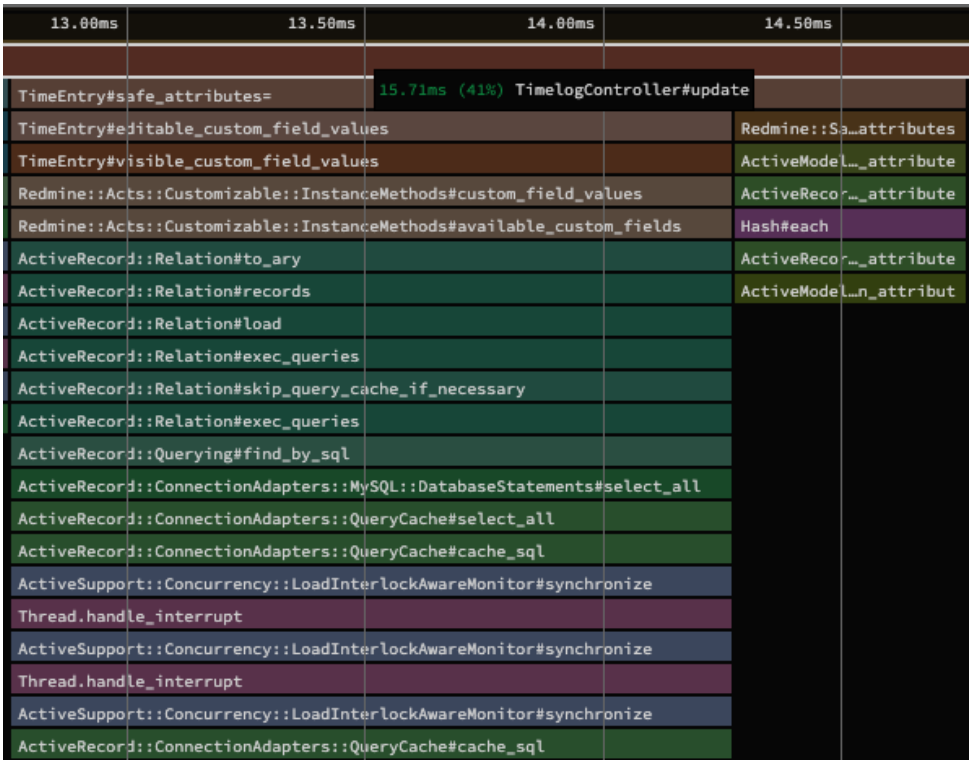
Analýza flamegrafu akcie úpravy vykázaného času

V tabuľke na obrázku 6.6 je táto akcia pomenovaná *update_time_entry* s metódou *POST*. Blok kontrolerovej akcie je *TimeLogController#update*. Pred blokom akcie je vykonaný blok pre načítanie objektu.

Pri minimálnej záťaži strávil request vykonaním kódu 38.75ms. Blok *TimeLogController#update* predstavoval 41% celkového času. Objekt výkazu času je pomerne malý, v porovnaní s objektom úlohy. Pri jeho úprave nedochádza k úprave asociovaných objektov. Blok sa rovnako delí na dva menšie bloky:

- úprava načítaného objektu,
- uloženie upraveného objektu.

Úprava načítaného objektu (obrázok 6.10) pracuje na úrovni kódu len s načítanou inštanciou vykázaného času, preto nie je časovo náročná.



Obrázok 6.10: Blok úprava načítaného objektu vykázaného času

Uloženie upraveného objektu (obrázok 6.9) pracuje v rámci transakcie pre zápis, ale zapisuje len inštanciu bez ďalších asociovaných objektov. Toto síce poskytuje priestor pre neuloženie objektu, ale v značne menšej miere ako uloženie objektu úlohy.

Pri zvýšenej záťaži request vykonal kód za 94,71ms. V bloku *TimeLogController#update* strávil 18% celkového času. V porovnaní s minimálnou záťažou nedošlo takmer k žiadnej zmene. Veľká zmena znova nastala z pohľadu bloku *garbage collection*. Tu došlo k navýšeniu na 49% celkového času vykonania kódu.

Analýza flamegrafu akcie zobrazenia prihlasovacieho formulára

Endpoint je označený v tabuľke na obrázku 6.6 ako *login form*. Blok kontrolerovej akcie je v tomto prípade priamo *ActionController::Instrumentation#render*.

Pri minimalnej záťaži strávil request 35ms vykonávaním kódu. V bloku *render* strávil 44% z celkového času, čo je 15.51ms.

Počas zvýšenej záťaže request vykonával kód 42,07ms. V bloku *render* strávil 33% z celkového času, čo je 13,95ms.

Zvýšenie záťaže znamenalo malý časový rozdiel strávený pri renderovaní prihlasovacieho formulára. Okrem toho sa zvýšil aj čas, počas ktorého pracoval blok *garbage collection*.

Garbage collection

Počas analýzy flamegrafov dochádzalo pri zvýšenej záťaži k nárastu práce bloku nazvaného *garbage collection*.

V prípade jazyka Ruby sa *garbage collector* nestará len o odstránenie alokovaných nepoužívaných objektov, ale aj o alokáciu pamäte pre nové objekty. Drží si všetky potrebné informácie pre správu pamäte.

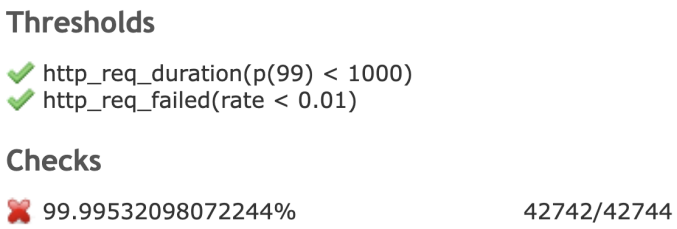
Garbage collector pri svojej práci používa mechanizmus *Mark and Sweep*. V prvej fáze tohto mechanizmu dôjde k označeniu objektov pomocou troch farebných značiek. V druhej fáze sa vykoná funkcia *sweep* a vymaže všetky objekty, ktoré sú označené na vymazanie. Počas práce *garbage collectoru* je program pozastavený a po dokončení je opätovne spustený.

Pre analýzu nárastu vyťaženia pamäte boli použité funkcie *profile-memory* a *profile-gc*. Analýza prebehla pre minimálnu, aj pre zvýšenú záťaž. Pri zvýšenej zaťaží došlo v niektorých prípadoch ku niekoľko násobnému navýšeniu počtu objektov a alokovanej pamäte. Tomu zodpovedá aj zvýšená aktivita *garbage collectoru*.

6.4 Vyhodnotenie profilingu

Analýza ukázala, že profilované miesta v aplikácii sú relatívne optimalizované.

Analýza pomocou profilingu neodhalila žiadne zásadné úzke miesta v aplikácii. Pozornosť vzbudila iba zvýšená aktivita procesu *garbage collection*. To je pochopiteľné vzhľadom na situáciu, kedy pri simulovaní záťaže dochádza k nadmernej alokácii a uvoľňovaniu miesta v pamäti. Okrem práce *garbage*



Obrázok 6.11: Kritéria úspešnosti záťažového testu scenára *Priechod širšou funkcionalitou Redminu* po úprave nastavení web serveru.

collectoru bolo zaznamenané, že renderovanie stránky prebieha na *backende*, čo môže ovplyvňovať výkon spracovania requestu.

Renderovanie stránky na *backende* môže byť veľmi náročné, čím je stránka komplikovanejšia. Pri veľkej návštevnosti to môže mať značný vplyv na trvanie requestu. Trochu nápomocný je v tomto ohľade *Rails cache* mechanizmus, ale ani ten nemusí stačiť.

Na spracovanie requestov môže mať vplyv aj nastavenie webového servera *Puma*. Pokiaľ toto nastavenie nie je prispôsobené pre daný stroj, tak môže práve toto nastavenie brániť využitiu dostatočného množstva zdrojov. V takom prípade sa z toho stáva úzke miesto.

Pre uľahčenie práce *garbage collectoru* a správu pamäte je vhodné vyhnúť sa používaniu globálnych objektov a referovaniu dvoch objektov medzi sebou.

Aplikácia Redmine má pomerne jednoduché stránky, preto renderovanie na *backende* nespôsobuje výrazné problémy. Ak by však obsahoval rozšírenia, ktoré zvyšujú komplexnosť stránok a *View*, odporúča sa využiť modernejšie *frontendové* technológie. Napríklad *React*, alebo *Vue.js*.

Čo sa týka webového serveru *Puma*, jeho konfigurácia dovoľuje spustiť aplikáciu v *clustrovom* móde. Počet procesov, na ktorých pobežia inštancie aplikácie, sa definuje pomocou premennej prostredia *WEB_CONCURRENCY*. Každý z týchto procesov má vlastnú sadu vlákien. To sa definuje pomocou premennej prostredia *RAILS_MAX_THREADS*. Manipuláciu s menovanými nastaveniami treba prispôsobiť hardwaru, na ktorom aplikácia pobeží.

Po zmene nastavení webového serveru *Puma* na využitie troch procesov v *clustrovom* móde, vykazuje aplikácia známky zvýšenia výkonnosti. Došlo k zlepšeniu priemernej doby trvania requestu a bol znížený počet neúspešných requestov. Zároveň boli splnené kritéria úspešnosti testu. Počet iterácií prie-

Request duration statistics

Request name	Method	Count	Avg	Min	Max	p(90)	p(95)	p(99)
login form	GET	1781	74.56	11.89	1879.04	89.58	178.94	1051.81
login	POST	1781	55.14	10.88	194.96	89.99	105.5	146.65
login	GET	1781	186.62	44.85	475.92	270.47	298.47	357.66
issues	GET	5343	225.94	46.92	1628.61	317.29	348.53	408.86
new_issue	GET	1781	241.41	41.15	517.1	352.51	382.53	444.01
create_issue	POST	1781	166.36	27.03	380.6	249.01	275.69	320.42
create_issue	GET	1781	446.59	81.37	933.37	618.13	660.61	738.89
issue	GET	3562	428.02	65.35	1333.88	594.75	629.82	702.49
edit_issue	GET	1781	311.85	46.43	705.18	433.66	467.59	540.69
update_issue	POST	1781	420.09	28.37	2963.44	1024.6	1378.71	2033.01
update_issue	GET	1168	458.38	86.4	2378.99	643.78	691.55	811.27
projects	GET	1781	141.74	20.67	2292.06	204.72	235.71	315.27
project	GET	1781	184.33	28.4	1267.57	266.25	299.78	358.99
new_time_entry	GET	1781	175.62	25.8	1931.86	250.55	282.85	347.2
create_time_entry	POST	1781	102.13	14.26	314.19	157.66	175.87	232.92
create_time_entry	GET	1781	282.87	51.65	599.89	400.6	430.23	472.63
time_entries	GET	1781	259.93	50.12	672.03	370.01	404.29	458.85
edit_time_entry	GET	1781	204.68	27.63	1523.04	308.12	405.07	908.64
update_time_entry	POST	1781	111.47	12.0	1479.27	157.53	324.74	828.05
update_time_entry	GET	1781	267.84	51.83	1499.25	387.25	416.9	546.6
logout form	GET	1781	54.03	9.24	1613.73	84.68	101.57	181.8
logout	POST	1781	38.11	6.52	1500.2	62.23	74.19	134.61
logout	GET	1781	50.01	10.02	1432.72	79.56	95.92	153.07

Obrázok 6.12: Štatistika trvania requestov počas záťažového testu scenára *Priechod širšou funkcionalitou Redminu* po úprave nastavení web serveru.

chodu testovacím scenárom sa zdvojnásobil. Výstup testovania je zobrazený na obrázkoch 6.11 a 6.12.

Detailnejšie obrázky flamegrafov, výstupov profilingu pamäte a *garbage collectoru* sú priložené k technickej časti práce. Súčasťou prílohy sú aj JSON súbory všetkých použitých flamegrafov. Tieto JSON súbory je možné otvoriť ako flamegraf na adrese `https://www.speedscope.app`.

Kapitola 7

Záver

Cieľom diplomovej práce bolo navrhnúť a naimplementovať framework pre testovanie výkonnosti Redmine aplikácie. Implementovaný framework následne použiť na otestovanie inštancie Redmine aplikácie a vyhodnotiť výsledky testovania.

V rámci diplomovej práce sa podarilo navrhnúť a naimplementovať funkčný nástroj na výkonnostné testovanie, ktorý bol implementovaný ako rozšírenie Redmine aplikácie. Nástroj spĺňa stanovené kritériá a funkcionálne požiadavky. Podarilo sa vytvoriť funkčný mechanizmus výkonnostného testovania, ktorý využíva voľne dostupný nástroj k6 a webové rozhranie, ktoré z nakonfigurovaných parametrov vygeneruje testovací skript.

Naimplementovaný nástroj sa podarilo nasadiť do navrhnutého prostredia dockeru, kde úspešne otestoval výkon základnej verzie Redminu. Výsledky výkonnostného testovania boli nástrojom v dostatočnej miere interpretované. Podľa interpretovaných výsledkov sa podarilo identifikovať pomalé časti aplikácie.

Na pomalých častiach aplikácie bol vykonaný profiling. Nepodarilo sa preukázať, že by spomalenie v identifikovaných miestach bolo spôsobené nedostatočnou optimalizáciou kódu Redmine aplikácie. Podarilo sa však odhaliť, že prostredie, v ktorom bola nasadená testovaná aplikácia, nebolo správne nastavené, čo bolo dôvodom, že boli využité len obmedzené zdroje. V dôsledku toho nebola dostatočne zvládnutá záťaž aplikáciou, čo preukázali výstupy interpretované implementovaným nástrojom. Nástroj je teda možné použiť aj v prípade, keď chceme overiť, že stroj, na ktorý bude aplikácia nasadená, dokáže

udržať predpokladanú záťaž, ktorá nastane behom prevádzky v produkčnom prostredí.

Prínosom práce je vytvorenie nástroja, ktorý môže v budúcnosti poslúžiť pri vývoji Redminu a pri vývoji vlastných rozšírení. Nástroj tiež môže pomôcť pri zavádzaní aplikácie Redmine do prevádzky tak, že overí možnosti technického vybavenia, na ktorom aplikácia pobeží.

Dodatok A

Literatúra

[1] *Apache JMeter™*, Documentation, Apache Software Foundation, `https://jmeter.apache.org/index.html`, accessed 2022-06-01.

[2] *Cloud*, Documentation, Grafana Labs, `https://k6.io/docs/results-output/real-time/cloud/`, accessed 2022-12-21.

[3] *FURPS*, `https://cs.wikipedia.org/wiki/FURPS`, accessed 2023-01-02.

[4] *Gatling*, Tech. report, Gatling Corp, `https://gatling.io/open-source/`, accessed 2022-06-01.

[5] *Grafana Cloud*, Documentation, Grafana Labs, `https://grafana.com/products/cloud/`, accessed 2022-12-21.

[6] *Grafana Cloud*, Documentation, Grafana Labs, `https://k6.io/docs/results-output/real-time/grafana-cloud/`, accessed 2022-12-21.

[7] *How does software testing work?*, `https://www.ibm.com/topics/software-testing`, accessed 2023-01-02.

[8] *JSON*, Documentation, Grafana Labs, `https://k6.io/docs/results-output/real-time/json/`, accessed 2022-12-21.

[9] *K6*, Documentation, Grafana Labs, `https://k6.io/docs/`, accessed 2022-06-01.

[10] *k6 Cloud documentation*, Documentation, Grafana Labs, `https://k6.io/docs/cloud/`, accessed 2022-12-21.

[11] *Load testing*, Documentation, Grafana Labs, `https://k6.io/docs/test-types/load-testing/`, accessed 2022-06-01.

[12] *Metrics*, Documentation, Grafana Labs, https://k6.io/docs/using-k6/metrics/#metric-types, accessed 2022-12-27.

[13] *Profiling tools*, Documentation, IBM, https://www.ibm.com/docs/en/aix/7.1?topic=reference-profiling-tools, accessed 2023-01-04.

[14] *Redmine*, Documentation, https://www.redmine.org/projects/redmine/wiki, accessed 2022-06-01.

[15] *Redmine API*, Documentation, https://www.redmine.org/projects/redmine/wiki/rest_api, accessed 2022-06-01.

[16] *Results output*, Documentation, Grafana Labs, https://k6.io/docs/get-started/results-output/, accessed 2022-12-21.

[17] *Smoke testing*, Documentation, Grafana Labs, https://k6.io/docs/test-types/smoke-testing/, accessed 2022-06-01.

[18] *Soak testing*, Documentation, Grafana Labs, https://k6.io/docs/test-types/soak-testing/, accessed 2022-06-01.

[19] *Stress testing*, Documentation, Grafana Labs, https://k6.io/docs/test-types/Stress-testing/, accessed 2022-06-01.

[20] *Web Performance Testing: What It Is and Why You Need It*, https://performancelabus.com/importance-of-web-application-performance-testing/, accessed 2022-6-1.

[21] *Why Artillery?*, Documentation, Artillery Software Inc, https://www.artillery.io/docs/guides/overview/why-artillery, accessed 2022-06-01.

[22] Len Bass, Paul Clements, and Rick Kazman, *Software architecture in practice, third edition*, Addison-Wesley Professional, September 2012.

[23] Nate Berkopec, *rack-mini-profiler - the Secret Weapon of Ruby and Rails Speed*, (2015), https://www.speedshop.co/2015/08/05/rack-mini-profiler-the-secret-weapon.html, accessed 2022-12-22.

[24] Eric Goebelbecker, *Rack Mini Profiler: A Complete Guide on Rails Performance*, (2019), https://stackify.com/rack-mini-profiler-a-complete-guide-on-rails-performance, accessed 2022-12-22.

[25] Thomas Hamilton, *9 BEST Performance Testing Tools (Load Testing Tool) in 2022*, (2022), https://www.guru99.com/performance-testing-tools.html, accessed 2023-01-01.

[26] Burak Kantarcı, *Why Should You Run All Your Tests in Docker?*, https://www.runforesight.com/blog/why-should-you-run-all-your-tests-in-docker, accessed 2023-01-04.

[27] Nishi Srivastava, Ujjwal Kumar, and Pawan Singh, *Software and performance testing tools*, Journal of Informatics Electrical and Electronics Engineering (JIEEE) **2** (2021), no. 1, 1–12.

[28] Martin Williams, *Performance Testing - Tools, Steps, and Best Practices*, (2019), `https://www.keycdn.com/blog/performance-testing`, accessed 2022-06-01.

Dodatok B

Zoznam skratiek

CLI Command Line Interface. 13

HTTP Hypertext Transfer Protocol. 6

HTTPS Hypertext Transfer Protocol Secure. 44

SaaS Software as a Service. 17

STI Single-Table inheritance. 40

SUT System under test. 57

URL Uniform Resource Locator. 56

Dodatok C

Zoznam príloh

- Príloha č. 1: loadtester.zip - zdrojové kódy, ktoré obsahujú implementáciu frameworku.
- Príloha č. 2: loadtester_redmine.zip - zdrojové kódy aplikácie redmine, do ktorej je možné zapojiť implementovaný framework.
- Príloha č. 3: profiling.zip - zložka obsahujúca súbory použité pri profilingu.
- Príloha č. 4: bdip_tex.zip - zložka, ktorá obsahuje zdrojové textové súbory.