Sem vložte zadání Vaší práce.

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

Diplomová práce

Systém pro automatizaci nasazování aplikací

Bc. Adam Staněk

Vedoucí práce: Ing. Tomáš Bartoň

4. května 2015

Poděkování

Rád bych poděkoval své přítelkyni Míše a celé svojí rodině za podporu při psaní této práce i během celého mého studia.

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 4. května 2015 .....

Odkaz na tuto práci

Staněk, Adam. *Systém pro automatizaci nasazování aplikací.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Diplomová práce se zabývá problematikou nasazení Continuous Integration systému na vlastní infrastruktuře. Součástí práce je návrh a implementace vlastního CI řešení jako doplněk pro Gitlab. CI je zaměřeno na automatizaci procesu vystavení webových aplikací a zajištění zabezpečeného přístupu k vystaveným službám.

Klíčová slova průběžná integrace, sestavení aplikace, CI, Mesos, Docker, PaaS, Phoebo, nginx

Abstract

This thesis introduces reader to Continuous Integration systems. It focuses on design and implementation of own CI solution as a complement to existing Gitlab setup. The main goal is to implement platform for automatic deployment of web applications while maintaining secure access to deployed web services.

Keywords Continuous Integration, CI, application build, Mesos, Docker, PaaS, Phoebo, nginx

Obsah

Seznam obrázků

Úvod

Softwarový vývoj ušel za posledních několik let obrovský kus cesty. Díky open-source iniciativě již jen zřídka kdy píšeme aplikace od základů ale využíváme obsáhlých frameworků. Současný hardwarový výkon umožňuje běh robustních aplikací. To vše usnadňuje tvorbu stále komplexnějších softwarových projektů.

S rostoucí složitostí projektů přestaly tradiční metodiky pro vývoj software dostačovat. V roce 2013 více než polovina softwarových projektů skončila selháním [1]. Reakcí ze strany vývojářů byl rozvoj množství alternativních, tzv. agilních, metodik adresujících některé z největších problémů vývoje.

Pojícím rysem všech agilních metodik je snaha o získání větší flexibility při vývoji softwaru. Místo dlouhých vývojových fází zavedených iterativním vývojem nastupuje snaha o vývoj v malých přírůstcích. Projekty často procházejí v průběhu svého vývoje podstatnými změnami požadavků. Úlohou agilních metodik je definovat způsob jak na ně reagovat. Nelze již vyvíjet software s rok dlouhým vývojovým cyklem, jako tomu bylo doposud. Je třeba zajistit dodávku softwaru v pravidelných intervalech, provádět jeho kontrolu a dynamicky reagovat na změny.

Díky snaze o zkrácení dodávky funkčních přírůstků roste režie způsobená prací nutnou pro vydání verze aplikace. V případě využití SCRUM metody je doporučená délka jednoho vývojového cyklu 1 - 2 týdny. Započteme-li čas nutný k otestování nové verze aplikace a dobu k jejímu sestavení pro všechny dostupné platformy, je čas pro možný vývoj méně než polovina cyklu. Za účelem zachování efektivity vývoje se proto snažíme všechny kroky spojené s dodávkou aplikace co nejvíce automatizovat. Systém, který zajišťuje automatizaci dodávky, se nazývá Continuous Integration.

Kapitola 1

Cíl práce

V této práci se zaměřuji na návrh Continuous Integration systému (dále CI) pro automatické nasazování webových aplikací do integračního prostředí.

Mým cílem je vytvořit automatizovanou platformu umožňující běh aplikací v před-produkčním prostředí, ve kterém budou vystavené aplikace podrobeny finálnímu testování.

1.1 Integrační prostředí

Integrační prostředí je prostředí, ve kterém je aplikace nasazena pro finální testování před začleněním nové funkcionality do hlavní větve projektu.

Testování probíhá zpravidla ve dvou fázích:

1. **Testerem,** který zkontroluje správnou integraci všech komponent a splnění požadavků definovaných v rámci dokumentace - tzv. princip dvou očí.

2. **Zákazníkem,** který rozhodne o akceptaci změn.

Integrační prostředí by mělo co nejblíže simulovat běh v produkčním prostředí (stejná architektura, jednotné verze externích závislostí, atd.). Obvykle je aplikace testována s kopií produkčních dat nebo s pečlivě vygenerovanými daty tak, aby simulovala reálné využití aplikace.

1.2 Uživatelské testování

Automatizované testování se těší velké popularitě. Stále však existují věci, které automatizovat nelze a nebo je to příliš nákladné. Jednou z nich je vizuální stránka aplikace. To je v případě webových prezentací velmi velkou složkou celého díla.

Je možné automatizovat testování nižších vrstev aplikace (např. modelů a controllerů). Je možné také automatizovat testování i samotných pohledů (pomocí PhantomJS, Selenia či jiných nástrojů). To bohužel ale stále nestačí. V případě webových aplikací se většina testů odehrává na front-endu aplikace. Je třeba zajistit funkčnost aplikace v heterogenním prostředí různých webových prohlížečů a mobilních zařízení.

Ať už se rozhodneme front-end testovat manuálně či pomocí některých polo-automatických řešení, jako je např. BrowserStack, je třeba zajistit vystavení aplikace a její dostupnost.

1.3 Rozdíl oproti existujícím řešením

Nejvyužívanějšími CI řešeními pro popsané účely jsou v současnosti Jenkins CI, Travis CI a nyní nově Gitlab CI. Hlavním zaměřením všech těchto řešení jsou automatizace spouštění testů a sestavení aplikace.

Cílem této práce je navrhnout CI systém, který umožní běh na vlastní zabezpečené infrastruktuře a vyřešit problém přístupu k publikovaným aplikacím.

Zatímco v případě testů se jedná o jednorázové spuštění úlohy, v případě běžící aplikace (služby) je třeba zajistit její trvalý provoz a postarat se o její dostupnost z vnějšího prostředí. V tomto ohledu se velmi liší pohled na celkovou architekturu CI řešení.

1.4 Vymezení práce

Ačkoliv popisované metody lze aplikovat i pro nasazování aplikací v produkčním prostředí, není to cílem této práce. Nasazování aplikací do produkčního prostředí lze dosáhnout stejnými prostředky ale vyžaduje odlišný návrh architektury systému.

Navrhované řešení je založeno na předpokladu, že je stav aplikace replikovatelný (testovací data) a neřeší jeho persistenci ani přechod mezi verzemi. Zatímco v testovacím prostředí je naším cílem poskytnout současný provoz několika konkurenčních verzí aplikace, v produkčním prostředí je třeba zajistit kontinuální přechod na novou verzi aplikace (graceful transition).

Kromě zmiňovaných rozdílů je v produkci kladen důraz na dostupnost provozovaných aplikací a je tedy potřeba zajistit potřebnou redundanci, kterou testovací prostředí postrádá.

Kapitola 2

Definice požadavků

V této kapitole budou blíže představeny požadavky, které jsem kladl při návrhu CI systému.

2.1 Formální požadavky

2.1.1 Běh na vlastní infrastruktuře

Cílem projektu je vytvořit systém, který bude snadné nasadit na vlastní infrastrukturu. Systém by měl být dobře škálovatelný a umožnit běh, jak na jediném stroji, tak i na celém výpočetním svazku. Aplikační server by měl být zcela izolovaný od spouštěných úloh.

2.1.2 Izolace

Jednotlivé úlohy by měly být spouštěny v izolovaném prostředí. Ideálně s využitím kontejnerů či virtualizační technologie. Musí být zajištěno, že aplikace nemůže čerpat prostředky na úkor jiné (tzn. nesmí využívat celý výkon serveru). Izolace by měla probíhat alespoň na úrovni síťové vrstvy a souborového systému.

2.1.3 Běhové prostředí

Je očekáváno nasazení společně se systémem Gitlab. Je proto vhodné aplikaci co nejvíce přizpůsobit již existujícímu prostředí. Ideálním běhovým prostředím je proto Ruby 1.9.3+, případně s využitím databázového serveru PostgreSQL a Redis úložiště.

2.1.4 Redundance

CI systém není nutné provozovat ve více instancích, ale je požadováno navrhnout architekturu tak, aby to bylo do budoucna pro zvýšení dostupnosti služby možné.

2.2 Funkční požadavky

2.2.1 Integrace s aplikací Gitlab

Navrhované řešení by mělo být integrovatelné s využitím aplikace Gitlab.

Systém by měl umožnit:

1. **Přihlášení uživatele pomocí Gitlab účtu** - zajištění jednotného přihlašování pro existující uživatele Gitlabu a to bez ohledu na to, zda-li jsou interními nebo externími uživateli (LDAP, Shibboleth, atd.).
2. **Práci s Gitlab projekty** - s jednotlivými úlohami by mělo být možné hromadně pracovat na úrovni Gitlab projektů (třídit, spravovat, atd.).
3. **Sdílení uživatelského oprávnění** - uživatel, který je vývojářem některého z Gitlab projektů, by měl mít přístup k projektovým úlohám bez explicitního nastavování.
4. **Automatizaci vystavování deploy klíčů** - aplikace by se měla sama starat o generování a registraci deploy klíčů pro zajištění přístupu ke zdrojovému kódu projektu.

2.2.2 Zabezpečení služeb

Spuštěné služby (např. webové aplikace) by měly být dostupné pouze pro:

1. **uživatele systému**, který se prokáže uživatelským jménem a heslem a je zároveň vývojářem nebo správcem projektu.
2. **externího uživatele (zákazníka)**, který se prokáže znalostí hesla (PIN).

Pro autorizaci služeb by mělo být využito single-sign-on principu, tzn., že již jednou přihlášený uživatel (v rámci navrhované aplikace) se nebude muset znovu autentifikovat pro přistoupení k dané službě.

2.2.3 Konfigurace způsobu sestavení

Konfigurace sestavení by měla být rozdělena do dvou částí:

1. **Konfigurace popisující postup, jak aplikaci sestavit.**

   Tato konfigurace by měla být umožněna pro každou verzi aplikace zvlášť. Konfigurace by měla být souborem, který je součástí repozitáře a bude převážně spravována samotnými vývojáři aplikace.

2. **Konfigurace parametrů pro sestavení.**

   Tato konfigurace slouží pro uložení parametrů, které součástí repozitáře být nesmějí. To jsou např. parametry popisující závislost na vnějším prostředí, hesla, apod. Tato konfigurace by měla být dostupná skrze uživatelské rozhraní aplikace. Konfigurace by měla zajistit potřebnou bezpečnost uložení hesel.

2.2.3.1 Dědičnost parametrů

Parametry pro sestavení by měly využívat těchto stupňů dědičnosti:

1. **Individuální projektové nastavení** - parametry specifické pouze pro jeden konkrétní projekt. Zde budou např. uloženy deploy klíče pro daný projekt.

2. **Nastavení skupiny (namespace)** - v rámci Gitlabu náleží všechny projekty do tzv. projektové skupiny. Mělo by být umožněno sdílení nastavení mezi všemi projekty ve skupině. Příkladem podobného nastavení je např. nastavení přístupu ke sdílenému databázovému serveru.

3. **Defaultní nastavení** - parametry sdílené pro všechny projekty. Jsou nastavované administrátorem a slouží jako šablona.

Všechny stupně nastavení by mělo být možné upravovat skrze uživatelské rozhraní aplikace.

2.2.4 Zprostředkování výstupu aplikace

Navrhované řešení by mělo nabídnout výstup spuštěné aplikace (stdout, stderr) prostřednictvím svého uživatelského rozhraní. K výstupu by měli mít přístup pouze vývojáři (a správci) daného projektu. Aplikační výstup by měl být přenášen pokud možno v reálném čase, aby bylo usnadněno debuggování potenciálních problémů.

Výstup aplikace lze omezit velikostí. Persistence celého výstupu bude řešena buď aplikací samotnou (v rámci sandboxu) a nebo některou z externích služeb (fluentd, logstash, ...).

Kapitola 3

Běh aplikací na výpočetním svazku

Jedním z hlavních požadavků, které jsem si při návrhu CI systému kladl, je zajištění nasazování aplikací na dynamickou infrastrukturu. Mým cílem bylo zbavit se jakéhokoliv statického rozdělení fyzických serverů. V této kapitole přiblížím některé prostředky, které jsem za tímto účelem použil.

3.1 Sdílení prostředků

V dnešní době je stále častěji skloňovaný termín cloud computingu. Jakkoliv je tento termín zavádějící, vnesl do našeho podvědomí způsob, jak přestat přemýšlet nad jednotlivými fyzickými prostředky a začít uvažovat nad logickým celkem.

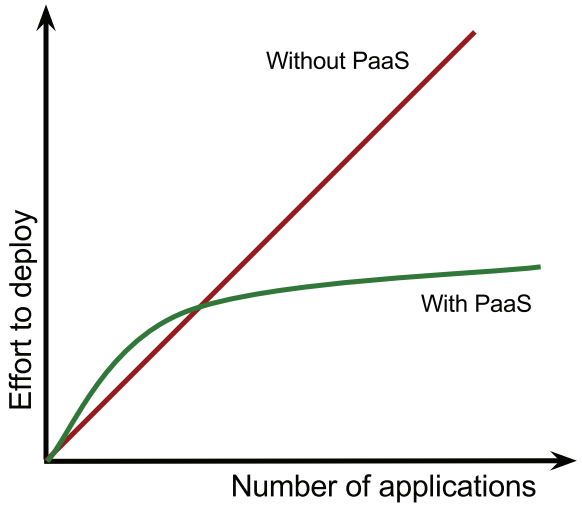
Díky velkému nárůstu výkonu serverů se stává neefektivní dedikovat fyzické prostředky jednotlivým aplikacím. Dynamického členění se nejčastěji dosahuje využitím virtualizačních technologií.

Mezi největší výhody, které nám toto řešení přináší patří:

1. **Abstrakce nad fyzickými prostředky** - aplikace se stávají přenositelnou jednotkou. V případě poruchy nebo změny fyzické architektury lze provozované aplikace snadno migrovat. Toho lze při vhodné architektuře dosáhnout i zcela bez výpadku.

2. **Sdílení a škálování prostředků** - ne každá aplikace vždy využije plného výkonu serveru. Sdílením nevyužitých prostředků s ostatními aplikacemi dochází ke snížení nákladů obětovaných příležitosti (opportunity costs). Každá aplikace má obvykle svoje výkyvy v době špičky. Díky škálování můžeme zátěž rozložit a využít tak prostředků celého svazku.

3. **Inkrementální růst** - s využitím sdíleného výpočetního svazku není třeba pro každou novou aplikaci, která vyžaduje specifické běhové pro-



Obrázek 3.1: Automatizace procesu vystavení pomocí PaaS

středí, hned pořizovat nový fyzický server. Fyzické prostředky lze postupně navyšovat s rostoucími požadavky na výkon, které lze snadno měřit a předvídat.

4. **Dynamická alokace** - díky dynamické alokaci prostředků není nutné provádět konfiguraci low-level aspektů systému pro běh každé nové aplikace. Stačí vyjádřit pouze konfiguraci prostředků, na nichž je nasazovaná aplikace bezprostředně závislá. Nemusíme se tak při vystavení každé aplikace starat o stav a údržbu fyzické architektury. **Tohoto aspektu využíváme právě pro navrhované CI řešení.**

3.2 Správa a konfigurace fyzických prostředků

Při nasazování vlastního PaaS řešení je třeba jako první zajistit konfiguraci fyzických serverů (implementaci IaaS). Cílem je vytvořit na každém serveru totožné podmínky pro běh všech plánovaných aplikací, čímž zajistíme, že z pohledu aplikace nezáleží na tom, na kterém serveru bude nasazena.

Konfigurace fyzických serverů není přímo předmětem této práce, ale je nezbytná proto, abychom mohli nasadit navrhované řešení.

3.2.1 Provisioning

Jedním z nejjednodušších způsobů jak dosáhnout jednotné konfigurace je využití tzv. provisioningu. Nejrozšířenější řešení pro daný úkol jsou systémy Chef a Puppet. Cílem obou systémů je popsat požadovaný stav konfigurace. Provisioner se postará o aplikaci potřebných pravidel k jeho dasažení. Pravidla jsou popsána v podobě tzv. manifestů, které lze spravovat centrálně (Puppet Master).

3.2.2 CoreOS

O něco složitější je nasazení některého ze specializovaných operačních systémů jako je např. CoreOS. CoreOS je mladá linuxová distribuce, která využívá k běhu aplikací Docker kontejnery. Díky tomu je zajištěna izolace procesů a aplikační přenositelnost [2].

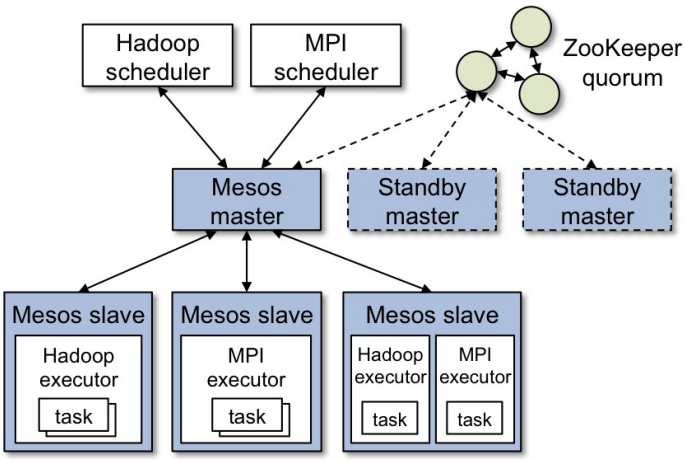
Ke správě svazku využívá CoreOS distribuované key-value úložiště *etcd*. To umožňuje sdílení konfigurace napříč jednotlivými výpočetními uzly. Zároveň slouží pro tzv. service discovery, což umožňuje dynamickou orchestraci služeb bez explicitní definice jejich závislosti.

Běh a správu samotných aplikací má na starosti nástroj *fleet*. Ten slouží jako centralizovaný init systém pro celý výpočetní svazek. Ke svému běhu využívá vazby na *systemd* každého z výpočetních uzlů [3].

3.3 Orchestrace služeb a dynamická alokace prostředků

Za předpokladu, že máme na každém fyzickém serveru nakonfigurované běhové prostředí, můžeme začít na výpočetním svazku provozovat naše aplikace. To, jakým způsobem je budeme v rámci svazku distribuovat, mají na starosti tzv. orchestrační systémy.

Díky tomu, že jsou všechny servery nakonfigurovány jednotně, můžeme abstrahovat rozdíly mezi jednotlivými výpočetními uzly. Orchestrační systémy se starají právě o tento stupeň abstrakce. V roli plánovače se starají o dynamickou distribuci výpočetních úloh na serverech s volnými výpočetními prostředky.



Obrázek 3.2: Diagram architektury Mesosu (současný běh Hadoop a MPI frameworku)

3.3.1 Mesos

Pro správu výpočetních úloh jsem při návrhu CI systému vybral systém Mesos. Mesos je osvědčeným orchestračním systémem s modulární architekturou.

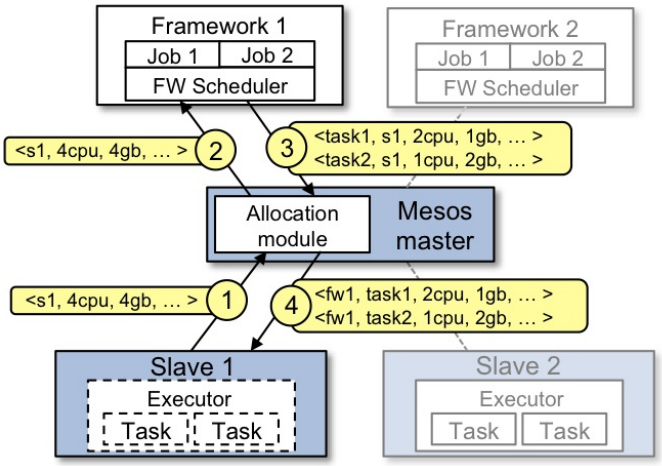
Architekturu systému Mesos lze rozdělit na tzv. Mesos master a množinu Mesos slave hostů. Mesos master slouží jako řídící prvek pro distribuci požadavků a synchronizaci stavu úloh. Jako Mesos slave host označujeme každý z výpočetních uzlů, který je k masteru přihlášen. Na Mesos slave hostech jsou spouštěny jednotlivé naplánované úlohy.

Úlohy jsou v rámci Mesosu spravovány pomocí tzv. frameworků. Každý Mesos framework se skládá ze 2 komponent:

1. Plánovač (scheduler), který se stará o registraci frameworku a zpracování nabídek prostředků.

2. Spouštěč (executor), který má na starosti spuštění procesu na požadovaném Mesos slave hostu.

Mesos využívá tzv. 2-stupňového plánování (viz. další sekce).



Obrázek 3.3: Ukázka nabídky prostředků

3.3.1.1 Distribuce prostředků

Mesos master nabízí sdílení prostředků mezi frameworky pomocí systému nabídky prostředků (resource offer). Nabídka prostředků je zasílána Mesos masterem každému zaregistrovanému frameworku a obsahuje seznam volných prostředků každého dostupného Mesos slave. Mesos master určuje kolik prostředků bude nabídnuto konkrétnímu frameworku pomocí předem vybrané politiky (např. prioritní plánování nebo rovnoměrné sdílení).

Každý z frameworků si vybírá, kterou z nabídek využije. Rozhoduje se tak v závislosti na požadovaných prostředcích svých naplánovaných úloh. Mezi základní prostředky patří: CPU, paměť, disk a síťové porty. Při přijímání nabídek se každý framework snaží o minimalizaci fragmentace prostředků a uspokojení maximálního množství naplánovaných úloh.

Narozdíl od plánování centrálním plánovačem, je díky využití systému nabídek rozhodování plně v režii frameworku. Vlastní framework se tak může rozhodnout, zda-li nabídka splňuje i všechny složitě kvantifikovatelné požadavky, jako je např. dostupnost lokálních dat potřebných pro výpočet [4].

3.3.1.2 Zajištění vysoké dostupnosti služby (HA)

Mesos je ze své podstaty distribuovaným systémem. Architektura umožňuje běh několika instancí Mesos masteru. Pro synchronizaci je využit Zookeeper, který lze taktéž provozovat ve více instancích - v tzv. Zookeeper quoru. V rámci quora je vždy jeden z Masterů zvolen za hlavní. Ten je potom zodpovědný za dohled nad jednotlivými Mesos slaves. Veškerá data jsou dostupná všem redundantním instancím.

3.4 Virtualizace a izolace

Pro současný běh několika naplánovaných úloh je důležité zajistit jejich izolaci. Izolací rozumíme, že žádná z úloh nebude schopná ovlivnit běh jiné úlohy. Naším cílem je zajistit, že každá aplikace bude mít k dispozici pouze prostředky nezbytně potřebné pro svůj běh a žádné jiné. Aplikace nesmí mít přístup k výpočetním prostředkům vyhrazeným jiné aplikaci, ani k jejím souborům.

Virtualizací rozumíme sdílení fyzického prostředku (CPU, pamět, disk, ...) pomocí jeho rozdělení do logických (virtuálních) celků. Virtualizaci můžeme provádět na několika úrovních:

1. na úrovni HW (plná nebo částečná virtualizace) tím, že simulujeme skutečnou činnost jednotlivých zařízení. Virtualizovaný OS běží bez jakékoliv modifikace (a znalosti, že nekomunikuje s fyzickým HW). Toho využívají řešení jako Oracle VM, VirtualBox, VMWare, Parallels a další.

2. na úrovni low-level API (paravirtualizace) tím, že modifikujeme virtualizovaný OS takovým způsobem, že nahradíme volání na nejnižší vrstvě OS naším vlastním, virtualizovaným rozhraním. Mezi tyto virtualizační systémy řadíme např. Xen.

3. na úrovni operačního systému tím, že umožníme sdílení API jádra OS pomocí virtuálních prostředí (namespaces). Příkladem tohoto způsobu virtualizace je např. BSD Jails nebo nyní popularizovaný Docker. Tento druh virtualizace se také někdy nazývá kontejnerizace.

Při volbě vhodné virtualizační technologie, jsem se rozhodl pro využití kontejnerizace, a to konkrétně pomocí Dockeru.

Ačkoliv plná virtualizace umožňuje provoz libovolného operačního systému, je tento fakt vykoupen vysokými nároky na výpočetní prostředky. Pro každou instanci virtualizovaného stroje musí běžet vlastní operační systém včetně všech jeho vrstev. To je pro naše účely zbytečné. Většina provozovaných úloh bude psána pro běh v jednom OS, resp. pro běh na linuxovém jádře. Díky tomu můžeme mnohem efektivněji virtualizovat pomocí kontejnerů.

3.4.1 Docker

Při virtualizaci pomocí Dockeru využívá každá aplikace vlastního jmenného prostoru (namespace). To znamená, že každá instance spravuje svoji vlastní množinu procesů, a nemá přístup k žádné jiné. Mimo procesů využívá Docker jmenné prostory také pro virtualizaci síťových zařízení, IPC, přípojných bodů (mount points) a UTS [5].

Kromě vlastního běhového prostředí je také třeba zajistit rozložení výpočetních prostředků. Toho je v případě Dockeru dosaženo pomocí tzv. Control groups (cgroups). Control groups umožňují pro každou instanci nastavit limit

potřebných výpočetních prostředků. Nehrozí tak, že by si některá z aplikací přivlastnila celý výpočetní výkon fyzického serveru.

Aplikace provozované s použitím Dockeru jsou uzavřené do tzv. kontejnerů. Kontejner je specifický formát umožňující popis virtualizovaného prostředí a jeho souborového systému.

Jako souborový systém využívá Docker UnionFS. UnionFS je souborový systém umožňující pouze čtení souborů (read-only fs). Zápis je prováděn pomocí vrstvení metodou copy-on-write. Díky tomu je zajištěna možnost sdílení části dat mezi různými kontejnery (neměnná systémová část).

Kontejner je základní přenositelnou jednotkou, se kterou můžeme pracovat. Umožňuje nám finální vrstvu abstrakce pro naše aplikace tím, že popisuje bezprostřední běhové prostředí.

3.5 Zajištění dostupnosti (Service discovery)

Proto, abychom mohli plně využít dynamického plánování úloh, musíme navrhovaný systém zbavit jakékoliv závislosti na umístění jednotlivých instancí. V případě mezi-aplikačních vazeb označujeme tento problém jako service discovery.

V rámci service discovery je naším cílem vytvořit systém vnějších závislostí. Každá aplikace by měla definovat pouze ty služby, které ke svému běhu potřebuje, ale neměla by se již starat o to, kde jsou služby fyzicky provozovány a jak se k nim dostat.

Jedním ze způsobů jak toho dosáhnout je využitím vlastního DNS serveru, který se postará o potřebný překlad názvu služby na adresu jejího nasazení.

Mesos-DNS je jednoduchý DNS server umožňující adresování provozovaných služeb pomocí virtuální domény '*.mesos'. V případě jeho využití lze jednotlivé běžící služby (aplikace) jednoduše adresovat jako 'sluzba.framework.mesos'. Překlad jména bude adresován přímo na výpočetní uzel, na kterém je daná aplikace provozována.

3.5.1 Proxy

Pro přístupu z vnějšího prostředí již podobného přístupu využít nelze (pokud nemůžeme provozovat veřejný DNS server).

Vhodnějším způsobem jak situaci vyřešit je nasazení brány v podobě proxy serveru. Vytvoříme tím tzv. bod jednotného přístupu (single point of entry).

Z pohledu uživatele bude díky tomu adresa služby neměnná (adresa proxy serveru) a proxy server se postará o směrování požadavků na správný fyzický server.

Tento princip je hojně využíván zejména díky tomu, že umožňuje vyvažování zatížení (load balancing) na úrovni protokolu.

Nejvyužívanějšími řešeními jsou HAProxy a nginx (ve funkci reverzní proxy).

3.5.1.1 HAProxy

HAProxy je specializovaný proxy server pro provádění load balancingu a zajištění trvalé dostupnosti služeb (high availability). V případě většího množství požadavků nabízí neporovnatelný výkon a umožňuje fungovat jako prostředník pro libovolný protokol na vrchu TCP. HAProxy je stabilní a osvědčené řešení a nelze než doporučit pro produkční nasazení.

3.5.1.2 Nginx

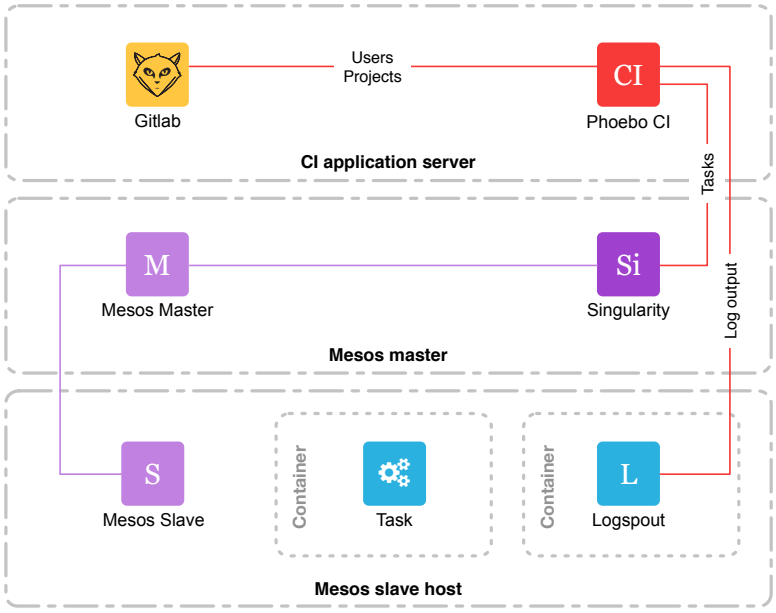
Jako alternativu k HAProxy jsem zvolil server Nginx. V současné době je velmi často využíván jako velmi rychlý webserver pro statické soubory a reverzní proxy pro aplikační servery (Mongriel, Puma, ...). Nginx byl od začátku vývoje navrhován jako generická TCP proxy, není proto nijak omezený na použití HTTP protokolu. Díky implementaci embedded jazyka (lua) umožňuje plnou kontrolu nad zpracováním požadavků.

Kapitola 4

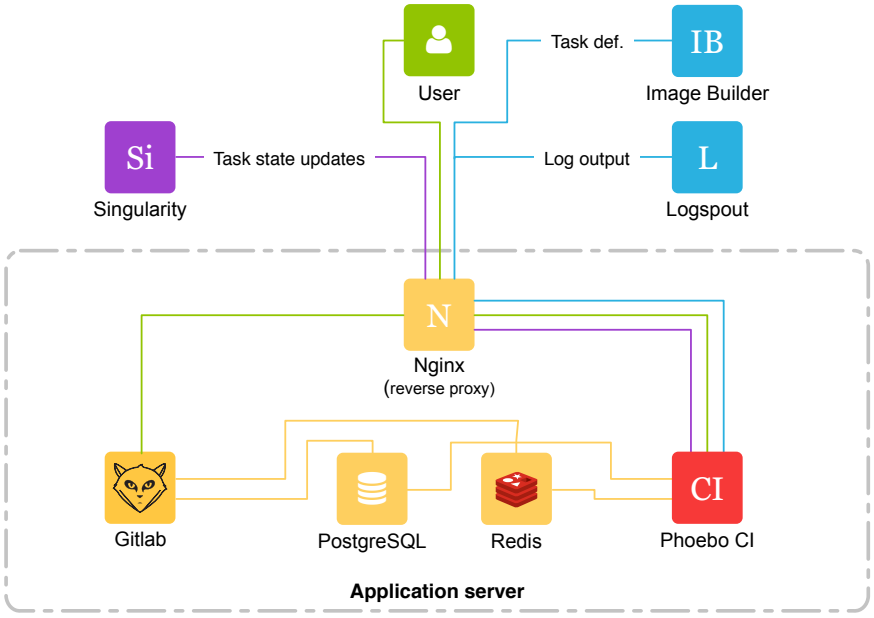
Návrh architektury pro nasazení CI

V této kapitole podrobněji rozeberu představené technologie z pohledu integrace do CI systému.

Pro zjednodušení budu architekturu popisovat na 3 serverech. Není to však podmínkou. Škálování výpočetního svazku není z tohoto pohledu nijak omezeno. Můžeme aplikaci provozovat jak na desítkách serverů, tak i na jediném stroji. Komunikační principy zůstanou stejné.



Obrázek 4.1: Diagram rozdělení serverů



Obrázek 4.2: Diagram vnějších závislostí aplikačního serveru

4.1 Aplikační server

Pro nasazení doporučuji instalaci Gitlabu a Phoebo CI na stejném serveru. Obě aplikace spolu komunikují pouze pomocí REST API, takže není problém je provozovat i separátně, ale zbytečně bychom tím zesložiťovali cílovou architekturu. Phoebo CI byl od počátku navrhovaný tak, aby mohl využít maximum z běhového prostředí Gitlabu a ulehčil tak svou vlastní instalaci.

Pro běh Gitlabu je potřeba zajistit Ruby nejméně ve verzi 1.9.3 a běh podpůrných služeb:

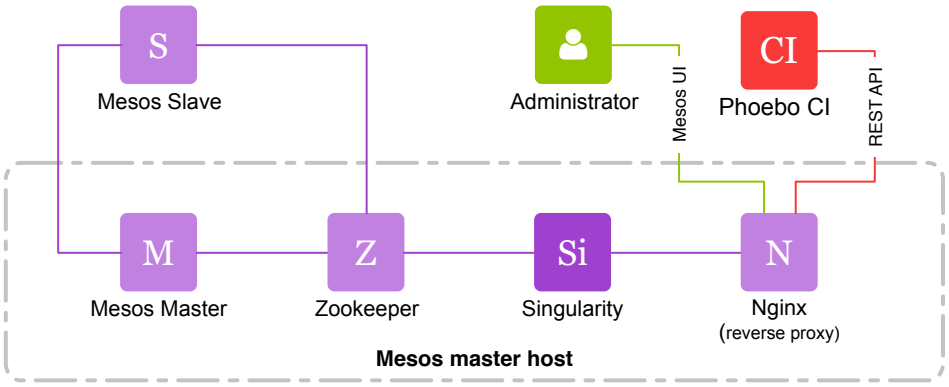
1. reverzní proxy jako prostředníka pro HTTPS transport a obsluhu statických souborů.

2. databázového serveru - PostgreSQL nebo MySQL - pro správu uživatelských dat.

3. Redisu - pro asynchronní zpracování úloh (pub/sub) a cachování.

Phoebo CI využívá všech zmíněných služeb. Instalace je proto velmi jednoduchá a šetří výpočetní prostředky.

Komunikace s vnějším prostředím probíhá pouze skrze reverzní proxy. Veškeré ostatní služby by měly být z vnějšího prostředí nedostupné.



Obrázek 4.3: Diagram vnějších závislostí na Mesos master hostu

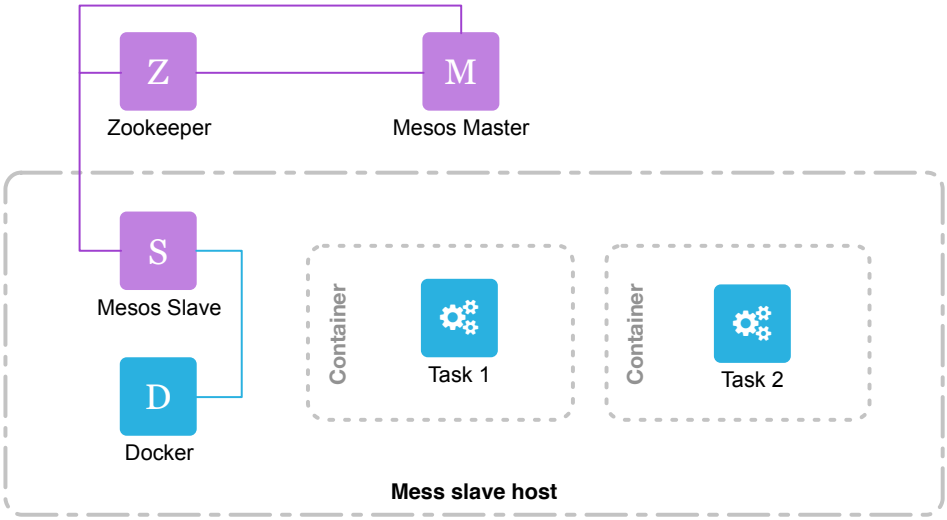
4.2 Mesos master a Singularity

Pro plánování úloh v rámci Mesosu jsem zvolil framework Singularity. Zvolil jsem jej proto, že jako jediný z dostupných frameworků umožňuje plánování jak jednorázových úloh (one-off), tak i služeb (services). Většina dostupných frameworků se specializuje pouze na jeden druh plánování (Marathon, Chronos, a další).

Singularity nabízí pro plánování úloh REST API přes HTTP protokol. Neumožňuje ale žádnou formu řízení přístupu. Rozhodl jsem se proto při implementaci schovat samotné API za reverzní proxy - konkrétně v podobě nginx serveru. Díky tomu je možné nabídnout veřejné rozhraní s autentifikací pomocí klientských SSL certifikátů.

Pro komunikaci s Mesosem využívá Singularity Zookeeperu. Jelikož v rámci navrhovaného řešení nepočítám s jeho využitím mimo Mesos, rozhodl jsem se pro jeho nasazení přímo na Mesos master hostu. V případě redundantního nasazení je rozumné mít alespoň tolik instancí Zookeeperu, jako je Mesos master instancí. Nebude to tedy činit žádné omezení ani v tomto případě.

Zookeeper je dále využíván pro synchronizaci mezi Mesos master a Mesos slave daemonem. Proto je nutné zajistit jeho dostupnost zvenčí. V případě Zookeeperu je řízení přístupu složitější. Samotný Zookeeper podporuje komunikaci pomocí dedikované síťové vrstvy Netty, která má široké možnosti. V současné verzi ale nejsou všechny možnosti podporovány Mesosem. Jediným podporovaným ověřovacím mechanismem ze strany Mesosu je HTTP Basic. Jelikož jsou data při použití této metody přenášena v plain-text podobě, je třeba zajistit bezpečnost komunikace na jiné úrovni. Doporučuji proto vytvořit mezi Mesos master hostem a jednotlivými Mesos slave hosty IPSec tunel. Díky tunelu bude zajištěno šifrování i integrita přenášených dat.

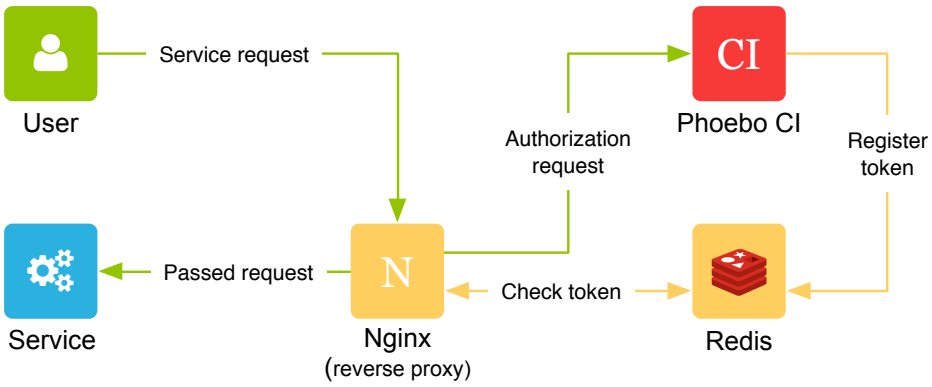


Obrázek 4.4: Diagram vnějších závislostí na Mesos slave hostu

4.3 Výpočetní uzly (Mesos slave)

Posledním prvkem navrhované architektury jsou samotné výpočetní uzly - Mesos slave hosty. Mesos slave slouží ke spouštění naplánovaných úloh. Každý výpočetní uzel obsahuje běžící instanci Mesos slave daemonu, která komunikuje s Mesos masterem s pomocí externího Zookeeperu (běžícího na Mesos master hostu).

Pro izolaci jednotlivých aplikací využívá Mesos slave Dockeru. Využití Dockeru bude podrobněji popsáno v kapitole č. 6.



Obrázek 4.5: Diagram autorizace požadavku pomocí proxy serveru

4.4 Proxy pro běžící služby

Jako předsazený proxy server jsem zvolil nginx. Zvolil jsem jej zejména pro to, že umožňuje snadnou implementaci vlastního autorizačního modulu pomocí *lua*. Zároveň můžeme využít již běžící instance na Mesos master hostu, takže není potřeba provozovat další službu.

Nginx v posledních verzích podává pro zvolené účely vyrovnaný výkon s HAProxy, a jelikož pro naši aplikaci nepotřebujeme žádný komplexní proxy backend, není důvod pro nasazení separátní instance.

Pro implementaci jsem využil distribuce *Openresty*, která v rámci balíku nabízí i některé lua moduly (konkrétně využívám *LuaNginxModule* a *LuaRestyRedisLibrary*).

Kromě směrování požadavků jednotlivým výpočetním uzlům, se proxy server stará také o řízení přístupu k běžícím službám. Pro ověření přístupu jsou vytvářena uživatelská sezení pomocí přesměrování na autorizační stránku aplikačního serveru. Sezení jsou ukládána v Redisu, ze kterého čte přímo nginx. Identifikátor uživatelského sezení je náhodný řetězec vygenerovaný při prvním přístupu a uložen jako cookie. Přístup ke službě je automaticky odebrán při zavření prohlížeče nebo při přerušení komunikace po dobu delší jak 1 hodina.

Tento způsob ověřování probíhá zcela ve vlastní vrstvě a nijak nekoliduje s vlastní autentifikací, kterou může vystavená aplikace implementovat. Zároveň je toto řešení zcela transparentní pro použití protokolu, tedy lze na testované aplikaci simulovat provoz přes zabezpečený i nezabezpečený HTTP protokol, a to včetně podpory pro WebSocket.

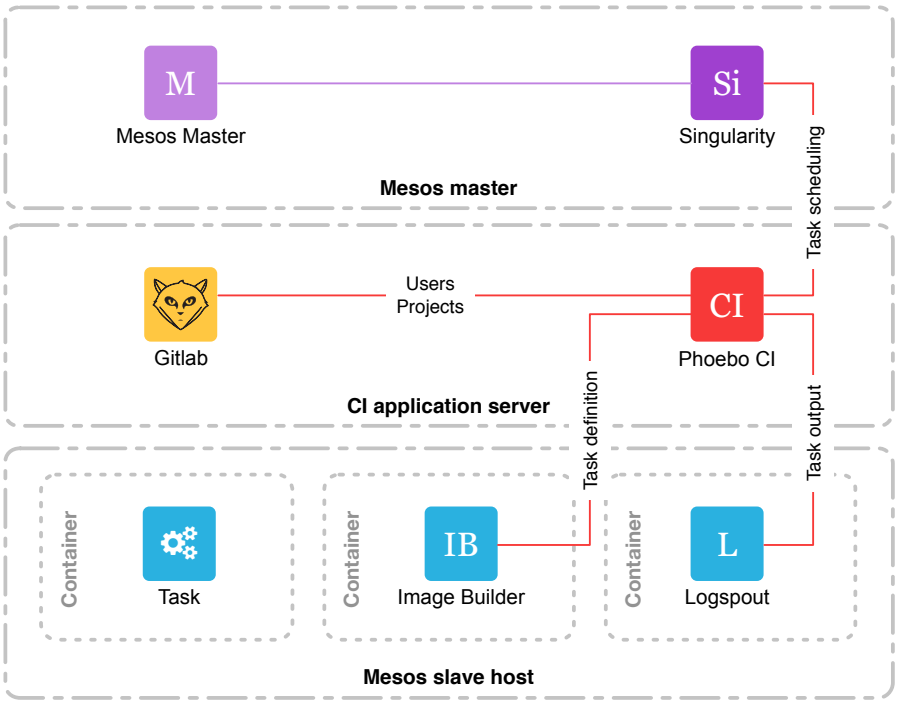
Kapitola 5

Implementace: Phoebo CI

Aplikační server Phoebo CI je základní komponentou celého systému. Zprostředkovává webové uživatelské rozhraní a slouží jako řídící komponenta pro ostatní služby.

Hlavní funkce aplikačního serveru jsou:

1. **Nabízet uživatelské rozhraní** umožňující:
   a) přehled a správu nad běžícími službami a úlohami,
   b) zadávat vlastní požadavky k sestavení (build),
   c) registrovat projekty do CI a měnit jejich nastavení.
2. **Přijímat výstup ze spuštěných aplikací** a nabídnout rozhraní pro jeho zobrazení.
3. **Přijímat požadavky na naplánování projektových úloh** z procesu sestavení projektu (Image Builder) a plánovat je.
4. **Přijímat změny stavu spravovaných úloh (Singularity)** a reagovat na ně.
5. **Umožnit autorizaci uživatele pro přístup k běžícím službám.**



Obrázek 5.1: Diagram zasazení aplikace Logspout

5.1 Aplikační server

Při návrhu CI jsem se chtěl vyhnout nutnosti implementace samostatného daemona. Vzhledem k tomu, že všechna komunikace bude probíhat pouze na úrovni webových služeb, je implementace vlastního síťového stacku zbytečná. Rozhodl jsem se proto vystavět řešení kompletně navrchu Rack vrstvy, konkrétně pomocí web serveru Puma.

Puma je velmi rychlá implementace více-vláknového HTTP serveru pro Ruby / Rack aplikace. Vyniká zejména velmi dobrým výkonem při konkurenčním zatížení a nízkou paměťovou náročností. Zvolil jsem ji kvůli požadavku na zvládání dlouho trvajících spojení.

Pro plné využití více vláken doporučuji serverovou aplikaci používat na Ruby implementaci Rubinius nebo JRuby, protože MRI (Matz's Ruby Implementation) v současné verzi neumožňuje běh vláken na více než jednom procesoru.

5.1.1 Framework

Samotná aplikace je vystavěna s použitím frameworku Rails ve verzi 4.2. Od verze 4 je oficiálně podporováno více-vláknové zpracování požadavků.

5.2 Uživatelské rozhraní

Při návrhu uživatelského rozhraní bylo mým cílem:

1. zachovat look-and-feel Gitlabu,
2. zobrazovat stav úloh v reálném čase.

Pro design aplikace bylo z velké části použito existujících šablon přímo z Gitlab projektu. Vlastní implementace se týká pouze zobrazování a správy úloh.

Pro úlohy jsem se rozhodl využít protokolu WebSocket. Samotné HTML stránky se negenerují na serveru, jako je zvykem, ale na straně klienta na základě informací obdržených v update streamu (WebSocket). Díky tomu je stránka schopná reagovat na jakoukoliv událost v reálném čase bez nutnosti její aktualizace uživatelem.

Na rozdíl od serverového renderu umožňuje tento způsob implementace efektivně vyřešit problém počátečního stavu, kdy díky velké prodlevě mezi samotným renderem do HTML a registrací k update streamu mohlo dojít ke ztrátě dat. To se stávalo zejména při zobrazování výstupu aplikace, kdy aplikace mohla v čase pár desítek milisekund vygenerovat i několik řádků výstupu, které se k uživateli díky zmíněné prodlevě vůbec nedostaly.

Uživatel může mít běžně naplánovaných i více než 100 úloh. Zasílání výstupu takového množství úloh pro všechna sezení by bylo velmi neefektivní. Proto je implicitně v update streamu zasílána pouze informace o změnách stavu úloh. Samotné přihlášení k odběru (subskripce) výstupu spuštěné aplikace probíhá až na základě uživatelského požadavku. K tomu využívá aplikace obousměrné komunikace po otevřeném WebSocket spojení. Odpadá tak nutnost složitě evidovat kanály a vše probíhá naprosto přirozeně - v rámci kontextu jednoho spojení. Díky tomu dochází i k redukci zatížení serveru dalšími požadavky a nutnosti alokovat další pracovní vlákno a jeho explicitní synchronizaci pro provedení subskripce.

Protokol WebSocket je v rámci aplikace implementován pomocí hijackingu. Socket Hijacking je jednou z novinek, kterou přineslo rozšíření Rack ve verzi 1.5. Jedná se o možnost přímého sdílení socketu, který byl otevřen HTTP serverem. Web server jednoduše přenechá aplikaci plnou kontrolu nad samotným spojením. Stará se pouze o jeho alokaci a uvolnění, zatímco aplikace má k socketu přístup pro zápis i pro čtení.

5.3 Datový model

Datový model aplikace je velmi úzce spjat se službami, které aplikace využívá. Většina dat totiž není uložena přímo v aplikaci, ale je načítána externě.

V prvotních verzích aplikace jsem prováděl persistenci všech použitých dat, ale později se ukázalo, že to nepřináší žádná významná pozitiva. Jelikož se předpokládá, že aplikace poběží na stejném stroji jako Gitlab, není výkonnostní propad natolik velký, aby to vyvážilo složitost modelu. Naopak dochází k nekonzistenci mezi aplikacemi. Raději jsem se proto rozhodl zapracovat na řádném cachování použitých dat.

5.3.1 Gitlab

Pro snazší integraci je umožněno pracovat přímo v kontextu Gitlab projektů. Projekty nejsou nijak ukládány. Pro načítání je využíváno služeb Gitlab API.

Při každém přihlášení uživatele dochází ke stažení seznamu jeho projektů, který je udržován v nacachovaném stavu v rámci uživatelského sezení. Vyjma cachování na úrovni uživatelského sezení také dochází ke cachování v globálním kontextu (např. informace o detailech projektů apod.).

Cachování je implementováno pomoci `ActiveSupport::Cache`. V současné době je využit in-memory adapter, ten lze v případě potřeby nahradit za jakýkoliv jiný, pokročilejší.

5.3.2 Vlastní data aplikace

Kromě projektových informací, které jsou sdílené s Gitlabem, je potřeba zajistit uchování vlastních nastavení, která obsahují informace potřebné pro sestavení projektu. Tato konfigurace obsahuje především deploy klíče a definici parametrů pro Image Builder.

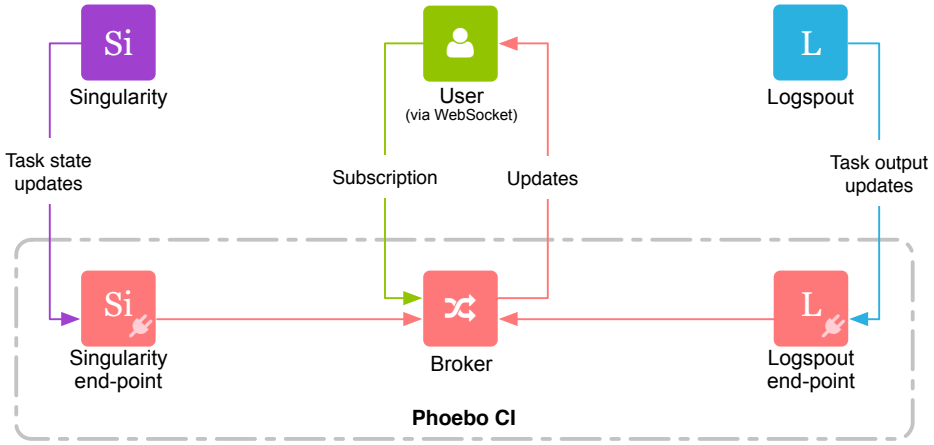
Uložení aplikačních dat je provedeno pomocí modelu vystaveném na `ActiveRecord`. Pro fyzické uložení dat jsem zvolil databázový server PostgreSQL. Aplikace má ve skutečnosti velmi malé nároky na DB. Pro tuto volbu jsem se rozhodl zejména proto, že PostgreSQL server bude zpravidla již na serveru k dispozici kvůli běžícímu Gitlabu. Díky `ActiveRecordu` ale není problém využít i jakoukoliv jinou SQL databázi bez změny jediného řádku kódu.

5.3.3 Úlohy a požadavky na sestavení

Všechny informace o běžících úlohách jsou uloženy pouze uvnitř Singularity. Potřebná specifická data (nad rámec informací potřebných pro deploy) jsou uložena pomocí metadata požadavků (deploy request).

Data jsou načítána při každém spuštění aplikačního serveru pomocí startup procesu a následně jsou udržována v konzistentním stavu pomocí webhooků (viz. dále).

Data jsou uloženy pouze v paměti a spravována pomocí Brokeru.



Obrázek 5.2: Zjednodušený diagram komunikace Brokeru

5.4 Broker

Srdcem celé aplikace je implementace Brokeru. Broker se stará o distribuci událostí mezi jednotlivými vlákny (požadavky) aplikace a o správu in-memory dat.

Broker plní klíčovou funkci při obsluze update streamu uživatelského rozhraní (WebSocket) - implementuje tzv. Pub/Sub model.

V prvních verzích aplikace jsem pro tyto účely využíval Pub/Sub model Redisu. Později se ale ukázalo, že tato implementace není ideální a rozhodl jsem se proto pro vlastní řešení.

Hlavním kamenem úrazu při komunikaci pomocí Redisu je, že vyžadoval výhradní spojení pro každý update stream. Subscribe je v Redisu blokující operace, a tudíž s sebou nese nutnost vytvoření vlastního vlákna pro každé nové spojení. Ani to ale nestačilo, protože Redis v rámci implementace Pub/Sub modelu neumožňuje čtení jiných klíčů při aktivní subskripci. To znamená, že pro každé spojení muselo existovat:

- jedno vlákno pro komunikační zpracování - to je dáno použitím vícevláknového serveru, na kterém je aplikace postavena
  (tzn. 1 spojení = 1 vlákno),
- jedno vlákno pro subskripci pro Redis, které obsahovalo exkluzivní spojení pro subskripci a druhé spojení pro čtení počátečního stavu.

Podobná režie by se mohla stát velmi rychle kritickou i při připojení menšího množství klientů. Z toho důvodu jsem se rozhodl Redis úplně vypustit a nahradit jej vlastním řešením.

Broker nevyžaduje pro běh žádné vlastní vlákno. Díky nízké složitosti implementace jednotlivých požadavků, lze vše pohodlně zvládnout pouze v komunikačních vláknech. V případě asynchronní potřeby lze do budoucna worker thread lehce implementovat bez jakékoliv změny vázaného kódu.

5.4.1 Mezi-vláknová synchronizace

Při návrhu mezi-vláknové synchronizace bylo potřeba odhadnout četnost jednotlivých požadavků. Funkce vyžadující zápis probíhají pouze při vytvoření nového spojení nebo při změně atributů úlohy, což jsou oboje řádově omezená čísla. Čtení je pro webovou aplikaci mnohonásobně četnější. Implementace s tím počítá a provádí explicitní zamykání pouze při zápisu. Díky tomu, a využití immutable objektů / kolekcí, je model Brokeru plně thread-safe, a to při zachování optimální efektivity.

5.4.2 Správa dat

Vyjma distribuce událostí je Broker zodpovědný také za udržování konzistentního stavu dat úloh pro všechna vlákna. Nabízí rutiny pro atomický update atributů úloh, přičemž se stará o zamykání (pokud je potřeba) a notifikaci přihlášených Subscriberů o provedené změně.

```ruby
# Update task state of #123
broker.update_task(123) do |task|
  task.state += 1
  task.state_message = 'Task finished'
end
```

Ukázka 5.4.1: Ukázka API pro update atributů úlohy

```ruby
# Create new subscriber
subscriber = broker.new_subscriber

# Handle task updates
subscriber.handle :task_update do |new_task, old_task, diff|
  puts "Task ##{new_task.id} updated:"
  diff.each do |key, value|
    puts "#{key}: #{old_task[key]} -> #{value}"
  end
end

# Subscribe for updates to all tasks from project #1
subscriber.subscribe_for :task_update, project_id: 1
```

Ukázka 5.4.2: Ukázka registrace subscriberu

5.5 Startup proces

Vzhledem k tomu, že je většina dat načítána z externích služeb, je vhodné si některá pro naši aplikaci předem načíst, abychom dosáhli lepšího uživatelského pohodlí.

V rámci startup procesu se aplikace stará o:

1. **Načtení seznamu běžících úloh ze Singularity frameworku** - tímto krokem dojde k inicializaci Brokeru.

2. **Registrace webhooku pro Singularity** - webhook zaručí, že aplikace bude notifikována při každé změně, která se v Singularity odehraje. Webhook slouží zejména k tomu, abychom udržovali data v Brokeru konzistentní s jejich stavem v Singularity.

3. **Naplánování spuštění Logspout procesu** pro každého z Mesos slave hostů. Proces je naplánován jako úloha pomocí Singularity frameworku.

Startup proces probíhá ve svém vlastním vlákně, které je skartováno po jeho doběhnutí. Je tomu proto, aby mohl webový server po dobu startu aplikace obsluhovat HTTP požadavky (jako je např. zmíněný webhook). Zároveň to umožňuje informovat uživatele o potenciální chybě v podobě komfortního prostředí webové aplikace.

5.6 API

5.6.1 Singularity end-point

Aplikace nabízí API end-point pro webhooky Singularity frameworku. Singularity nabízí notifikaci pro následující druhy událostí:

1. `REQUEST` - notifikace o změně požadavků, které je využíváno v případě, že byl požadavek smazán externě (např. z UI Singularity nebo Mesosu)

2. `DEPLOY` a `TASK` - notifikace o změně stavu úlohy, které je využíváno pro aktualizaci stavu úlohy a jejích atributů.

Při zpracování požadavků je důležité dbát na to, abychom kontrolovali jejich správnou posloupnost. Z podstaty síťové komunikace se totiž může snadno stát, že nám dorazí notifikace v nesprávném pořadí - např. prodleva mezi `TASK_LAUNCHED` a `TASK_RUNNING` je v podstatě neexistující a často se stává, že je notifikace pro `TASK_RUNNING` zpracována jako první. Této kontroly je docíleno pomocí implementace stavového automatu uvnitř modelu `Broker::Task`.

5.6.2 Logspout end-point

Logspout end-point slouží k příjmu výstupu spuštěných procesů. Výstup je získáván pomocí samostatného procesu Logspout na jednom z Mesos slave hostů (viz. kapitola č. 7). Proces se připojuje na Logspout end-point aplikačního serveru a zasílá data pomocí keep-alive spojení využívajícího WebSocket protokolu.

Zpracování je řešeno úplně stejně jako v případě update streamu. Veškerá přijatá data jsou zparsována a předána Brokeru.

5.6.3 Image Builder end-point

Image Builder end-point slouží k definici úloh, které jsou načteny z konfiguračního souboru `Phoebofile`, který se nachází uvnitř projektu (v rámci repozitáře). Tato definice je zasílána Image Builder procesem, který je rozebrán v kapitole 6.

5.6.4 Zabezpečení API

API předpokládá využití protokolu HTTPS pro šifrování samotného přenosu dat a validace identity komunikačních stran. Díky jeho použití dochází k omezení možnosti replay útoku. Samotné end-pointy jsou zabezpečeny 16-znakovým hexadecimálním heslem předávaným společně s požadavkem. Pro webhooky (Singularity a Logspout) je toto heslo generováno náhodně při jejich registraci. Pro Image Builder end-point je toto heslo vygenerováno pro každý požadavek na sestavení a zároveň dochází k zneplatnění hesla, pokud již byl požadavek zpracován.

V případě, že dojde k použití nesprávného klíče vrací API chybový kód a dochází ke zdržení odpovědi na požadavek o 1s. Pro prolomení hesla by v tomto případě bylo potřeba 16^{16} vteřin, což je řádově více, než je samotná životnost požadavku (a očekáváný uptime aplikačního serveru).

Aby ale tato ochrana byla účinná je důležité uchovávat heslo v tajnosti. Je třeba zajistit, že žádný potenciálně nebezpečný proces nebude mít přístup k API Singularity ani Mesosu, kteří toto heslo z principu věci znát musejí.

5.7 Autentifikace a autorizace uživatele

Aplikace využívá k autentifikaci služeb externího poskytovatele. Konkrétně OAuth API přímo Gitlabu. Díky tomu má aplikace přehled, které projekty je uživatel oprávněn vidět a spravovat. Dodatečně je uživatelům, kteří jsou administrátory Gitlabu, umožněna i kompletní správa CI systému.

5.8 Testování

Aplikace je pokryta unit testy pomocí testovacího frameworku RSpec. Testovací balík obsahuje testy modelu, controllerů i cest (routes).

Chování oproti externím službám je testováno za pomocí Mock objektů, aby byla zajištěna řádná reakce ve všech možných stavech.

Kapitola 6

Implementace: Image Builder

Základním kamenem navrhovaného CI systému je nástroj pro vytváření Docker Image (dále Image Builder). Hlavním úkolem Image Builderu je převedení požadované verze aplikace do podoby spustitelné Mesos exekutorem.

Při návrhu nástroje jsem si kladl následující požadavky:

1. **Uniformní chování procesu** - proces Image Builderu by se měl z pohledu CI systému chovat stejně jako kterýkoliv jiný naplánovaný proces - tzn., že by měl být spustitelný v rámci zaváděné architektury (na některém z dostupných Mesos slaves) a správa a dohled nad procesem by měla být dostupná z webového rozhraní aplikace.

2. **Robustní definice obrazu** - recept pro vytváření obrazů by měl být co nejlépe rozšiřitelný, aby se zamezilo jeho opakování napříč projekty. Měl by být srozumitelný pro programátora a dobře testovatelný.

3. **Lokální vývoj** - kromě samotného běhu v rámci CI procesu by měla být aplikace schopná běžet i přímo na stroji vývojáře. Je nezbytné, aby vývojář měl možnost ověřit výsledný obraz ještě před commitem projektu na server.

4. **Izolované prostředí** - aplikace by měla být schopná běhu uvnitř kontejneru a měla by klást minimální nároky na vnější závislosti.

6.1 Popis procesu sestavení obrazu

6.1.1 Získání zdrojového kódu

V prvním kroku sestavení dojde k vytvoření pracovní kopie požadované verze zdrojového kódu aplikace. Image Builder umožňuje použití lokálního adresáře (pro snadný vývoj a testování) a nebo stažení zdrojového kódu ze vzdáleného Git repozitáře (pro využití v rámci CI). Pro stažení Git repozitáře lze volitelně provést autentifikaci pomocí SSH klíče.

6.1.2 Zpracování konfiguračního souboru

Po získání pracovní kopie zdrojového kódu aplikace dojde k načtení konfiguračního souboru `Phoebofile`. Tento soubor obsahuje specifikaci pokynů k sestavení výsledného obrazu a seznamu úkolů, které se předají CI systému k naplánování, jakmile bude obraz k dispozici.

6.1.3 Sestavení obrazu

Z receptu uvedeného v konfiguračním souboru `Phoebofile` je sestaven tzv. *Docker Tarball*. To není nic jiného než TAR archiv obsahující všechny soubory potřebné k vytvoření obrazu a vygenerovaný *Dockerfile*. Tento archiv je pomocí Docker API předán Docker deamonu ke zpracování.

6.1.4 Publikace obrazu

Po dokončení sestavení je obrazu přiřazen tag s pojmenováním projektu a jeho verzí. Celý výsledný obraz je pak publikován na *Docker Registry* dle specifikovaných parametrů.

6.1.5 Publikace seznamu úkolů

V posledním kroce dojde k sestavení seznamu úkolů, které budou na daném obrazu spuštěny. Tento seznam je předán v podobě REST požadavku zpět naší CI aplikaci.

6.2 Popis konfiguračního souboru

Konfigurační soubor `Phoebofile` je implementován jako DSL v jazyce Ruby. Tento formát jsem zvolil proto, že dává programátorovi velkou volnost konfigurace. Ta nemusí být pouze deklarativní seznam kroků, ale může být dynamicky závislá na stavu aplikace (může např. využít části aplikační konfigurace, Gemfile, atd.).

Díky DSL odpadá jakákoliv nutnost vytváření vícero konfiguračních souborů pro různá prostředí, ale lze vše přehledně definovat v jediném parametrizovaném předpisu.

Konfigurační soubor je zpracováván v izolovaném prostředí Image Builderu (každý build běží ve svém vlastním kontejneru) a proto se nemusíme bát, že by potenciální škodlivý kód mohl jakkoliv ovlivnit funkčnost jiného projektu než toho, ke kterému má již jeho autor přístup.

Pro maximální flexibilitu není konfigurační předpis nijak omezen. Může obsahovat libovolnou konstrukci jazyka Ruby včetně definicí tříd a metod (ať už přímo v souboru nebo externě). Veškeré konfigurační direktivy uzavíráme do následujícího bloku:

```
Phoebo.configure(1) {
    # ...
}
```

Ukázka konfigurace 6.2.1: Definiční blok DSL

Dopředná kompatibilita formátu je zajištěna explicitním verzováním uvedeným parametrem.

6.2.1 Definice obrazů

Konfigurační soubor slouží primárně pro definici předpisů k vytváření obrazů. Každá definice musí obsahovat název rodičovského obrazu (base image) a seznam kroků jak vytvořit požadovaný obraz.

```
Phoebo.configure(1) {
  image('phoebo/nodejs-example', from: 'gliderlabs/alpine:3.1') {
    run('apk', '--update', 'add', 'nodejs')
    add('./app', '/app')
  }

  # ...
}
```

Ukázka konfigurace 6.2.2: Definice obrazu

6.2.1.1 Příkazy

V základní verzi Image Builderu jsou obsaženy následující příkazy:

`add(source_path, destination_path)`
: pro přidání existujícího souboru do výsledného obrazu

`create(destination_path, content)`
`create(destination_path, &block)`
: pro vytvoření nového souboru s daným obsahem nebo pomocí bloku. Blok dostává jako parametr output stream, takže je možné tímto způsobem vytvářet i binární soubory a mít vše plně pod kontrolou.

`run(command, *args)`
: pro spuštění příkazu uvnitř kontejneru.

6.2.1.2 Defince vlastních příkazů

Ačkoliv lze pomocí základních příkazů vytvořit libovolnou Docker Image, mohlo by časem docházet k tomu, že se velké části konfiguračních souborů začnou napříč projekty opakovat. Tomu lze pochopitelně předejít vytvořením vlastní base image. Bylo by ale také vhodné, aby Image Builder umožnil automatizovat některé komplexnější úlohy, jako je např. generování app-specific konfigurace apod.

Pro tyto účely nabízí Image Builder jednoduché rozhraní, jakým lze přidat podporu pro další příkazy. Stačí v adresáři `lib/phoebo/config/image_commands` vytvořit vlastní soubor s definicí třídy dle následujícího předpisu:

```
module Phoebo::Config::ImageCommands
  class MyExtension
    def self.id
      :my_command
    end

    def self.action(myparam)
      return Proc.new do |build|
        build << "RUN some_script.sh --with #{myparam}"
      end
    end
  end
end
```

Ukázka konfigurace 6.2.3: Definice vlastních příkazů

```
Phoebo.configure(1) {
  image('phoebo/nodejs-example', from: 'gliderlabs/alpine:3.1') {
      my_command('foo')
  }
  # ...
}
```

Ukázka konfigurace 6.2.4: Použití vlastního příkazu

6.2.1.3 Plugin systém

Do budoucna plánuji vytvořit plugin systém, který umožní přidávat nové příkazy i bez jakéhokoliv zásahu do existujícího kódu. Ruby pro to nabízí prostředek v podobě nástroje Bundler. Inspirací pro toto řešení jsem hledal v aplikaci Vagrant.

Jednotlivé zásuvné moduly budou jednoduše publikované gemy implementující vlastního potomka `Phoebo::Plugin`. Cílem plugin systému je, abychom mohli závislost specifikovat přímo v souboru `Phoebofile` a nemuseli ji řešit předem. Image Builder by si potřebný plugin nainstaloval sám na vyžádání. Celá instalace by probíhala uvnitř izolovaného prostředí kontejneru, tedy by nemohlo docházet k žádným konfliktům. Definice bude vypadat nějak takto:

```
Phoebo.configure(1) {
  require_plugin('jnovak/my_phoebo_plugin:1.0')

  image('phoebo/nodejs-example', from: 'gliderlabs/alpine:3.1') {
      my_command('foo')
  }

  # ...
}
```

Ukázka konfigurace 6.2.5: Ukázka použití pluginu v nadcházející verzi

Navrhované řešení umožňuje podobnou modulárnost jako má populární systém buildpacků pro PaaS Dokku / Heroku. Řeší však jeho největší nedostatky. Pluginy mohou využít plné síly Ruby a hlavně se dají velmi dobře testovat a rozšiřovat (narozdíl od sady BASH skriptů).

6.2.1.4 Parametrizace

Při vytváření obrazu můžeme využít parametrů předávaných z prostředí CI. Tyto parametry jsou konfigurovatelné pro každý projekt v rámci webového rozhraní aplikace. Díky tomu dochází k oddělení od závislosti na běhovém prostředí.

Na ukázce č. 6.2.6 můžeme vidět, jakým způsobem lze parametrů využít např. pro to, abychom nastavili naší aplikaci přístup k externímu databázovému serveru.

```
require 'yaml'

Phoebo.configure(1) { |phoebo|
  image('phoebo/nodejs-example', from: 'gliderlabs/alpine:3.1') {
    run('apk', '--update', 'add', 'nodejs')
    add('./app', '/app')

    # Generate config with passed secrets
    create('/app/config.yaml') do |out_stream|
      data = {
        database: {
          password: phoebo.params[:dbpassword]
        }
      }

      out_stream.write(YAML::dump(data))
    end
  }

  # ...
}
```

Ukázka konfigurace 6.2.6: Použití předaných parametrů

6.2.2 Definice úloh

Kromě definice obrazů k sestavení slouží konfigurační soubor také k definici úloh, které se mají naplánovat ke spuštění. Každý úloha obsahuje informaci o spustitelném příkazu, jeho parametrech (a volitelně názvu obrazu, pokud jich naše konfigurace obsahuje více). Úlohy jsou předány CI systému ihned, jakmile je k dispozici hotový obraz. Pokud nechceme některé úlohy provádět automaticky při každém buildu (jako je například skript generování dokumentace apod.), je možné je označit k ručnímu spuštění pomocí `on_demand: true`. Takto označené úlohy budou založeny v systému, ale budou čekat na akci uživatele.

```
Phoebo.configure(1)
  task('Run unit tests',
    image: 'phoebo/nodejs-example',
    command:   'rspec',
    arguments: 'spec/*',
    on_demand: true
  )
}
```

Ukázka konfigurace 6.2.7: Definice úlohy

6.2.3 Definice služeb

Definice služeb je velmi podobná běžným úlohám. Liší se pouze způsobem jakým se k definovaným procesům přistupuje ze strany plánovače. Běh služeb je kontrolován a v případě nedostupnosti služby je proveden její automatický restart. Proces lze v tomto ohledu spíše připodobnit k init skriptům. Konfigurace má navíc parametr `:ports`, pomocí kterého lze předat seznam portů, které má plánovač zpřístupnit. První z definovaných portů také slouží pro tzv. *healthcheck* běhu služby.

```
Phoebo.configure(1) { |phoebo|
  # Run NodeJS application as a service and expose it on port 80
  service('Web server',
    command:   '/usr/bin/node',
    arguments: '/app/main.js',
    ports: [
      { tcp: 80 }
    ]
  )
}
```

Ukázka konfigurace 6.2.8: Definice služby

6.3 Komunikace s Dockerem

Pro vytvoření Docker Image využívá aplikace Image Builder spojení přímo s Dockerem. Docker má pro tyto účely k dispozici API dostupné na síťovém rozhraní nebo unix socketu. Samotná CLI utilita `docker` je jen thin client využívající API Docker daemonu. Díky této architektuře je velmi snadné napsat aplikaci, která bude využívat existujících služeb Dockeru. Znamená to ale, že musíme mít pro naše účely běžící Docker daemon, což nám situaci poněkud komplikuje, pokud chceme aplikaci pouštět v izolovaném prostředí.

Jelikož chceme, aby aplikace Image Builder běžela ve svém vlastním kontejneru, máme v podstatě dvě možnosti: umožnit aplikaci přístup ke sdílenému Docker daemonu běžícím na hostu a nebo zajistit aby měla aplikace k dispozici svůj vlastní Docker daemon uvnitř kontejneru.

6.3.1 Sdílení Dockeru s hostem

Nejjednodušší pro implementaci je přímé sdílení Docker daemona s hostem. Daemon v tomto případě poslouchá na nějakém TCP portu nebo, v lepším případě, je nasdílen unix socket dovnitř kontejneru jako sdílený svazek (volume).

Problém tohoto řešení spočívá zejména v tom, že kontejnerizace našeho Image Builderu je téměř zbytečná. V případě, že by DSL obsahovalo škodlivý kód umožníme útočníkovi provést jakoukoliv akci nabízenou Docker daemonem. Může tak např. spustit libovolný kontejner s připojeným síťovým svazkem, který mu nenáleží, nebo přepsat existující obraz Image Builderu, za nějaký jiný - škodlivý, který by mohl např. logovat veškeré SSH klíče a odesílat je útočníkovi.

Tento problém by šlo do jisté míry vyřešit zabezpečením pomocí nějaké proxy vrstvy, která by povolila pouze příkazy potřebné k sestavení obrazu (`/build`) [6].

Kromě samotného problému bezpečnosti ale také dochází k nechtěnému zatěžování hostu. Celý proces vytváření obrazu totiž probíhá na straně daemonu, tedy s neomezenými výpočetními prostředky. V takovém případě si může jedna úloha snadno přivlastnit všechen výpočetní výkon hostu a omezit provoz ostatních služeb.

6.3.2 Vlastní instance Dockeru

Od verze 0.6 nabízí Docker `--privileged` flag, který umožňuje přístup zevnitř kontejneru k zařízením hostu (a k částečnému nastavení AppArmoru a pravidel SELinuxu). Izolovaný proces má tak téměř stejné pravomoce, jako kdyby běžel na hostu samotném [7].

Toho můžeme využít k tomu, abychom provozovali Docker uvnitř Dockeru (DinD). Díky tomu může mít každá instance Image Builderu kompletně izolovanou vlastní instanci Docker daemonu.

Kromě běhu v `--privileged` režimu je pro běh Dockeru potřeba provést rekurzivní připojení hierarchie *cgroups* do kontejneru a uvolnit přebytečné file deskriptory, které mohly vzniknout při běhu rodičovského procesu.

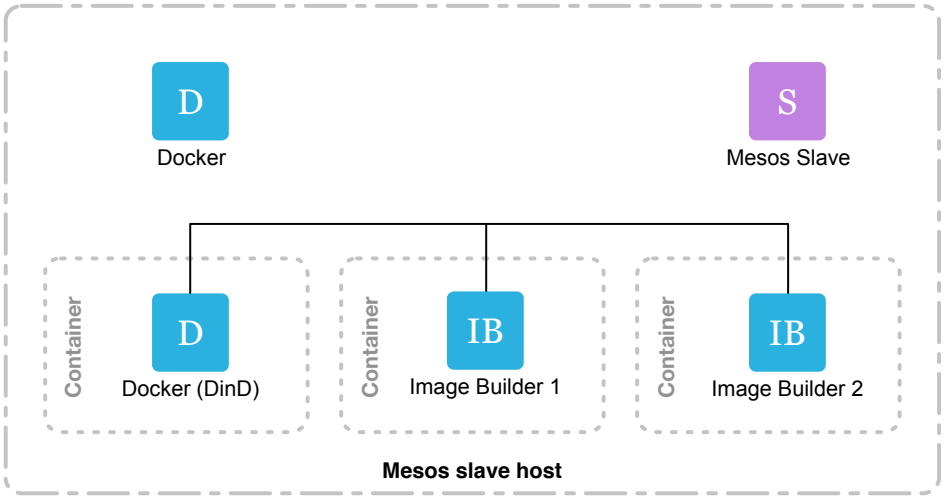
Ani toto řešení není ale zcela bez problému. Při používání *DinD* jsem narazil na problém, kdy nedocházelo ke správnému uzavírání *loopback devices* a po několika spuštěních tak docházelo k jejich vyčerpání. Děje se tomu především proto, že kontejner nemá ve skutečnosti žádný init systém, který by se postaral o to, aby po doběhnutí aplikace navrátil vše do původního stavu. To lze ale částečně vyřešit opatrným přidělováním prostředků a jejich explicitním uvolňováním v rámci wrapper skriptu [8].

6.3.3 Implementace v rámci Image Builderu

Při implementaci Image Builderu jsem se snažil o to, aby šlo nástroj provozovat v obou dvou situacích. Aplikace tedy obsahuje upravený DinD wrapper skript, který umožňuje plně izolovaný provoz, ale lze také spustit s využitím externího standalone Dockeru. V testovacím prostředí využívám sdíleného Dockeru s hostem. Tato varianta umožňuje snížit celkové nároky na systém vývojáře a také skutečnost, že není potřeba distribuovat vytvořené obrazy. Testovací prostředí se tak obejde kompletně bez vlastního *Docker Registry*.

Pro nasazení v produkčním prostředí bych doporučil provoz **jedné** instance uvnitř izolovaného kontejneru pro každý slave. Jednotlivé instance Image Builderu by se k ní pak připojovali. Vlastní Image Builder takto může běžet s nižším oprávněním (bez `--privileged` módu) a využívat cachování kontejnerů díky samostatnému daemonu.

Provoz DinD v kontejneru společně s Image Builderem doporučuji pouze v případě, že je třeba zajistit striktní oddělení mezi projekty. Toto řešení má oproti ostatním velkou výkonnostní nevýhodu, protože dochází k vytváření obrazů **bez jakékoliv cache**. Všechny intermediate kontejnery (nebo stažené base images) budou po doběhnutí skriptu ztraceny a je třeba je stahovat / vytvářet pro každý build zvlášť. To je zcela zbytečné, neboť je větší část obrazu zpravidla neměnná a ztrácí se tak jedna z největších přednostní Dockeru.



Obrázek 6.1: Diagram navrhované produkční architektury pro Image Builder

6.4 Komunikace s CI systémem

6.4.1 Požadavek na sestavení

Pro lepší integraci do CI procesu umožňuje Image Builder převzít vstupní parametry ze vzdáleného serveru pomocí HTTP požadavku (nebo z lokálního souboru). V rámci CI je toho využito zejména proto, že je problematické předávat některá rozsáhlejší data, jako jsou např. SSH klíče, pomocí parametrů Mesos frameworku. Některé frameworky sice umožňují automatické stahování tzv. artefaktů, ale na straně CI je toto stejně třeba vyřešit a nabídnout pro ně vlastní end-point. Proto jsem se rozhodl zaintegrovat vše potřebné přímo do aplikace a dosáhnout tak lepší nezávislosti na použitém frameworku. Kromě předávání SSH klíčů je tento přístup použit i pro předávání objektu parametrů, které lze při vytváření obrazu využít (viz. ukázka konfigurace č. 6.2.6).

```
{
  "id": "2e2996fe8420",
  "repo_url": "ssh://gitlab.fit.cvut.cz/phoebo/nette-example.git",
  "ssh_public": "ssh-rsa AAAAB3NzaC1yc2E...",
  "ssh_private": "-----BEGIN RSA PRIVATE KEY----- ...",
  "docker_user": "joe",
  "docker_password": "secret123",
  "docker_email": "joe@domain.tld",
  "params": {
    "dbpassword": "soemSecret11"
  }
}
```

Ukázka 6.4.1: Požadavek na sestavení (JSON)

6.4.2 Ping back

Po dokončení sestavení obrazu, je zaslána zpět informace o úlohách, které se mají na daném obraze spustit.

```
{
  "id": "2e2996fe8420",
  "tasks":[
    {
      "name":"Web server",
      "service": true,
      "image":"phoebo/nette-example",
      "command": "/usr/bin/node",
      "arguments": [
        "/app/main.js"
      ],
      "ports": [
        {
          "tcp": 80
        }
      ]
    }
  ]
}
```

Ukázka 6.4.2: Ping back (JSON)

6.5 Lokální vývoj

Aplikaci lze nainstalovat jako gem přímo na stroji vývojáře. Jedinou závislostí v tomto případě je existence běžícího Docker daemonu (bude se využívat přímo lokální instalace).

Možnost lokální instalace je určená k psaní a testování konfiguračního souboru `Phoebofile`.

```
gem install phoebo
```

Ukázka 6.5.1: Instalace Image Builderu

Kapitola 7

Implementace: Logspout

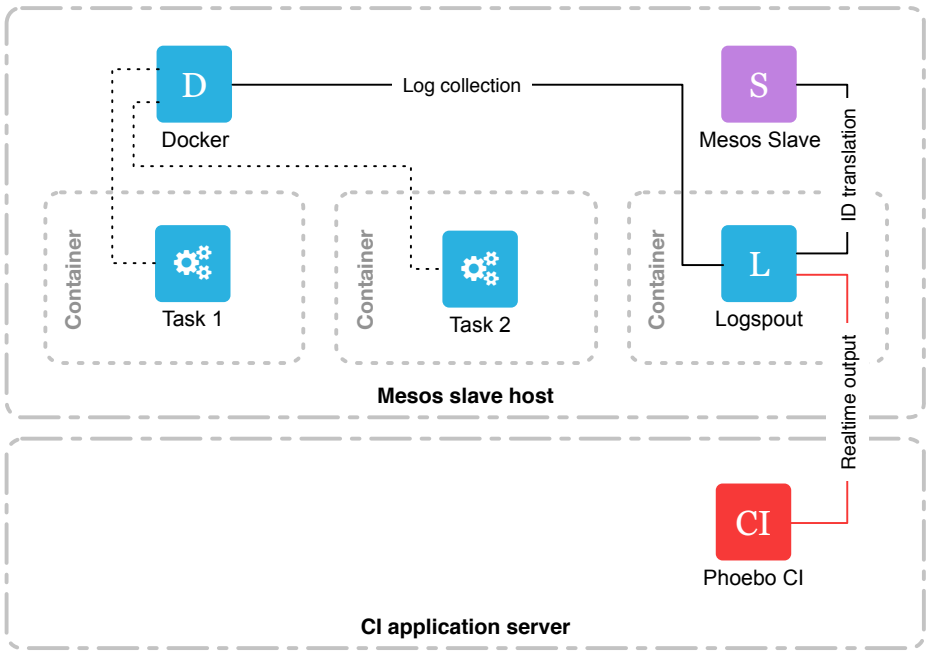
Jednou z klíčových vlastností jakéhokoliv CI systému je zajištění zpětné vazby spouštěného procesu uživateli. Díky využití Mesos frameworku máme v rámci CI k dispozici informaci o aktuálním stavu úlohy. Zároveň nám Mesos executor ukládá zachycený standardní výstup programu (stdout a stderr) do tzv. sandboxu aplikace.

Sandbox je ve skutečnosti připojený svazek z `/tmp/mesos/slaves/**` do `/mnt/mesos/sandbox` každého kontejneru spuštěného Mesosem. K jakémukoliv souboru v tomto svazku lze přistupovat pomocí API Mesosu.

Různé Mesos frameworky vytvářejí nad tímto API nadstavbu a umožňují pohodlný přístup v rámci kontextu vlastních naplánovaných úloh. Například mnou použitý framework Singularity obsahuje experimentální doplněk executoru pro tailing těchto logů. Bohužel v době psaní této práce měla tato část daleko do použitelného stavu. Rozhodl jsem se proto vytvořit vlastní implementaci.

Při návrhu implementace nebylo mým cílem pouze zachytávání logů, chtěl jsem výstup aplikace dopravovat k uživateli pokud možno v reálném čase. Abych mohl docílit něčeho podobného, nemohl jsem se už obrátit na Mesos API, ale musel jsem postavit aplikaci o úroveň níže a využít přímo API Dockeru.

Podobný princip zachycování výstupu využívá open-source nástroj Logspout. Tato aplikace se nyní těší velké popularitě díky sponzorování společností DigitalOcean. V době implementace obsahoval nástroj však pouze zlomek současné funkcionality. Provedl jsem proto vlastní fork, ve kterém jsem doimplementoval nezbytné úpravy pro integraci do CI systému.



Obrázek 7.1: Diagram zasazení aplikace Logspout

7.1 Popis aplikace

Logspout je jednoduchá aplikace napsaná jazyce Go. Běží kompletně uvnitř kontejneru, tudíž přímo padne do navrhované architektury. Jediné, co je potřeba zajistit, je její spuštění na každém Mesos slave hostu.

Aplikace využívá API Dockeru a naslouchá standardnímu výstupu všech kontejnerů. Principiálně se nijak neliší od volání `docker logs -f [Container ID]`.

Přečtené záznamy jsou po přijetí zpracovány a publikovány na REST rozhraní aplikace.

Mimo vlastního API nabízí aplikace také možnost publikovat logy přímo na externí syslog server. Logy lze třídit do uživatelsky definovaných rout a určit pro každou routu jinou destinaci.

7.2 Integrace s CI

Pro zasílání výstupu kontejneru do webového rozhraní využívá aplikace WebSocket protokol. Zvolil jsem ho zejména proto, že jej v rámci webové aplikace využívám k synchronizaci událostí s prohlížečem uživatele. Díky tomu není třeba vytvářet úplně nový end-point, ale lze sdílet velkou část kódu.

Původní verze projektu Logspout obsahovala jednoduchý server pro streaming dat přes HTTP. Pro naše využití by bylo ale vhodnější komunikaci otočit. Implementoval jsem proto WebSocket klienta, který se sám připojí k aplikačnímu serveru CI systému.

Celá komunikace probíhá v rámci jednoho persistentního spojení a na straně serveru jsou data zpracovávána uvnitř samostatného vlákna.

Jednotlivé logy jsou odesílány ve formátu JSON. Do budoucna plánuji využít vlastní binární formát pro zajištění lepší efektivity přenosu. Při testovacím provozu jsem ale nezaznamenal žádná omezení ani s využitím JSON formátu.

Napojení přímo na Docker sebou nese jedno nepříjemné omezení. Samotný Docker nemá o běžící úloze žádnou informaci. Jediné podle čeho lze jednotlivé logy na této úrovni odlišit je ID kontejneru.

Proto, abychom mohli zaslaná data využít v prostředí našeho aplikačního serveru, je třeba nejprve toto ID přeložit na nějaký srozumitelný identifikátor. V implementaci si proto udržuji překladovou tabulku ID na Mesos Task ID, kterou synchronizuji pomocí API Mesos Slave daemonu.

Dalším problémem, který ale nelze prozatím jednoduše vyřešit je, že Docker nám neumožňuje přistoupit k joined outputu kontejnerů. Můžeme zachytávat pouze separátně stdout a stderr (další FD bohužel také nejsou podporovány). Toto je bohužel nepříjemné z pohledu koncového uživatele. Aplikace totiž obvykle svůj výstup bufferují a dochází tak ke špatnému proložení výstupu. Bohužel neexistuje způsob jakým bychom mohli dostat z odděleného výstupu joined output bez časových značek. Logspout toto částečně řeší, protože data předává v reálném čase, ale i tak není výstup zcela přesný, právě kvůli zmíněnému bufferování.

Snažil jsem se tento problém vyřešit pomocí přesměrování veškerého outputu do jediného file deskriptoru, ale to bohužel v současném Mesos executoru nefunguje.

Podpora pro další FD je přislíbena v jedné z nadcházejících verzí Dockeru, doufejme tedy že se dočkáme i joined outputu.

Kapitola 8

Vývojové prostředí

Pro usnadnění vývoje a testování CI systému jsem připravil předinstalované prostředí pomocí aplikace Vagrant. Balík obsahuje sadu skriptů a konfiguračních Puppet manifestů, který lze využít k snadnému sestavení testovací infrastruktury.

Předpřipravené prostředí lze jednoduše využít k vyzkoušení vyvinutého řešení. Stačí pouze doplnit konfiguraci připojení k Gitlab serveru.

8.1 Architektura

Infrastruktura je simulována na 2 virtuálních strojích. Na Master node běží aplikační server Phoebo CI, Mesos master a Singularity. Na Slave node běží Mesos slave daemon. Jedná se tedy o mírně zjednodušenou architekturu než v navrhovaném systému představeném kapitole č. 4.

Pro sloučení Master hostu a aplikačního serveru jsem se rozhodl zejména pro to, že jinak by byly nároky pro běh třetího virtuálního stroje nepříjemně vysoké (i po sloučení, vyžadují VM k běhu přibližně 2.5 GB RAM).

Striktní rozdělení Slave a Master uzlu jsem se ale rozhodl dodržet, protože je důležité pro věrnou simulaci reálného prostředí a pomohlo mi odhalit některé komunikační nedostatky.

8.2 Technologie

Virtuální stroje jsou sestaveny pomocí aplikace Vagrant. Ta nabízí flexibilní nástroj pro vytváření a správu virtuálních strojů. Jelikož jsem vyvíjel projekt na platformě Mac OS X, použil jsem pro Vagrant Parallels provider. To však využití prostředí nijak neomezuje. Díky vagrantu jsou VM přenositelné i na jakoukoliv jinou platformu. Stačí v souboru `Vagrantfile` vybrat jiný base box pro preferovaný provider.

Prostředí bylo navrhováno pro Ubuntu Server 14.04 LTS (Trusty Tahr).

8.3 Testování

Pro testování systému jsem vytvořil 2 zkušební projekty. Prvním z nich je jednoduchá *NodeJS* aplikace. V současné době je po této platformě stále větší poptávka. Zároveň je pro testovací účely téměř ideální, protože v ní vytvořené aplikace mají velmi malé množství externích závislostí. Díky použití minimalistických linuxových distribucí (jako je např. Alpine Linux) lze dosáhnout velmi malé velikosti výsledného obrazu. Testovací aplikace má se všemi závislostmi méně než 20 MB.

Jako druhý projekt jsem zvolil o něco složitější aplikaci postavenou na PHP frameworku *Nette*. V tomto případě jsem vytvořil běhové prostředí s využitím *Apache* a *mod_php* a řešil dynamické závislosti pomocí *Composeru*.

Oba testovací projekty jsou součástí elektronické přílohy této práce.

Závěr

Cílem této práce bylo navrhnout a implementovat Continuous Integration systém jako doplněk k aplikaci Gitlab. Navržené řešení mělo umožnit distribuci úloh v rámci výpočetního svazku, čehož jsem dosáhl využitím orchestračního systému Mesos.

V rámci své práce jsem se věnoval běhu aplikací na vypočetním svazku, jejich izolaci a řízení přístupu ke spuštěným službám. Blíže jsem se seznámil s light-virtualizací pomocí Docker kontejnerů a vytvořil jsem nástroj, který se postará o sestavení jejich obrazů ze zdrojových kódů aplikací.

Při implementaci hlavní aplikace jsem vytvořil vlastní thread-safe backend a vyřešil jsem problém zobrazování stavu úloh v reálném čase.

Celkem jsem implementoval:

1. **Aplikační server CI systému**
   - jako více-vláknovou webovou aplikaci psanou v jazyce Ruby.

2. **Nástroj pro sestavování Docker obrazů**
   - jako CLI aplikaci v jazyce Ruby.

3. **Aplikaci pro přenos výstupů spouštěných aplikací**
   - program logspout, psaný v jazyce Go.

4. **Proxy vrstvu pro webový server nginx** psanou v jazyce Lua.

5. **Předpřipravené prostředí pro vývoj** s využitím nástroje Vagrant a Puppet receptů.

V rámci implementační části jsem vytvořil funkční koncept navržené architektury a ověřil jsem si možnosti její realizace. Pro produkční nasazení je však třeba dotáhnout několik aspektů.

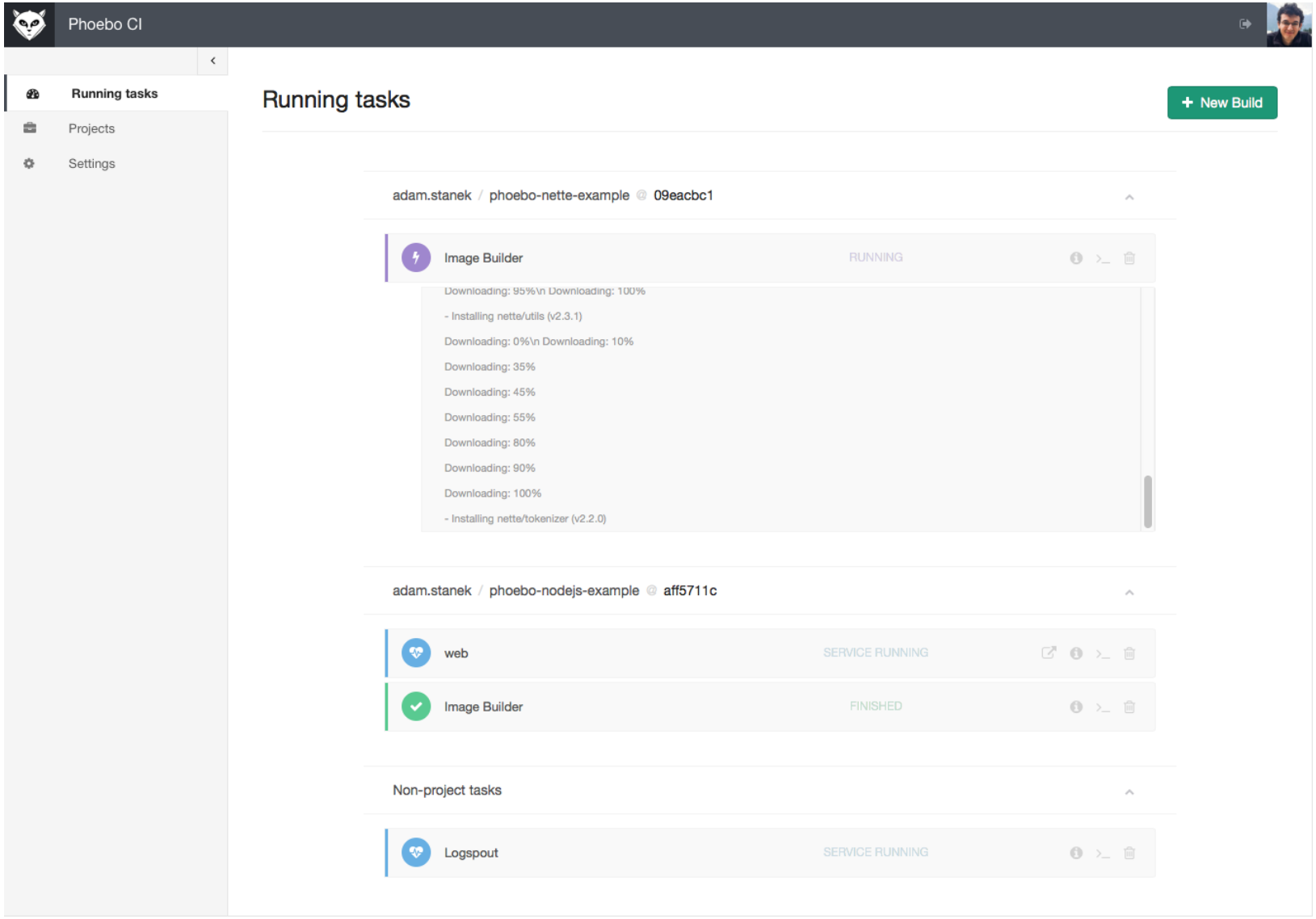
Plánuji zejména přepsat startup část aplikačního serveru s využitím některých funkcí přislíbených v nadcházejích verzích Singularity frameworku. Před

produkčním nasazením je také třeba dořešit čištění již nepotřebných kontejnerů a jejich obrazů. Toto je prozatím ponecháno na administrátorovi (lze to snadno vyřešit skriptem spouštěným v pravidelných intervalech). V rámci UI aplikace by bylo vhodné začlenit možnost přístupu k souborům uložených v sandboxu bez nutnosti odkazu do rozhraní Singularity frameworku (běžní uživatelé k němu nemají přístup). Posledním krokem by měla být integrace přímo do UI Gitlabu. Tuto možnost jsem prozatím nechal otevřenou, protože se integrační API mění s živým rozvojem Gitlab CI.

Pro nástroj Image Builder plánuji v jedné z nadcházejících verzí uvolnit plugin systém, představený v kapitole č. 6. Dále plánuji aplikaci rozšířit o další možnosti pro lokální vývoj. Konkrétně chci umožnit přímé spouštění vygenerovaných kontejnerů s volitelnou možností využití přípojných svazků na místo kopií souborů. Díky tomu bude umožněno automaticky reflektovat provedené změny v souborech bez nutnosti regenerace kontejneru. Toho bude možné využít k jednoduchému vývoji webových aplikací s využitím běhového prostředí uvnitř kontejneru.

Všechny části implementace navrhovaného řešení jsem uvolnil jako open-source a zveřejnil v repozitářích:

- `https://github.com/phoebo/phoebo`
- `https://github.com/phoebo/image-builder`
- `https://github.com/phoebo/logspout`
- `https://github.com/phoebo/vagrant`



Obrázek 8.1: Ukázka uživatelského rozhraní aplikace

Literatura

[1] AMBLER, S. W.: The Non-Existent Software Crisis: Debunking the Chaos Report. 2014. Dostupné z: http://www.drdobbs.com/architecture-and-design/the-non-existent-software-crisis-debunki/240165910

[2] CoreOS, Inc.: *Using CoreOS*. Dostupné z: https://coreos.com/using-coreos/

[3] Ellingwood, J.: An Introduction to CoreOS System Components. Dostupné z: https://www.digitalocean.com/community/tutorials/an-introduction-to-coreos-system-components

[4] Hindman, B.; Konwinski, A.; Zaharia, M.; aj.: Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. Technická zpráva, University of California, Berkeley, 2010. Dostupné z: http://mesos.berkeley.edu/mesos_tech_report.pdf

[5] Docker, Inc.: *Understanding Docker*. Dostupné z: https://docs.docker.com/introduction/understanding-docker/

[6] Docker, Inc.: *Docker Remote API v1.18*. Dostupné z: https://docs.docker.com/reference/api/docker_remote_api_v1.18/

[7] Docker announcement: Docker in Docker support. 2013. Dostupné z: https://blog.docker.com/2013/09/docker-can-now-run-within-docker/

[8] DinD Issue #19: Running out of loopback devices. Dostupné z: https://github.com/jpetazzo/dind/issues/19

Příloha **A**

Seznam použitých zkratek

API Application programming interface

Cgroups Control groups

CI Continuous integration

DinD Docker in docker

FD File descriptor

HA High availability

HTTP Hypertext transfer protocol

HTTPS Hypertext transfer protocol secure

HW Hardware

IaaS Infrastructure as a service

IPC Inter-process communication

MRI Matz's Ruby implementation

OS Operating system

PaaS Platform as a service

Pub/Sub Publisher / Subscriber model

REST Representational State Transfer

SQL Structured query language

Stderr Standard error output

Stdout Standard output

SW Software

UTS Unix timesharing system

VM Virtual machine

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `DP_Stanek_Adam_2015.pdf` ..... text práce ve formátu PDF