ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Franc** Jméno: **Martin** Osobní číslo: **420804**

Fakulta/ústav: **Fakulta informačních technologií**

Zadávající katedra/ústav: **Katedra počítačových systémů**

Studijní program: **Informatika**

Studijní obor: **Informační technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Minimalistický CI systém

Název bakalářské práce anglicky:

Minimalistic CI System

Pokyny pro vypracování:

Cílem práce je navrhnout a implementovat minimalistický CI (Continuous Integration) systém vhodný pro menší komunitní projekty.
Systém musí poskytovat následující funkcionality/vlastnosti:
- modulární podporu pro různá běhová prostředí (např. LXC kontejnery, VM),
- konfiguraci projektu pomocí textového souboru v repositáři daného projektu,
- možnost v rámci jedné úlohy spouštět více paralelních běhů s různými parametry (tzv. build matrix),
- terminálové rozhraní (přes SSH) pro přidávání projektů a základní správu,
- webové a terminálové rozhraní pro sledování stavu aktuálních i minulých úloh (vč. "streamingu" logů),
- snadnou rozšiřitelnost a integraci na další systémy.

Konkrétně:
1. Porovnejte existující open-source CI řešení.
2. Navrhněte systém splňující uvedené vlastnosti a implementujte ho v jazyce Rust, Lua, Ruby, nebo Python. Kód musí být rozumně okomentovaný a pokrytý automatizovanými testy.
3. Vytvořte uživatelskou dokumentaci a postup nasazení.

Seznam doporučené literatury:

Dodá vedoucí práce.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Jakub Jirůtka, katedra softwarového inženýrství FIT

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

Datum zadání bakalářské práce: **14.11.2016** Termín odevzdání bakalářské práce: _____

Platnost zadání bakalářské práce: _____

Podpis vedoucí(ho) práce	Podpis vedoucí(ho) ústavu/katedry	Podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

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

. _____ Datum převzetí zadání

_____ Podpis studenta

České vysoké učení technické v Praze

Fakulta informačních technologií

Katedra počítačových systémů

Bakalářská práce

Minimalistický CI systém

Martin Franc

Vedoucí práce: Jakub Jirůtka

16th of May, 2017

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl 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 16th of May, 2017 .....

České vysoké učení technické v Praze

Fakulta informačních technologií

Odkaz na tuto práci

FRANC, Martin. *Minimalistický CI systém*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato bakalářská práce pojednává o návrhu a implementaci minimalistického systému kontinuální integrace softwaru vhodného pro menší komunitní projekty, který je snadno rozšiřitelný pro různá běhová prostředí (LXC kontejner, virtuální stroj). Rešeršní část obsahuje porovnání existujících řešení s rozborem použitých technologií. Výsledkem práce je systém spravovatelný přes konfigurační soubory, terminálové SSH rozhraní a webový klient pro sledování stavu testovacích úloh. K implementaci je použit programovací jazyk Python. Vývoj systému je podpořen automatizovanými testy. Součástí práce je i uživatelská dokumentace a postup nasazení.

Klíčová slova Nahraďte seznamem klíčových slov v češtině oddělených čárkou.

Abstract

Tato bakalářská práce pojednává o návrhu a implementaci minimalistického systému kontinuální integrace softwaru vhodného pro menší komunitní projekty, který je snadno rozšiřitelný pro různá běhová prostředí (LXC kontejner, virtuální stroj). Rešeršní část obsahuje porovnání existujících řešení s rozborem použitých technologií. Výsledkem práce je systém spravovatelný přes konfigurační soubory, terminálové SSH rozhraní a webový klient pro sledování stavu testovacích úloh. K implementaci je použit programovací jazyk Python. Vývoj systému je podpořen automatizovanými testy. Součástí práce je i uživatelská dokumentace a postup nasazení.

Keywords Nahraďte seznamem klíčových slov v angličtině oddělených čárkou.

Obsah

Seznam ukázek kódu

Úvod

Programování a procesy s ním spojené prochází neustálým vývojem, který tuto disciplínu stále mění. Přidáváním vyšších vrstev abstrakce se vzdalujeme od stroje samotného a přecházíme k simulaci fungování reálného světa, čímž se stává vývoj softwaru neustále rychlejším a dynamičtějším.

Jsou to také nově vznikající vývojové techniky, které urychlují vývoj softwaru. Jednou z nich je i technika kontinuální integrace (zkráceně CI). Tato technika spočívá v časté integraci kódu mezi vývojáři ve společném repozitáři a následným testováním. Mezi testy například patří:

1. Integrační a jednotkové testy kontrolující funkčnost programu
2. Testy kontrolující jednotný styl kódu v projektu
3. Syntaktické testy

Kontinuální integrace je úzce spjatá s verzovacím systémem, který spravuje repozitář projektu. Samotný systém kontinuální intgrace, lze definovat jako množinu softwaru realizující následující úkony:

1. Naslouchat a reagovat na změny v repozitáři projektu
2. Připravit běhové prostředí vhodné pro testovací proces
3. Stáhnout zdrojové soubory projektu z repozitáře
4. Sestavit stážený projekt včetně jeho závislostí

5. Spustit testovaní dle zadání projektu
6. Informovat vývojáře a další systémy o výsledku testování

Kapitola 1

Běhová prostředí

Prostředí pro integraci by mělo být:

- Stejné na začátku každé integrace
- Odolné vůči vnějším vlivům prostředí
- Izolované proti neoprávněným zásahům do vnějšího systému

Těchto požadavků lze dosáhnout virtualizovaním prostředí pomocí virtualizace. Virtualizace je technika při které se jeden fyzický prostředek transformuje na několik virtuálních prostředků. Fyzický prostředek slouží jako hostitel pro virtuální prostředky.

1.1 Plná virtualizace

Virtualizace je technika při které se jeden fyzický prostředek transformuje na několik virtuálních prostředků. Fyzický prostředek slouží jako hostitel pro virtuální prostředky.

Výsledkem plné virtualizace je virtuální stroj, který simuluje hardware, na kterém je možné provozovat další instanci operačního systému. Prostředníke mezi virtuálním strojem a hostitelským systémem se nazývá hypervizor.

Mezi úkoly hypervizoru patří:

- Stejné na počátku každé integrace

- Odolné vůči vnějším vlivům prostředí
- Izolované proti neoprávněným zásahům do vnějšího systému

Virtualizační programy: VirtualBox, QEMU

1.1.1 Softwarová virtualizace

Softwarová virtualizace funguje na principu úpravy instrukcí virtualizovaného stroje. Je nutné přepsat privilegované instrukce na neprivilegované a upravit přístupy do paměti.

1.1.2 Hardwarové asistovaná virtualizace

Hardwarové asistovaná virtualizace funguje na principu simulace hardwaru za podpory specializovaných instrukcí procesoru. Na tomto simulovaném hardwaru je pak možné spustit další instanci operačního systému, která bude kompletně odstíněna od svého hostitele. Nevýhodou tohoto řešení je nutnost alokace systémových prostředků na běh dalšího operačního systému i samotnou simulaci hardwaru.

1.2 Virtualizace na úrovni operačního systému

Systémová paměť moderních operačních systémů je dělena na dvě části: prostor jádra a uživatelský prostor. Prostor jádra je přímo využíván pro běh jádra operačního systému a jeho služeb. V uživatelském prostoru běží procesy spuštěné uživatelem bez přístupu do prostoru jádra. Uživatelské procesy komunikují s prostorem jádra pouze skrz systémové metody. Na x86 procesorech se k rozdělení paměti používá chráněný režim, ve kterém lze stránky virtuální paměti rozdělit podle privilegovanosti.

Virtualizace na úrovni operačního systému je virtualizační technika, která dovoluje koexistenci několika oddělených uživatelských režimů, které fungují nad stejným prostorem jádra. Izolovanému uživatelskému prostoru se často označuje jako tzv. „kontejner“.

Výhodou je menší režie oproti plné virtualizaci, protože už není nutné spouštět celou novou instanci operačního systému. Oproti plné virtualizaci je naopak

nutné zajistit, aby se jednotlivé uživatelské režimy nemohly neoprávněně ovlivňovat. Toto je netriviální úloha a proto je vhodné zavést některá bezpečností opatření. Jedním z opatření je vytvoření kontejneru pod neprivilegovaným uživatelem (ne-root, ne-administrátor), takže případní „únik“ z kontejneru nezpůsobí velkou nebo žádnou škodu. Dalším opatřením může být spuštění nové instance operačního systému pomocí plné virtualizace, která bude využívána pro běh kontejnerů. Únikem z kontejneru se pak útočník dostane pouze do virtualizovaného operačního systému bez možnosti ovlivnit hostitele.

Mezi technologie pro tvorbu kontejnerů patří:

- Jmenné prostory a kontrolní skupiny (Linux)
- Jails (BSD)
- Hyper-V

1.2.1 Nástroje operačního systému Linux

Linux kernel sám o sobě nenabízí přímé kontejnerové řešení, ale pouze sadu nástrojů, pomocí kterých je kontejner vytvořen. Jedná se o jmenné prostory a kontrolní skupiny. Nad těmito nástroji je postaveno například kontejnerové řešení LXC nebo Docker.

1.2.1.1 Jmenné prostory (namespaces)

Nástroj Linux kernelu sloužící k vytvoření izolovaných skupin procesů. Každý proces má svůj jmenný prostor, který mu vymezuje viditelnost pouze na vlastní prostředky a prostředky podřízených jmenných prostorů. Při startu systému existuje nejméně jeden namespace. Jmenné prostory tvoří hierarchickou stromovou strukturu. Existuje několik druhů jmenných prostorů.

PID Proces vidí procesy pouze ze svého a podřízených PID jmenných prostorů.

NET Síťové rozhraní náleží právě do jednoho jmenného prostoru a není sdíleno s podřízenými jmennými prostory. Každé síťové rozhraní má svoje IP adresy, routovací tabulky, firewall a další síťové prostředky.

MNT Kontrolujeme jaké mount pointy budou viditelné. Vytvořením nového MNT namespacu přeneseme všechny mount pointy rodiče. Změny v aktuálním namespacu neovlivňují rodičovský namespace.

IPC Meziprocesová komunikace. Linux kernel ve výchozím stavu přiděluje sdílenou paměť na základě uživatelského ID, což by vedlo ke kolizím mezi jmennými prostory.

UTS Dovoluje systému vystupovat pod více hostitelskými jmény (hostnames).

user Umožňuje libovolně mapovat uživatelská ID z kontejneru na ID uživatelů z hostitele. Uživatel v kontejneru se pak může tvářit jako root i když vně kontejneru se jedná pouze o obyčejného uživatele (bez práv roota).

O vytvoření jmenného prostoru se stará systémové volání FORK.

1.2.1.2 Kontrolní skupiny (cgroups)

Nástroj Linux kernelu umožňující přidělovat skupinám procesů limit na systémové prostředky a prioritizovat jednotlivé skupiny při přístupu k nim. Typy prostředků jsou:

- CPU čas
- Operační paměť
- Disková paměť
- Síťový provoz
- Provoz na disku

Mezi další vlastnosti kontrolních skupin patří možnost zmrazit skupinu procesů v určitém stavu a následně je znovu odmrazit.

1.2.1.3 Copy on write

Technika optimalizace správy dat, při které se při kopírování dat namísto vytvoření jejich kopie pouze označí jako sdílená. Fyzickou kopii dat je třeba

vytvořit až v okamžiku jejich modifikace. Za cenu vyšší režie můžeme takto ušetřit mnoho paměti.

Použití v Linuxu v systémovém volání FORK nebo v různých filesystémech (ZFS).

1.2.2 BSD jails

Kontejnerová technologie nacházející v operačních systémech typu BSD (FreeBSD, DragonflyBSD, ...). Na rozdíl od Linuxových kontejnerů, kde jsme si museli poskládat konečný kontejner z jednotlivých nástrojů, nám BSD nabízí přímo sadu systémových volání (`jail_*`).

Kontejner nám zajišťuje izolaci:

- Proces vidí procesy ze svého kontejneru
- Modifikace síťového nastavení zevnitř kontejneru není povolena (pevná IP adresa, routovací tabulky)
- Kontejner nemá přímý přístup k síťovému socketu (ICMP protokol nebude fungovat)
- Kontejner je vymezen pouze na adresářovou strukturu, ve které byl založen

Kontejneru lze také omezovat systémové prostředky (RAM, disk, síť, ...)

1.2.3 Hyper-V

Proprietární technologie firmy Microsoft. Dostupné pouze na některých verzích Windows (profesionální verze, dražší oproti klasickým verzím). Kontejnerové řešení Docker je schopné využít tuto technologii a fungovat nad ní jako frontend.

Kapitola 2

Gitlab CI

2.1 Uživatelské použití

Konfigurace projektů je řešena pomocí souboru `.gitlab-ci.yml`, který je umístěn přímo v kořenovém adresáři repozitáři projektu.

2.2 Průběh integrace

Proces integrace je rozdělen na fáze (*stages*), úkoly (*jobs*) a příkazy. Fáze se vykonávají sekvenčně za sebou v zadaném pořadí. Pokud jedna z fází selže, tak celý proces integrace končí chybou.

```
1 stages:
2   - test
3   - build
4   - deploy
```

Jednotlivé fáze se skládají z několika úkolů. Úkoly jsou spouštěny nezávisle na sobě a tím pádem je lze paralelizovat. Selhání jednoho z úkolů vede k neuspěchu celé fáze. Úkoly jsou vždy provedeny všechny, nezávisle na výsledku ostatních v rámci fáze.

```
1 job1:
2   stage: test
3   script:
```

```
    - cmd 1
    - cmd 2
```

Každý úkol je složen z několika příkazů. Pokud jeden z příkazů skončí chybovým návratovým kódem, pak je ukončeno provádění úkolu.

Mezi speciální úkoly patří `before\_script` a `after\_script`. Tyto úkoly jsou provedeny vždy před nebo po každým uživatelsky definovaným úkolem.

```
before_script:
  - cmd 1
  - cmd 2

after_script:
  - cmd 1
  - cmd 2
```

Jednotlivým úkolům lze přiřadit podmínečné spuštění pomocí notace `when`.

- `on\_success` – vykonej pouze pokud předchozí fáze skončila úspěchem (výchozí)
- `on\_failure` – vykonej pouze pokud předchozí fáze skončila neúspěchem
- `always` – vykonej vždy

2.3 Architektura

2.3.1 NGINX a Gitlab Pages

Frontend aplikace. Úloha NGINX jako reverzní proxy z hlediska cachování je předávána spíše na Gitlab Workhorse. Stará je o šifrování venkovního spojení pomocí HTTPs.

2.3.2 Gitlab Workhorse

Prvotním úkolem bylo podržet dlouhotrvající HTTP spojení vytvořeném GIT clonem přes HTTP a ne SSH. Důvodem pro to bylo, že Unicorn tyto spojení po vyprchání času ukončoval a zdálo se technologicky jednodušší implementovat

další vrstvu, která obslouží tyto požadavky, než měnit existující modul. Po přidání možnosti stažení ZIP souboru s repozitářem a vytváření tzv. artefaktů během buildu se zrodil stejná problém jako s GITem přes HTTP a byl také vyřešen stejným způsobem. Při dalším vývoji se zdálo moc pracné stále udržovat NGINX server, aby nějaké typy požadavků směroval na Unicorn a jiné na Workhorse, tak se začly všechny požadavky směrovat přes Workhorse. Workhorse nyní funguje jako chytřejší proxy poskytující statické soubory, GIT přes HTTP a směruje požadavky dále do Gitlab infrastruktury.

2.3.3 Unicorn

Ruby HTTP server.

2.3.4 Gitlab shell

Shell obsahující set příkazů pracující na originálním GITem. Po připojení na server přes SSH je možné konfigorovat repozitáře přímo přes shell příkazy. Řada oprávnění je kontrolována přímo přes GIT hooky, takže není nutné využívat Gitlab Shell. Pokud se jedná o komplexnější akci, vyžadující komplexnější kontrolu oprávnění, pak je nutné využít Gitlab Shellu.

2.3.5 Gitaly

Každý GIT příkaz zpracovávaný Gitlabem nakonec skončí v Gitaly. Cíl Gitaly je urychlit GIT příkazy decentralizováním uložiště a použitím cache. V současné době se uvažuje nad využitím projektu Git Ketch. Git Ketch replikuje uložiště mezi několik serverů, u kterých není pevně určené rozdělení na hlavní a sekundární. Hlavní server je vybrán podle hlasování na základě aktuálnosti dat mezi servery.

2.3.6 Uložiště

PostreSQL persistentní. Cache a data broker Redis.

2.3.7 Komunikace

GRPC uvnitř Gitlab systému.

2.3.8 Gitlab CI multi-runner

CI systém spolupracuje s několika běhovými servery (*Runners*), které se starají o samotné vykonání definovaných příkazů (sestavení). Běhové servery využívají exekutory. Implementuje několik exekutorů. Volba běhováho serveru je definována na úrovni konfigurace projektu a je omezena globálním nastavením Gitlab CI.

2.3.8.1 Shell exekutor

Příkazy provádí pod stejným uživatelem jako samotný běhový server. Příkazy přeneseny jako soubor.

2.3.8.2 Docker exekutor

Docker je kontejnerová technologie. Kontejnery jsou sestavovány z Docker obrazů definující sadu dostupného softwaru. Pro jednotlivá sestavení je vytvořena nová instance Docker kontejneru. Volba obrazu pro sestavení a pro jednotlivé joby je definována klíčovým slovem `image`. Sestavení lze obohatit o tzv. služby, kterými můžeme každému buildu dát k dispozici přístup k další Docker instanci, která bude dostupná pod zadanou hostname. Služby lze opět konfigurovat na úrovni sestavení, tak na úrovni jobu.

```
1 image: ruby:2.2
2
3 services:
4   - postgres:9.3
5
6 test:
7   script:
8   - bundle exec rake spec
```

2.3.8.3 VirtualBox a Parallels

Virtualizační technologie. Virtualní stroj musí podporovat Bash kompatibilní shell a být dostupný přes SSH spojení. Virtualní stroje jsou klonovány pro udržení čistého prostředí pro sestavení a snapshotovány pro urychlení dalších sestavení.

2.3.8.4 SSH exekutor

Podobné jako Shell exekutor, jen je využito SSH spojení.

2.3.9 Cachování

Ve výchozím stavu je cache sdílena mezi *branch* a *job*. Toto chování lze upravit nastavením klíče, který bude identifikovat danou cache. Při tvorbě klíče lze využít předdefinované CI proměnné. Soubory a složky určené k cachování uvedeme jako výčet v konfiguraci projektu. Cesta k souboru je uvedená jako relativní ke kořeni projektu, cachování souborů mimo projekt není touto cestou možné. Cache lze také omezit pouze na soubory a složky, které nejsou verzovany GITem.

```
cache:
  key: "$CI_BUILD_NAME/$CI_BUILD_REF_NAME"
  untracked: true
  paths:
    - binaries/
    - .config
```

Cache je ve výchozím nastavení ukládána přímo na běhovém serveru. Pokud fáze běží na různých běhových serverech, tak se cache nesdílí. Alternativně můžeme použít sdílené cachování na S3 kompatibilních serverech. S3 je REST API, které bylo původně vytvořeno jako část Amazon services. Toto API implementuje samotný Amazon, tak i nějaké self-hosted projekty.

Kapitola 3

Travis CI

Travis CI je open source služba kontinuální integrace spjatá s verzovací hostovací službou Github. Travis je naprogramován v Ruby.

3.1 Architektura

Systém je rozdělen na několik částí, které mezi sebou komunikující přes distribuovanou frontu RabbitMQ. Datové uložiště je realizováno v SQL databázi PostreSQL. Pro potřeby cachování je využiván Redis.

Travis Core Stará o většinu logiky pro Travis CI. Obsahuje model aplikace a služby, které jsou sdíleny se zbytkem systému.

Travis API Rozhraní překladající HTTP požadavky na volání jádra Travis Core.

Travis Hub Shromažďuje údálosti a komunikuje s aplikacemi mimo vnitřní systém.

Travis Listener Naslouchá notifikacím z Githubu a informuje o nich zbytek systému přes distribuovanou frontu.

Travis Build Překládá uživatelskou konfiguraci integrace na koncový skript, který bude vykonán Travis Workerem.

Travis Worker Zodpovědný za spouštění skriptů integrace a tvorbou čistého prostředí pro jejich běh. Úzce spolupracuje s Travis Loggerem, který ukládá výstup skriptu integrace.

Travis Web Obstaravá webové rozhraní pro koncového uživatele.

Travis Tasks Stará se o komunikace mezi Travis Hubem a zbytkem vnitřního systému.

3.2 Průběh integrace

Podobně jako u Gitlab CI je konfigurace řešena souborem `.travis.yml`, který je umístěn přímo v kořenovém adresáři repozitáře projektu.

Proces integrace je rozdělen na fáze (*stages*), úkoly (*jobs*) a příkazy. Fáze se vykonávají sekvenčně za sebou v zadaném pořadí. Selhání fáze znamená zastavení provádění všech následujících fází.

```
1 stages:
2   - test
3   - build
4   - deploy
```

Úkol náleží právě do jedné fáze a jeho přiřazení se provede v konfiguraci následovně.

```
1 stages:
2   - test
3 jobs:
4   include:
5     - stage: test
6       script: pytest
```

Každý úkol je spoštěn ve svém vlastním běhovém prostředí.

Další možností definice úkolů je přes tzv. Build Matrix.

```
1 language: python
2 python:
```

```
3     - '3.5'
4     - '3.4'
5     - '2.7'
6   env:
7     - TOXENV=first
8     - TOXENV=second
9   script:
10    - tox
```

Výsledkem této konfigurace bude vytvoření 6 úkolů podle kartézského součinu. Oba zmíněné přistupy konfigurace lze kombinovat.

3.2.1 Běhová prostředí

Travis CI podporuje několik běhových prostředí vyjmenovaných v následující tabulce.

	Superuživatel	Technologie	Konfigurace
Ubuntu Precise	Ano	Virtulní stroj	`sudo: required` `dist: precise`
Ubuntu Trusty	Ano	Virtulní stroj	`sudo: required` `dist: trusty`
Ubuntu Trusty	Ne	Kontejner	`sudo: false` `dist: trusty`
OS X	Ano	Virtuální stroj	`os: osx`

Tabulka 3.1: My caption

Výhodou kontejnerového prostředí je jeho rychlejší start.

```
1   sudo: required
2   dist: trusty
3   script:
4     - sudo apt add ...
5     - pytest
```

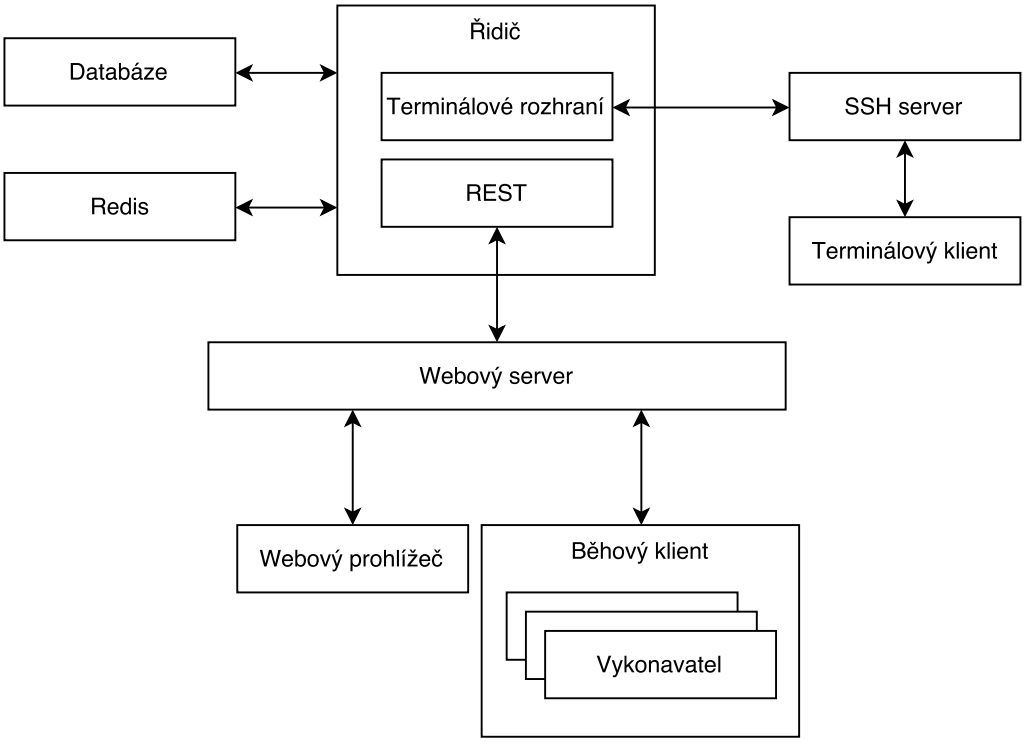
Ukázka kódu 3.1: Example from external file

Kapitola 4

Návrh architektury systému

Systém CI pod sebou bude spravovat několik Runnerů, každý Runner v sobě bude spouštět jednotlivé exekutory. Komunikace bude probíhat přes API. Jakákoliv funkcionalita by měla být zprvu naimlementovaná jako API endpoint a až pak zavedena do systému.

4.1 Části systému



4.1.1 Řidič

Zajišťuje komunikaci s uživatelem, informačním rozhraním a Runnery na pevně daném protokolu. Řidič by měl být podle vytížení Runnerů (API endpoint na Runneru) a požadovaných služeb rozhodnout, kterému Runneru práci předá.

4.1.2 Běhový klient

Zadaný API vstup rozparsuj pro potřeby Exekutoru a nech ho provést dané příkazy. Runner by měl spravovat pouze jeden typ exekutoru (Unixová architektura). Runner by měl běžet na samostatném stroji, aby neubíral systémové prostředky ostatním. Runner v sobě většinou bude spouštět více instancí Exekutoru.

4.1.2.1 Vykonavatel

Konečný vykonavatel příkazů. Řidič by neměl o existenci exekutorů vědět a komunikovat pouze s Runnery. Exekutorem je například VirtualBox, Docker a jiné. Samotná exekutor by měl být co nejméně modifikovaný. Nechceme například aby VirtualBox exekutor přijímal pouze image, které mají nainstalovaný nějaký obsáhlý set závislostí.

4.1.3 Webové rozhraní

JS only

4.1.4 Terminálové rozhraní

TODO

4.2 Komunikace

Komunikace bude navazována vždy ze směru od běhového klienta

4.2.1 REST architektura

Za REST kompatibilní rozhraní je považováno rozhraní splňující následující požadavky:

- Model klient-server
- Bezstavový model
- Správa mezipaměťi
- Uniformní rozhraní
- Vrstvená architektura

4.2.1.1 Model klient-server

Model klient-server je styl návhru systému, který dělí jeho části na poskytovatele služeb (servery) a jejich uživatele (klienty). Rozdělením systému na více

(na sobě nezávislých) částí je docíleno vylepšené portabtability a škálovatelnosti.

4.2.1.2 Bezstavový model

Bezstavové odbavení požadavku znamená, že každý požadavek musí obsahovat všechny informace k jeho vyřízení. Přínosem je jednodušší odbavení požadavku z hlediska serveru, který nemusí udržovat jednotlivým klientům jejich stav (*session*) nebo řešit chyby na základě nevalidního stavu. Nevýhoda spočívá v redundanci přenesených dat.

4.2.1.3 Správa mezipaměti

Součástí odpovědi od serveru může být i informace o tom, zda je možné tuto odpověď znovupoužít. Klient pak místo posílání dalšího stejného požadavku může použít již obdrženou odpověď z mezipaměti. Výhodou je uvolnění systémových prostředků serveru. Na druhé straně se může stát, že odpověd v mezipaměti je již neplatná.

4.2.1.4 Uniformní rozhraní

Rozhraní serveru je navrženo obecně, bez přizpůsobení požadavkům jednoho určitého klienta. Tímto je dosaženo zjednodušení serverové části za cenu snížení efektivity.

4.2.1.5 Vrstvená architektura

Příjemce požadavku a jeho vykonavatel nemusí být tentýž server. Z hlediska klienta se v komunikaci se serverem nic nemění. Nevýhodou je zvýšení režie a latence přenosu dat. Přínos spočívá v umožnění rozdělení serverové části na více menších systémů, které jsou snažší na správu a umožňují lepší škálovatelnost.

4.2.1.6 Reprezentace dat a přístup ke zdrojům

Základním stavebním kamenem REST rozhraní jsou zdroje. Každá pojmenovatelná informace může být zdrojem (např. počasí dnes v Praze, ...), kolekce

dalších zdrojů (např. počasí v hlavních městech Evropy) a další. Zdroj je identifikován URI adresou[1].

4.2.2 HTTP

4.2.3 Websocket

Websocket protokol umožňující oboustranout komunikaci mezi klientem a hostitelem.

4.2.4 Message Based

Implementace distribuovaných prioritních front ve stylu producenta a konzumenta. Použití: máme více Runnerů a chceme jim na základě jejich vytížení přiřazovat úkoly.

4.3 Komuninakce s běhovým prostředím

Cílem je vytvořit rozhraní s VM, které přijme zadaný příkaz nebo jejich sadu a rozhraní schopno streamovat výstup z příkazů. Rozhraní by mělo udržovat stály stav (proměnné shellu, ...).

4.3.1 Virtualbox

4.3.1.1 COM port pro čtení/zápis

Pomocí VirtualBox administračního rozhraní vytvoříme COM port, který nasměrujeme do souboru/socketu. Každý spuštěný příkaz poté přesměrujeme na zvolený COM port a necháme aplikaci na rodičovském stroji číst výstupy. Spojení je oboustranné, takže je možné předávat příkazy i dovnitř stroje, které je ale nutné parsovat a předávat systému další aplikací běžící uvnitř VM. Tento způsob předpokládá už přihlášeného uživatele.

[1] schéma:[//[uživatel[:heslo]@]server[:port]][/cesta][?dotaz][#fragment]

4.3.1.2 SSH spojení

SSH je kryptografický protokol, který umožňuje oboustrannou komunikaci mezi účastníky. Přihlašování uživatelů do operačního systému virtuálního stroje bez hesla lze docílit uložením jejich veřejných klíčů do souboru *known_hosts*. Soubor *known_hosts* je uložen v domovském adresáři uživatele a bude nutno zajistit jeho modifikaci buďto přes dodatečné připojení VDI obrazu disku virtuálního stroje nebo předpřípravou obrazu systému.

4.3.1.3 Vzdálený terminál přes COM port

Podobné SSH spojení. Nutná editace konfigurace VM systému, aby daný COM port viděl jako terminálový. Přihlašování uživatele a spuštění příkazů za pomoci přímého vpisu/čtení ze socketu.

4.3.2 Docker

Příkazy lze předávat do kontenejneru přímo z hostujícího stroje. Výpis výsledku je přímo na stdout, který lze přesměrovat do roury a z té dál do CI systému.

```
1 docker start ecstatic_perlman
2 docker exec -i ecstatic_perlman bash -c 'ls -al'
```

Ukázka kódu 4.1: Ukázka Docker exekutoru

4.3.3 LXC

4.4 Aktivní komunikace mezi prohlížečem a CI

Pro zaručení co největší uživatelské přívětivosti by bylo vhodné, aby webové rozhraní CI systému umělo zobrazit některé informace ihned jakmile budou dostupné, bez nutnosti manuálně vyvolat obnovení celé webové stránky. Server tedy posílá pouze žádané informace a ne celou webovou stránku. Jedná se například o stavy buildů nebo výpis výstupu aktuálně běžícího buildu (tzv. „streaming logu“). Klientem je v této komunikaci webový prohlížeč a serverem CI systém.

4.4.1 Short-polling

Spočívá v opakovaném dotazování ze směru klienta (webového prohlížeče) směrem k serveru (CI systém) v určitém časovém intervalu. Nevýhoda tkvý v plýtvání systémových prostředků z důvodu velkého počtu dotazů. Výhodou je velmi jednoduchá implementace.

```
00:00:00 Klient -> Máš pro mě něco?
00:00:01 Server -> Ne, čekej.
00:00:01 Klient -> Máš pro mě něco?
00:00:02 Server -> Ne, čekej.
00:00:02 Klient -> Máš pro mě něco?
00:00:03 Server -> Ano.
```

K uskutečnění dotazu ze směru prohlížeče využijeme JavaScript a jeho třídu `XMLHttpRequest`. Ze strany serveru se jedná o klasický HTTP GET požadavek.

4.4.2 Long-polling

Server po přijmutí požadavku místo okamžité záporné odpovědi čeká dokud nemůže vrátit kladný výsledek. Čekání je omezeno časovým limitem spojení na straně serveru i klienta. Na straně serveru je nutné si pohlídat maximální počet aktivních spojení.

```
00:00:00 Klient -> Máš pro mě něco?
00:00:15 Server -> Ano.
00:00:15 Klient -> Máš pro mě něco?
```

Implementace téměř totožná s Short-pollingem. Rozdílem je přidání čekací smyčky na straně serveru.

4.4.3 Server-Sent Events

Jednostranné spojení ve směru od serveru ke klientovi. Oproti long-pollingu není nutné stále otevírat nová spojení, drží se stále jedno. Spojení, které nekončí chybou, jsou automaticky znovuotevírána.

```
00:00:00 Klient -> Posílej mi všechno!
00:00:15 Server -> Tady to je.
00:00:16 Server -> Tady to je.
```

Serverová implementace založena na HTTP se speciální `Content-Type` hlavičkou. Jednotlivé zprávy jsou identifikovány pomocí pole `id` a odřádkováním.

```
Content-Type: text/event-stream
Cache-Control: no-cache

id: <id zpravy>
data: <data>

id: <id zpravy>
data: <data>
```

O příjem komunikace na straně klienta se stará JavaScript třída `EventSource`.

4.4.4 Websocket

Oboustranné spojení mezi klientem a serverem. Prvnotní požadavek přes HTTP protokol se změnou na WebSocket spojení. Většína jazyků určených pro vývoj webových aplikací nemá podporu WebSocketů přímo zabudovanou a tak je nutnost sáhnout po knihovně třetí strany. Implementace na straně webového prohlížeče implementována pomocí JavaScript třídy `WebSocket`.

4.4.5 HTTP2

4.4.6 Závěr

Nejsilnějším nástrojem jsou zajisté WebSockety. Nicméně naše použití nepotřebuje plnou oboustrannou komunikaci a z hlediska implementace jsou WebSockety tou nejsložitější cestou. Nejvýhodnějším řešením je technologie SSE, nepotřebuje žádné další externí závislosti a implemetace na serverové straně je velmi jednoduchá.

4.5 Ukládání dat systému

Co je třeba ukládat?

- Uživatelé a jejich nastavení
- Výsledky buildů včetně jejich výstupů
- Vazby mezi CI a SCM

Data by měla být i jednoduše seřaditelná podle různých kritérií.

4.5.1 Souborový systém

Nejjednodušší persistentní datové uložiště, které lze strukturovat pomocí adresářů a symbolických odkazů. Jednotlivé soubory pak uložíme v nějakém strojově zpracovatelném formátu, například JSON nebo XML. Z hlediska složité filtrace a řazení nevhodné pro data nad kterými chceme provádět tyto operace.

Vhodné pro:

- logy
- binární data
- velké soubory
- konfigurační soubory

4.5.2 In-memory datová struktura (Redis)

Klíč-hodnota struktura uložená v operační paměti podporující následující datové typy:

- řetězce
- seznamy
- sady
- seřazené sady

- hashe (asociativní pole)

Asociativní pole lze zanořovat do sebe a tím vytvářet struktury. Funguje na principu klient-server a je tedy snadné sdílet data mezi více systémy, které běží například na různých strojích. Operační pamět není persistentním uložištěm a tak je třeba data ukládat také na pevný disk. To je možné udělat pomocí průběžných zápisových logů nebo exportem celé struktury do souboru. Oba způsoby jsou přímou součástí Redisu.

Vhodné pro:

- dočasná data ke kterým potřebujeme rychlý přístup
- fronty
- zámky

4.5.3 Databáze

Vhodné pro:

- filtrovaná data
- řazená data
- data se složitějšími vazbami