Nástroj pro monitoring více instancí MongoDB

Bc. Jan Machala

Diplomová práce
2016

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2015/2016

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Jan Machala**
Osobní číslo:	**A14510**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Forma studia:	**kombinovaná**
Téma práce:	**Nástroj pro monitoring více instancí MongoDB**
Téma anglicky:	**A Monitoring Tool for Multiple Instances of MongoDB**

Zásady pro vypracování:

1. **Proveďte rešerši možností monitoringu databáze MongoDB.**
2. **Analyzujte požadavky na monitorovací nástroj.**
3. **Proveďte návrh a analýzu vhodné architektury pro aplikaci.**
4. **Navrhněte aplikaci sloužící k monitoringu databází.**
5. **Realizujte prototyp aplikace.**
6. **Věnujte pozornost zabezpečení aplikace a přenášených dat.**
7. **Popište formou případové studie nasazení monitorovací aplikace.**

Rozsah diplomové práce:
Rozsah příloh:
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **CHODOROW, Kristina. MongoDB: the definitive guide. 2nd ed. Sebastopol: O'Reilly, 2013, xix, 409 s. ISBN 978-1-4493-4468-9.**
2. **Practical mongodb: architecting, developing, and administering MongoDB. New York, NY: Springer Science+Business Media, 2015, pages cm. ISBN 9781484206485.**
3. **NIALL O'HIGGINS. MongoDB and Python. Farnham: O'Reilly, 2011. ISBN 9781449310370.**
4. **FOWLER, Martin. Destilované UML. 1. vyd. Praha: Grada, 2009, 173 s. Knihovna programátora (Grada). ISBN 978-80-247-2062-3.**
5. **BONNET, Laurent, et al. Reduce, you say: What nosql can do for data aggregation and bi in large repositories. In: Database and Expert Systems Applications (DEXA), 2011 22nd International Workshop on. IEEE, 2011. p. 483-488.**
6. **VRÁNA, Jakub. 1001 tipů a triků pro PHP. Vyd. 1. Brno: Computer Press, 2010, 456 s. ISBN 978-80-251-2940-1.**
7. **MITCHELL, Lorna Jane. PHP web services : [APIs for the modern web]. First edition. Beijing: O'Reilly Media, 2013, x, 104 pages. ISBN 9781449356569.**

Vedoucí diplomové práce: **doc. Ing. Zdenka Prokopová, CSc.**
Ústav počítačových a komunikačních systémů

Datum zadání diplomové práce: **5. února 2016**
Termín odevzdání diplomové práce: **20. května 2016**

Ve Zlíně dne 5. února 2016

L.S.

doc. Mgr. Milan Adámek, Ph.D.
děkan

doc. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 20.5.2016

…………………….
podpis diplomanta

ABSTRAKT

Cílem diplomové práce je vytvoření nástroje pro monitorování většího množství MongoDB instancí. V teoretické části se práce zabývá rešerší dostupnosti informací ke sledování v MongoDB. Následuje popis asynchronního programování v jazyce PHP, postupná integrace a tvorba grafického rozhraní. V praktické části se práce zabývá analýzou požadavků na monitorovací nástroj, využití frameworku Icicle a navrhuje komunikaci pomocí WebSocketů. V další části se pak zabývá použitím integračních nástrojů Travis CI a Scrutinizer CI a realizací grafického rozhraní v ReactJS. V případové studii je popsán způsob nasazení do produkčního prostředí využitím Docker kontejnerů, Docker Cloudu a Amazon Web Services. V poslední části se práce věnuje zabezpečení aplikace.

Klíčová slova:

Monitorování MongoDB, Icicle, ReactPHP, Asynchronní aplikace v PHP, Generátory, Korutiny, Koprogramy, Smyčka událostí, Amazon Web Services, EC2, S3, CloudFront, Docker, Docker Cloud, Postupná integrace, Travis CI, Scrutinizer CI, Composer, Awaitables, Promises, WebSokety, SPA, ReactJS, Babel

ABSTRACT

The aim of this thesis is to build a tool for monitoring multiple instances of MongoDB. The theoretical part deals with the retrieval of possible available information to monitor MongoDB. Following part is a description of asynchronous programming in PHP, continuous integration and graphical interface. In the practical part thesis analyzes the requirements for monitoring utility usage and propose a framework Icicle and suggest a communication via WebSockets. The next section deals with integration tools Travis CI and CI Scrutinizer and implementation of graphical user interface in ReactJS. The case study describes a deployment method to production environments using Docker containers, Docker Cloud and Amazon Web Services. The last part deals with application security.

Keywords:

Monitoring MongoDB, Icicle, ReactPHP, Asynchronous applications in PHP, Generator, Coroutines, Event loop, Amazon Web Services, EC2, S3, CloudFront, Docker, Docker Cloud, Continuous integration, Travis CI, Scrutinizer CI, Composer, Awaitables, Promises, Websokets, SPA, ReactJS, Babel

Chtěl bych poděkovat svojí manželce za podporu při studiích a také mé rodině za podporu při studiích a při vypracovávání diplomové práce. Dále bych chtěl poděkovat za pomoc kolegům Ing. Janu Görigovi a Romanu Antlovi.

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

OBSAH 7
ÚVOD 9
I. TEORETICKÁ ČÁST 10
1 MONITOROVÁNÍ DATABÁZE 11
1.1 REŠERŠE MONITOROVÁNÍ MONGODB 11
1.1.1 ZÁKLADNÍ MNOŽINA REAL-TIME REPORTOVACÍCH NÁSTROJŮ MONGODB 11
1.1.2 PŘÍMÉ PŘÍKAZY DO DATABÁZE 14
1.1.3 NÁSTROJE TŘETÍCH STRAN 15
1.1.4 LOGOVÁNÍ PROCESŮ 16
2 POUŽITÉ TECHNOLOGIE 17
2.1 PHP 17
2.1.1 GENERÁTORY 17
2.1.2 COMPOSER 17
2.2 PHP FRAMEWORK ICICLE 18
2.2.1 EVENT LOOP 18
2.2.2 AWAITABLES 19
2.2.3 KOPROGRAMY 19
2.2.4 BALÍČEK WEBSOCKET SERVER 20
2.2.5 BALÍČEK HTTP SERVER 20
2.3 OVLADAČE PRO MONGODB 20
2.3.1 REACTPHP ADAPTÉR 20
2.4 REACTPHP 21
2.5 PROMISES/A+ 21
2.6 POSTUPNÁ INTEGRACE - CONTINOUS INTEGRATION 22
2.6.1 TRAVIS CI 23
2.6.2 SCRUTINIZER CI 23
2.7 GRAFICKÉ UŽIVATELSKÉ ROZHRANÍ 23
2.7.1 WEBPACK 23
2.7.2 ŠABLONOVÝ SYSTÉM REACTJS 24
3 POŽADAVKY NA MONITOROVACÍ NÁSTROJ 25
II. PRAKTICKÁ ČÁST 26
4 ANALÝZA POŽADAVKŮ NA MONITOROVACÍ NÁSTROJ 27
4.1 KOMUNIKACE S INSTANCEMI 27
4.2 KONFIGURACE SEZNAMU INSTANCÍ 27
4.3 KOMUNIKACE PŘES WEBOVÉ API 27
4.3.1 REST API 27
4.3.2 WEBSOCKETS 28
4.3.3 VOLBA API ROZHRANÍ 28
5 IMPLEMENTACE 29
5.1 SOUBOROVÁ STRUKTURA 29

5.2 MONITORING INSTANCÍ MONGODB ... 30
5.2.1 Smyčka událostí ... 30
5.2.2 Použití ReactPHP adaptéru ... 30
5.2.3 Získání seznamu databází ... 31
5.2.4 Práce s WebSockety ... 31
5.2.5 Komunikační protokol ... 31
5.3 IMPLEMENTACE KONFIGURAČNÍHO SOUBORU ... 32
5.4 INTEGRAČNÍ NÁSTROJE ... 33
5.4.1 Travis CI ... 33
5.4.2 Scrutinizer CI ... 36
5.5 GRAFICKÉ UŽIVATELSKÉ ROZHRANÍ ... 37
5.5.1 Lokální vývoj klientské aplikace ... 37
6 POPIS TECHNOLOGIÍ POUŽITÝCH K NASAZENÍ ... 38
6.1 AMAZON CLOUDFRONT ... 38
6.2 S3 ... 38
6.3 EC2 ... 38
6.4 DOCKER ... 38
6.5 DOCKER CLOUD ... 39
7 NASAZENÍ MONITOROVACÍHO NÁSTROJE ... 40
7.1 PŘÍPADOVÁ STUDIE NASAZENÍ ... 40
7.2 POSTUP NASAZENÍ SPA DO PRODUKCE ... 40
7.3 POSTUP NASAZENÍ API DO PRODUKCE ... 41
7.3.1 Vytvoření Docker kontejneru ... 41
7.3.2 Nastavení Docker Cloud ... 42
7.4 BĚŽÍCÍ APLIKACE ... 45
7.5 VYHODNOCENÍ PŘÍPADOVÉ STUDIE NASAZENÍ MONITOROVACÍHO NÁSTROJE ... 46
8 ZABEZPEČENÍ APLIKACE ... 47
8.1 ZABEZPEČENÍ KONFIGURACE SLEDOVANÝCH INSTANCÍ ... 47
8.2 ZABEZPEČENÍ WEBSOCKETŮ ... 47
8.3 BEZPEČNOST KOMUNIKACE S INSTANCEMI ... 48
8.4 SSL ... 48
ZÁVĚR ... 49
SEZNAM POUŽITÉ LITERATURY ... 50
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK ... 52
SEZNAM OBRÁZKŮ ... 53
SEZNAM TABULEK ... 54
SEZNAM PŘÍLOH ... 55

ÚVOD

Diplomová práce řeší problém s nedostupností nástroje na monitorování více instancí databázových serverů MongoDB. Existující nástroje jsou schopné monitorovat jednu instanci MongoDB, ale selhávají při monitorování větších jednotek, či dokonce desítek instancí. Diplomová práce si klade za cíl poskytnout tento nástroj formou monitorovacího serveru a dát tak uživateli vhled do aktuálního stavu všech databázových instancí, které ho zajímají.

Práce se zabývá rešerší dostupných údajů z MongoDB a možnostmi jak tyto údaje získávat pro použití monitorovacího nástroje. V další kapitola pak obsahuje požadavky na monitorovací nástroj a v následující kapitole se práce zabývá analýzou požadavků na monitorovací nástroj, dostupným technologickým řešením. Z analýzy pak vyplynula i potřeba grafického rozhraní pro zobrazení aktuálního stavu monitorovaných instancí. V předposlední kapitole se práce zabývá zabezpečením takto běžící aplikace a možnými bezpečnostními riziky a v poslední kapitole je popsána případová studie nasazení nástroje do produkčního prostředí pomocí nástroje Docker Cloud.

Téma diplomové práce bylo vybráno za účelem získání praktických zkušeností s dlouhodobě běžícími aplikacemi v jazyce PHP a využití asynchronních knihoven pro zpracování velkého množství požadavků naráz.

I. TEORETICKÁ ČÁST

1 MONITOROVÁNÍ DATABÁZE

Monitorování databáze je nedílnou součástí náplně práce každého správce systému. Správce systému by měl znát aktuální stav databáze v jakýkoliv okamžik. Správce by měl vždy předcházet problémům se systémem a to včasnou reakcí na možné obtíže s provozem spojené. Ke zjištění aktuálního stavu je nutné znát databázový systém a vědět, jak klíčové informace získat a právě k tomu slouží nástroje na monitorování databází. Následující kapitoly se zabývají získáváním cenných dat z MongoDB.

1.1 Rešerše monitorování MongoDB

Dostupné informace k monitorování MongoDB jsou k dle dokumentace [[9]][[1]] následující:

1. základní množina real-time reportovacích nástrojů MongoDB
2. přímé příkazy do databáze, které vracejí aktuální využití databáze s mnohem větší přesností než nástroje v 1. Bodu
3. Využitím MongoDB Cloud Manažeru, služby poskytované přímo asociací MongoDB. Dále pak Ops Manager dostupný jako placená služba pro MongoDB Enterprise Advanced řešení – poskytuje monitorovací nástroj na sběr statistik běžících MongoDB instancí s přidanou vizualizací a upozorněními na základě těchto dat.
4. Logování procesů

Každý z dostupných nástrojů dokáže odpovědět na různé otázky v různých chvílích a situacích. Tyto nástroje se navzájem doplňují. V následujících sekcích se budeme každému z výše zmíněných bodů věnovat podrobněji.

1.1.1 Základní množina real-time reportovacích nástrojů MongoDB

Mezi základní reportovací nástroje se řadí nástroje, jež jsou základní součástí standardního distribučního balíčku s instalací MongoDB.

Mongostat

Nástroj *mongostat* zachycuje a vypisuje součty databázových operací seskupených podle typu (vložení, dotazování, aktualizace, mazání, apod.). Tyto údaje jsou důležité pro zjištění aktuálního zatížení serveru. Tyto údaje lze využít ke zjištění rozložení typů operací v databázi a tím k lepšímu plánování rozvoj infrastruktury a k identifikování potenciálně slabých míst.

Nástroj se připojuje k běžící *mongod* instanci a sbírá z něj statistické data, které následně vypíše do konzole.

Tabulka 1-1 Ukázkový výstup z nástroje *mongostat*

```
insert query update delete getmore command flushes mapped vsize  res faults qr|qw ar|aw netIn netOut conn    time
    *0    *0     *0     *0     0     1|0     0 496.0M  1.2G 105.0M    0   0|0   0|0   79b    10k   33 2016-04-20T00:25:39+02:00
     2     4      2     *0     0     1|0     0 496.0M  1.2G 105.0M    0   0|0   0|0    3k    12k   33 2016-04-20T00:25:40+02:00
     1     2      1     *0     0     1|0     0 496.0M  1.2G 138.0M    0   0|0   0|0    1k    11k   33 2016-04-20T00:25:41+02:00
     3     6      3     *0     0     1|0     0 496.0M  1.2G 105.0M    0   0|0   0|0    4k    13k   33 2016-04-20T00:25:42+02:00
    *0    *0     *0     *0     0     4|0     0 496.0M  1.2G 105.0M    0   0|0   0|0  250b    11k   33 2016-04-20T00:25:43+02:00
     2     2      1     *0     0     1|0     0 496.0M  1.2G 105.0M    0   0|0   0|0    2k    11k   33 2016-04-20T00:25:44+02:00
    *0     1     *0     *0     0     3|0     0 496.0M  1.2G 105.0M    0   0|0   0|0  253b    66k   33 2016-04-20T00:25:45+02:00
    *0    *0     *0     *0     0     1|0     0 496.0M  1.2G 105.0M    0   0|0   0|0   79b    10k   33 2016-04-20T00:25:46+02:00
    *0    *0     *0     *0     0     1|0     0 496.0M  1.2G 105.0M    0   0|0   0|0   79b    10k   33 2016-04-20T00:25:47+02:00
     4     8      4     *0     0     1|0     0 496.0M  1.2G 105.0M    0   0|0   0|0    5k    13k   33 2016-04-20T00:25:48+02:00
```

Mongotop

Nástroj *mongotop* monitoruje aktuální stav čtení a zápisu MongoDB instance a reportuje a sbírá je v pravidelných intervalech specifikovaných argumenty při spuštění.

Tabulka 1-2 Ukázkový výstup z nástroje *mongotop*

```
                     ns    total    read    write    2016-04-20T00:46:22+02:00
     stahovac.raw_seznam     32ms     0ms     32ms
                   admin      0ms     0ms      0ms
   admin.system.indexes      0ms     0ms      0ms
        admin.system.js      0ms     0ms      0ms
admin.system.namespaces      0ms     0ms      0ms
     admin.system.roles      0ms     0ms      0ms
     admin.system.users      0ms     0ms      0ms
   admin.system.version      0ms     0ms      0ms
                      db      0ms     0ms      0ms
```

HTTP Konzole

HTTP Konzole je webová stránka dostupná na každém MongoDB serveru do verze 3.2. Server naslouchá na portu 28017, pokud není nastaveno v konfiguraci jinak. Po přistoupení na adresu např.: http://localhost:28017/ jsou uživateli zobrazeny základní informace o běžící MongoDB instanci:

- Verze MongoDB, stručné informace o serveru, čas běhu od posledního startu
- Přehled stavu zámků
- Tabulka aktuálně připojených uživatelů k této instanci včetně aktuální prováděné operace, stavu zámků, IP adresy a portu

- Aktuální vytížení indexů z pohledu počtu čtení, zápisu, dotazů, vkládání, aktualizací a mazání
- Log událostí MongoDB instance od posledního startu

Výše zmíněné informace jsou zobrazeny ve stručné podobě na jedné stránce bez jakékoliv další vizualizace či zobrazení těchto údajů v čase.

Rozšířené možnosti HTTP Konzole pak nabízí přepínač *–rest*, který je nutné zadat při spouštění databáze. Po zapnutí přepínače nabízí webové rozhraní možnosti pro spouštění přednastavených dotazů na databázi:

- *buildInfo*
- *cursorInfo*
- *features*
- *hostInfo*
- *isMaster*
- *listDatabases*
- *replSetGetStatus*
- *serverStatus*
- *top*

mongod VS134.coolhousing.net

List all commands | Replica set status

Commands: buildInfo cursorInfo features hostInfo isMaster listDatabases replSetGetStatus serverStatus top

```
db version v2.4.14
git hash: 05bebf9ab15511a71bfbded684bb226014c0a553
sys info: Linux ip-10-154-253-119 2.6.21.7-2.ec2.v1.2.fc8xen #1 SMP Fri Nov 20 17:48:28 EST 2009 x86_64 BOOST_LIB_VERSION=1_49
uptime: 1642468 seconds
```

overview (only reported if can acquire read lock quickly)

```
time to get readlock: 0ms
# databases: 5
# Cursors: 0
replication:
master: 0
slave:  0
```

clients

Client	OpId	Locking	Waiting	SecsRunning	Op	Namespace	Query	client	msg	progress
DataFileSync	1		{ waitingForLock: false }		0			:27017		
initandlisten	7		{ waitingForLock: false }		2002	local.startup_log		0.0.0.0:0		
journal	2		{ waitingForLock: false }		0			:27017		
TTLMonitor	187427		{ waitingForLock: false }		2004	magickio.system.indexes	{ expireAfterSeconds: { $exists: true } }	0.0.0.0:0		
clientcursormon	5		{ waitingForLock: false }		0			:27017		

Obrázek 1 Ukázka webového rozhraní HTTP Konzole MongoDB

Využívání HTTP Konzole je od verze 3.2. nedoporučené a tento nástroj bude v budoucích verzích vypnut. Jako hlavní motivaci pro vypnutí tohoto nástroje vidím to, že je v základu automaticky zapnutý a tudíž dostupný z vně sítě (pokud není specifikováno jinak). Ostatně lze tento případ vidět na Obrázku 1, který byl pořízen z veřejně dostupné MongoDB instance.

1.1.2 Přímé příkazy do databáze

Přímými příkazy do databáze jsou myšleny dotazy do databáze přes databázový dotazovací jazyk. Mezi tyto dotazy patří *serverStatus*, *dbStats*, *db.currentOp*, *collStats*, a *replSetGetStatus*. Tyto příkazy si probereme podrobněji.

Příkaz serverStatus

Příkaz *serverStatus* nebo příkaz *db.serverStatus()* z konzole vrací celkový pohled na stav databázové instance, podrobné využití disku, využití paměti, konektivity, žurnálování a přístupy k indexům. Příkaz vrací údaje rychle a nemá žádný dopad na výkon MongoDB instance.

Příkaz dbStats

Příkaz *dbStats* nebo příkaz *db.stats()* z konzole poskytuje základní statistické údaje na úrovni kolekcí. V odpovědi nalezneme informaci o aktuálně zabraném prostoru na disku, množství dat v databázi, počty objektů, kolekcí a indexů.

Příkaz je vhodné využívat ke zjištění stavu a kapacity vybrané databáze. Tento výstup také dovoluje porovnat a určit průměrnou velikost jednoho dokumentu.

Příkaz db.currentOp

Je příkaz, který vrací dokument obsahující informace o aktuálně běžících operacích na databázové instanci. Příkaz lze využít ke zjištění dlouho trvajících dotazů.

Příkaz collStats

Příkaz *collStats* nebo příkaz *db.collection.stats()* z konzole poskytuje stejné statistiky jako *dbStats*, které jsou však zaměřeny na konkrétní databázovou kolekci.

Příkaz replSetGetStatus

Příkaz *replSetGetStatus* vrací přehled o replikách MongoDB instance. Příkaz slouží především k ujištění se, že jsou repliky správně nastaveny.

1.1.3 Nástroje třetích stran

Nástroje třetích stran nejsou součástí základní instalace MongoDB a je třeba se starat o ně vlastními silami. Mezi takovéto nástroje patří: *Ganglia*, *Motop* a *mtop*.

Ganglia

Ganglia je škálovatelný distribuovaný monitorovací systém pro vysoce výkonné výpočetní systémy jako jsou clustery a gridy.

Obrázek 2 Monitorovací nástroj Ganglia zobrazuje v grafech aktuální vytížení jednotlivých částí systému

Nebyly nalezeny žádné nevýhody u tohoto nástroje a splňuje požadavky na monitorovací nástroj.

Motop

Real-time monitorovací nástroj který funguje jako *top* pro MongoDB. Nástroj monitoruje aktuálně probíhající operace na instanci. Umožňuje monitorování více instancí naráz. Nevýhodou nástroje je jeho úzké zaměření pouze na běžící operace, z čehož vyplívá, že uživatel nezná ostatní údaje o stavu databází, serveru a stavu kolekcí.

Mtop

Mtop je jednoduchý python script monitorující aktuálně běžící procesy MongoDB. Umožňuje monitoring více instancí najednou, má však stejné nevýhody jako nástroj motop.

Hostované monitorovací nástroje (SaaS)

Dostupné hostované monitorovací nástroje pro MongoDB: MongoDB Cloud Manager, Scout, Server Density, Application Performance Management, New Relic, Datadog

1.1.4 Logování procesů

V průběhu normální operace MongoDB instance se každá informace, aktivita serveru a výstup operací loguje na standardní výstup do konzole nebo do logovacího souboru. Instanci je možné nakonfigurovat a zvolit co vše je potřeba logovat a kam. Díky tomu lze zjišťovat stav informace z logů databáze naživo při běhu serveru bez přístupu k samotnému souboru. Nastavit lze *verbosity*, nebo-li úroveň logování (příkaz *db.setLogLevel()*). K souboru pak lze přistoupit přímo na serveru, nebo přes administrátorský dotaz nad instancí: *db.adminCommand({getLog: "global"})*.

2 POUŽITÉ TECHNOLOGIE

Předchozí kapitola se zabývala způsoby získávání detailních informací z MongoDB. V této kapitole se budeme zabývat použitými technologiemi, které vyplynuly z analýzy v kapitole 4. V kapitole jsou popsány základní koncepty, které budou využity pro implementaci nástroje jako jsou Generátory v PHP, Framweork Icicle, Smyčka událostí, Promises, Ovladače pro MongoDB, Integrační nástroje a Grafické uživatelské rozhraní.

2.1 PHP

2.1.1 Generátory

Od verze PHP 5.5.0 a ve verzi PHP 7.0 podporuje PHP tzv. generátory. Generátory poskytují jednoduchý způsob, jak vytvořit tzv. *iterátor* (třída implementující \Iterator).

Generátor dovoluje psát kód, který využívá *foreach* pro procházení přes datový soubor bez nutnosti alokace celého vstupního datového pole v paměti. Předčasné alokování způsobovalo přetečení paměti, překročení paměťových limitů a enormní množství času k optimalizaci chodu takto paměťově náročného programu. Místo toho lze využít funkci generátoru, která je podobná jako normální funkce s tím rozdílem, že funkce vrací následující hodnotu až na vyžádání (*yield*) – generuje ji až na vyžádání. Následující hodnotu funkce vrací, dokud generátor nenarazí na poslední hodnotu.

Jako jednoduchý příklad lze tímto způsobem implementovat funkci *range()* (vrací rozsah hodnot od-do) jako generátor. Standardní funkce při použití *range(0, 1000000)* alokuje přes 100 MB paměti načež generátor jí využije jen 1kB.

2.1.2 Composer

Composer je nástroj pro řešení závislostí v PHP. Composer umožňuje deklarovat závislosti na externích knihovnách projektu a spravovat tyto závislosti pro něj. Správa závislostí je především při instalaci závislostí, dále pak nabízí možnost aktualizaci závislostí a zjištění dostupnosti novější verze. Manažer závislostí umí řešit i závislosti v instalovaných externích knihovnách a řeší tak rekurzivně všechny závislé balíčky a jejich případné konflikty. Balíčky se instaluji do adresáře *vendor* v projektu. Composer vytvoří i tzv. autoloader pro PHP – funkce, která automaticky načítá soubory když jsou vyžádány. Autoloader podporuje řadu standardů pro načítání souborů, jeden z nich je PSR-4 – načítání tříd podle souborové struktury aplikace odpovídající struktuře jmenných prostorů tříd.

2.2 PHP framework Icicle

Icicle je PHP knihovna pro psaní *asynchronního* kódu pomocí *synchronního* stylu zápisu kódu.

Icicle používá generátory a *yield* k zabudování tzv. *awaitables* a koprogramů, které umožňují psát asynchronní kód běžným způsobem zápisu synchronního kódu, jako jsou například návratové hodnoty z funkcí a vyhazování výjimek. Zápisem funkcí tímto způsobem nám umožňuje se zaměřit na význam kódu namísto na předávání zanořených callbacků.

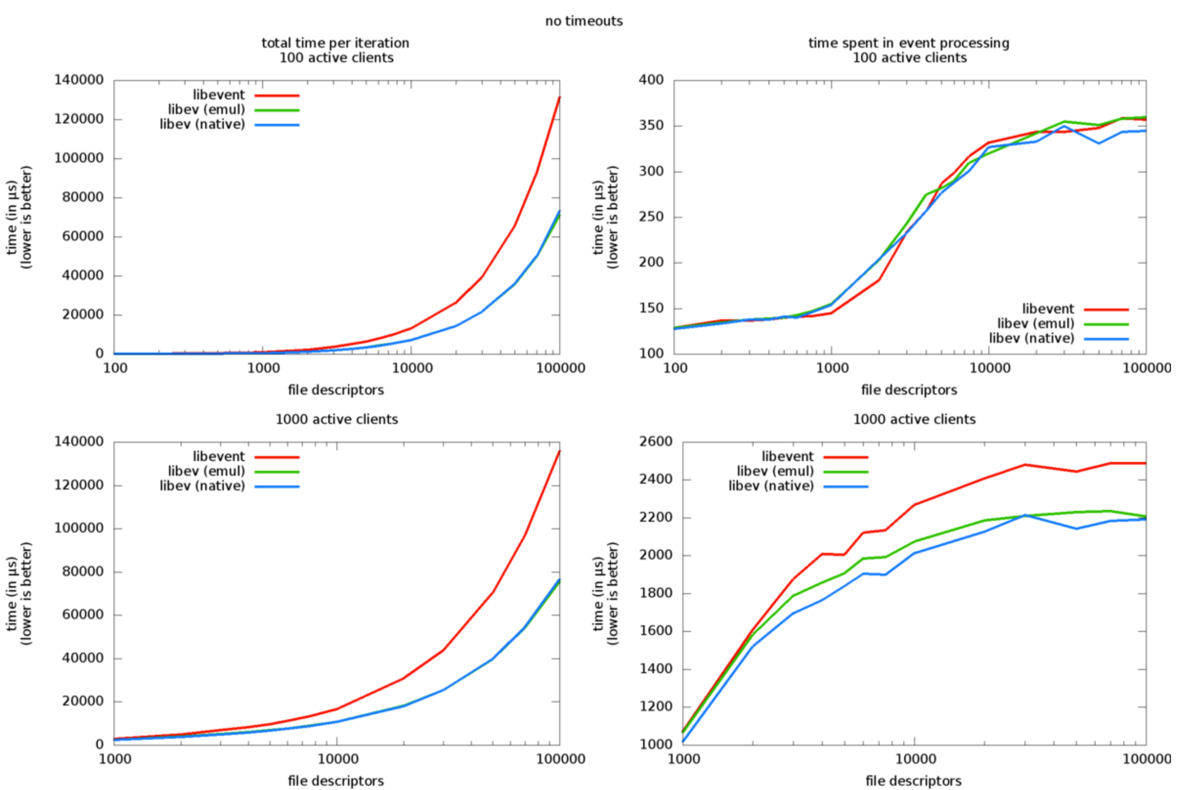
Důvodem proč knihovna Icile jde tímto směrem je ten, že ve většině procedurálních programovacích jazycích znamená zápis asynchronního kódu používání callbacků. Volání funkce pak znamená, že je kromě vstupních parametrů předána i reference na funkci, která se má zavolat po vyhodnocení funkce. Tímto způsobem je možné psát asynchronní kód. Nevýhodou tohoto zápisu je náročnost na pochopení. V komunitě programátorů zabývající se asynchronním psaní kódu (třeba v JavaScriptu) je tento způsob zápisu označován jako „callback hell“.

2.2.1 Event loop

Event loop, nebo-li smyčka událostí (dále jen smyčka) plánuje volání funkcí, běh časovačů, zpracování signálů a aktualizace čtených či zapisovaných dat ze socketů. Existují 3 implementací smyčky, které jsou dostupné pro Framework Icicle (*SelectLoop*, *EvLoop* a *UvLoop*). Nad rámec Icicle smyček existují i smyčky pro ReactPHP (*StreamSelectLoop*, *LibEventLoop*, *LibEvLoop* a *ExtEventLoop*).

Základní implementace *SelectLoop* a *StreamSelectLoop* využívá nativních PHP funkcí *stream_select()* a *usleep()* a je ji tedy možné využít bez problémů. Pro ostatní implementace smyček je nutné nainstalovat rozšiřující moduly PHP.

Základní implementace má dostatečný výkon a není nutné využít jiných smyček. Pokud bychom naznali, že smyčka je právě to, co brzdí výkon, lze ji bez problémů vyměnit. Všechny poskytují stejné rozhraní *LoopInterface*. Porovnání výkonu LibEventLoop a LibEvLoop lze nalézt na [[8]].



Obrázek 3 Porovnání *libevent* a *libev* a emulované *libev* smyčky, převzato z [[8]]

Na obrázku výše je porovnání *libevent* s *libev* a emulovanou *libev*, lze vidět, že *libevent* nepřepíná mezi událostmi tak rychle, jako *libev*. Výrazného rozdílu si lze povšimnout až při 1000 aktivních klientech, kdy přepínání je téměř dvojnásobné. Podrobnější údaje k metodice měření lze nalézt na [[8]].

2.2.2 Awaitables

Icicle implementuje tzv. Awaitables na základě specifikace Promisses/A+. Promisses/A+ přidává navíc rušení promise. Awaitables jsou objekty fungující jako výplň pro budoucí hodnotu asynchronní operace. Probíhající awaitable může být naplněna hodnotou (včetně jiné awaitable, *null*, nebo výjimkou) a nebo odmítnuta jakoukoliv hodnotou (ne-výjimky jsou automaticky zabaleny do výjimky). Jakmile je awaitable rozhodnuta (naplněna nebo odmítnuta) s hodnotou, awaitable již nemůže vstoupit do stavu probíhající případně změnit svou výslednou hodnotu.

2.2.3 Koprogramy

Koprogramy, nebo-li coroutines jsou přerušitelné programy implementované pomocí Generátorů. Generátor používá klíčové slovo *yield* k získání hodnoty z kolekce, tím lze generátor považovat za iterátor. Koprogramy používají *yield* k určení míst, kde může být chod

funkce přerušen. Když koprogram získává další hodnotu, vykonávání koprogramu je dočasně přerušeno a další koprogramy dostávají šanci na běh. Těmito programy můžou být třeba I/O operace, časovače nebo další koprogramy.

2.2.4 Balíček Websocket server

Balíček *icicleio/websocket* slouží k podpoře websocketů ve frameworku icicle. Tento balíček nabízí základní podporu pro připojení přes websockety k serveru pro webové prohlížeče. Balíček implementuje server pomocí nativní podpory *icicle/socket*.

2.2.5 Balíček HTTP Server

Balíček *icicle/http* poskytuje uživateli možnost psaní http klienta i serveru ve frameworku *icicle*. Za účelem implementace REST API jsem zvolil tento balíček, jako ideální možnost k implementaci webserveru.

2.3 Ovladače pro MongoDB

PHP obsahuje dva ovladače *mongo* a *mongodb*. Mongodb je nástupcem staršího ovladače *mongo*. Obě knihovny poskytují předchystané všechny základní třídy pro manipulaci s requesty a s odpověďmi. Bohužel, obě knihovny jsou napsány v blokujícím paradigma a tudíž je není možné použít pro reaktivní asynchronní implementaci monitorovacího nástroje.

Alternativou k těmto ovladačům je knihovna *jmikola/react-mongodb*, [[11]]. Tato implementace pro framework ReactPHP je už od základu neblokující a přichystaná pro asynchronní přístup. Knihovna je experimentální a neobsahuje ani z daleka tolik tříd, jako ovladače *mongo* či *mongodb*. Je však možné s ní vytvářet jakékoliv dotazy do MongoDB. Jedná se o jedinou implementaci knihovny s neblokujícím přístupem pro MongoDB.

Nevýhoda této knihovny neblokující knihovny pro MongoDB je její nedostupnost pro framework Icicle [[11]], neboť je implementována jako knihovna pro ReactPHP. Tento nedostatek lze vyřešit adaptérem pro ReactPHP, kterým se budeme zabývat v další kapitole.

2.3.1 ReactPHP adaptér

Knihovna *jmikola/react-mongodb* je určena pouze pro ReactPHP. Icicle a ReactPHP frameworky jsou na nejnižší úrovni velmi podobné. Oba používají smyčku událostí. Pokud bychom chtěli spojit tyto dva frameworky dohromady, resp. používat balíčky mezi sebou,

bylo by nutné vytvořit adaptér (návrhový vzor), kterým by se tyto frameworky propojili. Autoři Icicle vytvořili adaptér *icicleio/react-adapter*, který usnadňuje toto spojení obou frameworků.

Použitím tohoto adaptéru je možné používat přehlednější způsob zápisu ve stylu synchronního kódu za použití komponent z událostně řízeného ReactPHP frameworku.

2.4 ReactPHP

ReactPHP je událostně řízená, neblokující knihovna v jazyce PHP.

ReactPHP je nízko úrovňová knihovna pro událostně řízené programovaní v jazyce PHP. V jejím jádru je zabudována smyčka událostí a nad tou jsou poskytovány nízko úrovňové nástroje. Například abstrakce pro streamy, asynchronní DNS dekodér, síťový klient/server, HTTP klient/server, interakce s procesy. Externí knihovny pak nabízejí další komponenty na vytváření síťových klientů, serverů a dalších.

Smyčka událostí je založena na tzv. reaktivním vzoru a silně inspirovaná knihovnami jako je EventMachine v Ruby, Twiste v Pythonu a Node.js v JavaScriptu.

ReactPHP podporuje stejně jako Icicle více typů událostních smyček. Jedinou výhodou je lepší výkon – rychlejší přepínání kontextu.

ReactPHP poskytuje smyčku událostí jako samostatný balíček a ten je díky tomu znovupoužitelný v jiných aplikacích.

ReactPHP je od základu neblokující, pokud má uživatel potřebu řešit i blokující úlohy, měl by využít ReactPHP nadstavbu nad klasickými procesy.

ReactPHP poskytuje v základu práci se streamy a práci s Promisy. Promisy využívá také Icicle a bude tento koncept vysvětlen ve vlastním sekci.

Na závěr bych rád podotkl, že proti Icicle tato knihovna nepoužívá PHP Generátory ke svému běhu a tudíž nepoužívá klíčové slovo *yield*.

2.5 Promises/A+

Je otevřený standard pro řádné a spolupráce schopné JavaScriptové promises navržené vývojáři pro vývojáře.

Promise (česky příslib) reprezentuje potenciální výsledek asynchronní operace. Primární způsob práce s promise je přes metodu *then*, která registruje callbacky pro příjem jiné

promise, potenciální výsledek a nebo důvod zamítnutí promise, která nemohla být vyřešena (naplněna).

Promises/A+ je specifikace, která vychází z původní specifikace Promises/A, rozšiřuje ji a zaměřuje se na skutečné způsoby chování a řeší vynechané nebo nespecifikované části původní specifikace.

Závěrem lze říci, že specifikace Promises/A+ neřeší jak se mají vytvářet, naplňovat nebo zamítat promises, ale zaměřuje se na chování a spolupráci metody *then*.

Originální specifikace je uvedena jako Příloha P1 – Promisses/A+

2.6 Postupná integrace - continous integration

Postupná integrace je praxe při vývoji softwaru, kde členové týmu integrují jejich práci velmi často, obvykle každá osoba integruje své změny minimálně denně – což vede k několika integracím za den. Každá integrace je ověřena automatickým sestavením (zahrnujícím testy) za účelem zjištění integrační chyb jak jen to je možné. Hodně týmů shledává, že tento postup vede k významnému redukování integračních problémů a dovoluje týmu vyvíjet kohezivní software rychle. [[12]]

Díky postupné integraci dostává vývojář více času na práci, neboť počítače za něj obstarávají veškerou práci nutnou pro integraci nového zdrojového kódu do projektu.

Mezi základní funkcionalitu integračního nástroje patří:

- sestavení kódu
- spuštění jednotkových testů
- vytvoření kompletního instalačního balíku vhodného pro nasazení
- nasazení do produkce

Mezi rozšířené možnosti integračního nástroje lze zařadit následující:

- spouštění integračních testů
- spouštění akceptačních testů
- kontrola dodržování stylu psaní
- spuštění tzv. mess detektorů – nevhodně strukturované bloky, přílišná komplexita, velká NPath komplexita apod.

2.6.1 Travis CI

Travis CI je základní integrační nástroj poskytující integraci pro jazyk PHP, automatické spouštění unit testů, definici konfigurací, integraci s externími službami (Slack, Email, HipChat), paralelní spouštění testů, CLI a API. Nástroj lze pro open-source projekty používat zdarma.

2.6.2 Scrutinizer CI

Scrutinizer je integrační nástroj pro jazyky PHP, Ruby a Python. Oproti Travis CI je Scrutinizer zaměřený na vylepšování kvality kódu, eliminování chyb, bezpečnostní analýzy. Cenová politika není nakloněná použití zdarma, Scrutinizer nabízí trial verzi na 14 dní zdarma.

2.7 Grafické uživatelské rozhraní

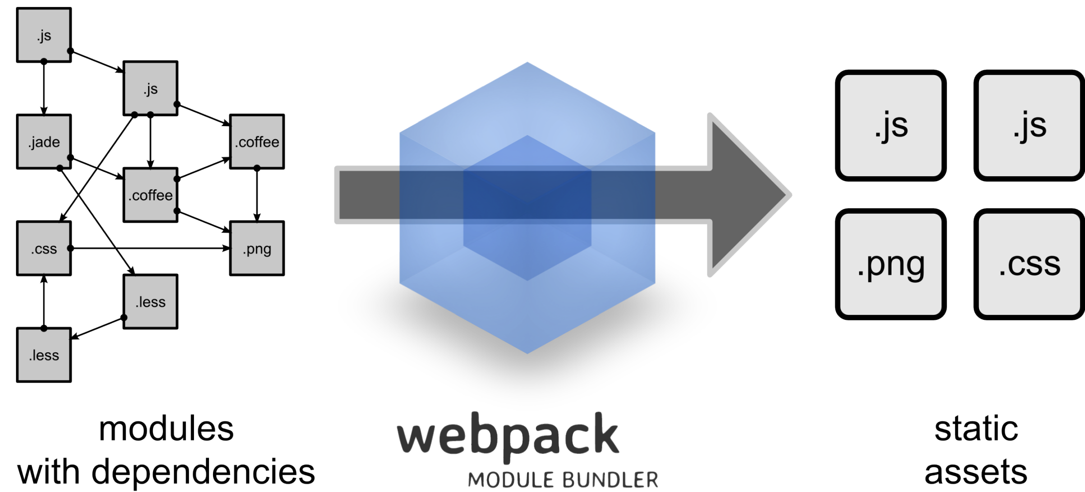
Přistoupením k realizaci monitorovacího nástroje pomocí WebSocketů se ukázalo, že je nutné vytvořit a navrhnout i grafické uživatelské rozhraní podporující tento událostně řízený přístup k monitorování instancí. Původní návrh využití REST API u kterého není třeba implementovat uživatelské rozhraní je tímto nevhodný.

2.7.1 Webpack

Webpack je chytrý nástroj na vytváření balíčku ze všech závislých modulů. Webpack vezme všechny moduly včetně jejich závislostí a vygeneruje statický soubor reprezentující tyto moduly [[12]].

Cíle Webpacku:

- Rozdělení stromu závislostí do shluků podle požadavků
- Zachování nízkého času pro první spuštění
- Každý statický balíček se může stát modulem
- Možnost integrace knihoven a modulů třetích stran
- Možnost upravit téměř jakoukoliv část balíčkovače
- Určeno pro velké projekty



Obrázek 4 Ilustrace práce balíčkovacího systému Webpack. Převzato z [[12]]

2.7.2 Šablonový systém ReactJS

ReactJS je velmi populární Javascriptový balíček vytvořený vývojáři ze společnosti Facebook. V klasické MVC architektuře, tedy Model View Controller je ReactJS pouze ta část „V“, tedy View. ReactJS neřeší Model a Controller, a lze jej tedy snadno nasadit do již existující aplikace. To u ostatních JavaScriptových frameworků není tak snadné a většina z nich se snaží řešit všechny části MVC architektury. Příkladem budiž AngularJS, který řeší kompletní strukturu aplikací [[13]].

ReactJS si vytváří ztv. virtuální DOM, což mu umožňuje jednodušší programování a lepší výkon. ReactJS umí renderovat i na serveru pomocí NodeJS a díky tomu lze v něm vytvářet i nativní aplikace (tzv. React Native).

ReactJS využívá tzv. jednosměrného reaktivního toku dat uvnitř aplikace, což výrazně redukuje práci s tradičním navázáním dat na časti aplikace.

3 POŽADAVKY NA MONITOROVACÍ NÁSTROJ

Účel monitorovacího nástroje je pravidelně monitorovat všechny MongoDB instance, které jsou pro uživatele zajímavé. Uživatel chce vědět dopředu, že se blíží problém s konkrétní instancí a podle toho pak případně zakročit. V tomto by mu měl být nástroj nápomocný a měl by mu usnadnit správu desítek instancí tím, že bude poskytovat aktuální data o všech sledovaných instancích.

Požadavky na monitorovací nástroj:

- Serverová aplikace monitorující MongoDB instance.
- Komunikuje přes webové API
- Na dotaz na API vrátí aktuální stav instancí
- Aplikace periodicky sleduje stav instancí
- Sledované informace:
    - Aktuální obsazený prostor na disku pro jednotlivé kolekce
    - Aktuální vytížení paměti RAM
    - Aktuální síťový tok
    - Velikosti indexů
- Nástroj zná seznam monitorovaných instancí

II. PRAKTICKÁ ČÁST

4 ANALÝZA POŽADAVKŮ NA MONITOROVACÍ NÁSTROJ

Z požadavků na monitorovací nástroj vyplynulo několik základních požadavků na programovou část nástroje, které si probereme v následujících odstavcích. Jedná se především o způsob komunikace s instancemi, konfigurace nástroje a zvolení vhodného komunikačního protokolu API.

4.1 Komunikace s instancemi

V požadavcích je uvedeno, že nástroj musí umět monitorovat několik desítek instancí ve stejný čas. Je tedy zřejmé, že nástroj musí umět komunikovat s těmito instancemi asynchronně a zvládat tak libovolný počet instancí přes libovolně vytíženou síť.

4.2 Konfigurace seznamu instancí

V požadavcích je uvedeno, že tento nástroj musí u sebe držet seznam monitorovaných instancí a monitorovat je. Zaměříme se tedy na seznam instancí. Na řešení seznamu navrhuji využití konfiguračního souboru, který bude obsahovat seznam instancí včetně IP adresy a portu na kterém se nachází zvolená instance.

Alternativní způsob držení seznamu monitorovaných instancí by mohl být speciální API endpoint, kterým by se tento seznam mohl plnit. Toto přináší flexibilitu nástroji a možnost rychlého přidávání instancí, bohužel to však sebou přináší určité bezpečnostní riziko. Pokud API bude omylem uveřejněno, mohlo by být zneužito k monitorování nechtěných instancí a tudíž k narušení spolehlivosti nástroje.

Způsob konfigurace přes konfigurační soubor byl po konzultaci s vedoucím práce a po zvážení bezpečnostních rizik byly návrh uznán za pravdivý a tudíž byl zvolen konfigurační soubor před přidáváním přes API.

4.3 Komunikace přes webové API

V požadavcích není přesně specifikováno, jak by mělo API fungovat a chovat se z vnějšku. Uvažujme tedy dvě možnosti: REST API a WebSockets.

4.3.1 REST API

REST API je aplikační rozhraní, které na každý požadavek odešle odpověď. Toto API je vhodné pro běžné aplikace, které nevyužívají živé data. Pokud však chceme sledovat desítky instancí najednou a pravidelně informovat uživatele, bylo by nutné se dotazovat v pra-

videlných intervalech API a zjišťovat aktuální stav. Nevýhodou tohoto řešení je fakt, že každý požadavek na API vytváří nové TCP spojení a s tím spojený handshake. Další nevýhodou je zbytečnost těchto dotazů v případě málo aktivních instancí MongoDB. Lze částečně redukovat požadavky na TCP spojení využití HTPP Keep-Alive, ale i tak je pořád nutné se pravidelně dotazovat. Taková zátěž zbytečně vytěžuje serveru a tím kladně vyšší požadavky infrastrukturu a server.

4.3.2 WebSockets

Jako alternativní řešení k REST API existuje implementace událostně řízeného komunikačního protokolu známého jako WebSockets.

WebSockets – je pokročilá technologie, která umožňuje otevření alternativního komunikačního rozhraní mezi webovým prohlížečem a serverem. S tímto API je možné poslat zprávy na server a dostat zprávy ve formě událostně řízených odpovědí bez nutnosti dotazování se serveru na odpověď. [[15]] Lze tak snadno implementovat méně náročnou komunikaci na infrastrukturu.

Podpora WebSockets v prohlížečích je variabilní. Všechny majoritní prohlížeče na trhu podporují standard RFC 6455. Mobilní prohlížeče také podporují RFC 6455 s výjimkou je mobilního prohlížeče Internet Explorer. [[15]]

4.3.3 Volba API rozhraní

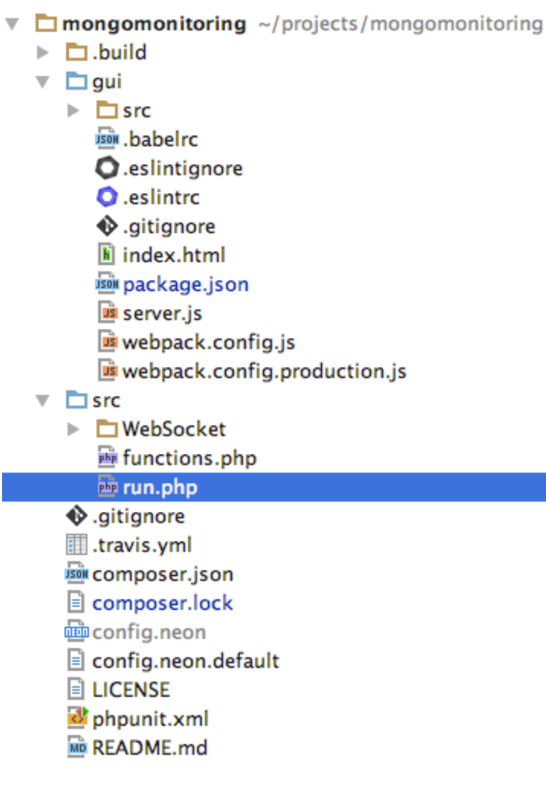
Z výše uvedených řešení bylo vybráno řešení postavené nad WebSocketech. Důvodem výběru je jeho menší náročnost na komunikaci. Redukce zbytečného dotazování serveru na nové informace. Další výhodou je schopnost událostně řízeného komunikačního protokolu. Návrhu komunikačního protokolu se budeme zabývat s kapitole 5.2.5.

5 IMPLEMENTACE

V kapitole implementace se budeme zabývat jednotlivými kroky, které vedly k implementaci monitorovacího nástroje na monitorování více instancí MongoDB a k implementaci grafického rozhraní k monitorovacímu nástroji.

5.1 Souborová struktura

Nástroj i klientská aplikace jsou součástí stejného git repozitáře. Klientská aplikace je v adresáři *gui*, monitorovací nástroj je v adresáři *src*. Git repozitář uplatňuje soubor *.gitignore*, který určuje seznam nezahrnutých souborů do verzovacího nástroje Git (vendor, node_modules, build, config.neon). *Readme.md* je textově strukturovaná nápověda ve formátu Markdown [https://en.wikipedia.org/wiki/Markdown]. Složka *.build* bude popsána v kapitole o integračním nástroji, podobně tak i soubor *.travis.yml*.



Obrázek 5 Struktura aplikace

5.2 Monitoring instancí MongoDB

Z analýzy zadání vyplynulo, že monitorovací nástroj má monitorovat více než jednu instanci MongoDB. Pro náš monitorovací nástroj předpokládejme, že chceme monitorovat cca 30 instancí. K implementaci byl zvolen PHP framework Icicle. Icicle umožňuje pomocí kooperativního multitaskingu přepínat kontext a lze tak pustit více koprogramů v jednom vláknu aplikace. Této vlastnosti je dosaženo využíváním smyčky událostí a generátorů, jak byly popsány v 2.1.1 a 2.2.1.

5.2.1 Smyčka událostí

V aplikaci je využita standardní smyčka událostí *Loop\loop()* z frameworku Icicle, jak lze vidět na následujících řádcích zdrojového kódu.

```
$loop = Loop\loop();
$server = new Server(new Handler(new ReactLoop($loop), $config['hosts']));
$server->listen($port);

$loop->run();
```

Obrázek 6 Zdrojový kód se smyčkou událostí a adaptérem na ReactPHP

Ve zdrojovém kódu umístěném výše si povšimněme třídy new *ReactLoop($loop)*. Zde je ukázán způsob, jak lze kombinovat událostní smyčku z frameworku Icicle s událostní smyčkou nízko úrovňové knihovny ReactPHP. Díky tomuto adaptéru lze využít neblokující knihovnu *react-mongodb* [[11]].

5.2.2 Použití ReactPHP adaptéru

Adaptér na spojení ReactPHP s Icicle používá metody *adapt()*, a vytváří tak vlastní *Awaitable*.

```
$connectionThenable = Awaitable\adapt($this->connectionFactory->create($host, $port, ['connectTi-
meoutMS' => 500, 'socketTimeoutMS' => 500]));
```

Obrázek 7 Zdrojový kód použití napojení Icicle na ReactPHP

Ve zdrojovém kódu uvedeném výše lze vidět, že za použitím metody *adapt()* je vytvořena nová třída *Awaitable*, s kterou lze nakládat stejně jako s ostatními metodami v Icicle.

5.2.3 Získání seznamu databází

V současné chvíli jsme spojeni s konkrétní instancí MongoDB a můžeme pokládat dotazy. Jak již bylo dříve zmíněno, k dotazování se použije *react-mongodb* knihovna. Dotaz do databáze je vytvářen přes třídu Query, kde první argument je namespace zprávy (většinou označuje název databáze a název kolekce oddělený tečkou), druhý argument je dotaz samotný. V ukázce níže je namespace „admin.$cmd“ a dotaz je „listDatabases:1“. Dalším krokem je příkaz *$connection->send()*, který pošle dotaz na instanci. Příkaz je uvnitř dříve popisované metody *adapt()*, která vytvoří z metody *send()* instanci třídy *Awaitable*. Povšimněme si operátoru *yield*, který přerušuje chod aplikace dokud nebude vyřešena *Awaitable* a pokračuje na další příkaz.

```
$listDatabasesQuery = new Query('admin.$cmd', ['listDatabases' => 1], null, 0, 1);
/** @var Reply $reply */
$reply = (yield Awaitable\adapt($connection->send($listDatabasesQuery)));
$listDbs = current(iterator_to_array($reply));
$initResponse = Messages\Init::create($instanceIp, $instanceIp, $listDbs);
$this->log($websocketConnection, $instanceIp, 'getting list of databases');
yield $websocketConnection->send($initResponse);

yield $listDbs;
```

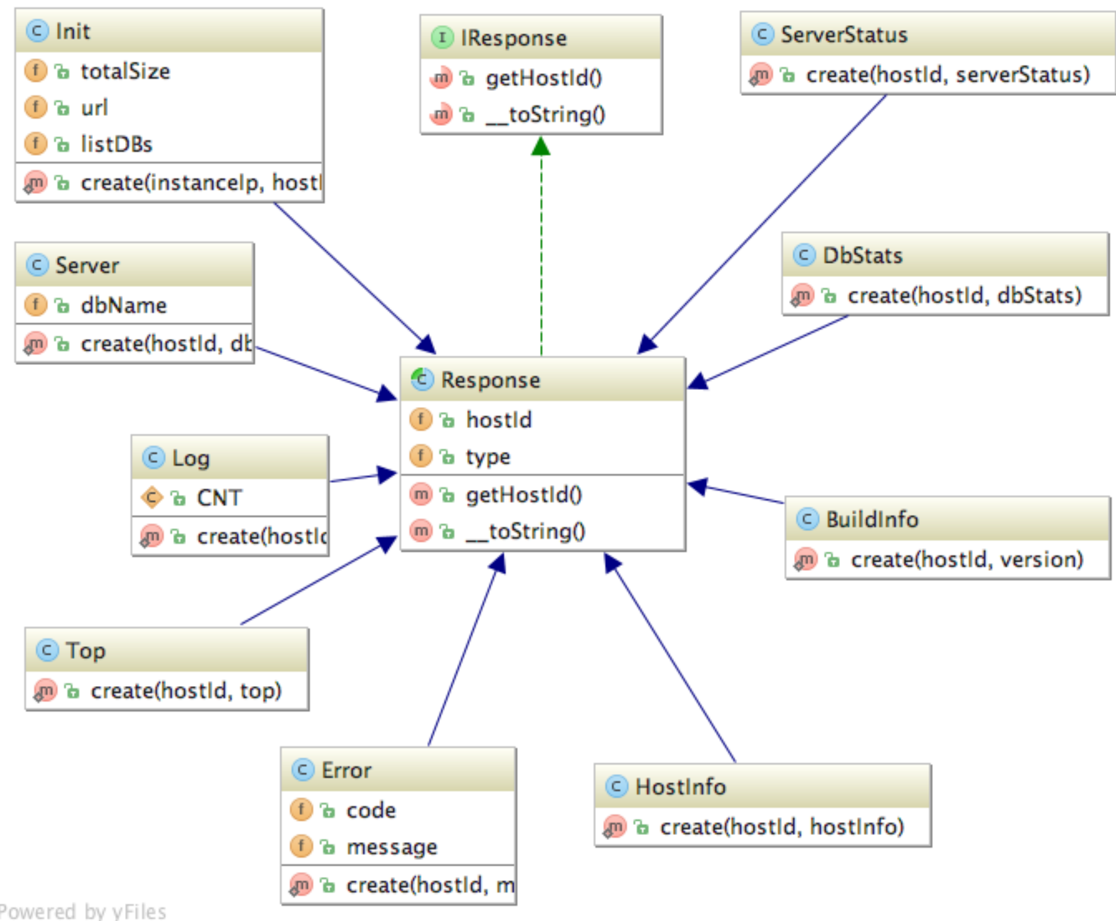
Obrázek 8 Práce s *react-mongodb* knihovnou, poslání dotazu a získání odpovědi

5.2.4 Práce s WebSockety

K implementaci WebSocketů je využita knihovna *icicleio/websockets* [[16]], díky které je práce s WebSockety jednoduchá. Z Obrázek 8 si povšimněme příkazu *$websocketConnection->send();*, který zašle zprávu klientské aplikaci. Tímto způsobem je vytvořena komunikace s klientem, popis struktury komunikačních zpráv je popsána v následující kapitole.

5.2.5 Komunikační protokol

Komunikační protokol aplikace byl zvolen s ohledem na požadavky serveru i klientské aplikace. Klientská aplikace je vytvořena v JavaScriptu a pro aplikaci psanou v tomto jazyce je nejvýhodnější datový formát JSON. MongoDB komunikuje přes REST API, kde výchozím komunikačním protokolem je taktéž JSON. Oběma stranám vyhovuje komunikace v JSONu a tudíž i samotná komunikace přes WebSockety bude ve formátu JSON [[17]].



Obrázek 9 Diagram tříd zpráv komunikačního protokolu [[4]]

Na Obrázek 9 je diagram tříd zpráv komunikačního protokolu. Každá zpráva implementuje rozhraní *IResponse*, která obsahuje dvě metody: *getHostId()* a *__toString()*. Metoda *getHostId()* vrací identifikátor konkrétní instance MongoDB. Tento identifikátor je důležitý pro párování zpráv na klientské aplikaci. Metoda *__toString()* slouží k exportu zprávy na komunikační kanál WebSocketu. Tato metoda vezme všechny atributy dané třídy a uloží je do formátu JSON. Pro zjednodušení a redukce duplicitního kódu existuje i abstraktní třída *Response* implementující *IResponse*. Každá z tříd dědících z abstraktní třídy Response má vlastní faktory metodu *create()*, která vytvoří novou instanci s potřebnými atributy.

5.3 Implementace konfiguračního souboru

Konfigurační soubor aplikace byl zvolen typ souboru Neon. Jedná se o soubor s rozšířenou syntaxí YAML formátu. Pro podporu formátu Neon byl využit balíček *nette/neon* [[18]], který poskytuje metodu *Neon::decode()* pro načtení vstupního konfiguračního souboru do PHP proměnné *$config*. Pro aplikování konfigurace je třeba mít v projektu vytvořený sou-

bor *config.neon*, který není součástí repositáře. Jako vzorový lze využít *config.neon.default*.

```
server:
    port: 9900

hosts:
    - 127.0.0.1:27017
```

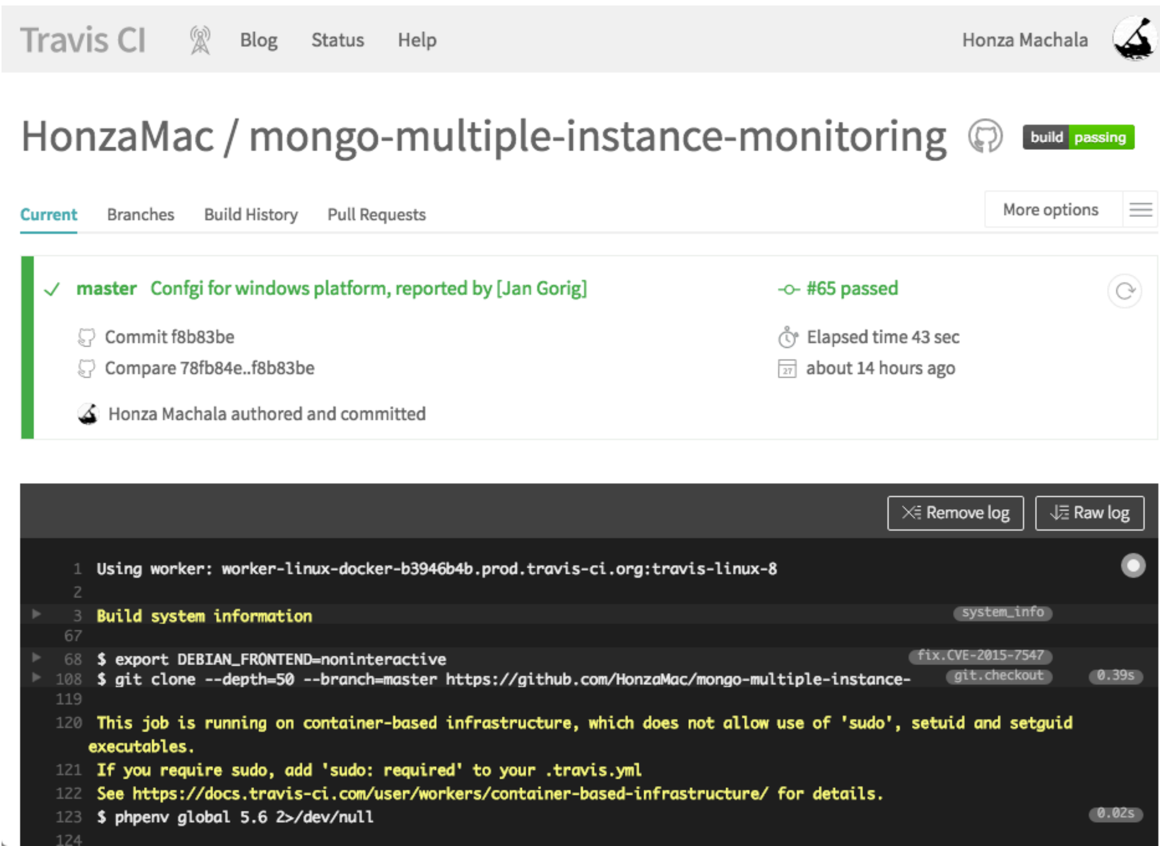
Obrázek 10 Ukázková struktura konfiguračního souboru *config.neon.default*

5.4 Integrační nástroje

Za účelem rychlejší integrace kódu a hlídání inkrementálních změn byly využity integrační nástroje *Travis CI* a *Scrutinizer CI*.

5.4.1 Travis CI

Na obrázku níže lze vidět nasazení integračního nástroje na aplikaci. Na obrázku lze vidět aktuální stav aplikace. Vidíme zde úspěšné sestavení na branchi *master*, čas běhu byl *43 sekund* a změnu provedl *Honza Machala*.



Obrázek 11 Uživatelské rozhraní integračního nástroje *Travis CI*

Pro využívání integračního nástroje *Travis CI* je nutné mít v kořenu projektu soubor *.travis.yml*. Soubor je ve formátu YAML, následuje ukázka jeho obsahu:

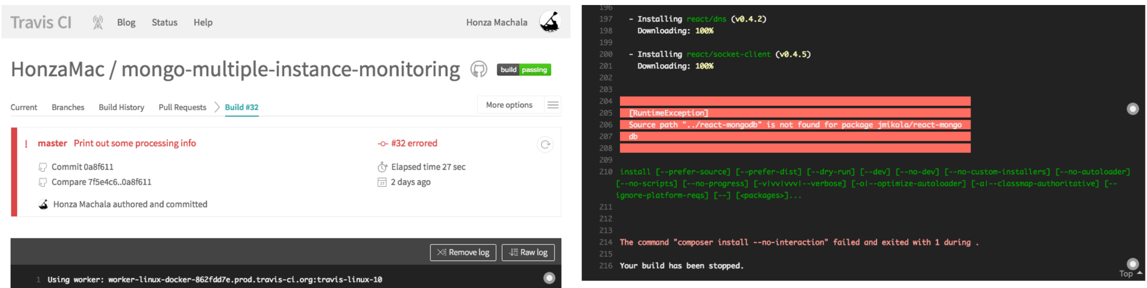
```
language: php
php:
  - '5.6'

install:
  - composer install --no-interaction

before_install: phpenv config-add .build/php.ini

script:
  - vendor/bin/phpunit
```

Tabulka 5-1 Konfigurační soubor *.travis.yml*

Soubor obsahuje 5 sekcí. První sekce *language* určuje který jazyk bude pro sestavení používán. V další sekci *php*, je již pak seznam podporovaných verzí zvoleného jazyka. Sekce *install* určuje kroky nutné ke kompletní instalaci aplikace. Sekce *before_install* je využívána k doplnění potřebných konfiguračních direktiv před spuštěním instalace. Většinou do této sekce patří seznam modulů na kterých aplikace závisí. V tomto případě je zde odkaz

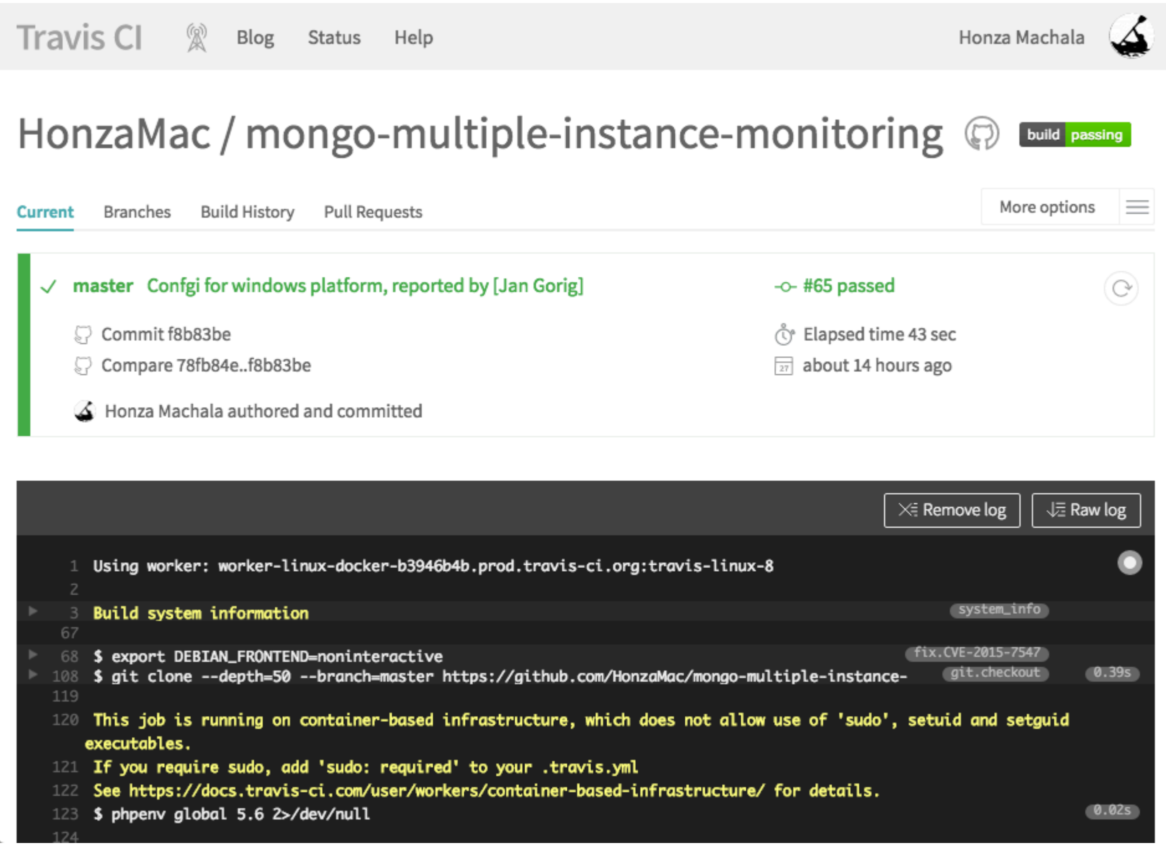
na soubor *.build/php.ini*, který obsahuje nutné konfigurační direktivy pro běh aplikace. Soubor *php.ini* obsahuje pouze „extension: mongo.so“, závislost na PHP modulu *mongo*.

V uživatelském rozhraní lze zobrazit historii jednotlivých integrací a z logů lze poznat důvod selhání integrace. Ukázka selhání integrace je na následujícím obrázku.

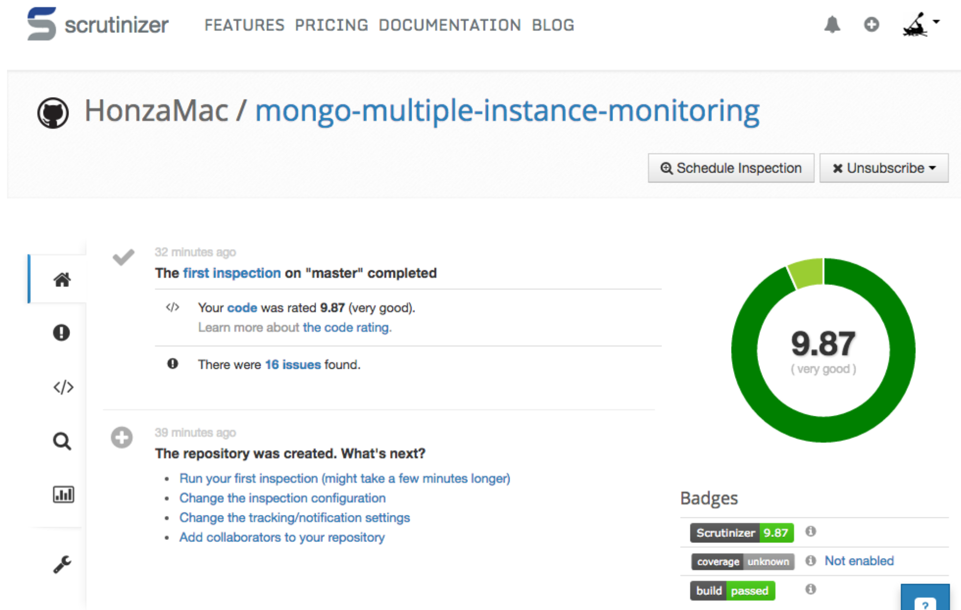


Obrázek 12 Selhání integračního nástroje při instalaci závislosti na balíčku

V případě selhání integračního nástroje je autor upozorněn emailem, případně může být upozorněn do jakéhokoliv externího nástroje (Slack, HipChat).



Obrázek 13 Uživatelské rozhraní integračního nástroje *TravisCI*



5.4.2 Scrutinizer CI

Integrační nástroj *Scrutinizer CI* hlídá kvalitu kódu, komplexitu kódu, potenciální programátorské chyby, coding style a bezpečnost. Na následujícím obrázku lze vidět, že kvalita aplikace je 9.8. Z uživatelského rozhraní lze vidět 18 problémů, které by bylo dobré vyřešit.

Obrázek 14 Nástěnka *Scrutinizer CI* po první integraci

Rating	Name	Duplication	Size	Complexity	Changes
B	MongoApplication::onConnection()	0	15	5	8
A	MongoApplication::fetchServerStatus()	15	10	2	5
A	WebSocket\MongoApplication::fetch()	0	10	1	3
A	WebSocket\Messages\Server::create()	0	8	1	1
A	MongoApplication::fetchListDatabase()	0	8	1	1
A	WebSocket\Messages\ServerStatus::create()	8	6	1	1
A	MongoApplication::connectToHost()	0	7	1	2

Obrázek 15 Výsledek testu komplexnosti metod

Na obrázku výše lze vidět komplexnost jednotlivých metod v aplikaci a jejich hodnocení. Metrika *duplication* označuje míru duplikace kódu – je dobré aby se tato metrika snižova-

la. Metrika *Size*, udává počet řádků metody. Metrika *Complexity* udává rozvětvení řádků kódu – typicky podmínka *If*, *else*, případně *switch* a *throw*. A poslední metrika *Changes* udává, kolikrát byla daná třída upravena.

5.5 Grafické uživatelské rozhraní

Grafické uživatelské rozhraní bylo naimplementováno jako SPA – nebo-li jednostránková webová aplikace pomocí základních jazyků HTML, CSS a JavaScript.

5.5.1 Lokální vývoj klientské aplikace

NodeJS je JavaScriptové jádro (V8) portované z webového prohlížeče na server a upravené pro běh na serveru. Pro klientskou aplikaci je využíván jako běhové prostředí pro lokální vývoj.

Na správu balíčků je využíván NPM - balíčkovací systém pro JavaScript. Definice pro balíčkovací systém je v soubor *gui/package.json*. V souboru jsou uvedeny závislosti ve dvou sekcích. První sekce je *dependencies*, sekce pro balíčky nutné do produkčního prostředí – balíčky pro Webpack. Druhá sekce *devDependencies* definuje seznam balíčku pro vývojové prostředí.

Babel

Důležitým balíčkem pro vývoj je kolekce balíčků Babel [[19]]. Babel je speciální kompilátor, který překládá zdrojový kód JavaScriptu nové generace do JavaScriptu současné generace podporované v prohlížečích. Tím je možné docílit nových konstrukcí a nových vlastností JavaScriptu při zachování zpětné kompatibility. Zde je seznam nejzásadnějších vylepšení, které Babel přidává do JavaScriptu:

- Arrow funkce
- Asynchronní funkce
- Generátory
- Syntaktické vylepšení, čárka jako sirotek
- Šablonové řetězce

6 POPIS TECHNOLOGIÍ POUŽITÝCH K NASAZENÍ

V předchozí kapitole jsme se zabývali implementací monitorovacího nástroje. V této kapitole se budeme zabývat použitými technologiemi pro případovou studii nasazení, která je v kapitole [7].

6.1 Amazon CloudFront

Amazon CloudFront je webová služba, která urychluje distribuci statického a dynamického webového obsahu jako jsou soubory *.html*, *.css*, *.php*, obrázky a mediální soubory koncovému uživateli. CloudFront doručuje tento obsah po celém světě. Základní vlastností CloudFrontu je jeho chytré směrování uživatelského obsahu. Pokud si uživatel v Brazílii vyžádá soubor, bude mu doručen z nejbližšího dostupného serveru (USA) namísto stahování obsahu ze serverů v Evropě. Obsah pro CloudFront je uložen na uložišti S3.

6.2 S3

Amazon Simple Storage Service (S3) je úložiště data pro internet. Na S3 je možné ukládat a načítat jakékoliv množství data kdykoliv a odkudkoliv z webu. Ke správě S3 slouží jednoduché uživatelské rozhraní, které uživatele provede veškerými možnostmi na staveními. Jako většina služeb od AWS, tak i S3 poskytuje ke svému běhu API rozhraní a CLI nástroj pro automatickou správu. Lze tak zařadit automatické nahrání nově integrované části kódu přímo do produkce pouhým voláním na konci build procesu.

6.3 EC2

Amazon Elastic Compute Cloud (EC2) je webová služba poskytující libovolně měnitelnou výpočetní sílu v cloudu. Služba je navržena k usnadnění škálování webů.

EC2 nabízí od malých instancí (0.5 GiB RAM, 1 CPU) až po velmi rychlé instance (8 GiB, 2 CPU) pro třídu T2. M4 pak nabízí od 2 CPU a 8 GiB RAM po 40 CPU a 160 GiB. AWS nabízí i další třídy, které vždy vynikají v konkrétní vlastnosti, CPU, paměť, SSD disk, síťová propustnost.

6.4 Docker

Docker je „jednoduché a tenké virtualizační prostředí pro přenosné aplikace“. Docker zaštiťuje firma Docker.inc. Docker je v podstatě aplikace rozšiřující základní koncept kontejnerů a přidává k ní snadnější způsob správy, možnost jednoduchého a rychlého nasazova-

cího procesu – díky zabalení aplikací do malých funkčních bloků (image), které můžou běžet kdekoliv. Zároveň tím odpadají problémy s rozdílným hardwarem a rozdílným běhovým prostředím operačního systému. Použitím dockeru je možné na libovolném počítači provozovat aplikaci, která je právě nasazená do produkčního prostředí.

Docker se skládá z těchto tří základních částí:

1. Démon s komunikačním rozhraním typu RESTful
2. CLI rozhraní s možností hledání a stahování vytvořených obrazů z registru
3. Podpora pro veřejné a privátní registry s obrazy

6.5 Docker Cloud

Docker Cloud je služba na správu a nasazení docker kontejnerů. Docker Cloud je služba pro uživatele Dockerů, kterým dává možnost správy celého spektra aplikací od jednoho kontejneru až po distribuované aplikace založené na microservisové architektuře. Docker Cloud neposkytuje serverové řešení pro běh kontejnerů, ale pouze jednoduché grafické rozhraní pro správu a nasazení. Kontejnery lze provozovat u několika různých poskytovatelů serverového řešení jako jsou např.: AWS, DigitalOcean, Microsoft Azure, SoftLayer a Packet.

7 NASAZENÍ MONITOROVACÍHO NÁSTROJE

V předchozí kapitole jsme se zabývali technologiemi, které využijeme v případové studii nasazení monitorovacího nástroje do produkce. V této kapitole se budeme zabývat případovou studií nasazení monitorovacího nástroje a grafické uživatelského rozhraní do produkčního prostředí. K nasazení použijeme technologii Docker kontejnerů, Docker Cloudu a Amazon Web Services.

7.1 Případová studie nasazení

Případová studie nasazení monitorovacího nástroje si klade za cíl zvolení vhodných kroků pro nasazená monitorovacího nástroje a klientského rozhraní do produkčního prostředí. Cílem je ověřit funkčnost a schopnost běhu v produkčním prostředí.

Moderní SPA aplikace vyžadují ke svému chodu pouze několik málo souborů. V této aplikaci se jedná pouze o soubor *index.html* a *app.js*. Ostatní funkcionalita je schovaná v API a tudíž není vyžadován další soubor.

7.2 Postup nasazení SPA do produkce

Amazon Web Services (AWS) nabízí několik zajímavých webových služeb, které lze využít pro snadné nasazení SPA do produkce. Níže jsou uvedeny jednotlivé kroky vedoucí k nasazení grafického rozhraní monitorovacího nástroje do produkčního prostředí.

Prvním krokem k nasazení je vytvoření balíku JavaScriptů *app.js*, to lze provést příkazem *npm run build*. Po dokončení příkazu je výsledný app.js uložen do adresáře *gui/dist/*.

Navštívíme adresu https://console.aws.amazon.com/s3 a vytvoříme tzv. bucket s názvem *monitoring-gui* v regionu *Ireland*. Do tohoto bucketu nahrajeme přes uživatelské rozhraní *index.html* a *app.js*. U obou souborů nastavíme v sekci permissions soubory jako veřejně dostupné. Tím je nastavené na S3 hotové.

Dalším krokem je vytvoření distribučního kanálu na službě CloudFront s adresou https://console.aws.amazon.com/cloudfront. To lze snadno provést přes tlačítko *Create distribution*, následně pak vybereme doručovací metodu WEB, vybereme jako zdroj dat vytvořený S3 bucket z předchozího kroku, vybereme další volitelné atributy, zvolíme Default Root Object *index.html* a potvrdíme tlačítkem *Create Distribution*. Po cca 15 minutách je obsah dostupný z celého světa na této adrese: d1f0meeslamq94.cloudfront.net. Ná-

zev url nás nemusí znepokojovat, neboť ji lze posléze nastavit DNS alias ve službě AWS Route 53.

7.3 Postup nasazení API do produkce

Pro nasazení aplikace do produkčního prostředí je nutné, aby webhosting měl plnou podporu pro API. Následující požadavky na webhosting:

1. Možnost volby portu pro naslouchání
2. Použití vlastního HTTP Serveru
3. Trvalé spuštění procesu

Všechny výše zmíněné požadavky vycházejí z charakteru frameworku Icicle, který mění standardní způsob používání klasické webové aplikace. Díky tomu je náročné najít vhodné prostředí webhostingu pro běh.

Jedním z řešení je zabalení aplikace do kontejneru. Tím lze vytvořit samostatně spustitelnou aplikaci vyžadující pro běh pouze prostředí Dockeru. Zabalením vyřešíme všechny výše zmíněné požadavky na poskytovatele webhostingu.

Výběrem kontejnerové architektury již nemůžeme využít klasického webohostingového řešení a musíme využití řešení pro správu kontejnerů. Pro případovou studii bylo vybráno řešení Docker Cloud od společnosti Docker Inc.

7.3.1 Vytvoření Docker kontejneru

Pro vytvoření Docker kontejneru musíme vytvořit tzv. Dockerfile. Dockerfile je speciální soubor obsahující instrukce pro vytvoření kontejneru. Na obrázku níže je uvedena konkrétní definice Dockerfile. Dockerfile je složen z několika základních direktiv:

- FROM – určuje výchozí obraz
- RUN – instrukce spouštějící jakýkoliv příkaz
- ADD – přidá soubor či složku do nového image
- COPY – provede kopírování souboru do nového image
- WORKDIR – instrukce měnící aktuální pracovní adresář
- EXPOSE – instrukce určující seznam portů přístupných z vně kontejneru
- ENTRYPOINT – instrukce určující příkaz volaný při spuštění docker kontejneru

Z níže uvedené definice Dockerfile vyplívá, že bude vytvořen kontejner založená na image *bileto/php-micro-service* obsahující balíček mongo a přednastavený php.ini, dále pak na-

kopíruje základní konfigurační soubor z výchozího a nainstaluje všechny závislosti. Nakonec nastaví port 9900 přístupný z vně kontejneru a nastaví *run.php* jako vstupní příkaz při spuštění kontejneru.

```
FROM bileto/php-micro-service

RUN pecl install mongo \
  && echo "extension=mongo.so" > /etc/php5/mods-available/mongo.ini \
  && php5enmod mongo \
  && sed -i 's/variables_order = .*/variables_order = "EGPCS"/' /etc/php5/cli/php.ini \
  && sed -i 's/safe_mode_allowed_env_vars = .*/safe_mode_allowed_env_vars = ""/' 
/etc/php5/cli/php.ini \
  && sed -i 's/;date\.timezone =.*/date.timezone = "UTC"/' /etc/php5/cli/php.ini \
  && rm -rf /tmp/*


ADD src /srv/src
COPY config.neon.default /srv/config.neon
ADD composer.json /srv/composer.json
ADD composer.lock /srv/composer.lock

RUN  php -r "readfile('https://getcomposer.org/installer');" | php -- --install-dir=/bin --
filename=composer \
  && cd /srv/ && composer install --no-interaction


WORKDIR /srv/src
EXPOSE 9900
ENTRYPOINT php run.php
```

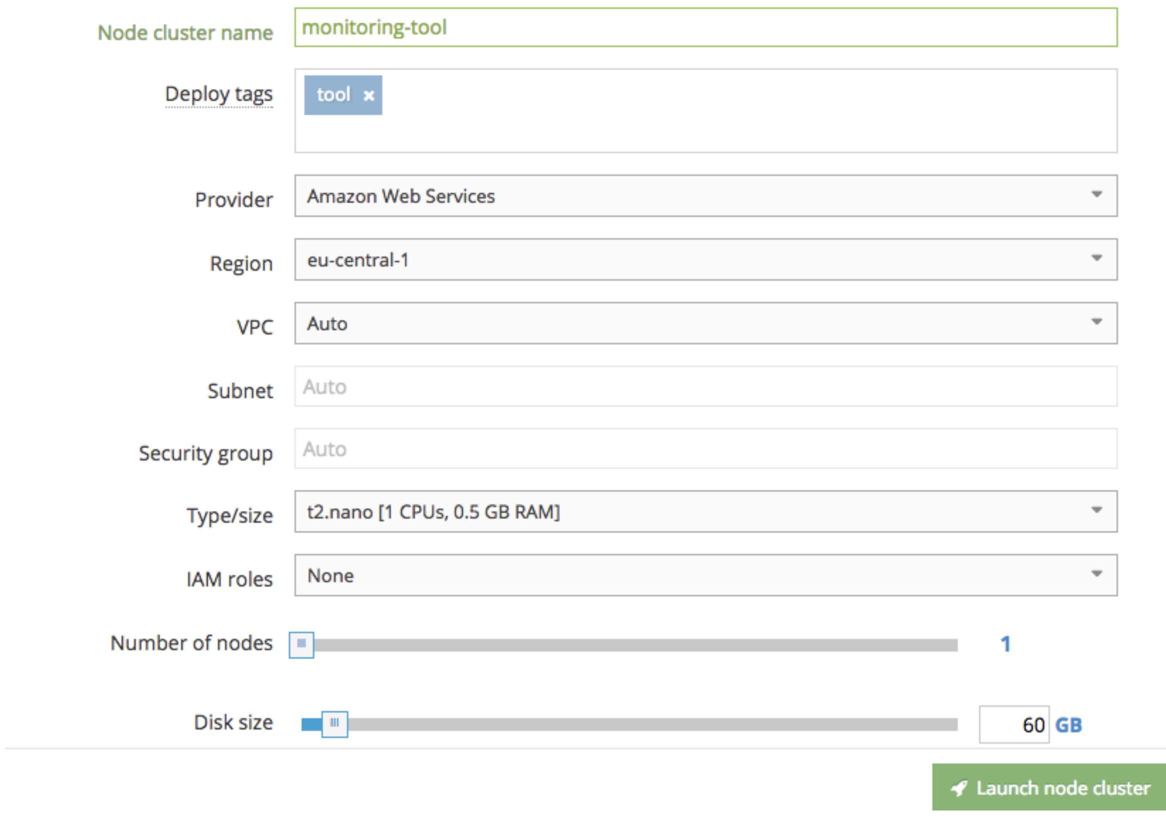
Obrázek 16 Dockerfile použitý k vytvoření Docker kontejneru

Za účelem vyšší flexibility docker kontejneru a jeho větší univerzality byl *config.neon* nastaven jako konfigurovatelný z ENV proměnné „HOSTS“. Touto úpravou je možné mít desítky běžících kontejnerů vždy monitorujících jinou sadu instancí MongoDB. Důvodem proč přibyla konfigurace přes ENV prostředí je fakt, že po vytvoření docker image již není možné upravovat soubory uvnitř – image je immutabilní a tudíž jej nelze upravit, jde však předávat libovolné ENV proměnné a tím jej konfigurovat.

7.3.2 Nastavení Docker Cloud

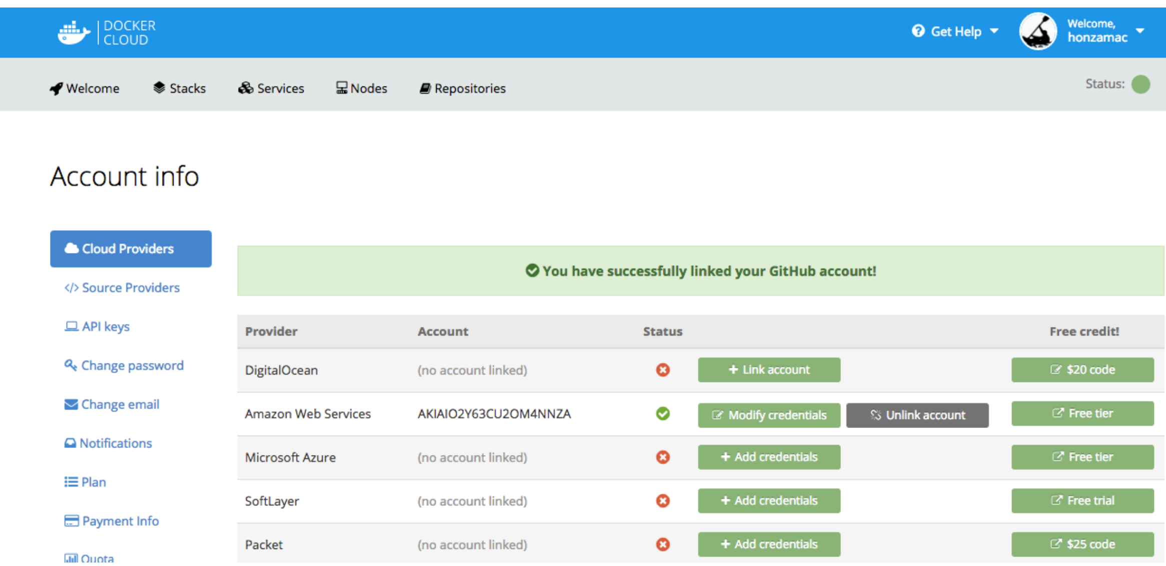
Začneme registrací do Docker Cloud na adrese http://cloud.docker.com. Po registraci vstoupíme do nastavení účtu a provedeme přidání poskytovatele serverového řešení AWS, jak ilustruje obrázek níže.

Nyní je třeba vytvořit AWS EC2 instanci, která bude sloužit pro spouštění kontejnerů. V záložce Nodes zvolíme *Launch new node cluster*, a vyplníme údaje podle obrázku níže.



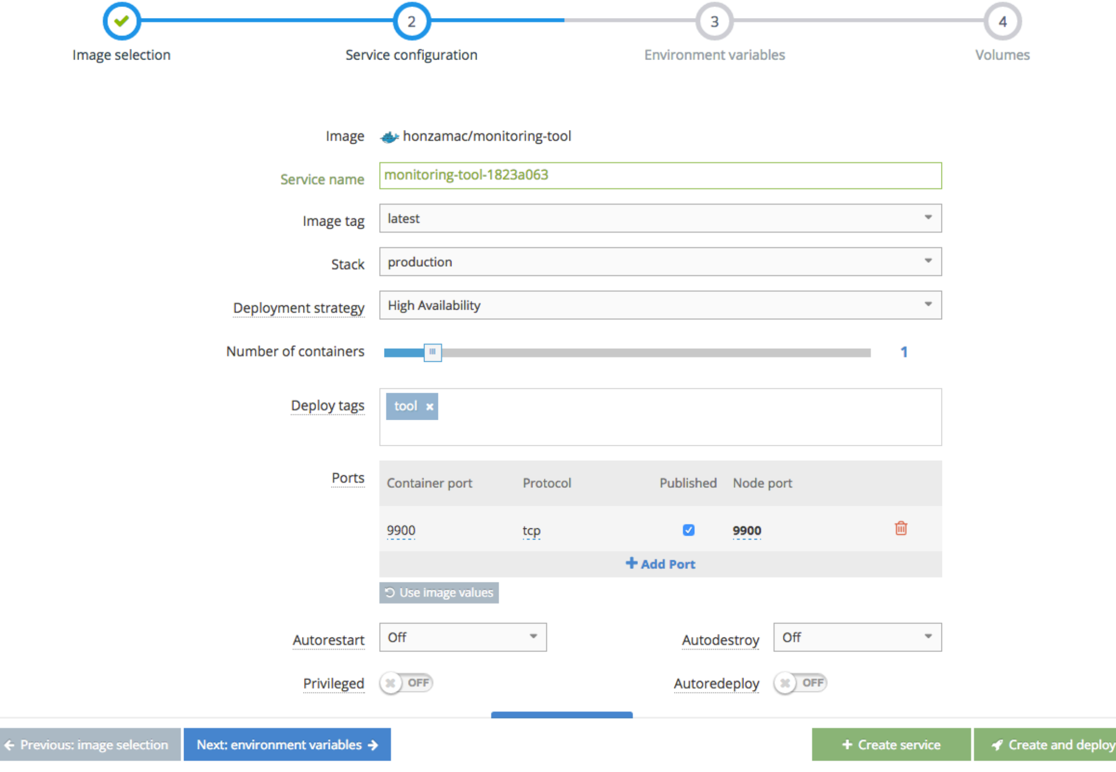
Obrázek 17 Vytvoření nové EC2 instance pomocí Docker Cloudu.

Dalším krokem je vytvoření registru pro docker image. Zvolíme tedy záložku *Repositories* a vytvoříme propojený repositář *honzamac/monitoring-tool* propojený s github repositářem *HonzaMac/mongo-multiple-instance-monitoring*. Po nastavení bude jakýkoliv commit do větve master na githubu automaticky spuštěno vytvoření kontejneru dle dostupného Dockerfile. Úspěšně vytvořený kontejner je pak uložen do registru s tagem :latest.



Obrázek 18 Výběr poskytovatele serverů ve službě Docker Cloud

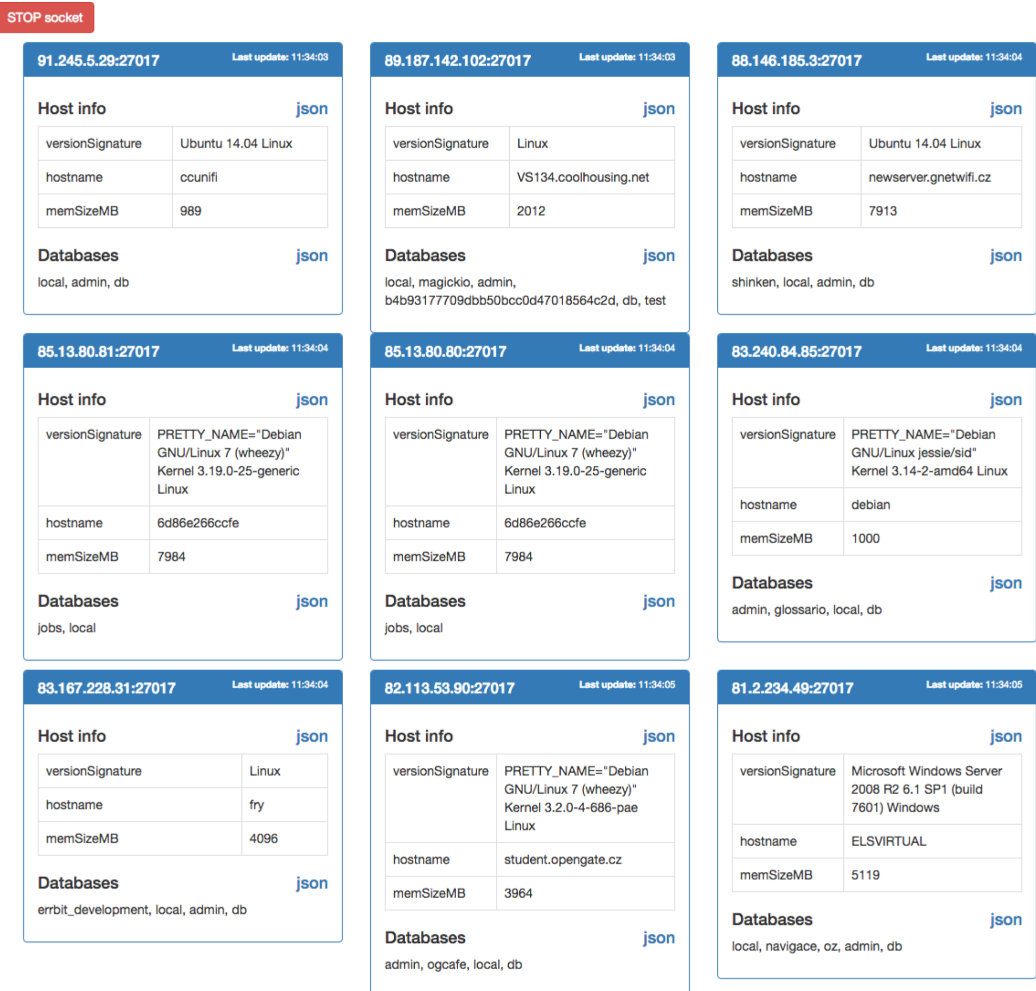
Po prvním úspěšném vytvoření kontejneru pak lze stisknutím tlačítka *Launch* spustit průvodce pro první spuštění kontejneru. V následujícím kroku pak vybere parametry dle obrázku níže a necháme vytvořit a spustit kontejner. Po několika minutách bude vytvořen Stack *Production* s jedním běžícím kontejnerem. Otevřením nově vytvořeného kontejneru pak v záložce endpoints uvidíme url adresu pro přístup k API (monitoring-tool-051d819c.production.62b68e85.svc.dockerapp.io:9900/).



Obrázek 19 Vytvoření kontejneru z registru ve službě Docker Cloud

7.4 Běžící aplikace

Po úspěšném provedení předchozích kroků byla úspěšně nasazena aplikace do provozu. Níže je obrázek ukazující běh klientské aplikace s daty poskytovanými z API. Aplikace je dostupná na adrese: http://bit.ly/23Sq9it



Obrázek 20 Běžící SPA aplikace napojená na API

Na obrázku vidíme 9 instancí MongoDB a základní údaje o každé z nich. V záhlaví je vidět IP adresa instance a informace a čas poslední aktualizaci z API. V sekci *Host info* jsou údaje o instanci, zde je uvedena velikost RAM, hostname a popis operačního systému. Pro podrobnější informace lze využít tlačítko *json*, které zobrazí surové data. V další sekci je seznam databází.

Kliknutím na hlavičku zobrazíme podrobnější informace o instanci, jak lze vidět na obrázku v příloze P2. Detail zobrazení poskytuje podrobnější údaje o každé databázi v rámci instance. Zobrazuje počet kolekcí, obsazený prostor a velikost indexů. Toto je pouze výňa-

tek z údajů které posílá API. Podrobnější informace jde nalézt v dokumentaci, případně v sekci analýzy Monitorovacích nástrojů.

7.5 Vyhodnocení případové studie nasazení monitorovacího nástroje

Cílem případové studie bylo ověření, zda je možné nasadit monitorovací nástroj do produkčního prostředí. Případová studie krok za krokem ukazuje způsob, jakým lze nasadit aplikaci a spravovat aplikaci za pomocí webových služeb AWS a Docker Cloud. Podobný způsob nasazování do produkce používá firma Bileto Technologies s.r.o. pro nasazení API a SPA.

Zvolená cesta vyhovuje této specifické aplikaci a je možné postup zopakovat i u konkurenčních služeb jako je AWS Container Services, Kubernetes, Shippable a jiné.

Zvolené řešení je minimální a lze jej vylepšit. V následujících odstavcích je uvedeno, jak lze process vylepšit.

Problémovým místem je neautomatické nasazení SPA a tedy ideální je automatizovat komunikaci s webovými službami S3 a CloudFront. Lze tak učinit přidáním příkaz přímo do integračního nástroje Travis CI.

```
aws s3 sync --acl "public-read" --delete dist/ s3://$AWS_S3_BUCKET/monitoring-tool
```

Obrázek 21 Příkaz pro nahrání nové verze SPA do S3

Pokud by se změnil účel aplikace a nástroj by byl dostupný pro velké množství uživatelů, bylo by nutné tuto aplikaci horizontálně i vertikálně škálovat. K vertikálnímu škálování by bylo vhodné umístit před API proxy server a tím rovnoměrně rozvrstvit zátěž mezi API servisy. Horizontálně škálovat pak lze pomocí Docker Cloud pouhým spuštěním více kontejnerů.

V této fázi by již bylo vhodné rozdělit aplikace na vývojové a produkční prostředí a tím znemožnit nasazení nekvalitní verze aplikace.

8 ZABEZPEČENÍ APLIKACE

V předchozí kapitole jsme se zabývali nasazením monitorovacího nástroje a nasazením grafického nástroje do produkčního prostředí. V této kapitole se budeme zabývat zjištěnými bezpečnostními riziky a případným řešením.

Monitorovací nástroj byl od začátku navrhován tak, aby jej nebylo možné zneužít k ovládání instancí MongoDB a případně k obdobné formě útoku.

8.1 Zabezpečení konfigurace sledovaných instancí

Nástroj umožňuje přidávat další instance MongoDB pouze pomocí konfiguračního souboru a ENV proměnné definované při spuštění kontejneru. Díky to lze vyloučit možnost modifikace tohoto seznamu sledovaných instancí. Pokud bychom chtěli změnit tento seznam, museli bychom mít:

1) přístup do správy Docker Cloud a zde modifikovat ENV proměnnou HOSTS a
2) přístup ke konfiguračnímu souboru a ten modifikovat

Add 1) zde je slabé místo, neboť pro přístup do administrace stačí znalost kombinace uživatelského jména a hesla. Docker Cloud neposkytuje 2FA.

Add 2) přístup ke konfiguračnímu souboru je velmi náročný. Soubor je součástí Docker kontejneru a jediný způsob přístupu je přes hostitelský server (server na kterém běží Docker kontejner). Hostitelský server je v našem případě AWS EC2 instance. Pro přímý přístup na tyto instance je nutné mít povolený přístup přes *ssh*, a ten lze získat pouze vytvořením IAM uživatele v AWS.

8.2 Zabezpečení WebSocketů

SPA aplikace se napojuje na API WebSocket dostupný veřejně a komunikuje tak se serverem. Tato komunikace není nijak šifrovaná a je zde riziko odposlouchání či případného ovládání serveru. Serverová část nástroje využívá WebSockety pouze k navázání spojení a komunikaci, ta probíhá pouze jedním směrem a to od serveru ke klientovi. Server neumí reagovat na jakékoliv pokyny přes komunikační kanál a tudíž nelze spustit jakýkoliv příkaz na straně serveru.

Jedním z řešení by bylo využití šifrované SSL komunikace přes protokol *wss* (WebSocket-Secured) obdobně jako u protokolu *http* a zabezpečeně přes *https*.

8.3 Bezpečnost komunikace s instancemi

Komunikace s instancemi probíhá přes klasické nezabezpečené TCP spojení. Je to forma komunikace, kterou nabízejí vybrané instance MongoDB a tudíž z pohledu provozovatele instance to zřejmě není bezpečnostní riziko. Pokud by to byl z jejich pohledu problém, určitě by instance nebyla veřejně přístupná z prostředí internetu. Pro monitorování instancí byly vybrány veřejně přístupné MongoDB instance ze seznamu na schodan.io viz. [[20]].

8.4 SSL

Nasazená aplikace nekomunikuje přes zabezpečenou síťovou vrstvu. Pro zabezpečení komunikace a znemožnění odposlouchávání lze aplikaci umístit na HTTPS protokol a nastavit tak SSL s certifikátem pro komunikaci. AWS ve své službě CloudfFront nabízí HTTPS automaticky pro všechny hostované služby. Bohužel Docker Cloud nenabízí SSL automaticky a není tedy možné přepnout celý nástroj na SSL. Určitě však lze tento prolém vyřešit pomocí nástroje AWS Route 53.

ZÁVĚR

Cílem této diplomové práce bylo představení způsobů možností monitoringu instancí MongoDB a vytvoření nástroje na monitorování většího množství instancí.

V počátku teoretické části byly rozebrány aktuálně dostupné způsoby a možnosti sledování MongoDB a nástroje na zobrazení aktuálních. Mezi dostupné možnosti byl nalezen způsob jak získávat aktuální stav obsazení prostoru na disku, aktuální vytížení databáze, výpis logů, získání seznamu databází a získání detailních údajů o těchto databázích. V druhé části jsou popsány použité technologie, mezi důležité se řadí především Framework Icicle, ReactPHP, Promises, WebSockety, MongoDB ovladač, Postupná integrace a ReactJS s Webpackem.

Hlavním účelem praktické části pak byla samotná implementace monitorovacího nástroje na více instancí MongoDB. V počátku se práce zabývá analýzou požadavků na nástroj a volbou komunikačního protokoli. V další části je pak popsána samotná implementace pomoci frameworku Icicle a napojení na nízkoúrovňovou knihovnu ReactPHP. Další kapitola se pak zabývá nasazením a využíváním Integračních nástroje Travis CI a Scrutinizer CI. V poslední části před případovou studií se zabývám způsobem lokálního vývoje grafického uživatelského rozhraní. Následující kapitola obsahuje popis technologií nutných pro případovou studii nasazení aplikace do produkčního prostředí, aby v následující kapitole byl popsán nasazení aplikace do Docker kontejneru a vystavením API do Docker Cloudu u poskytovatele AWS EC2. Pro grafické rozhraní je pak vyžito AWS a jeho služby CloudFront a S3. V poslední části případové studie jsou popsány nápady na vylepšení a automatizaci procesu nasazení do produkce. V poslední kapitole se práce zabývá zabezpečením aplikace a možnými bezpečnostními riziky.

Aplikace je volně přístupná k vyzkoušení na odkaze http://bit.ly/23Sq9it nebo na přiloženém CD-ROM a lze si tak i prohlédnout úspěšně nasazený monitorovací nástroj v produkčním prostředí. Monitorovací nástroj byl řešený formou prototypu a je tedy nutné validovat požadavky uživatelů s obsahem a formou nástroje jako takového. Nástroje lze snadno rozšířit o jakýkoliv další údaje o instanci MongoDB. Grafické uživatelské rozhraní zobrazuje pouze malou část (kvůli přehlednosti) dat které získává.

SEZNAM POUŽITÉ LITERATURY

[1] CHODOROW, Kristina. MongoDB: the definitive guide. 2nd ed. Sebastopol: O'Reilly, 2013, xix, 409 s. ISBN 978-1-4493-4468-9.

[2] Practical mongodb: architecting, developing, and administering MongoDB. New York, NY: Springer Science+Business Media, 2015, pages cm. ISBN 9781484206485.

[3] NIALL O'HIGGINS. MongoDB and Python. Farnham: O'Reilly, 2011. ISBN 9781449310370.

[4] FOWLER, Martin. Destilované UML. 1. vyd. Praha: Grada, 2009, 173 s. Knihovna programátora (Grada). ISBN 978-80-247-2062-3.

[5] BONNET, Laurent, et al. Reduce, you say: What nosql can do for data aggregation and bi in large repositories. In: Database and Expert Systems Applications (DEXA), 2011 22nd International Workshop on. IEEE, 2011. p. 483-488.

[6] VRÁNA, Jakub. 1001 tipů a triků pro PHP. Vyd. 1. Brno: Computer Press, 2010, 456 s. ISBN 978-80-251-2940-1.

[7] MITCHELL, Lorna Jane. PHP web services : [APIs for the modern web]. First edition. Beijing: O'Reilly Media, 2013, x, 104 pages. ISBN 9781449356569.

[8] LEHMANN, Marc Alexander. Benchmarking libevent against libev. In: *Libev* [online]. Schmorpforge Software Repository, 2011 [cit. 2016-05-17]. Dostupné z: http://libev.schmorp.de/bench.html

[9] Monitoring for MongoDB. *MongoDB Documentation* [online]. World: MongoDB Inc., 2016 [cit. 2016-05-16]. Dostupné z: https://docs.mongodb.com/manual/administration/monitoring/

[10] PIOTROWSKI, Aaron a contributors. *Icicle documentation* [online]. 2016 [cit. 2016-05-13]. Dostupné z: https://icicle.io/docs/manual/

[11] MongoDB client for React PHP. *Github* [online]. 2014 [cit. 2016-05-17]. Dostupné z: https://github.com/jmikola/react-mongodb

[12] FOWLER, Martin. Continuous Integration. In: *Martin Fowler* [online]. 2006 [cit. 2016-05-17]. Dostupné z: http://martinfowler.com/articles/continuousIntegration.html

[13] *Webpack* [online]. [cit. 2016-05-13]. Dostupné z: http://webpack.github.io/

[14] *React: a JavaScript library for building user interfaces* [online]. Facebook Inc., 2013 [cit. 2016-05-13]. Dostupné z: https://facebook.github.io/react/

[15] WebSockets. In: *Mozilla Developer Network* [online]. 2016 [cit. 2016-05-13]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

[16] *Asynchronous WebSocket server for Icicle https://icicle.io* [online]. Github, 2016 [cit. 2016-05-13]. Dostupné z: https://github.com/icicleio/websocket

[17] *JSON: Úvod do JSON* [online]. 1999 [cit. 2016-05-13]. Dostupné z: http://www.json.org/json-cz.html

[18] *Nette NEON: loads and dumps NEON files https://nette.org* [online]. Github, 2016 [cit. 2016-04-10]. Dostupné z: https://github.com/nette/neon

[19] *Babel: is a JavaScript compiler.* [online]. 2016 [cit. 2016-05-17]. Dostupné z: https://babeljs.io

[20] *Shodan Search: "MongoDB" and "CZ"* [online]. Shodan, 2016 [cit. 2016-05-17]. Dostupné z: https://www.shodan.io/search?query=country%3A%22CZ%22+product%3A%22MongoDB%22

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

SaaS	Software As A Service – software jako služba
API	Application Programming Interface – rozhraní pro programování aplikací.
JSON	JavaScript Object Notation
YAML	YAML Ain't Markup Language
DOM	Document Object Model
MVC	Model View Controller
RESTfull	Representational State Transfer se striktní podporou REST specifikace (full)
AWS	Amazon Web Services
2FA	Two Factor Authentication
CI	Continuous Integration – postupná integrace

SEZNAM OBRÁZKŮ

Obrázek 1 Ukázka webového rozhraní HTTP Konzole MongoDB ... 13
Obrázek 2 Monitorovací nástroj Ganglia zobrazuje v grafech aktuální vytížení jednotlivých částí systému ... 15
Obrázek 3 Porovnání *libevent* a *libev* a emulované *libev* smyčky, převzato z [8] ... 19
Obrázek 4 Ilustrace práce balíčkovacího systému Webpack. Převzato z [12] ... 24
Obrázek 6 Zdrojový kód se smyčkou událostí a adaptérem na ReactPHP ... 30
Obrázek 7 Zdrojový kód použití napojení Icicle na ReactPHP ... 30
Obrázek 8 Práce s *react-mongodb* knihovnou, poslání dotazu a získání odpovědi ... 31
Obrázek 9 Diagram tříd zpráv komunikačního protokolu [4] ... 32
Obrázek 10 Ukázková struktura konfiguračního souboru *config.neon.default* ... 33
Obrázek 11 Uživatelské rozhraní integračního nástroje *Travis CI* ... 34
Obrázek 12 Selhání integračního nástroje při instalaci závislosti na balíčku ... 35
Obrázek 13 Uživatelské rozhraní integračního nástroje *TravisCI* ... 35
Obrázek 14 Nástěnka *Scrutinizer CI* po první integraci ... 36
Obrázek 15 Výsledek testu komplexnosti metod ... 36
Obrázek 16 Dockerfile použitý k vytvoření Docker kontejneru ... 42
Obrázek 17 Vytvoření nové EC2 instance pomocí Docker Cloudu. ... 43
Obrázek 18 Výběr poskytovatele serverů ve službě Docker Cloud ... 44
Obrázek 19 Vytvoření kontejneru z registru ve službě Docker Cloud ... 44
Obrázek 20 Běžící SPA aplikace napojená na API ... 45
Obrázek 21 Příkaz pro nahrání nové verze SPA do S3 ... 46
Obrázek 22 Detail zobrazení instance MongoDB v SPA ... 62

SEZNAM TABULEK

Tabulka 1-1 Ukázkový výstup z nástroje *mongostat* 12

Tabulka 1-2 Ukázkový výstup z nástroje *mongotop* 12

SEZNAM PŘÍLOH

Příloha P1: Promises/A+

Příloha P2: Detail instance nasazené SPA

Příloha P3: Seznam příloh na disku CD-ROM

PŘÍLOHA P I: PROMISES/A+

An open standard for sound, interoperable JavaScript promises—by implementers, for implementers.

A *promise* represents the eventual result of an asynchronous operation. The primary way of interacting with a promise is through its then method, which registers callbacks to receive either a promise's eventual value or the reason why the promise cannot be fulfilled.

This specification details the behavior of the then method, providing an interoperable base which all Promises/A+ conformant promise implementations can be depended on to provide. As such, the specification should be considered very stable. Although the Promises/A+ organization may occasionally revise this specification with minor backward-compatible changes to address newly-discovered corner cases, we will integrate large or backward-incompatible changes only after careful consideration, discussion, and testing.

Historically, Promises/A+ clarifies the behavioral clauses of the earlier Promises/A proposal, extending it to cover*de facto* behaviors and omitting parts that are underspecified or problematic.

Finally, the core Promises/A+ specification does not deal with how to create, fulfill, or reject promises, choosing instead to focus on providing an interoperable then method. Future work in companion specifications may touch on these subjects.

Terminology

1. "promise" is an object or function with a then method whose behavior conforms to this specification.
2. "thenable" is an object or function that defines a then method.
3. "value" is any legal JavaScript value (including undefined, a thenable, or a promise).
4. "exception" is a value that is thrown using the throw statement.
5. "reason" is a value that indicates why a promise was rejected.

Requirements

Promise States

A promise must be in one of three states: pending, fulfilled, or rejected.

1. When pending, a promise:
    1. may transition to either the fulfilled or rejected state.
2. When fulfilled, a promise:
    1. must not transition to any other state.
    2. must have a value, which must not change.
3. When rejected, a promise:
    1. must not transition to any other state.
    2. must have a reason, which must not change.

Here, "must not change" means immutable identity (i.e. ===), but does not imply deep immutability.

The then Method

A promise must provide a then method to access its current or eventual value or reason.

A promise's then method accepts two arguments:

promise.then(onFulfilled, onRejected)

1. Both onFulfilled and onRejected are optional arguments:
    1. If onFulfilled is not a function, it must be ignored.
    2. If onRejected is not a function, it must be ignored.
2. If onFulfilled is a function:
    1. it must be called after promise is fulfilled, with promise's value as its first argument.
    2. it must not be called before promise is fulfilled.
    3. it must not be called more than once.
3. If onRejected is a function,
    1. it must be called after promise is rejected, with promise's reason as its first argument.
    2. it must not be called before promise is rejected.
    3. it must not be called more than once.

4. onFulfilled or onRejected must not be called until the execution context stack contains only platform code. [3.1].
5. onFulfilled and onRejected must be called as functions (i.e. with no this value). [3.2]
6. then may be called multiple times on the same promise.
    1. If/when promise is fulfilled, all respective onFulfilled callbacks must execute in the order of their originating calls to then.
    2. If/when promise is rejected, all respective onRejected callbacks must execute in the order of their originating calls to then.
7. then must return a promise [3.3].

promise2 = promise1.then(onFulfilled, onRejected);

1. If either onFulfilled or onRejected returns a value x, run the Promise Resolution Procedure[[Resolve]](promise2, x).
2. If either onFulfilled or onRejected throws an exception e, promise2 must be rejected withe as the reason.
3. If onFulfilled is not a function and promise1 is fulfilled, promise2 must be fulfilled with the same value as promise1.
4. If onRejected is not a function and promise1 is rejected, promise2 must be rejected with the same reason as promise1.

The Promise Resolution Procedure

The **promise resolution procedure** is an abstract operation taking as input a promise and a value, which we denote as [[Resolve]](promise, x). If x is a thenable, it attempts to make promise adopt the state of x, under the assumption that x behaves at least somewhat like a promise. Otherwise, it fulfills promise with the value x.

This treatment of thenables allows promise implementations to interoperate, as long as they expose a Promises/A+-compliant then method. It also allows Promises/A+ implementations to “assimilate” nonconformant implementations with reasonable then methods.

To run [[Resolve]](promise, x), perform the following steps:

1. If promise and x refer to the same object, reject promise with a TypeError as the reason.
2. If x is a promise, adopt its state [3.4]:
    1. If x is pending, promise must remain pending until x is fulfilled or rejected.
    2. If/when x is fulfilled, fulfill promise with the same value.
    3. If/when x is rejected, reject promise with the same reason.
3. Otherwise, if x is an object or function,
    1. Let then be x.then. [3.5]
    2. If retrieving the property x.then results in a thrown exception e, reject promise with e as the reason.
    3. If then is a function, call it with x as this, first argument resolvePromise, and second argument rejectPromise, where:
        1. If/when resolvePromise is called with a value y, run [[Resolve]](promise, y).
        2. If/when rejectPromise is called with a reason r, reject promise with r.
        3. If both resolvePromise and rejectPromise are called, or multiple calls to the same argument are made, the first call takes precedence, and any further calls are ignored.
        4. If calling then throws an exception e,
            1. If resolvePromise or rejectPromise have been called, ignore it.
            2. Otherwise, reject promise with e as the reason.
    4. If then is not a function, fulfill promise with x.
4. If x is not an object or function, fulfill promise with x.

If a promise is resolved with a thenable that participates in a circular thenable chain, such that the recursive nature of [[Resolve]](promise, thenable) eventually causes [[Resolve]](promise, thenable) to be called again, following the above algorithm

will lead to infinite recursion. Implementations are encouraged, but not required, to detect such recursion and reject promise with an informative TypeError as the reason. [3.6]

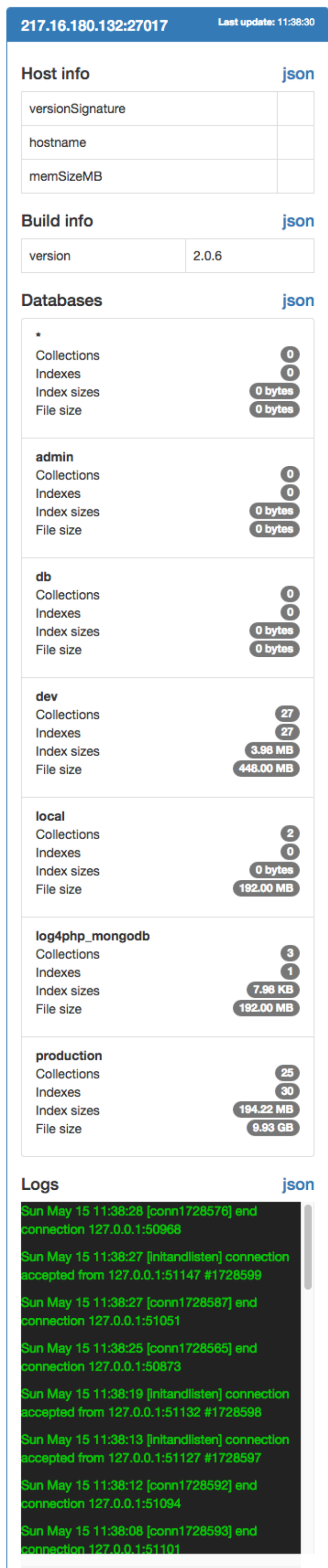
Notes

1. Here "platform code" means engine, environment, and promise implementation code. In practice, this requirement ensures that onFulfilled and onRejected execute asynchronously, after the event loop turn in which then is called, and with a fresh stack. This can be implemented with either a "macro-task" mechanism such as setTimeout or setImmediate, or with a "micro-task" mechanism such asMutationObserver or process.nextTick. Since the promise implementation is considered platform code, it may itself contain a task-scheduling queue or "trampoline" in which the handlers are called.
2. That is, in strict mode this will be undefined inside of them; in sloppy mode, it will be the global object.
3. Implementations may allow promise2 === promise1, provided the implementation meets all requirements. Each implementation should document whether it can produce promise2 === promise1and under what conditions.
4. Generally, it will only be known that x is a true promise if it comes from the current implementation. This clause allows the use of implementation-specific means to adopt the state of known-conformant promises.
5. This procedure of first storing a reference to x.then, then testing that reference, and then calling that reference, avoids multiple accesses to the x.then property. Such precautions are important for ensuring consistency in the face of an accessor property, whose value could change between retrievals.
6. Implementations should *not* set arbitrary limits on the depth of thenable chains, and assume that beyond that arbitrary limit the recursion will be infinite. Only true cycles should lead to a TypeError; if an infinite chain of distinct thenables is encountered, recursing forever is the correct behavior.

To the extent possible under law, the Promises/A+ organization has waived all copyright

and related or neighboring rights to Promises/A+ Promise Specification. This work is published from: United States.

PŘÍLOHA P2: DETAIL INSTANCE NASAZENÉ SPA



Obrázek 22 Detail zobrazení instance MongoDB v SPA

PŘÍLOHA P3: SEZNAM PŘÍLOH NA DISKU CD-ROM

Přiložený CD-ROM obsahuje v kořenovém adresáři, naskenované oficiální zadání a samotný text diplomové práce ve formátu pdf. Složka mongomonitoring obsahuje spustitelnou aplikaci včetně všech nezbytných závislostí a veškeré zdrojové kódy.