ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název: Nástroj pro optimalizaci výkonu Rails aplikací
Student: Ondřej Ezr
Vedoucí: Ing. Tomáš Bartoň
Studijní program: Informatika
Studijní obor: Softwarové inženýrství
Katedra: Katedra softwarového inženýrství
Platnost zadání: do konce letního semestru 2015/16

Pokyny pro vypracování

Nastudujte a popište možnosti měření výkonu aplikací psaných ve frameworku Ruby on Rails. Analyzujte existující dostupné nástroje a postupy, které je možné použít pro měření výkonu a detekci nejčastějších problémů při optimalizaci výkonu webových aplikací. Na základě analýzy navhrněte a implementujte vlastní nástroj ve formě webové aplikace, která bude v reálném čase monitorovat dotazy běžící Rails aplikace a reportovat případné problémy. Demonstrujte využití nástroje na několika příkladech.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof.Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 17. prosince 2014

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Bakalářská práce

Nástroj pro optimalizaci výkonu Rails aplikací

Ondřej Ezr

Vedoucí práce: Ing. Tomáš Bartoň

11. května 2015

Poděkování

Děkuji zejména svému vedoucímu, Ing. Tomáši Bartoňovi, za nezměrnou trpělivost a cenné rady, kterými pomohl vzniku samotného nápadu na tuto bakalářskou práci a později směroval její vývoj. Rodičům děkuji za úžasnou podporu psychickou i materiální jak při psaní této práce, tak v průběhu celého studia. Marku Plaštiakovi jsem vděčný za pomoc s udržením zdravého rozumu, když se věci všedního života snažily dostat v prioritním žebříčku před psaní této práce. Ještě jednou Markovi za jeho trpělivost s korekturou této práce. Dominiku Dragounovi bych rád poděkoval za diskuze při pauzách, které mi pomohly udržet směr mých myšlenek.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 11. května 2015 .....

Odkaz na tuto práci

Ezr, Ondřej. *Nástroj pro optimalizaci výkonu Rails aplikací.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Hlavním cílem této práce je implementovat nástroj pro analýzu a monitorování výkonu webových aplikací napsaných ve frameworku Ruby on Rails. Práce nejprve analyzuje základní výkonnostní problémy webových aplikací. Součástí analýzy je rovněž seznam dostupných nástrojů, které se snaží tyto problémy v Ruby on Rails řešit. V souladu s touto analýzou je posléze realizován návrh a implementace nástroje. Ta kombinuje výhody dostupných nástrojů za účelem poskytnutí komplexního spektra funkcionalit.

Klíčová slova Ruby on Rails, výkon, monitoring

Abstract

Main goal of this thesis is implementation of tool for monitoring and analysing performance of web applications written on top of framework Ruby on Rails. To the design of this tool predeces an analysis of the most common performance problems in web applications. Part of analysis is a list of available tools attempting to provide solutions for these problems. Design and implementation part describes creation of tool based on previous analysis. It combines advantages of existing tools in order to provide complex functionality.

Keywords Ruby on Rails, performance, monitoring

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Web je dnes masivně se rozvíjející médium, přenášející stále více informací a poskytující stále více služeb. Poptávka po těchto službách rychle roste a webové aplikace se rozšiřují do nejrůznějších odvětví, ve kterých jsme si využití webu neuměli před dvaceti lety ani představit. Webové aplikace začínají převládat nad aplikacemi lokálními. Všechny aplikace se přesouvají na internet zejména kvůli snadné přenositelnosti, sdílení dat, propojení dat uživatelů vzájemně, jednoduchým aktualizacím a dalším nesporným výhodám internetu. Web je dnes zkátka opravdu všude kolem nás.

Vzniká tedy velké množství webových aplikací. A jelikož není dostatek profesionálních vývojářů, velké množství z nich pochází z rukou vývojářů neprofesionálních.

Tito vývojáři mají často pocit, že díky nárůstu výpočetního výkonu není třeba příliš se ohlížet na výkon aplikace. Proto větší a rozšířenější aplikace, které na tento aspekt nemyslí od začátku, mají velké problémy. Objeví se větší množství uživatelů, začne narůstat objem dat, či přibývat mnoho vzájemně se ovlivňujících funkcí. Nemusejí zvládnout svůj vstup na trh a my v těchto aplikacích můžeme i přijít o excelentní nápad, či dobrého pomocníka v každodenním životě. I u aplikací, které toto ustojí, je tato fáze velmi obtížná a stojí mnoho zdrojů.

Je velmi důležité, aby vývojáři měli k dispozici nástroje, které je na případné problémy upozorní dříve, než se do této fáze dostanou, případně jim pomohou tuto fázi co nejrychleji překlenout. Mělo by se jednat o nástroje, které jsou jednoduše použitelné a snadno implementovatelné. Jelikož vývojáři jsou často velmi sebevědomí a pokud by měli vynaložit větší úsilí, většinou takový nástroj použijí, až když je pozdě.

Tento problém se pro framework Ruby on Rails snaží částečně řešit tato bakalářská práce.

S výkonnostními problémy bojují samozřejmě i zkušení vývojáři, kteří takový nástroj jistě také ocení. Mou cílovou skupinou jsou však vývojáři méně zkušení. Pokud se podaří nástroj vytvořit tak, aby byl užitečný méně zkušeným vývojářům, bude užitečný všem.

Motivace

Již čtvrtým rokem jsem vývojářem webových aplikací a framework Ruby on Rails jsem si velmi oblíbil. Nabízí to, co od frameworku očekávám: jednoduchost, elegantní a přehledný kód a minimální konfiguraci. Se vzrůstající zkušeností si však začínám všímat méně zkušených vývojářů a vidím na vlastní oči, jaké konstrukce jsou schopní vytvořit.

Jelikož pracuji na aplikaci, která se vyvíjí již sedmým rokem, vidím zároveň, kam tyto konstrukce aplikaci vedou. K práci mě přivedla aplikace EasyRedmine, kterou v součastnosti aktivně vyvíjím. Tato aplikace za dobu své existence nashromáždila velké množství funkcionalit. Avšak firma vyvíjející tuto aplikaci nemá striktní politiku na kontrolu kvality kódu. Díky tomu se aplikace vyvinula velmi rychle a její změny jsou dynamičtější, než u konkurence. Tento přístup však také vede k tomu, že některé části aplikace jsou relativně kvalitní, jiné však křičí po úpravách. Jednotlivé komponenty aplikace jsou vzájemně velmi provázané, a proto se chyba na špatném místě může projevit opravdovou katastrofou. Toto je přesně stav, kterému by měl můj nástroj předcházet, případně pomáhat řešit. V praxi jsem vyzkoušel mnoho řešení, žádné však plně nevyhovovalo mým požadavkům a požadavkům aplikace. Proto jsem se rozhodl, že takové řešení navrhnu a pokusím se dostat dále, než se zatím dostala konkurenční řešení.

Cíl práce

Cílem bakalářské práce je navrhnout otevřený modulární analyzační nástroj, který by pomohl vývojářům středně velkých webových aplikací postavených na frameworku Ruby on Rails s kontrolou a analýzou výkonu jejich aplikací. A to jak při vývoji tak i u ostrých aplikací. Nástroj by měl nabízet funkcionality pro sledování základních parametrů aplikace. Neměl by vyžadovat příliš nastavení. Měl by být použitelný jednorázově bez jakéhokoli zásahu do aplikace. Pokud má být nástroj využívaný déle, je nastavení samozřejmě nutné. Nastavení by mělo být snadné a jednoduché. Nástroj by měl poskytovat jednoduše pochopitelné výstupy.

V rámci této práce bych chtěl navrhnout základ monitorovací knihovny, která by byla snadno použitelná a snadno rozšiřitelná. Řešení by mělo být otevřené pro komunitu a nabídnout jí kostru, na které by bylo možné jednoduše vystavět komplexní monitorovací nástroj. Knihovna má být použitelná i v ostrých aplikacích pro monitorování provozu aplikace. Bude tedy kladen důraz na rychlost, zejména u částí zamýšlených pro použití v ostré aplikaci. Použití nástroje by mělo mít co nejnižší dopad na rychlost aplikace v produkčním prostředí.

Kapitola 1

Současný stav

1.1 Problémy

S roustoucí poptávkou po webových aplikacích a zjednodušováním návrhu aplikací, které dnešní webové frameworky nabízí, se vývoj těchto aplikací zrychluje. Vývojáři poté realizují aplikace co nejjednodušší a nejrychlejší cestou. Tato cesta většinou však nebere ohledy na správnost a efektivitu kódu. Tato cesta je navíc podporována stále rostoucím hardwarovým výkonem a vede tak vývojáře a firmy k domněnkám, že jejich aplikace je výjimečná a obsahuje náročnou logiku, proto si pronajímají výkonnější a výkonnější hardware, opomínaje alternativu zefektivnění práce aplikace. Toto je poté problém zejména pro rozpočty firem aplikace provozujících. Ačkoli výrobci hardwaru by se mnou nejspíše nesouhlasili.

Webové aplikace již přestávají být dílem školených odborníků. Velké množství začínajících vývojářů začíná programovat webové aplikace bez předešlé zkušenosti či vzdělání v oboru. Proto si často neuvědomují problémy efektivnosti aplikace. Programují stylem „aby to fungovalo“.

Zároveň s přísunem nezkušených vývojářů do tohoto odvětví se webové aplikace stávají mnohem komplexnějšími, než dříve. To souvisí samozřejmě s tím, že většina funkcionalit, potřebných v dnešních aplikacích, se stále opakuje. Do toho přichází otevřená komunita a jednoduchost, s jakou může spolupracovat a vytvářet úžasné nástroje. Tyto nástroje zvládají složitější úkony za vývojáře a jeho úkolem je tedy hlavně tyto knihovny „pospojovat“. To však často přináší mnohá úskalí, kdy jsou knihovny využívány bez rozmyslu, špatným způsobem a jsou tedy slabým místem pro výkon aplikace. Zároveň tento přístup zanáší do aplikace velké množství kódu, který vývojáři nedokáží optimalizovat. Neví, jak kód pracuje a je pro ně tedy náročnější objevit pomalá místa.

Dalším problémem je globalizace internetu. Vyvíjené aplikace jsou často používány celosvětově. Avšak vývojáři znají zvyky své kultury a vyvíjí tedy aplikaci pro ní na míru. Světový trh však může používat zcela jiné postupy, jak k aplikaci přistupovat, jak o ní smýšlet. Tím se může snadno stát, že vývojáři netuší, jak je jejich aplikace opravdu využívána. Od vývojářských stolů pak často můžeme po

konzultacích s nespokojeným klientem slyšet věty typu: „Na to přece klikat neměl, ať nediví!“ nebo „Kdo měl vědět, že to chtěli používat na mobilu?!“.

1.2 Výkon aplikace

V této části bych rád shrnul měřítka výkonu aplikace a typické problémy, které v RoR aplikacích mají negativní dopad na rychlost. Zaměřuji se na řešení v RoR aplikaci, ale většina měřítek je platná pro každou serverovou webovou aplikaci.

1.2.1 Čas zpracování

Základním měřítkem výkonu aplikace je čas zpracování požadavku na aplikaci. Za jak dlouho od chvíle, kdy webový server zadal aplikaci dotaz, aplikace odpoví. Toto měřítko je to, které nás na straně aplikačního serveru zajímá nejvíce. Nese však jen velmi malou informační hodnotu. V podstatě se dozvíme pouze zda bylo zpracování pomalé, či rychlé. Říká nám tedy zejména, které části aplikace by potřebovaly bližší prozkoumání.

1.2.2 Počet dotazů na databázi

Základem drtivé většiny webových aplikací je databáze. Databáze je však uložena na disku a tudíž při čtení z databáze probíhá čekání na disk. Disk je však velmi pomalý a tím se i databáze stává relativně pomalou. Což v důsledku znamená, že chceme data, potřebná pro zpracování dotazu na server, načíst v co nejméně SQL dotazech.

U dotazů na databázi je zajímavý čas a počet. Oba údaje by měl mít vývojář stále na očích, jelikož zde je možné jednoduše nalézt největší problémy aplikace. Vývojář tuší, co se na dané stránce vykonává a je schopen odhadnout počet potřebných dotazů. Proto je i schopen použít číslo k nalezení zřejmých chyb.

1.2.3 Rychlost SQL dotazů

U složitějších aplikací se již nelze vyhnout složitějším dotazům na databázi, zejména pokud vykonává složitější operace s velkým množstvím dat. Proto je velmi zajímavé vidět, jak dlouho trvají jednotlivé dotazy, abychom mohli nalézt ty, které by bylo dobré podrobit bližšímu zkoumání.

S rychlostí dotazů velmi úzce souvisí indexy nad tabulkami databáze. Tento problém řeší nové verze frameworku RoR za vývojáře. Tedy minimálně tím, že přidávají indexy téměř automaticky na spojovací sloupce mezi tabulkami za vývojáře. Touto problematikou se zabývá následující text.

1.2.4 Chybějící indexy

Jedním z fyzických úložišť nabízených většinou systémů pro správu SQL orientovaných systémů je index, který nabízí rychlý přístup k řádkům tabulky, založeném na

jednom nebo více sloupcích [1]. Pokud pomineme index na primárním klíči, který je většinou databázových systémů generován automaticky, je jedním z nejpoužívanějších indexů index na sloupci cizího klíče, který je používán na spojení subentit s jejich rodiči. Ten přináší vysoký nárůst výkonnosti při vyhledávání subentit. To se děje například při dotazech, které pracují s daty z více tabulek databáze, takzvaných JOIN dotazech. Index by měl být na všech sloupcích tabulky, podle kterých probíhá časté vyhledávání záznamů.

1.2.5 Načítání dat

1.2.5.1 N+1 query

Problémy s načítáním dat jsou v `ActiveRecord` řešené pomocí takzvaných preload dotazů. Toto zjednodušuje řešení N+1 query, kdy ve většině případů stačí přidat preload na potřebné místo.

Vývojáři však často zapomínají, že přidaná funkcionalita začne používat data, která dříve použita nebyla. Díky lehce čitelnému kódu lze snadno zapomenout, že za ním často stojí drahé dotazy do databáze.

1.2.5.2 Načítání více dat, než je nutno

Další problém s N+1 query je, že je řešen tam, kde by nenastal a jsou načtena data, která nejsou použita. V RoR není často jasné, která data jsou načtena, jelikož jsou načítána do interních struktur frameworku. Pokud si o nějaká data řekneme, framework nám je načte. Avšak později tato data již nemusíme potřebovat, ale zapomeneme odebrat jejich načítání. Proto je pro vývojáře obtížnější tento problém v RoR odhalit.

1.2.5.3 Counter Cache

Jedná se o problém s dotazem na počet subentit. Pokud potřebujeme zobrazit pouze jejich počet, provádíme jeden dotaz na databázi pro každou entitu. Tento problém se dá řešit jednoduchým přidáním počtu buď do dotazu načítajícího rodičovskou entitu, nebo do tabulky rodičovské entity přidat sloupec s počtem subentit. Toto rozhodnutí je již na vývojáři a oba přístupy jsou v Rails snadno dosažitelné.

1.2.6 Paměť

Velmi důležitým měřítkem je také práce s pamětí. Příliš mnoho spotřebované paměti může vést k drastickému snížení výkonu aplikace, zejména kvůli Garbage Collectoru, který se stará o její čištění a bude popsán v kapitole Použité technologie v sekci věnované Ruby 3.1.1.1. Jeho práce není na první pohled viditelná, ale o to důležitější je uvědomovat si, že stojí procesorový čas. Samozřejmě pokud vyčerpáme dostupnou paměť úplně, stojíme před opravdovým problémem a často i zvýšenými náklady. Avšak i pokud paměti máme více, než naše aplikace využívá, je důležité

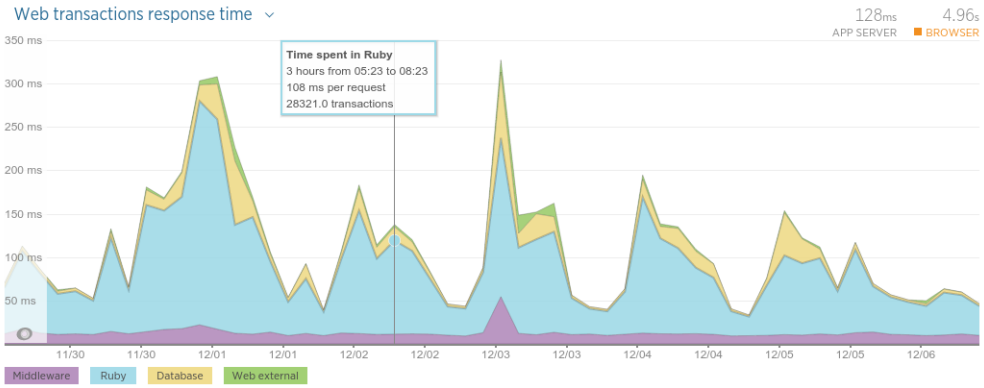
Garbage Collectoru šetřit práci. Před konstrukcemi, které jsou potenciální hrozbou pro spotřebu paměti, se tedy musíme mít na pozoru.

1.3 Existující řešení

1.3.1 New Relic

New Relic je velmi silný analytický a monitorovací nástroj, který nabízí velmi mnoho funkcionalit, na které se chci zaměřit ve své bakalářské práci. Z tohoto důvodu byl i hlavní inspirací při implementaci. Tento nástroj nenabízí žádné specifické postupy pro odhalení problémů typických pro framework Ruby on Rails. Avšak jeho nevětší nevýhodou je cena, v neplacené variantě je velmi ochuzený a pro menší aplikaci je $200 / server / měsíc velmi vysoká cena. Lze monitorovat více aplikací, avšak každá je zpoplatněna zvlášť.

Nástroj nabízí velké množství informací a grafů. Jsou velmi zajímavé, avšak zabere dost času zorientovat se v jejich významu. Velká část z nich ani není pro většinu aplikací příliš zajímavá. Na první pohled není rozhraní příliš intuitivní. New Relic se tedy hodí k monitorování opravdu velkých aplikací, které mají málo instancí a potřebují podrobný monitoring. Zároveň je nejspíše zapotřebí minimálně jednoho školeného odborníka na čtení informací nabízených tímto nástrojem, který by byl schopný z dostupných dat vytěžit co nejvíce informací. Na obrázku 1.1 je vidět jeden z velmi užitečných grafů, který je pravděpodobně hlavním zájmem u většiny sledovaných aplikací.



Obrázek 1.1: New Relic graph

1.3.2 Miniprofiler

Miniprofiler nabízí asi nejvíce funkcionalit, které jsem u dostupných řešení hledal. Je to velmi známé řešení, které bylo původně napsáno pro framework .NET, ale nyní existuje jeho varianta pro mnoho webových frameworků. Jedním z nich je

také Ruby on Rails. Avšak chybí mu těžba dat, z minulosti lze dostat pouze velmi omezené informace. Tím se stává nástrojem spíše pro vývoj a ladění nalezených problémů. Tomuto nástroji také chybí přehlednější rozhraní. Nabízí spíše hlubší pohled na chování aplikace v reálném čase, což může pomoci, až když se přímo soustředíme na její jednotlivé části.

1.3.3 Bullet

Dalším nástrojem, který stojí za povšimnutí, je Bullet, což je velmi užitečná knihovna. Zaměruje se však pouze na načítání dat v Rails aplikacích a základní tři problémy v nich:

- N+1 query
- Nadbytečné načítání dat
- Counter cache

Jedná se tedy o velmi úzce specializovaný nástroj bez samostatného grafického výstupu. Podporuje však mnoho cest, jak problémy reportovat do externích monitorovacích aplikací.

1.3.4 Peek

Peek je nástroj soustředící se na lehký vhled do aplikace. Nástroj je líbivý a pěkně napsaný. Nepřináší však příliš mnoho informací. Dodává informace spíše systémové. Zhruba načrtává, v jakém se aplikace nachází stavu. Je vhodný zejména pro monitorování ostré či režijní aplikace bez nutnosti vzdáleného připojení.

Obrázek 1.2: Peek bar

Na obrázku 1.2 je vidět, že Peek nabízí monitoring relativně mnoha součástí Rails aplikace. Bohužel však o všech podává jen málo informací. Nenabízí také žádný monitoring v čase.

1.3.5 Rack insight

Rack inside je velmi zajímavý nástroj s dlouhou historií. Nástroj se snaží podat vývojáři jakýsi přehled a umí nabídnou relativně dost informací. Jeho vývoj byl několikrát přerušen a převzat jiným vývojářem. Bohužel však i podle toho vypadá samotný nástroj. Kód není konzistentní, není příliš intuitivní na instalaci a jeho jednotlivé komponenty občas nefungují tak, jak mají.

Na obrázku 1.4 je vidět jedna z podávaných informací. Jedná se o počet objektů `ActiveRecord`, neboli RoR ORM, inicializovaných v průběhu zpracování. Tato

Obrázek 1.3: Rack insight bar

informace je bohužel mylná, jsem si jist, že je jich v příkladu, ze kterého byl obrázek pořízen, mnohem více. Přesto, že by tedy tato informace mohla být relativně užitečná, takto je zavádějící a tedy spíše ke škodě.

Obrázek 1.4: Rack insight panel

1.3.6 Sentry

Sentry je aplikace pro monitorování, oznamování a sledování chyb. Jedná se o takzvaný „bug tracking“ systém, který má na starost jednoduchou správu chyb v aplikaci. Sentry do tohoto procesu přidává zejména automatické zaznamenávání chyb. Jedná se o velmi užitečný nástroj ke sledování aplikace. Zaměřuje se však pouze na chybovost.

1.3.7 Scout

Scout je další velmi zajímavý jednoúčelový nástroj. Slouží k „monitoringu“ aplikace pomocí prohlížení jejích systémových logů. Avšak zde naráží na omezení, které vyplývá z omezeného logování v ostrém prostředí z důvodu rychlosti zápisu na disk.

1.3.8 Oink

Oink je též velmi úzce profilovaný nástroj. Zaměřuje se na analýzu využití paměti. Jediná možnost záznamu výsledků je do speciálního souboru. Nabízí však velmi rozšířené možnosti průzkumu na základě tohoto souboru. Nutí však vývojáře zkoumat přímo lokální soubory, čili je vhodný zejména ve vývojovém režimu.

1.4 Shrnutí

Nejdále v problematice je samozřejmě nástroj New Relic, avšak ten je velmi draze placený. Navíc poskytuje velmi mnoho informací, což vede k relativně složité orientaci mezi nimi.

Nástroj MiniProfiler nabízí úplně jiný pohled na aplikaci a zaměřuje se spíše na vývojové prostředí. Nabízí však základní potřebné informace o požadavcích na aplikaci a při vývoji aplikace je velmi užitečným pomocníkem.

Nástroje Peek a RackInsight nabízí základní informace o aplikaci. Opět bychom zde však marně hledali funkci monitorování aplikace.

Ostatní nástroje jsou již velmi úzce profilované a nenabízí komplexní řešení. V tabulce 1.1 jsou přínosy nástrojů shrunuty v tabulce. Parametry jsou hodnoceny znaky *+* nebo *-*. Pokud je parametr nástrojem splněn lépe či hůře, přidávám zdvojené znaménko. Pokud parametr u nástroje nemá význam hodnotit, je tato skutečnost jednoslovně naznačena.

Tabulka 1.1: Srovnání řešení

Řešení	**Monitoring**	**Přehlednost**	**Informace**	**Komplexnost**
New Relic	*++*	*+*	*+*	*++*
Miniprofiler	-	*+*	*+*	*+*
Bullet	*externě*	-	-	*Ne*
Peek	-	*+*	-	-
Rack insight	*?*	-	-	-
Scout	*+*	-	*+*	-
Oink	-	*+*	*+*	*Ne*

Kapitola 2

Požadavky

2.1 Funkční požadavky

- F1. Sledování SQL dotazů
- F2. Analýza problémů s načítáním dat
- F3. Analýza využití paměti
- F4. Záznam výsledků
- F5. Rozhraní pro dlouhodobý monitoring
- F6. Analýza výsledků
- F7. Sledovaní více aplikací

2.1.1 F1. Sledování SQL dotazů

Nástroj bude umožňovat sledování SQL dotazů v rámci zpracování požadavku. Bude zaznamenávat čas trvání jednotlivých dotazů. Posléze nabídne jejich počet, přehled a celkovou dobu, kterou aplikace strávila komunikací s databází.

2.1.2 F2. Analýza problémů s načítáním dat

Nástroj umožní analyzovat případné problémy s efektivitou načítání dat. Zaměří se hlavně na analýzu problémů s N+1 dotazy, načítání nepoužitých dat a counter cache.

2.1.3 F3. Analýza využití paměti

Nástroj umožní sledovat využití paměti a základní analýzu. Zároveň upozorní na nápadné využití velkého množství paměti.

2.1.4 F4. Záznam výsledků

Nástroj umožní zaznamenávat výsledky do různých úložišť. Nabídne implementaci rozhraní pro základní úložiště. Umožní jednoduché přidání nového rozhraní.

2.1.5 F5. Rozhraní pro dlouhodobý monitoring

Nástroj nabídne rozhraní pro dlouhodobý monitoring aplikace. Bude zaznamenávat a analyzovat dlouhodobé používání aplikace.

2.1.5.1 F6. Analýza výsledků

Nástroj poskytne rozhraní pro zpětnou analýzu výsledků a následné vytěžování informací o fungování aplikace. Jelikož se jedná o velmi pokročilou funkcionalitu, nástroj nabídne pouze základní rozhraní. V rámci této bakalářské práce nebudou implementovány pokročilejší metody analýzy výsledků.

2.1.6 F7. Sledovaní více aplikací

Nástroj umožní v dlouhodobém monitoringu sledovat více aplikací. Ošetří zároveň autorizaci aplikací. Aplikace se jednoduše zaregistruje do monitorovacího rozhraní.

2.2 Nefunkční požadavky

- N1. Minimální dopad na produkční aplikace
- N2. Rozšiřitelnost měřených parametrů
- N3. Jednoduché přidání nového úložiště
- N4. Minimální konfigurace
- N5. Přehlednost výstupů

2.2.1 N1. Minimální dopad na produkční aplikace

Nástroj bude sloužit i ke sledování aplikací v produkčním prostředí. Musí proto dbát na minimální dopad na rychlost těchto aplikací.

2.2.2 N2. Rozšiřitelnost měřených parametrů

Nástroj poskytne jednoduché rozšíření, jak přidat další měřené parametry. Pokud tedy vývojář shledá užitečným sledovat nějaké vlastní knihovny, bude pro něj jednoduché přidat parametry těchto knihoven do sledovaných parametrů.

2.2.3 N3. Jednoduché přidání nového úložiště

Nástroj umožní vývojáři přidat rozhraní pro své vlastní úložiště, které není mezi implementovanými.

2.2.4 N4. Minimální konfigurace

Nástroj bude navržen tak, aby bylo jednoduché ho přidat do aplikace bez rozsáhlé konfigurace. Zejména ve vývojovém prostředí, kde by měl sloužit i pro jednorázové použití k odladění jednoho problému. Pro tento účel by měl být použitelný bez jakékoli konfigurace.

2.2.5 N5. Přehlednost výstupů

Nástroj bude poskytovat přehledné výstupy naměřených parametrů a provedených analýz. Jak ve vývojovém prostředí přímo při používání aplikace, tak i v dlouhodobém sledování aplikace.

2.3 Shrnutí

Hlavním cílem nástroje je poskytnout vývojáři snadno použitelné, bez složitého nastavování dostupné řešení. Měl by vývojáři poskytnout dostatek informací, aby byl při vývoji schopen odhalit potenciální výkonnostní problémy a rizika. Neměl by obtěžovat vývojáře, čili vizuální rozhraní nesmí být příliš nápadné, přesto přehledné. Neměl by příliš zpomalit fungování aplikace, zejména v produkčním prostředí.

Cílové řešení by mělo být komunitní a otevřené. Samozřejmě je zde možnost hostovaného řešení a podpory. Avšak základní použití by mělo být jednoduché, zdarma a pokud je potřeba, upravitelné na míru.

Kapitola 3

Návrh a implementace

3.1 Použité technologie

3.1.1 Ruby

Ke zrození programovacího jazyka Ruby došlo v 90. letech 20. století. Jeho předním vývojářem je japonský programátor Yukihiro Matsumoto, přezdívaný „Matz“. Ten se rozhodl vytvořit Ruby, když po mnoha pokusech nenalezl jazyk, který by mu vyhovoval a který by splňoval jeho představy silného, avšak jednoduše čitelného jazyka. “I wanted a scripting language that was more powerful than Perl, and more object-oriented than Python [2].” Ruby je moderní, dynamický, objektově orientovaný jazyk se zaměřením na jednoduchost a produktivitu. Nabízí elegantní syntaxi, která je přirozená pro psaní a snadno čitelná [3].

Díky čitelnosti a intuitivnosti se někteří zastánci Ruby dokonce domnívají, že Ruby není počítačový jazyk. To by totiž naznačovalo, že je to jazyk počítače a programátor je překladatel do tohoto jazyka. Ruby je však tak čitelné, že v něm lze i myslet a může být tedy považován za programátorovi přirozenější než počítači. Tím se tedy Ruby interpreter stává tímto překladatelem a programátor pouze komunikuje s interpreterem [4].

Citovaná kniha [4] zároveň přináší vhled do myšlení hlavních zastánců jazyka. Jedná se o otevřený a hravý jazyk, ve kterém je radost programovat a je lehké ho přizpůsobovat vlastním potřebám.

3.1.1.1 Garbage Collector

Garbage Collector zajišťuje vývojáři abstrakci od paměti. Vývojář se tedy nemusí starat o uklid místa použitého pro jeho proměnné. GC je dnes již zabudován do většiny moderních jazyků. Postup uklízení v paměti je však stále diskutované téma. Sledování použité paměti je tím nejmenším problémem a můžeme říct, že již vyřešeným. Avšak pro úklid to úplně neplatí. Jelikož proces úklidu je relativně náročný, vedou se rozsáhlé debaty o tom, jak často tento proces spouštět. Velké množství alokované paměti zpomalí běh programu, časté uklízení také. Ruby nabízí způsoby,

kterými lze do tohoto procesu zasáhnout. Ty však jsou spíše až zoufalým řešením, neboť pro většinu aplikací se vyplatí spíše důvěřovat přednastaveným hodnotám. Vývojář by měl mít na paměti zejména to, že vytváření proměnných stojí čas při úklidu. Měl by tedy zejména dbát na to, aby udržel datový otisk v programu na nezbytném minimu.

3.1.2 Ruby gem

Jako každý jazyk, i Ruby má knihovny, které pomáhají programátorům sdílet kód. V Ruby byl systém knihoven posunut na ještě vyšší úroveň. Pomocí RubyGems se sdílení kódu a používání knihoven stává naprosto triviální záležitostí. RubyGems pro programátora, který gemy využívá, zvládá instalaci gemu i se všemi závislostmi. Pro programátora gemů je připraveno intuitivní rozhraní pro definování závislostí, předpřipravené souborové struktury, jednoduché vytvoření a publikování balíčku (gemu) z kódu. Zároveň poskytuje velmi intuitivní správu verzí gemů.

Z popsaných důvodů je logické, že naprostá většina knihoven pro Ruby je publikována a používána skrze gemy. Proto jsem se i já rozhodl tento standard využít a mé řešení je publikováno pomocí gemů.

3.1.3 Rack

Rack je webserver interface pro Ruby. Nabízí minimalistické rozhraní mezi webovým serverem a Ruby aplikacemi [5].

V průběhu posledních let se rack stal vpodstatě standardem při vývoji Ruby webové aplikace.

Takto vypadá aplikace, která na jakýkoli požadavek odpoví „Hello world.“ s HTTP status kódem 200(OK) a v hlavičce nastavený typ odpovědi na HTML.

```
app = Proc.new do |env|
  [
    '200',
    {'Content-Type' => 'text/html'},
    ["Hello world."]
  ]
end
```

Aplikace může být naprosto jakýkoli objekt, pokud odpovídá na metodu `call` s jedním parametrem, kterým je proměnné prostředí. Návratovou hodnotou této metody musí být trojice status, hlavičky, obsah (ten musí implementovat metodu `each`, kterou Rack používá pro „vykreslení“ odpovědi).

3.1.4 Rack middleware

Rack je však více, než pouhým rozhraním. Používá se zároveň k seskupování a řazení modulů zodpovědných za výslednou odpověď aplikačního serveru. Tato tech-

nika je známá jako rack middleware a spočívá v postupném probublávání requestu mezi jednotlivými moduly, přičemž každý přidá svou trošku do výsledné odpovědi.

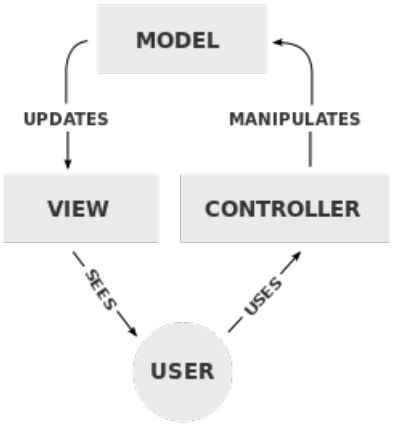
Aplikace je poté výsledkem všech middlewarů a kořenové aplikace. Middleware musí implementovat mimo rackové metody `call` také konstruktor, který přijímá jako parametr aplikaci, která následuje v řetězu po něm. Tato aplikace může být jak Rack aplikace tak další Rack middleware.

V metodě `call` potom volá následující aplikaci. Middleware samozřejmě může řetěz přerušit a vytvořit svojí vlastní odpověď, což se ve výjimečných případech může hodit. Toto provádí například Rails Exception handler v případě výjimky. Avšak pokud bychom v Middlewaru přerušovali řetězení častěji, začíná Middleware ztrácet smysl, jelikož se z něj stává Rack aplikace.

3.1.5 Ruby on Rails

Ruby on Rails je framework napsaný v jazyce Ruby a je jedním z hlavních důvodů rozšíření tohoto jazyka i v anglicky mluvících zemích.

RoR je vystavěn na architektuře Model2, která je odvozena z architektury MVC, kterou Model2 upravuje pro použití v serverových webových aplikacích. Vizuální komponenty nemohou být přímo řízeny a aktualizovány modely. Serverové aplikace totiž pohledy vyrenderují a odešlou zpět uživateli do prohlížeče. Server tedy nemá možnost pomocí návrhového vzoru Observer pohled aktualizovat v případě, že se změní model. Toto je však jedním ze základů MVC návrhového vzoru viz. 3.1. Oproti tomu Model2 je více uzpůsoben tomuto problému. Tento problém se stává diskutovaným až v poslední době, kdy se rozmáhají javascriptové frameworky. Ty jsou na straně klienta a mohou tedy implementovat MVC architekturu. A zde nastává problém terminologie.



Obrázek 3.1: MVC diagram [6]

Ruby on Rails je publikováno pomocí gemu, a proto je i velmi jednoduché vyvíjet na jednom systému pro více verzí frameworku, bez nutnosti přímého publikování

kódů frameworku zároveň s kódy aplikace. Postačí definovat verzi frameworku ve speciálním souboru `Gemfile`, který spravuje závislosti Ruby aplikací.

3.1.5.1 Convention over configuration

Jedná se o návrhový vzor, který se snaží minimalizovat počet rozhodnutí, která musí vývojář učinit. Aplikace by měla být tedy bez nutnosti jakéhokoli nastavení v použitelném stavu, v jakém vyhovuje co největšímu počtu vývojářů. Tento návrhový vzor se RoR snaží dodržovat. A RoR tým odvádí velmi dobrou práci. Většina vývojářů se o rozšířených nastaveních dozvídá až po několikaleté zkušenosti s RoR a začátečníci často ani nevědí, kde hledat konfigurační soubory. Tento fakt patří mezi hlavní aspekty frameworku, které mu zajišťují masovou oblíbenost.

3.1.5.2 Don't Repeat Yourself

Princip DRY požaduje, aby každá část systému měla jedinečnou, konečnou a opodstatněnou reprezentaci. Částí systému mohou být data, metadata, logika, funkcionalita nebo algoritmus [7]. RoR velmi přispívá k dodržování tohoto principu. Příkladem může být `ActiveRecord`, který načítá seznam sloupců databáze z databáze samotné a není tedy nutné je znovu definovat v kódu.

3.1.5.3 Middleware v RoR

Aplikace v Ruby on Rails je Rack aplikace a hojně využívá technologie Rack middleware. Poskytuje vývojáři možnost do řetězce Rack middleware vložit na kteroukoli pozici vlastní middlaware a tím ovlivnit předzpracování požadavku nebo finální odpověď. Pomocí příkazu `rake middleware` lze zjistit, jaké middlewary jsou použity a v jakém pořadí jsou volány.

3.1.6 `ActiveSupport::Notifications`

`ActiveSupport` je knihovna funkcionalit pro Ruby, která je součástí RoR frameworku. Nabízí užitečné nástroje, které vznikly při tvorbě frameworku Ruby on Rails. Tyto nástroje nesouvisí přímo se samotnou aplikací, ale jedná se o jakési helpery.

Jednou z užitečných knihoven, které `ActiveSupport` nabízí, je knihovna `Notifications`. Ta nabízí jednoduchou implementaci návrhového vzoru Observer. Pomocí `Notifications` lze informovat o dění uvnitř aplikace, bez nutnosti „monkey patchingu". Využívají event systému, který nabízí jednoduché API.

Rails aplikace využívá `Notifications` v postačujícím množství pro monitoring základních komponent aplikace. Dozvíme se tak pomocí nich například o všech dotazech na databázi, vykreslení snippetu, začátku a konci requestu.

Jelikož Observer návrhový vzor je pro monitoring nejsmysluplnější, rozhodl jsem se `ActiveSupport::Notifications` využívat v maximální míře. Bohužel na všechny monitorované problémy použít nelze.

3.1.7 Rails engine

Engine může být považován za malou aplikaci, která poskytuje funkcionalitu svojí hostující aplikaci. Rails aplikace je vlastně „superset“ enginu, s její hlavní třídou `Rails::Application` dědící mnoho funkcionality od třídy `Rails::Engine`, která je základem enginu [8]. Engine je používán pro vytváření zásuvných modulů, které zvládají části funkcionality aplikace. Velmi dobrý příklad pro použití enginu je fórum či chat. Tyto komponenty se hodí použít jako engine dokonce i v případě, že ji vyvíjí stejný tým lidí a nemá v plánu tyto komponenty použít v jiné aplikaci. Řada vývojářů v enginech totiž vidí možnost, jak efektivně rozdělit zodpovědnost aplikace na menší části.

3.1.8 Bootstrap

Bootstrap je mocný framework, nabízející sadu CSS tříd a JavaScriptových funkcí, které zjednodušují proces vývoje front-endu aplikace [9]. Bootstrap pomáhá s tvorbou front-endu aplikace a s jeho použitím vývojář nemusí vytvářet CSS a teoreticky ani JavaScript kód. Minimálně dokud postačuje základní design nabízený bootstrapem, který je velmi flexibilní a u většiny základních aplikací tedy dostačuje.

3.1.9 Chartkick

Jedná se o JavaScriptovou knihovnu s Ruby rozhraním pro snadnou tvorbu grafů. Podporuje pouze základní typy grafů a pro vykreslení používá externí vykreslovací engine. Základní podporovaný engine je Google Charts. Jeho hlavní předností je velmi jednoduché použití a předpřipravené knihovny pro předzpracování vykreslovaných dat.

3.1.10 Git

Git je volně šiřitelný distribuovaný verzovací systém s otevřeným kódem. Je navržen tak, aby zvládal spravovat malé i velké soubory rychle a efektivně.

Git byl navržen linuxovou komunitou, vedenou Linusem Torvaldem, tvůrcem Linuxu, pro vývoj Linuxového kernelu. Základními cíli při vývoji Gitu byly:

- Rychlost
- Jednoduchost
- Podpora pro nelineární vývoj (tisíce zároveň existujících větví)
- Kompletní distribuovanost (každý vývojář má kompletní kopii repozitáře)
- Efektivně zvládat i obrovské projekty (jako je Linux kernel)

Od jeho zrodu roku 2005 se Git vyvinul do jednoduše použitelného nástroje. Stále však dodržuje tyto předsevzaté cíle [10].

3.1.11 GitHub

GitHub je komerční server hostující Git repozitáře. Nabízí však zdarma neomezené funkcionality pro veřejné repozitáře. Za dobu svého působení si vydobyl prvenství v hostování projektů s otevřeným kódem. Toto prvenství mu zajistilo zejména přehledné rozhraní a mnoho nadstandardních služeb, které poskytuje nad Git repozitáři. Projektu s otevřeným kódem tak nabízí plnohodnotné prostředí se vším, co takový projekt potřebuje k životu a správě. Což je zejména užitečné pro malé projekty, u kterých se nevyplatí zvlášť vytvářet prezentační web, správu úkolů a hosting zdrojových kódů. Tyto základní funkcionality mají vývojáři na GitHubu pohromadě a dokud je vše veřejné, je také vše zdarma.

3.1.12 Shrnutí

Jako cílové aplikace byly vybrány aplikace postavené na RoR. Jelikož je nutné analýzu provádět přímo v aplikaci, použité technologie nebylo příliš těžké vybrat. Základními použitými technologiemi bude tedy jazyk Ruby, na něm vystavěný framework Ruby on Rails. Nástroj bude Rails engine publikovaný pomocí gemů přes rozhraní RubyGems. Celá práce na nástroji bude verzována pomocí Git a zdrojový kód i s dokumentací a domovskou stránkou bude na GitHubu.

3.2 Možná řešení

V této kapitole bych rád načrtl možné cesty, kterými se bylo pro splnění požadavků možné vydat, a jejich zhodnocení.

3.2.1 Měření času požadavku

Doba trvání požadavku se dá měřit mnoha způsoby. Pokud vynecháme měření zvenčí či na úrovni webového serveru, zbyde nám stále měření od předání požadavku Rackové aplikaci a měření po zpracování middlewary. Já jsem se rozhodl vydat se cestou, která by byla co nejblíže samotné aplikaci a byla tak co nejméně ovlivněna předzpracováním požadavku. Jelikož předzpracování je většinou stejné u všech požadavků, není příliš zajímavé ho zahrnovat do měření. Navíc se jedná o relativně zanedbatelné hodnoty.

3.2.2 „Monkey patching“

Jedná se o metodu, která je v Ruby velmi oblíbená. Při této metodě upravujeme chování aplikace za běhu a ovlivňujeme pouze právě běžící instanci. Tato metoda je velmi oblíbená pro obalování původních metod a přidávání funkcionalit před a po jejich volání. Jedná se o nejjednodušší způsob měření času běhu metody. Avšak zasahování do kódu aplikace tímto způsobem má mnoho úskalí. Zejména to je pak nepředvídatelnost takového kódu a obtížné nalézání chyb. Také s každou takto upravenou metodou přibývá počet volání metod a tím se zpomaluje běh celé aplikace.

Jelikož monitoring by měl aplikaci ovlivnit co nejméně je to možné, snažil jsem se této metodě vyhnout, kde to jen bylo možné.

3.3 Navržené řešení

3.3.1 Cílová platforma

Mnou navržené řešení je zaměřeno na framework Ruby on Rails. Snaží se však dodržovat základní pravidla pro pozdější snadnou rozšiřitelnost na další Ruby frameworky.

3.3.2 Rozdělení do komponent

Řešení se skládá ze dvou hlavních částí a jedné sdílené knihovny. První část je Rails engine, který zajišťuje samotné měření aplikace a ve vývojovém prostředí přidává panel s přehledem.

Druhou částí je opět Rails engine, který zajišťuje vizualizaci a ukládání historie výsledků. Tento engine je možné použít přímo v aplikaci samotné, nebo v oddělené aplikaci pro sbírání statistik z více aplikací. Pro malé aplikace je vhodné ukládání přímo v aplikaci, zejména kvůli jednodušší správě. Zatímco pro velké robustní aplikace je vhodné využít řešení oddělené aplikace. Zejména proto, že databáze výsledků může ve větší aplikaci rychle narůstat na objemu a pokud by byla spojená s aplikační databází, mohla by začít zpomalovat dotazy samotné aplikace. Pro ukládání se dá použít meziukládání do některého z paměťových úložišť a výsledky pak zpracovávat na pozadí. Tento přístup velmi urychlí proces ukládání na konci požadavku, vyžaduje však více nastavení.

Poslední částí je knihovna, sdílená oběma hlavními částmi a tou je knihovna takzvaných adaptérů. Adaptéry jsou rozhraní pro ukládání výsledků do různých úložišť. Volba úložišť je velmi široká. Základním úložištěm je paměť, což je vhodné, zejména pokud se výsledky využijí pouze pro zobrazení v ladícím panelu.

3.3.3 Využití dostupných řešení

Jelikož mé řešení má být otevřené a komunitní, je žádoucí v něm využít existující řešení pro dílčí úkoly. Je to zejména vhodné pro roztříštění zodpovědnosti. Některé existující knihovny již pokrývají části problematiky velmi obstojně a nemá smysl implementovat je znovu. Rozhodl jsem se tedy využít již dříve zmíněné nástroje Bullet a rack-mini-profiler, které jsou ve svých oblastech velmi dotaženými nástroji. Snažím se tedy využít jejich potenciál co nejvíce.

3.3.4 Adapter

Adapter slouží k výběru úložiště pro naměřené hodnoty. Prozatím jsou dostupné tyto adaptéry:

- **Memory** Základní adaptér, který výsledky uchovává pouze po dobu běhu aplikace. Je užitečný zejména pro nulovou konfiguraci a postačí pro zobrazení aktuálních informací. Neumožňuje však dlouhodobý monitoring aplikace. Velkým problémem tohoto úložiště je, že funguje pouze v serverech jednoprocesových jednovláknových.

- **File** Adaptér, který též nevyžaduje téměř žádnou konfiguraci a nemá žádné závislosti. Jeho velkou nevýhodou je, že vytváří velké množství souborů. Funguje však i ve vícevláknových serverech.

- **Redis** Umožňuje uložit výsledky do memory based storage. Nabízí jednoduchou konfiguraci, má však omezený objem dat. Proto jsou výsledky po určité době mazány. Doba se dá nastavit, pokud nastavena není, výsledky zůstanou uloženy půl hodiny.

- **Memcached** Jedná se o další memory based storage, které umožňuje monitorování více aplikací najednou. Jedná se pravděpodobně o nejlepší volbu pro produkční prostředí.

- **InfluxDB** Ukládá výsledky do databáze specializované na ukládání časových posloupností. Tuto možnost budou chtít využít zejména lidé, kteří se rozhodnou využít jiné zobrazování, než nabízenou komponentu, například pro nástroj Grafana.

- **Server** Výsledky jsou posílány na vzdálený server - Vizualizer - pomocí HTTP. Data je samozřejmě možné posílat na jakýkoli server, který implementuje rozhraní pro deserializaci dat. Výsledky jsou posílány asynchronně, aby odpověď nečekala, než se data na serveru uloží.

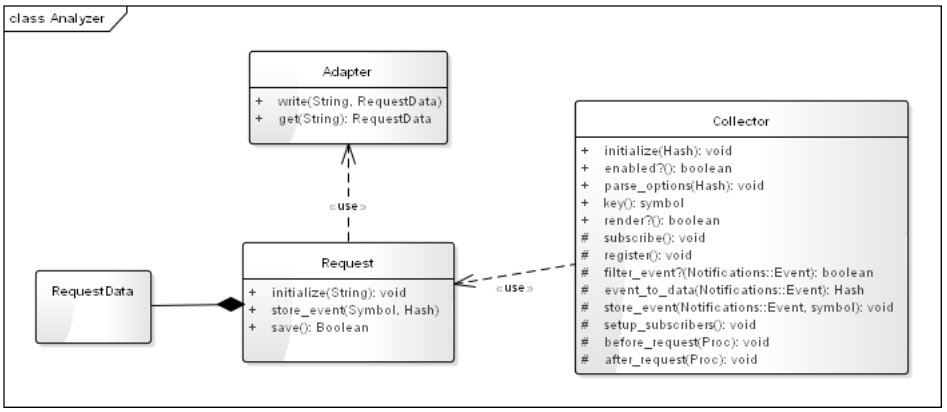
Je možné také doprogramovat vlastní adaptér. Adaptér musí implementovat pouze dvě základní metody:

- `write(request_id, data)` – uloží data do úložiště

- `get(request_id)` – načte data z úložiště v původním formátu.

Data jsou objektem třídy `Speedup::RequestData`, která je také obsažena v knihovně. Tato třída dědí od základní Ruby třídy `Hash`. Přidává pouze pomocné metody a drží si informaci o použitých kontextech. Kontexty většinou odpovídají použitým collectorům. Jeden collector však může zapisovat i do více kontextů, není to tedy striktním pravidlem.

Adaptérů lze použít i více zároveň. Například pro ukládání dat na vzdálený server a zároveň do InfluxDB. Zde je však potřeba myslet na to, že ukládání do každého adaptéru stojí čas.



Obrázek 3.2: Model základních tříd Analyzeru

3.3.5 Analyzer

3.3.5.1 Collector

Analyzer je modulární, což je zařízeno pomocí collectorů, které sbírají data z různých částí aplikace. Každý má své vyhrazené místo v ladícím panelu a svou sekci ve vizualizačním nástroji. Nejsou vzájemně závislé a proto je jejich volba na vývojáři. Je zároveň velmi jednoduché přidat další collector.

Všechny collectory dědí od třídy `Speedup::Collectors::Collector`, jehož rozhraní je znázorněno v diagramu 3.2. K dispozici poté mají konstruktor `initialize`, který je volán automaticky při inicializaci aplikace a jako parametr jsou mu předána nastavení z konfiguračního souboru. Programátor by neměl zapomenout použít volání kontruktoru v mateřské třídě metodou `super`. Konstruktor nemusí být implementován a collectoru postačí v metodě `parse_options` nastavit základní parametry. Další důležitou metodou je `setup_subscribers`, která slouží k zaregistrování na události a nastavení příslušné obsluhy – uložení.

Pokud collector využívá `ActiveSupport::Notifications`, jsou ve třídě připraveny pomocné metody, které pomohou se zpracováním událostí. Nejdůležitější z nich, metoda `register` se jménem události jako parametrem, slouží ke zpracování událostí tohoto jména. Bez dalších nastavení metoda zapíše všechny události tohoto jména v kontextu nesoucím název collectoru. Pro odfiltrování některých událostí slouží metoda `filter_event?(evt)`, která jako parametr přijímá událost typu `ActiveSupport::Notifications::Event` a dle návratové hodnoty `boolean` se událost uloží či nikoliv. Další metodou, kterou je možné implementovat, je `event_to_data(evt)`, které je předána událost a jako návratovou hodnotu očekává `Hash` s informacemi o události tak, jak budou uloženy. V základní implementaci tato metoda zaznamená veškeré pomocné informace, čas vzniku události a dobu zpracování události. Pokud je nutné událost obsloužit jiným než předpřipraveným způsobem, lze metodě `register` předat volitelný blok. V bloku je pak definována obsluha události.

Další dvě zajímavé pomocné metody nesou název `before_request` a `after_request`. Obě přijímají blok jako parametr a tento blok bude proveden před provedením dotazu v prvním případě a po dokončení dotazu v případě druhém.

3.3.5.2 Middleware

Základem Analyzeru je Rack middleware, starající se o inicializaci struktury, do které se v průběhu zpracování požadavku aplikací ukládají měřené hodnoty. Po dokončení zpracování odpovědi se postará o uložení naměřených hodnot do zvoleného úložiště. Pokud je povolen panel, přidává middleware k odpovědi placeholder a pomocné metody pro jeho načítání. Data jsou načtena do panelu dodatečně pomocí AJAX techniky, aby nezpomalovala request samotný.

3.3.5.3 Vývojový panel

Panel se vykresluje po collectorech, každý collector má svojí šablonu, která se nalézá v `views/speedup_rails/collectors/<nazev_collectoru>`. Do této šablony je předána proměnná `data` obsahující data kontextu odpovídajícímu názvu collectoru. V šabloně je také možné přidat panel s detailními informacemi. Vykreslí je do bloku s názvem `speedup_additional_details` a o jejich umístění a zobrazování / skrývání se postará hlavní šablona. Tento přístup je ukázán na příkladu 3.1

Listing 3.1: Vykreslení detailu SQL dotazů

```
<% content_for(:speedup_additional_details) do %>
  <div class="<%= key %>">
  <% data.each do |query_data| %>
    <div><%= query_data[:query] %></div>
  <% end %>
  </div>
<% end if data.any? %>
```

3.3.6 Grafana

Grafana je velmi intuitivní a silný nástroj pro zobrazování časových grafů z nejrůznějších zdrojů, jedním z nich je také InfluxDB. Je tedy jednou z alternativ k vizualizeru pro sledování výkonu aplikace v průběhu času. Vyžaduje použití InfluxDB databáze jako adaptéru. Grafana je organizována do nástěnek, které sdružují tématicky podobné grafy. Na přiloženém CD je k dispozici příklad nástěnky pro grafanu. Stačí nahrát nástěnku a nastavit svůj zdroj dat – influxDB databáze, nastavena jako cílové úložiště monitorované aplikace.

3.3.7 Vizualizer

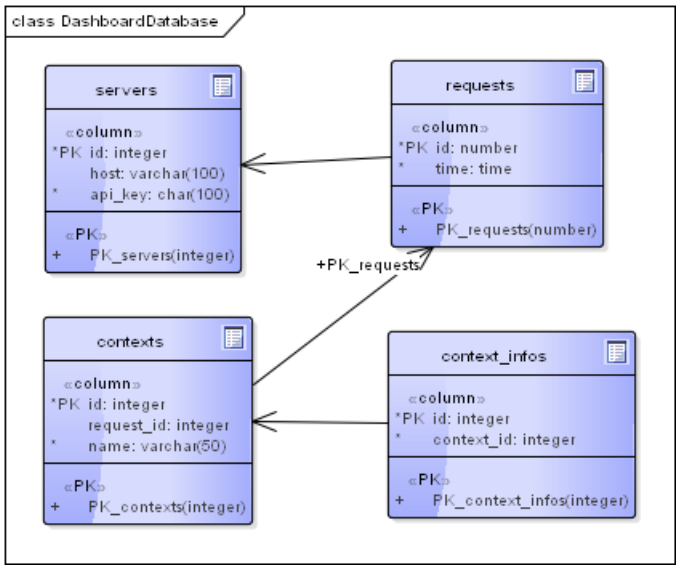
Vizualizer je hlavním viditelným výstupem zejména užitečným v produkčním režimu na ostré aplikaci, kde se nehodí zobrazovat toolbar (i když i to je možné). Umožňuje zobrazovat přehled za delší časové období (chybovost, průměrná rychlost jednoho requestu). Období lze nastavit.

Vizualizer zároveň přidává možnost ukládat výsledky do databáze. Do budoucna je potřeba zpracovat zobrazování dat i z jiných úložišť, aby nebylo nutné výsledky posílat přímo aplikaci, která se stará o zobrazování (jelikož se nejedná o nejrychlejší metodu, pokud aplikace nejsou v lokální síti).

Vizualizer může monitorovat více aplikací, avšak pokud výsledky posíláme z jiné aplikace, musíme ji nejdříve zaregistrovat. Ověření aplikace probíhá přes vygenerovaný klíč, který je nutné zkopírovat do konfiguračního souboru monitorované aplikace.

3.3.7.1 Datový model

Datový model pro uložení dat je velmi jednoduchý a přímo odpovídá struktuře, v jaké jsou data zaznamenávána. Na obrázku 3.3 je vidět jeho zjednodušené schéma. Schéma nezobrazuje všechny parametry.



Obrázek 3.3: Databázový model Vizualizeru

3.3.8 Neimplementované

V návrhu je také zamýšlena varianta, kdy Analyzer zapisuje do paměťového úložiště a Vizualizer si z tohoto úložiště hodnoty čte. V takovém řešení odpadá často velmi časově náročný HTTP přenos dat. Toto řešení bohužel nebylo naimplementováno.

Dále je v návrhu analyzační proces requestu, který po přijetí dat requestu data analyzuje pro vytěžení více informací, než čistá zaznamenaná data. Tento proces je připraven a je naimplementován základní analyzer, ale bohužel nezbyl čas tento proces doladit.

Kapitola 4

Použití

4.1 Konfigurace

Aplikace dodržuje návrhový vzor „Convention over configuration“ a ve vývojovém prostředí není třeba nic nastavovat. Je přednastaveno ukládání do paměti, cesty enginu využité při načítání dat toolbaru se automaticky namapují. Ve vývojovém prostředí se zobrazí toolbar. Po přidání Vizualizeru přímo do aplikace se adaptér automaticky nastaví na databázi. Vizualizer ve vývojovém prostředí tedy také funguje bez nutnosti konfigurace.

V produkčním prostředí je nutná alespoň minimální konfigurace, jelikož nám již nestačí ukládat výsledky do paměti. Nejspíše také nechceme ukládat výsledky ve stejné aplikaci, ale chceme Vizualizer přidat do aplikace, se kterou budeme komunikovat přes HTTP protokol.

Konfigurace gemu je možná přes konfiguraci aplikace, v podsekci speedup.

Do souboru config/environments/production.rb bychom tedy přidali například:

```
config.speedup.adapter = :server, {
  url: 'http://localhost:8080',
  api_key: ENV['SPEEDUP_API_KEY']
}
```

4.1.1 Adaptér

U většiny adaptérů je možné (často nutné) další nastavení, zejména umístění a přístupové údaje k úložišti.

4.1.1.1 Soubor

Adaptér `File`, ukládající údaje do souboru, má jediný parametr a tím je `:path` – relativní cesta, kde se ukládají soubory s hodnotami.

4.1.1.2 Server

- `:url` - string HTTP adresa na které běží monitorovací server
- `:api_key` - string přístupový klíč, vygenerovaný monitorovacím serverem pro ověření

U serverového adaptéru je nutné nastavit adresu serveru, na který chceme ukládat výsledky sledování. Tento server je pravděpodobně Vizualizer, lze však využít jakýkoli server, který správně implementuje deserializaci zasílaných dat. Pokud je serverem Vizualizer, je zároveň nutné nastavit přístupový klíč, který slouží k ověření aplikace.

4.1.1.3 `InfluxDB`

- `:host`
- `:database`
- `:username`
- `:password`
- `:expires_in`

Pro InfluxDB adaptér je potřeba nastavit host (server, na kterém databáze běží), databázi, do které chceme výsledky ukládat a uživatelské jméno a heslo s potřebnými přístupovými právy k této databázi. Konfigurace adaptéru je předávána přímo třídě `InfluxDB::Client` knihovny `InfluxDB`, kterou adaptér interně využívá. Proto je možné použít jakékoli nastavení, které tato třída podporuje. Detailní popis těchto nastavení je dostupný na GitHub v dokumentaci knihovny [11]. Jediný parametr, který není předán knihovně, je `:expires_in`. Tento parametr nastavuje dobu, po kterou jsou výsledky v úložišti uloženy.

4.1.1.4 Redis

- `:url`
- `:host`
- `:port`
- `:expires_in`

Za předpokladu, že Redis úložiště běží na stejném počítači jako aplikace, není potřeba Redis adaptér konfigurovat vůbec. Připadné nastavení je opět kromě parametru `:expires_in` předáváno knihovně, která se stará o propojení s Redis úložištěm. Zde se jedná o knihovnu `Redis`, jejíž dokumentaci lze opět nalézt na domovské stránce – GitHubu [12]. Lze si vybrat zadání umístění úložiště jak pomocí `:url`, tak zvlášť konfigurovat cílový server jako `:host` a `:port`.

4.1.1.5 Memcached

Stejně jako v předchozím případě, Memcached adaptér využívá nativní knihovny, zde se jedná o třídu `Dalli::Client` poskytovanou knihovnou `Dalli`. Detailní nastavení je tedy opět k nalezení v dokumentaci knihovny [13].

4.1.2 Analyzer

Použití Analyzeru je díky gemu velmi jednoduché a s pomocí Bundleru je možné instalaci provést pouhým přidáním následujícího řádku do souboru `Gemfile`. Důležité je zvolit správné nastavení parametru group. Pokud bychom parametr opomněli a nenastavili produkční prostředí, kódy nástroje by byly nahrány do paměti zbytečně.

```
gem 'speedup-rails', group: :development
```

U Analyzeru je nutné nastavit adaptér, samozřejmě kromě vývojového prostředí, kde, pokud nepotřebujeme dlouhodobý monitoring, dostačuje memory adaptér.

Další dostupné konfigurace:

- `show_bar` – (`boolean`) – říká, zda zobrazit toolbar, či nikoliv
- `automount` – (`boolean`) – říká, zda připojit cesty analyzeru do cest aplikace automaticky
- `collectors` – (`array`) – pole Collectorů, které budou použity

4.2 UI

4.2.1 Analyzer

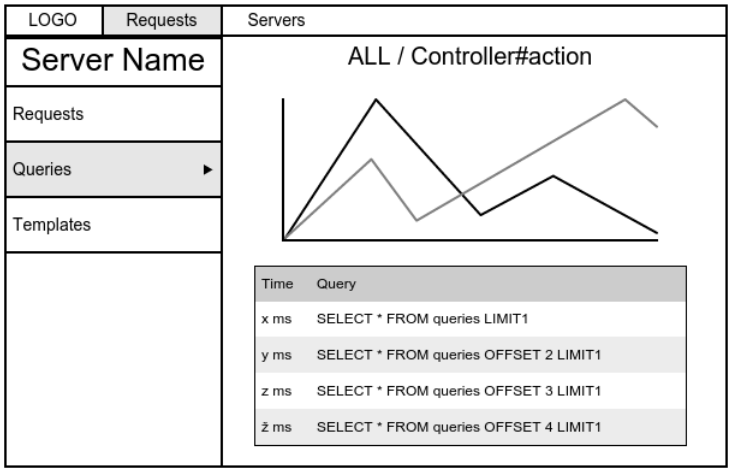
Rozhraní pro Analyzer je velmi jednoduché. Jeho základem je panel, který pro každý zapnutý collector vykresluje příslušné informace.

Obrázek 4.1: Toolbar

Na obrázku 4.1 je vidět základní panel Analyzeru. Po najetí na některé panely se otevře okno s detaily. Ze základních panelů jsou takto interaktivní SQL dotazy, problémy v načítání dat a seznam vykreslených šablon.

4.2.2 Vizualizer

Rozhraní pro Vizualizer je postaveno na frameworku Bootstrap, který zajišťuje přehledný, uživatelsky přívětivý a responzivní design. Základní rozhraní je zobrazeno ve wireframu 4.2. Základem každé části je graf, který je vykreslen pomocí knihovny Chartkick. Většinou jde o průměr hlavní měřené hodnoty za minutu. Ve spodní části se nachází detailní výpis. Požadavky jsou zároveň prokliknutelné, takže je možné zobrazení filtrovat pro jednotlivou akci.



Obrázek 4.2: Rozhraní Vizualizeru

4.3 Testování

4.3.1 Automatické testy

Nástroj je částečně pokryt automatickými testy. Pro automatické testování je využit populární testovací framework Rspec. Toto pomáhá vývojářům přidávajícímu nové funkcionality. Pomocí testů snadno zkontrolují, že nerozbili funkcionality původní. Toto je v komunitním vývoji pravda dvojnásobná, jelikož zde často vývojáři neznají celou logiku aplikace.

4.3.2 Manuální testování

Nástroj jsem testoval na aplikaci EasyRedmine, zmíněné v úvodu. V rámci testování se mi podařilo odhalit značné množství menších výkonnostních problémů. Jelikož aplikace EasyRedmine podléhá autorským právům, nemohu zabíhat s problémy do detailů. Nástroj se v současné podobě hodí zejména na odhalení problémů s SQL dotazy a těch se také týkala většina odhalených chyb.

Testování probíhalo ve dvou fázích. V první fázi jsem se pokusil simulovat vývoj jedné funkcionality aplikace a procházel jsem tedy vybranou skupinu míst s tímto

požadavkem souvisejících. Velmi užitečné mi připadají problémy nalezené nástrojem Bullet, které jsou stále při ruce, avšak ne tak protivné, jako výchozí JavaScriptové upozornění. Ostatní informace mohou spíše upozornit na evidentní chyby.

V druhé fázi jsem na vybraná místa aplikace pustil zátěžový skript a nechal si tak vegenerovat do monitorujícího rozhraní větší množství informací. Poté jsem se zaměřil na nejpomalejší požadavky. Mezi nimi mne často zaujal počet dotazů do databáze. Ve zkoumané aplikaci je běžný počet sto a více dotazů do databáze pro jeden požadavek. Některé požadavky měly toto číslo o poznání větší a v rámci nich jsem nalezl místa, kde bylo možné tento počet radikálně snížit.

Závěr

Cílem práce byla základní implementace nástroje pro sledování a ladění výkonu aplikace psané v RoR. Vytvořený nástroj splňuje základní požadavky na použitelnost a užitečnost. Je velmi dobrým společníkem vývojáře a může mu poskytnout velké množství užitečných informací. Snaží se neomezovat vývojáře a nabízí velkou škálu využití. Nalezne své využití při samotném vývoji aplikace, kdy vývojáři pomáhá prostřednictvím ladícího panelu. Jeho hlavním přínosem je však dlouhodobé sledování aplikace.

Nástoj má implementovány základní stavební prvky. Nabízí jednoduché rozhraní pro přidání dalších sledovaných parametrů. Má rozsáhlé možnosti ukládání dat o sledované aplikaci a lze opět přes jednoduché rozhraní přidat další možnosti. Jako nejpřínosnější hodnotím navržení rozhraní pro dlouhodobé sledování výkonu aplikace.

Díky využití implementovaných řešení má nástroj od počátku silnou, ač nepřímou podporu komunity, tu přímou si bude muset ještě zasloužit.

Při psaní této práce jsem se seznámil s problematikou sledování a ladění výkonu webové aplikace. Rozšířil jsem své znalosti vývoje Ruby Gemu, naučil jsem se využívat engine pro tvorbu části RoR aplikace. Osvojil jsem si postupy používané pro zveřeňování zdrojových kódů komunitě.

Budoucí vývoj

Nástroj má před sebou ještě dlouhou cestu, než bude plnohodnotnou konkurencí větším placeným řešením. Je potřeba doimplementovat více analyzačních komponent, zefektivnit samotné zápisy do sledovacího nástroje a přidat více sledovaných parametrů. Již nyní však je užitečným společníkem při vývoji aplikací v RoR.

Jelikož pracuji ve firmě, která má na využití tohoto nástroje velký zájem, doufám že se budu moci dalšímu vývoji nástroje aktivně věnovat.

Přínos

Nástroj by měl pomoci komunitě psát efektivnější aplikace. To, zda se tento cíl podaří naplnit, není možné samozřejmě předpovědět. Vše záleží zejména na tom, jak bude nástroj komunitou přijat a zda se mu podaří prosadit. Věřím však v jeho potenciál a doufám, že dalším vývojem se tento potenciál bude navyšovat.

Literatura

[1] James R. Groff and Paul N. Weinberg: *SQL, the complete reference*. New York ; Osborne/McGraw-Hill, c2002., ISBN 9780072225600.

[2] Stewart, B.: An interview with the Creator of Ruby. 2001, [Online; accessed 20-04-2015]. Dostupné z: `http://www.linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html`

[3] community, R.: Ruby. 2010, [Online; accessed 24-04-2015]. Dostupné z: `https://www.ruby-lang.org/en/`

[4] Why The Lucky Stiff, : *Why's (poignant) Guide to Ruby*. Simon Orr Studio, ISBN 9781458392466.

[5] Rack: *Rack documentation*. Dostupné z: `http://rack.github.io/`

[6] RegisFrey: The model, view, and controller (MVC) pattern relative to the user. 2010, [Online; accessed 8-04-2015]. Dostupné z: `http://commons.wikimedia.org/wiki/File%3AMVC-Process.svg`

[7] Bachle, M.; Kirchberg, P.: Ruby on Rails. *Software, IEEE*, ročník 24, č. 6, Nov 2007: s. 105–108, ISSN 0740-7459, doi:10.1109/MS.2007.176.

[8] Ruby on Rails: *Rails Guides [online]*. [cit. 2015-04-10]. Dostupné z: `http://guides.rubyonrails.org/engines.html`

[9] Balasubramanee, V.; Wimalasena, C.; Singh, R.; aj.: Twitter bootstrap and AngularJS: Frontend frameworks to expedite science gateway development. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, Sept 2013, s. 1–1, doi:10.1109/CLUSTER.2013.6702640.

[10] Patrick Aljord: *Pro Git (Pro)*. Apress, ISBN 9781430218333.

[11] Home of influxdb-ruby. [Online; accessed 21-04-2015]. Dostupné z: `https://github.com/influxdb/influxdb-ruby`

[12] Home of redis-rb. [Online; accessed 21-04-2015]. Dostupné z: https://github.com/redis/redis-rb

[13] Dalli: *Dalli documentation*. Dostupné z: http://www.rubydoc.info/github/mperham/dalli/Dalli/Client#initialize

Příloha A

Seznam použitých zkratek

RoR Ruby on Rails

UI User interface

GUI Graphical user interface

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

SQL Structured Query Language

API Application programming interface

AJAX Asynchronous JavaScript and XML

DRY Don't Repeat Yourself

ORM Object-relational mapping

GC Garbage Collector

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementovaných gemů
  - `grafana` ..... nástěnka pro nástroj grafana
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
  - `thesis.pdf` ..... text práce ve formátu PDF