**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
BRNO UNIVERSITY OF TECHNOLOGY

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
FACULTY OF INFORMATION TECHNOLOGY

**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**
DEPARTMENT OF INTELLIGENT SYSTEMS


# UDÁLOSTMI ŘÍZENÁ AUTOMATIZACE

EVENT-DRIVEN AUTOMATION

**BAKALÁŘSKÁ PRÁCE**
BACHELOR'S THESIS


**AUTOR PRÁCE** — **JAN HAVLÍN**
AUTHOR

**VEDOUCÍ PRÁCE** — **Ing. ONDŘEJ LENGÁL, Ph.D.**
SUPERVISOR



BRNO 2020

**Vysoké učení technické v Brně**
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2019/2020

# Zadání bakalářské práce

Student: **Havlín Jan**
Program: Informační technologie
Název: **Událostmi řízená automatizace**
**Event-Driven Automation**
Kategorie: Softwarové inženýrství

Zadání:

1. Seznamte se s procesy v týmu Testing Farm (TFT) firmy Red Hat.
2. Identifikujte v procesech úlohy vhodné k automatizaci.
3. Seznamte se s problematikou událostmi řízené automatizace a zmapujte existující řešení.
4. Navrhněte framework pro automatizaci úloh v prostředí týmu TFT a programové řešení vybraných úloh identifikovaných v bodu 2.
5. Implementuje framework a řešení navržené v předchozím bodě.
6. Vyhodnoťte přínosnost implementovaného řešení pro tým TFT.

Literatura:

- Brenda M. Michelson, Event-Driven Architecture Overview, Patricia Seybold Group, February 2, 2006
- Jeroen van Baarsen. GitLab Cookbook. Packt Publishing. 2014.
- Dokumentace projektu StackStorm.
- Dokumentace projektu RxPY.
- Dokumentace projektu AMQP.

Pro udělení zápočtu za první semestr je požadováno:

- První tři body zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Lengál Ondřej, Ing., Ph.D.**
Konzultant: Vadkerti Miroslav, RedHatCZ
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 31. července 2020
Datum schválení: 31. října 2019

## Abstrakt

Práce se zabývá automatizací procesů v rámci týmu Testing Farm firmy Red Hat Czech s.r.o. Automatizací vybrané úlohy *„Aktualizace a otestování systému průběžné integrace s novým sestavením operačního systému“* se docílilo snadnější údržby tohoto systému průběžné integrace. Implementace využívá Jenkins server ke spouštění úloh a nástroj tft-admin k vykonávání dílčích kroků vedoucích k automatizaci dané úlohy. Dále implementace umožňuje využití nástroje tft-admin k používání v automatizovaných skriptech, což do budoucna usnadňuje automatizaci dalších procesů.



## Abstract

The thesis deals with automation of processes in Testing Farm Team of company Red Hat Czech s.r.o. Automating the chosen task *„Update and test continuous integration system with new operating system build“* results in simplified maintenance of said continuous integration system. Implementation uses a Jenkins server to execute jobs and tft-admin tool to perform smaller steps resulting in automation of the task. Also, in the implementation part, modifications of the tft-admin tool allow usage in automated scripts which simplifies future automation of other processes.



## Klíčová slova

Automatizace, AWS, Beaker, Jenkins, Message bus, OpenStack, Průběžná integrace, Python

## Keywords

Automation, AWS, Beaker, Jenkins, Message bus, OpenStack, Continuous integration, Python


## Citace

HAVLÍN, Jan. *Událostmi řízená automatizace.* Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Ondřej Lengál, Ph.D.

# Událostmi řízená automatizace

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Ondřeje Lengála, Ph.D. Další informace mi poskytl technický konzultant pan Mgr. Miroslav Vadkerti. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

........................
Jan Havlín
29. července 2020

## Poděkování

V této sekci bych rád poděkoval technickému konzultantovi panu Mgr. Miroslavu Vadkertimu a vedoucímu práce panu Ing. Ondřeji Lengálovi, Ph.D. za jejich přispění ke vzniku této práce.

# Obsah


**1 Úvod** **3**

**2 Událostmi řízená automatizace** **4**
2.1 Událostmi řízená architektura . . . 4
2.1.1 Message bus . . . 4
2.2 Automatizační server Jenkins . . . 5
2.2.1 Úlohy . . . 5
2.2.2 Zásuvné moduly . . . 5
2.2.3 Způsob vytváření Jenkins úloh . . . 6

**3 Testování operačního systému** **8**
3.1 Testování softwaru . . . 8
3.1.1 Úrovně testování . . . 8
3.1.2 Typy testování . . . 9
3.1.3 Průběžná integrace . . . 9
3.2 Proces sestavení verze operačního systému Fedora . . . 10
3.2.1 Terminologie . . . 10
3.2.2 Distribution Git . . . 11
3.2.3 Koji . . . 12
3.2.4 Bodhi . . . 13
3.2.5 Fedora Infrastructure Message Bus . . . 13
3.3 Testing Farm Team . . . 14
3.3.1 Průběžná integrace balíčkového systému . . . 14
3.4 Manuální úlohy vhodné k automatizaci . . . 15

**4 Návrh automatizace manuálních úloh** **18**
4.1 Aktualizace a otestování systému průběžné integrace s novým sestavením operačního systému . . . 18
4.1.1 Ověření výskytu verze operačního systému na testovacích infrastrukturách . . . 20
4.1.2 Aktualizace konfigurace . . . 20
4.1.3 Spuštění integračních testů . . . 20
4.2 Nástroj tft-admin . . . 21
4.2.1 Implementační detaily . . . 21
4.3 Využití Jenkins serveru k automatizaci . . . 23
4.4 Výsledný stav . . . 23
4.4.1 Odlišnost od počáteční koncepce práce . . . 25

**5 Implementace automatizace manuálních úloh** **26**
5.1 Úpravy nástroje tft-admin . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
5.1.1 Rozlišení výpisů na standardním výstupu . . . . . . . . . . . . . . . . 26
5.1.2 Přepínač pro režim strojově čitelného výstupu . . . . . . . . . . . . . 26
5.1.3 Výsledný stav . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
5.2 Aktualizace a otestování systému průběžné integrace s novým sestavením operačního systému . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
5.2.1 Rozšíření konfigurace o nutné položky . . . . . . . . . . . . . . . . . . 28
5.2.2 Implementace příkazů nutných k vykonání dílčích kroků . . . . . . . 29
5.2.3 Jenkins úloha pro vykonání skriptu . . . . . . . . . . . . . . . . . . . . 34

**6 Vyhodnocení přínosnosti automatizace** **37**

**7 Závěr** **38**

**Literatura** **39**

**A Obsah paměťového média** **41**

# Kapitola 1

# Úvod

Bakalářská práce se zabývá problematikou událostmi řízenou automatizací. Zadání vzniklo ve spolupráci se společností Red Hat Czech s.r.o.

Cílem této práce je implementovat softwarový nástroj, který by automatizoval opakující se procesy v rámci týmu Testing Farm firmy Red Hat. Tyto procesy souvisejí s údržbou systému průběžné integrace. Implementací tohoto nástroje by došlo k ušetření práce lidskému faktoru. Nástroj má být do budoucna snadno rozšiřitelný a programovatelný k reakcím na nové události. Hotové řešení bude nasazeno do firemního provozu.

Kapitola 2 uvádí do problematiky událostí, zejména zpráv zasílaných mezi systémy a Jenkins serveru, který se využije v implementační části. Kapitola 3 se zabývá teoretickou části práce a popisu výchozího stavu v týmu Testing Farm. V této kapitole si představíme testování operačního systému a proces sestavení komponent operačního systému Fedora. V následující sekci 3.3 se rozebírá systém průběžné integrace a související úlohy a procesy patřící do údržby tohoto systému, jejichž automatizace je cílem bakalářské práce.

Návrhu implementační části práce se věnuje kapitola 4. Podrobně popisuje úlohu, která je nejvhodnější k automatizaci a také analyzuje dostupné nástroje k implementaci. Samotné implementaci se věnuje kapitola 5, která rozepisuje nově implementované příkazy do nástroje `tft-admin` a Jenkins úlohu `compose-update`, která tento nástroj využívá.

# Kapitola 2

# Událostmi řízená automatizace

Tato kapitola uvádí do problematiky událostmi řízené automatizace. Nejprve vysvětluje koncepty událostmi řízené architektury. Dále se věnuje konkrétnímu softwaru, který tuto architekturu podporuje, Jenkins serveru.

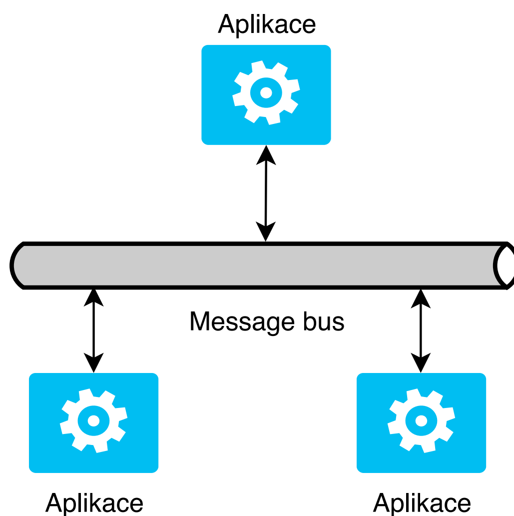
## 2.1 Událostmi řízená architektura

Tato sekce čerpá z publikace *Enterprise Integration Patterns.*[13]

Událostmi řízená architektura je přístup k architektuře softwarových systémů pomocí vytváření, detekce, zpracování a reakcí na události. Událost je definována jako „významná změna v systému“.

### 2.1.1 Message bus

Message bus je komunikační kanál, struktura, která spojuje aplikace, a tím jim umožňuje pracovat společně pomocí zasílání zpráv. Obrázek 2.1 zobrazuje diagram příkladu propojení message busu s aplikacemi.




**Obrázek 2.1:** *Schéma message busu*

*Message bus* je kombinace společného datového modelu, společné množiny příkazů a infrastruktury pro zasílání zpráv, která umožňuje různým systémům komunikovat skrze sdílenou množinu rozhraní. Toto je analogické k počítačové sběrnici, která slouží jako ústřední prvek pro komunikaci mezi procesorem, operační pamětí a periferiemi. Stejně jako v hardwarové analogii, i zde se nachází několik částí, které dohromady sestavují *Message Bus*:

- **Sdílené komunikační rozhraní** – Typicky je zvolen systém pro zasílání zpráv, který slouží jako fyzická komunikační infrastruktura poskytující multiplatformní univerzální adaptér mezi aplikacemi.
- **Adaptéry** – K propojení systémů na message bus.
- **Sdílená struktura příkazů** – Jednotná množina příkazů, kterým rozumí všechny účastníci v message busu.

## 2.2 Automatizační server Jenkins

Tato sekce rozebírá automatizační server Jenkins, protože mezi jeho součásti patří také schopnost reagovat na události. V kontextu této práce je podstatné zasílání zpráv mezi systémy, Jenkins server umožňuje na tyto zprávy reagovat a na základě toho vykonávat různé úlohy.

Jenkins server hraje klíčovou roli v systému průběžné integrace operačního systému, podrobněji popsaného v kapitole 3.3.1. Implementační část využívá Jenkins server k automatizaci procesů souvisejících s údržbou tohoto systému průběžné integrace.

Tato sekce čerpá informace z dokumentace o serveru Jenkins.[5]

Jenkins je nezávislý, open-source automatizační server, který může sloužit k automatizaci různých úloh spojených se sestavováním, testováním, dodáváním nebo nasazováním softwaru.

### 2.2.1 Úlohy

Úlohy (jobs) jsou základním prvkem jenkins serveru.

Úloha se skládá z několika částí:

- *Trigger* spouští úlohu, pokud byla splněna vstupní podmínka,
- *builder* vykonává danou úlohu,
- *post-build* poskytuje uživatelům zpětnou vazbu prostřednictvím různých komunikačních kanálů.

### 2.2.2 Zásuvné moduly

Moduly (angl. *plugins*) slouží k rozšiřování funkcionality Jenkins serveru pro přizpůsobení potřebám řešení daného problému. Existuje přes 1000 modulů, které mohou být na server nainstalovány.

Jenkins nabízí dva způsoby instalace modulů:

- Prostřednictvím webového rozhraní,
- použitím příkazu `install-plugin` v rozhraní příkazové řádky.

Mezi moduly, na které se tato práce bude odkazovat, patří například:

- *Pipeline*[1], který nabízí bohaté možnosti pro automatizované vykonávání různých procesů,
- *JMS Messaging*[2], který rozšiřuje funkcionalitu o spouštění úloh po zachycení zprávy na message busu založeném na protokolu AMQP, který je podrobněji popsán v kapitole 3.2.5.

### 2.2.3 Způsob vytváření Jenkins úloh

V současné době existuje několik způsobů, kterými lze vytvořit Jenkins úlohu, zde si uvedeme dva příklady, a to:

- Prostřednictvím webového rozhraní,
- textového souboru ve formátu *yaml* s popisem dané úlohy.

Obě tyto volby poskytují stejné možnosti z hlediska vytváření nových nebo upravování již existujících úloh, ovšem tato část věnuje větší pozornost druhé možnosti, neboť oproti webovému rozhraní nabízí několik výhod, např. jelikož se jedná o textové soubory, je tak možné snadno verzovat.

Následující výpis zobrazuje popis jednoduché úlohy ve formátu *yaml*:

```
- job:
    name: test-job
    project-type: pipeline
    sandbox: true
    pipeline-scm:
      scm:
        - hg:
            url: http://hg.example.org/test_job
            clean: true
      script-path: Jenkinsfile
      lightweight-checkout: true
```

**Výpis 2.1:** *Ukázka jednoduché Jenkins úlohy typu Pipeline.*

Jak je z atributů Výpisu 2.1 zřejmé, jedná se o úlohu typu *Pipeline*, která vyžaduje nainstalovaný stejnojmenný modul. Dále se v atributu `script-path` odkazuje na soubor `Jenkinsfile` nacházející se na adrese `url`, která může odkazovat například na gitový repozitář. Soubor `Jenkinsfile` definuje jednotlivé kroky, které má úloha vykonat.

Pokud by se měla tato úloha spouštět v reakci na nějakou událost, vyžadovala by vyplněný atribut `trigger`, v současné době by tedy tato úloha musela být spouštěna pouze manuálně.

Následující výpis ukazuje obsah souboru `Jenkinsfile` obsahující skript, který má tato úloha při spuštění vykonat:

[1] https://plugins.jenkins.io/workflow-aggregator/

[2] https://plugins.jenkins.io/jms-messaging/

```
pipeline {
    agent { docker { image 'python:3.5.1' } }
    stages {
        stage('build') {
            steps {
                sh 'echo Hello World!'
            }
        }
    }
}
```

**Výpis 2.2:** *Ukázka jednoduchého* `Jenkinsfile` *souboru.*

Pro shrnutí, pro definici Jenkins úlohy typu *Pipeline* jsou zapotřebí dva soubory – soubor ve formátu *yaml*, který definuje základní parametry úlohy (zobrazený ve Výpisu 2.1) a soubor *Jenkinsfile*, který definuje jednotlivé kroky dané *pipeline* (zobrazený ve Výpisu 2.2).

# Kapitola 3

# Testování operačního systému

## 3.1 Testování softwaru

Tato sekce čerpá informace z publikace *ISTQB Foundation Level Syllabus.*[16]

Testování softwaru je způsob, jak zajistit jeho kvalitu a snížení možnosti selhání v provozu. Software, který nefunguje správně může způsobit řadu problémů jako např. peněžní ztrátu, časovou ztrátu, snížení reputace společnosti, ale také i zranění a smrt člověka.

Některé testování vyžaduje spuštění komponenty nebo systému, který se testuje. Takové testování se označuje jako dynamické. Opakem toho je statické testování, které zkoumá pouze zdrojový kód a nespouští tedy daný program.

### 3.1.1 Úrovně testování

Testování softwaru je možné kategorizovat do několika skupin, které obsahují testovací procesy v souvislosti s úrovní, na které se testování pohybuje. Tyto úrovně se pohybují od individuálních jednotek nebo komponent ke komplexním systémům. Každá úroveň vyžaduje vhodné testovací prostředí.

#### Testování komponent

Testování komponent (také označováno jako jednotkové testování) se zaměřuje na komponenty, které lze testovat odděleně. Testování se obvykle provádí izolovaně od zbytku systému.

#### Integrační testování

Integrační testování se zabývá interakcí mezi komponentami a systémy.

Existují 2 různé úrovně integračního testování:

- Integrační testování komponent – zaměřuje se na interakci a rozhraní mezi integrovanými komponentami. Toto testování se provádí po dokončení testování komponent a je zpravidla automatizováno.
- Integrační testování systémů – zaměřuje se na interakci a rozhraní mezi systémy, aplikace se ověřuje jako celek.

#### Systémové testování

Systémové testování se zaměřuje na chování celého systému. Testovací prostředí by mělo odpovídat finálnímu nebo produkčnímu prostředí. Toto testování typicky provádí nezávislí testeři, kteří spoléhají na požadavky.

#### Akceptační testování

Akceptační testování se podobně jako systémové testování typicky zabývá chováním celého systému. Cílem je vytvořit důvěru v kvalitě systému jako celku, validovat, že je systém kompletní a pracuje, jak se od něj očekává a verifikovat, že funkcionální a nefunkcionální chování systému je podle specifikace.

### 3.1.2 Typy testování

Testování lze také dělit na různé typy, jsou to skupiny aktivit, které se zaměřují na testování určitých charakteristik softwaru.

#### Funkcionální testování

Funkcionální testování zahrnuje testy, které vyhodnocují funkce, které by měl systém vykonávat. Tyto funkce říkají, *co*, by systém měl dělat. Testovat by se mělo na všech úrovních.

#### Nefunkcionální testování

Nefunkcionální testování je sledování, *jak dobře* se systém chová. Vyhodnocují se charakteristiky systému a softwaru jako použitelnost, výkonnost nebo bezpečnost. Testovat by se mělo na všech úrovních, a to co nejdříve, kdy je to možné. Pozdní objevení nefunkcionálních chyb může negativně ovlivnit úspěch celého projektu.

### 3.1.3 Průběžná integrace

V předchozích částech této sekce jsou stručně popsané různé způsoby, kterými se software testuje. Speciální pozornost si zaslouží takzvaná *průběžná integrace* (angl. *Continuous Integration*, *CI*), neboť tato práce si klade za cíl zautomatizovat některé procesy související s údržbou konkrétního systému průběžné integrace popsaném v kapitole 3.3.1.

Tato sekce čerpá informace z publikace *Continuous Integration: Improving Software Quality and Reducing Risk.*[10]

Procesy průběžné integrace začínají přidáním nového commitu se zdrojovými kódy do repozitáře. V typickém projektu, různí lidé s různými rolemi mohou přidat commit, který spustí proces průběžné integrace, např. vývojář úpravou zdrojového kódu, administrátor databáze změnou definice tabulky, tým zodpovědný za sestavení upravením konfiguračních souborů...

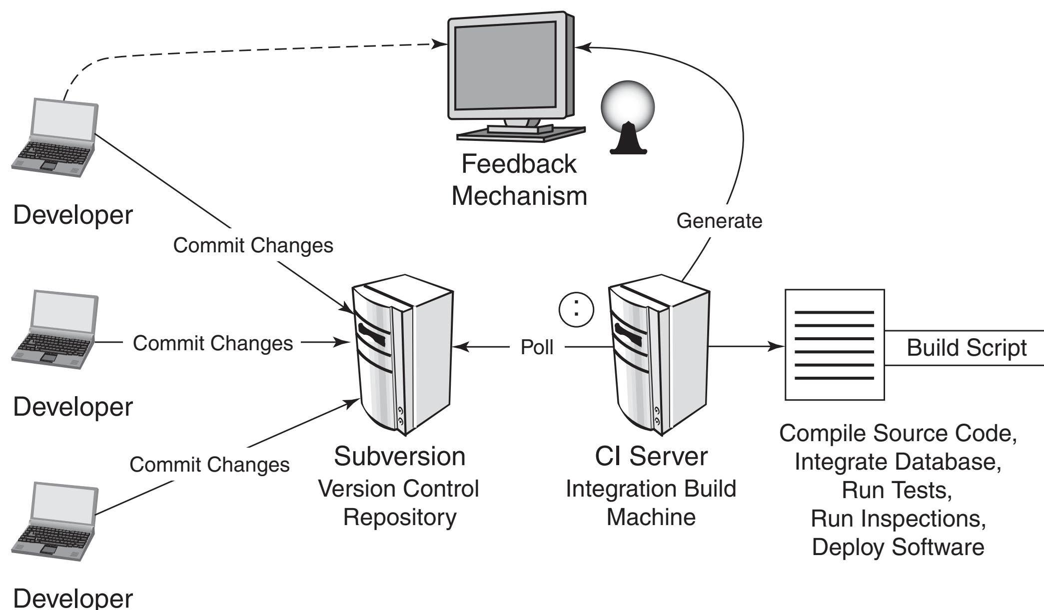
Při použití metody průběžné integrace se typicky provádí následující kroky:

1. Vývojář odešle své změny do vzdáleného repozitáře. Mezitím CI server na stroji určeném pro sestavení (angl. build) tohoto softwaru sleduje tento repozitář, zda v něm nenastala změna.

2. Brzy po odeslání tohoto commitu CI server zaznamená změnu v repozitáři a tuto nejnovější změnu si stáhne a spustí skript.

3. CI server vytvoří zpětnou vazbu a prostřednictvím komunikačního kanálu ji zašle vybraným členům projektu.
4. CI server pokračuje v sledování repozitáře, zda v něm nenastala změna.

Následující Obrázek 3.1 zobrazuje schéma komponent průběžné integrace:




**Obrázek 3.1:** *Schéma komponent průběžné integrace, obrázek převzat z [10].*

## 3.2 Proces sestavení verze operačního systému Fedora

Systém průběžné integrace, nad kterým tato práce staví, se odkazuje na různé komponenty související se sestavováním operačního systému. Je tedy namístě stručně představit, z jakých komponent a procesů se skládá sestavování verze operačního systému. Tato sekce se zabývá operačním systémem Fedora, nicméně podobné principy se aplikují v operačním systému RHEL.

### 3.2.1 Terminologie

V následujících sekcích se často uvádí pojem RPM balíček. Je proto vhodné si nejprve stručně vysvětlit význam.

#### RPM Package Manager

RPM Package Manager (rekurzivní akronym RPM)[7] je balíčkovací systém, který umožňuje:

- Sestavovat software ze zdrojových souborů do snadno distribuovatelných balíčků,
- instalovat, aktualizovat a odinstalovat balíčky,
- dotazovat se na detailní informace o zabaleném softwaru,
- verifikovat integritu zabaleného softwaru a výsledné instalace.

### RPM balíček

RPM balíček je soubor obsahující jiné soubory a informace o nich potřebné pro systém. Balíček sestává z cpio archivu, který se skládá ze souborů a RPM hlavičky, která obsahuje metadata o balíčku. RPM Package Manager používá tato metadata pro vyhodnocení závislostí, kam nainstalovat soubory a další informace.

Existují 2 typy RPM balíčků:

- Zdrojový RPM balíček (SRPM),
- binární RPM balíček.

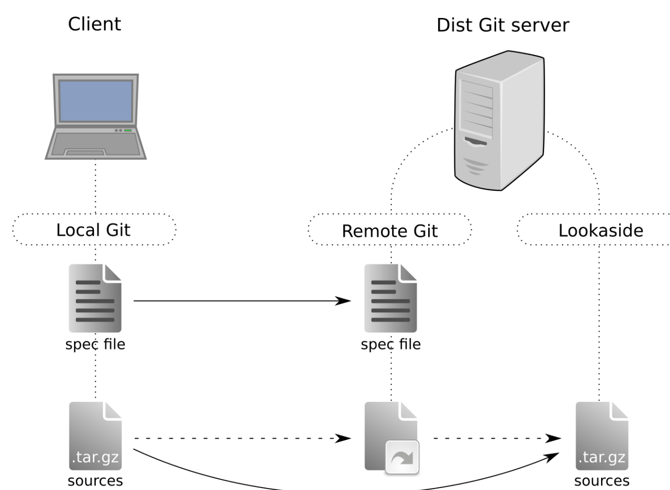
### 3.2.2 Distribution Git

Distribution Git (zkr. *DistGit*) je verzovací systém Git s přídavným datovým úložištěm.[6] Je určený k ukládání obsahu zdrojových RPM balíčků a skládá se ze tří základních komponent:

- Git repozitáře,
- mezipaměť k ukládání zdrojových tar archivů,
- skripty řízení obojího.

Zdrojový balíček RPM typicky obsahuje spec file a zdrojové soubory (upstream tar a přídavné záplaty). Zdrojové tar archivy, binární potenciálně velké soubory, nejsou příliš vhodné k ukládání do Git repozitáře. Při každé jejich aktualizaci by verzovací sysém vytvořil velký bezvýznamný diff. Z tohoto důvodu byl vytvořen DistGit, který poskytuje efektivní mezipaměť, do kterého se mohou ukládat tar archivy. Git repozitář může být ponechán k činnosti, kterou dělá nejlépe, a to k sledování změn ve spec souboru, downstream záplatám a přídavným textovým souborům pojmenovaných `sources`, které obsahují odkazy ke zdrojovým tar archivům v mezipaměti.

Obrázek 3.2 zobrazuje schéma tohoto systému.




**Obrázek 3.2:** *Schéma systému DistGit, obrázek převzat z [6].*

### 3.2.3 Koji

Tato sekce pojednává o balíčkovém systému Koji čerpá z její dokumentace[15] a Wiki stránek Fedory.[8]

Koji je software pro vytváření balíčků RPM (angl. *build system*). Využívá nástroj Mock k vytvoření prostředí pro sestavování balíčků. RPM balíčky, jež byly sestaveny build systémem Koji, se také nazývají *Koji build*, takto obdobně se nazývají balíčky sestavené i jinými build systémy, např. pro build systém *Brew* a *Copr* by se jednalo o *Brew build*, respektive *Copr build*.

V Koji je někdy nezbytné rozlišovat mezi balíčkem obecně, konkrétním sestavením balíčku a různými RPM soubory vytvořenými sestavením balíčku. Tyto tři termíny jsou přesněji definovány následovně:

- Balíček (Package) je název zdrojového RPM balíčku, referuje to obecně k balíčku, ne ke konkrétnímu sestavení (buildu) nebo podbalíčku,
- build označuje konkrétní build balíčku, toto zahrnuje kompletní build, a to konkrétní architekturu a podbalíčky,
- RPM specifikuje jednu konkrétní architekturu a podbalíček buildu.

#### Komponenty

Koji zahrnuje několik komponent:

- **Koji-Hub** je středem všech Koji operací. Jedná se o XML-RPC server, který je pasivní v tom, že pouze přijímá XML-RPC volání a spoléhá na build démony a další komponenty k zahájení komunikace. Koji-Hub je jedinou komponentou, která má přímý přístup do databáze a jedna ze dvou komponent, která má práva pro zápis do souborového systému.
- **Kojid** je build démon, který běží na každém buildovacím stroji.
- **Koji-Web** poskytuje vizuální rozhraní k správě Koji-Hub. Zobrazuje informace a také nabízí prostředky k některým operacím, jako např. zrušení buildu.
- **Koji-client** umožňuje uživateli se dotazovat na data a současně vykonávat určité akce jako např. přidávání uživatelů a zahajování build požadavků.
- **Kojira** je démon, který drží kořenová repozitářová data na aktuální verzi. Je odpovědný za odstraňování redundantních build kořenů a uklizení po dokončení build požadavku.

#### Organizace balíčků

Koji používá tagy k organizaci balíčků:

- tagy jsou sledovány v databázi, ale ne na disku,
- tagy podporují vícedědičnost,
- každý tag má vlastní seznam platných balíčků (dědičné),
- vlastnictví balíčku může být nastaveno pro každý tag (dědičné),

- při buildu se specifikuje cíl namísto tagu.

Cíl buildu specifikuje, kam by se měl balíček vybuildit a jak by měl být otagován. Toto umožňuje cílovým jménům zůstat fixní, jelikož tagy se mění v různých vydáních.

### 3.2.4 Bodhi

Bodhi je systém, který usnadňuje proces zveřejňování aktualizací pro software distribuce Fedora.[4] Je navržen k demokratizaci testování aktualizací balíků a procesu vydání linuxových distribucí založených na RPM. Bodhi je rozhraní pro publikování aktualizací pro Fedoru. Rovněž slouží jako nástroj pro zpětnou vazbu z testování. Mezi jeho hlavní rysy patří:

- Poskytuje rozhraní pro vývojáře k vydání aktualizací balíků pro vícero verzí distribucí.
- Generuje předběžné testovací repozitáře pro koncové uživatele a testery k instalaci navrhovaných aktualizací.
- Poskytuje testerům rozhraní pro zanechání zpětné vazby k aktualizacím balíků.
- Oznamuje nově příchozí balíčky, které vstupují do sbírky.
- Koncovému uživateli publikuje poznámky k novému vydání, které se označují jako *errata*.
- Generuje yum repozitáře.
- Dotazuje se ResultsDB pro automatizované výsledky testování a zobrazuje je v nových vydáních.

### 3.2.5 Fedora Infrastructure Message Bus

Služby, které jsme si zde popsali (*DistGit*, *Koji*, *Bodhi*), ale také řada dalších, které tvoří infrastrukturu Fedory, zasílají zprávy na *message bus*. Tento message bus je postaven na protokolu *AMQP*.

#### Advance Message Queueing Protocol

*Advance Message Queueing Protocol* (AMQP) je otevřený protokol sloužící ke komunikaci mezi systémy.[1] Mezi klíčové prvky protokolu patří:

- *server* (broker) je aplikace implementující AMQP model, která přijímá připojení od klientů pro výměnu zpráv.
- *zpráva* obsahuje přenášená data a jejich metadata,
- *příjemce* (consumer) je aplikace, která přijímá zprávy z front,
- *odesílatel* (producer), který zasílá zprávy do fronty prostřednictvím exchange.

AMQP Model definující operace se zprávami (příjem, směrování, ukládání a vkládání do front) závisí na definici následujících komponent:

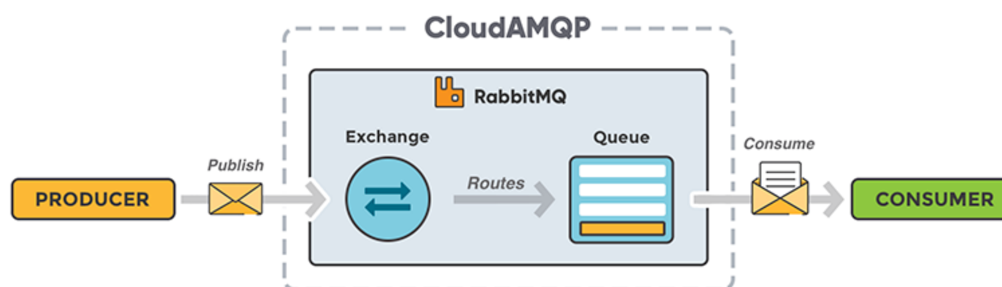
- *exchange* je část serveru, která přijímá zprávy od odesílatele a směruje je do front,

- *fronta* (message queue) je pojmenovaná entita, ke které se asociují zprávy a odkud je příjemci získávají,
- *bindings* jsou pravidla, pro distribuci zpráv z exchange do front.

Po obdržení zprávy od klienta (odesílatele) probíhá směrování do front, k tomuto slouží část serveru exchange. V současné době existují 4 typy exchange:

- *direct exchange* – zpráva je přiřazena do fronty na základě směrovacího klíče, který se nachází ve zprávě,
- *fanout exchange* – směrovací klíč ignoruje a zprávu přiřadí do front dle bindings pravidel,
- *topic exchange* – zpráva je přiřazena do fronty na základě směrovacího klíče ve zprávě a bindings pravidel,
- *headers exchange* – zpráva se směruje do fronty dle několika atributů, jde o podobný princip jako *direct exchange*, ale nabízí více možností.

Následující Obrázek 3.3 zobrazuje schéma AMQP:




**Obrázek 3.3:** *Schéma AMQP, obrázek převzat z [3].*

## 3.3 Testing Farm Team

Testing Farm Team je tým, který vyvíjí, udržuje a nasazuje systém průběžné integrace sloužící k spouštění různých druhů testů, které testují komponenty operačního systému Fedora. Dále se stará o infrastrukturu automatizačního serveru Jenkins a monitorovací infrastrukturu.

### 3.3.1 Průběžná integrace balíčkového systému

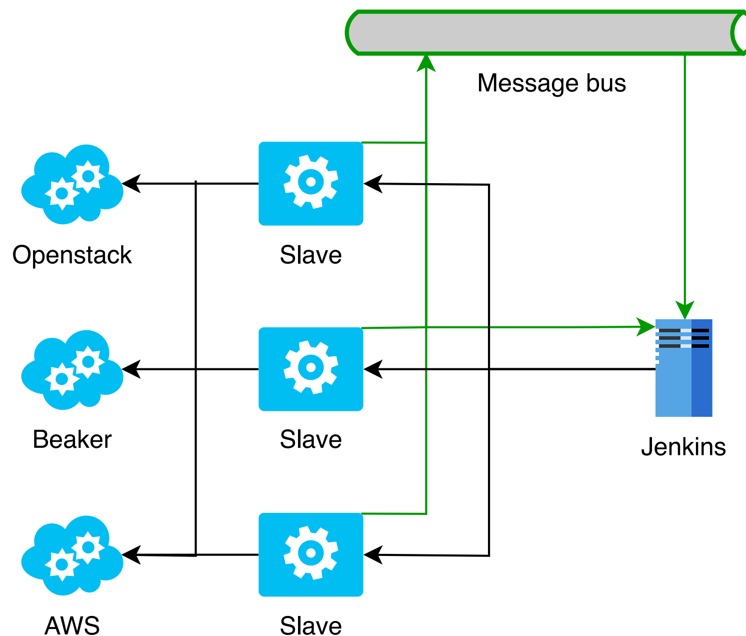
V kapitole 3.1.3 je popsáno obecně, co je to průběžná integrace. Tato podsekce pojednává o konkrétním systému průběžné integrace, na kterém tato práce staví a čerpá informace z diplomové práce *Podpora průběžné integrace v rámci systému Copr.*[14]

Různé balíčkovací systémy jako například Brew, Copr nebo Koji, který je popsaný v kapitole 3.2.3, po úspěšném sestavení dokončení sestavení RPM balíčku odešle zprávu na message bus, který je založen na protokolu AMQP (popsaný v kapitole 3.2.5).

Systém průběžné integrace zachycuje tyto zprávy prostřednictvím Jenkins serveru, následně se podívá do konfigurace, zda jsou pro tento balíček definované testovací úlohy a pokud ano, spustí je.

Testovací úloha vybere odpovídající sestavení operačního systému (tato verze operačního systému bývá také nazývána *compose*) dle atributů obsažených v RPM balíčku. Systém získá dostupný virtuální stroj prostřednictvím služeb Openstack, Beaker nebo AWS. Na tento stroj se nainstalují všechny prostředky nezbytné k testování (testovací frameworky a artefakt, který se testuje). Následně se spustí testovací framework a zkoumá se pomocí statické analýzy a funkcionálního testování, zda tato komponenta funguje správně. Výstupem testu jsou logy o testování, které se nahrají na Jenkins server a zároveň se odešle zpráva na message bus o tom, že testování skončilo.

Následující Obrázek 3.4 zobrazuje schéma tohoto systému průběžné integrace:




**Obrázek 3.4:** *Schéma průběžné integrace operačního systému.*

## 3.4 Manuální úlohy vhodné k automatizaci

S údržbou systému popsanému v předchozí sekci se váže několik různých činností nebo úloh, které musí lidé vykonávat manuálně. Tato sekce stručně popisuje několik těchto manuálních úloh, jejichž automatizace by usnadnila údržbu systému. Všechny tyto úlohy spojuje vlastnost, že musí být vykonávány pravidelně, z tohoto důvodu má smysl některé z nich zautomatizovat.

### Pravidelné vydání systému průběžné integrace

Pravidelné vydání (angl. release) slouží k informování koncových uživatelů o nových funkcionalitách a opravených chybách. Takovéto aktualizace systému hrají významnou roli v rámci agilního vývoje.

Release se skládá z:

- Procházení commitů v novém vydání, připravení merge requestu, pouštění integračních testů, provedení změny changelogu,
- příprava textu nového release pro koncové uživatele a odeslání e-mailu,
- vygenerování diffu mezi současnou produkcí a verzí, která se bude nasazovat.

Jednou z možností, jak zautomatizovat vydání systému průběžné integrace je vytvořením vhodně pojmenovaného merge requestu (např. *release*), který by neobsahoval žádnou změnu, do repozitáře na GitLabu. Systém následně zareaguje na změnu v repozitáři a v případě vytvoření takto pojmenovaného merge requestu do něj vloží informace spojené s releasem.

### Kontrola a upozornění na výpadky systémů

Funkčnost systému průběžné integrace je závislá na provozu několika dalších systémů (např. Jenkins, Openstack). V současnosti máme 2 významné zdroje informací:

- službu, která si vede informaci o současném stavu každého systému[1],
- druhým zdrojem informací je portál, který obsahuje výpadky a plánované údržby s časovými údaji.

V případě výpadku systému, který je klíčový pro provoz CI, chceme být informováni co nejdříve, tedy zareagovat na tuto změnu stavu systému a informovat prostřednictvím např. irc.

Dále by bylo vhodné informovat o plánovaných údržbách např. prostřednictvím mailu, irc, přidáním do Google kalendáře.

### Pouštění integračních testů

Při vývoji je nezbytné pouštět integrační testy, v současnosti se to většinou dělá manuálně. Musíme sami zjistit, které konkrétní testy chceme spustit.

Nástroj by měl umožnit například volbu různých verzí integračních testů z repozitáře na serveru GitLab.

### Aktualizace a otestování systému průběžné integrace s novým sestavením operačního systému

RCM je tým, který se stará o vydávání produktů zákazníkům a stará se o vytváření nového sestavení operačního systému (*compose*), která vydává na různých platformách, např. image pro virtuální stroje nebo na DVD.

V současnosti jsou 2 možné způsoby vydání compose:

- Vydání nového distra do Beakeru,
- vydání nového image do Openstacku.

Systém průběžné integrace lze aktualizovat na základě druhé možnosti, tedy když se v Openstacku nachází nová verze image. Manuální kroky jsou poté následující:

[1] https://status.fedoraproject.org/

- Nahrání image na službu AWS,
- aktualizace konfiguračního souboru `variables.yaml`,
- spuštění integračních testů, zda systém s novým compose pracuje správně,
- v případě, že systém pracuje správně, provedení merge v konfiguračním repozitáři a nasazení.

Nástroj by měl tuto činnost zautomatizovat tím, že na základě názvu compose ověří, že se vyskytuje jeho image na testovacích infrastrukturách, případně je tam nahraje, aktualizuje soubor `variables.yaml` s touto novou verzí, vytvoří pro tuto změnu merge request a spustí integrační testy.

### Procházení merge requestů, jejich prioritizace a urgování

V současnosti se pracuje s větším množstvím repozitářů, které se kategorizují do několika skupin a nacházejí se na několika různých serverech (*GitLab* a *GitHub*).

Cílem je vytvořit agregátor, který by vytvořil frontu merge requestů. Měl by urgovat na různých komunikačních kanálech (např. prostřednictvím e-mailu nebo IRC), když budou repozitáře dlouho beze změny.

Prioritizace merge requestů by měla brát v potaz čas od poslední změny a tag, který určuje prioritu (např. tag *WIP* by snižoval prioritu).

### Restart padlých testovacích úloh

V případě, že testovací úloha selže, je důležité zjistit, z jakého důvodu se to stalo. Je namístě vytvořit heuristiku pro restartování těchto úloh. Např. v případě výskytu problému s infrastrukturou by měl systém vyčkat na opětovné zprovoznění infrastuktury a potom úlohy spustit.

Semafor (služba informující o výpadcích systémů) řekne, že je vyřešený výpadek systému, ale tato informace není vždy pravdivá, proto si to chceme sami ověřit. Například po skončení výpadku Openstacku musíme ručně naplánovat několik testovacích úloh a zkoušíme, zda projdou. Tato činnost je vhodná k automatizaci.

# Kapitola 4

# Návrh automatizace manuálních úloh

V předchozí kapitole 3.4 jsme si představili procesy a úlohy související s údržbou systému průběžné integrace, které se musí periodicky vykonávat manuálně. Tato kapitola podrobně popíše vybranou úlohu, která je nejvhodnější k automatizaci a také zanalyzuje dostupné nástroje vhodné pro implementaci.

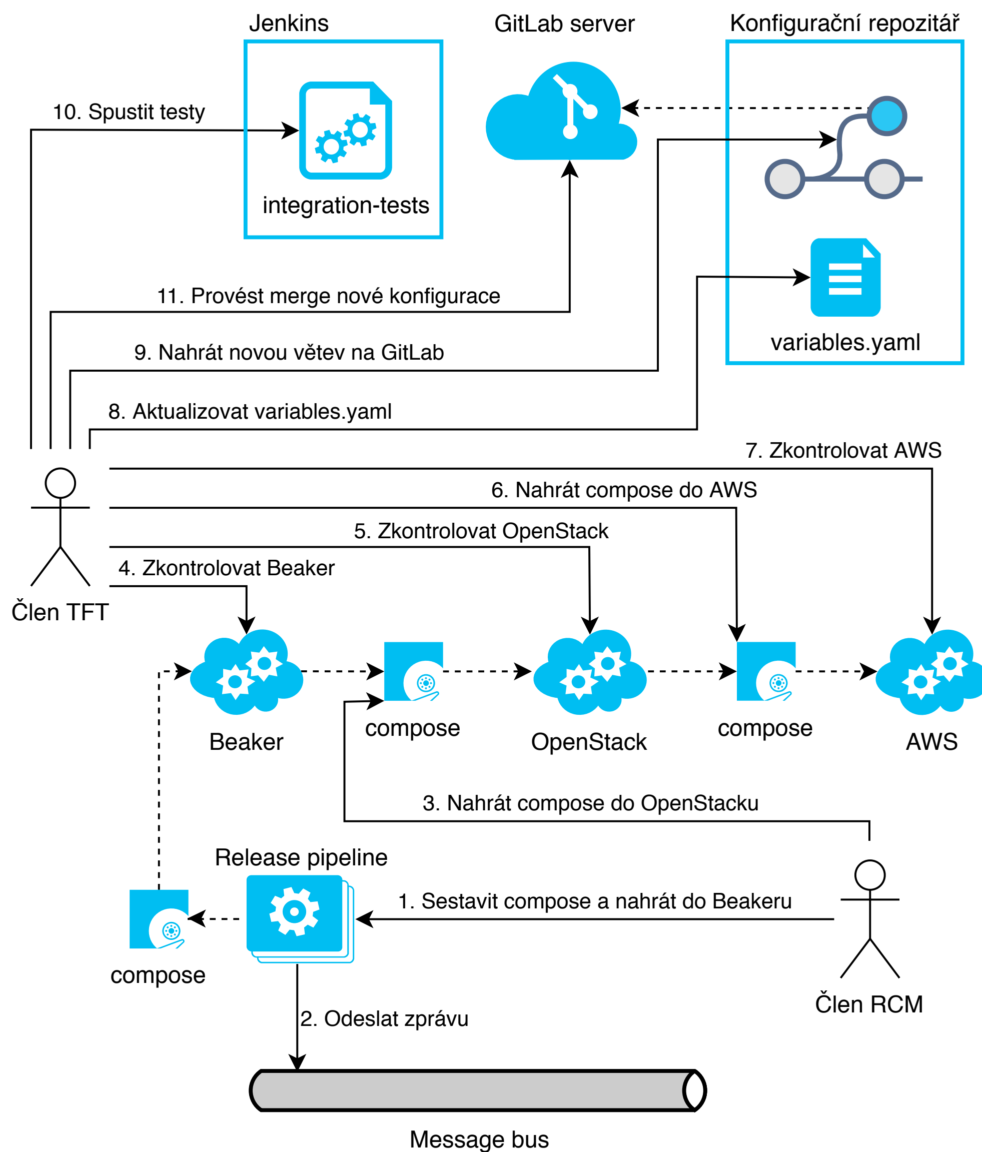
## 4.1 Aktualizace a otestování systému průběžné integrace s novým sestavením operačního systému

Nejvhodnější úlohou, která značně usnadní údržbu systému průběžné integrace je aktualizace konfigurace, podle které se systém rozhoduje, kterou verzi operačního systému vybere pro vykonávání testů.

Aby bylo testování balíčků RPM na dané verzi operačního systému co nejaktuálnější, je nutné také zajistit, aby systém testoval na co nejnovější verzi tohoto operačního systému. Nové verze operačního systému (v této a následující kapitole budou označovány slovem *compose*) se sestavují pravidelně jednou až dvakrát denně.

Aktualizace konfigurace spočívá v úpravě jednoho souboru `variables.yaml`, který se nachází v konfiguračním repozitáři. Než je ovšem tuto změnu možné provést, je nutné provést několik kroků pro ověření, že systém s novým compose bude pracovat správně.

Následující diagram v Obrázku 4.1 zobrazuje výchozí stav, kde největší pozornost je kladena na *člena týmu Testing Farm* a operacím, které musí vykonat, aby zaktualizoval systém průběžné integrace.




**Obrázek 4.1:** *Diagram zobrazující výchozí stav.*

Z obrázku vyplývají dvě věci, zaprvé vyobrazuje *Člena TFT*, který vykonává velké množství operací. Cílem je vytvořit dílčí skripty, které budou snadno spustitelné člověkem i použitelné v rámci skriptu. Skripty musí být vytvořeny pro následující kroky:

4. Zkontrolovat Beaker
5. Zkontrolovat OpenStack
6. Nahrát compose do AWS
7. Zkontrolovat AWS
8. Aktualizovat variables.yaml
9. Nahrát novou větev na GitLab

10. Spustit testy

Následující sekce podrobněji rozepisuje, co tyto kroky obnáší.

Dále diagram zobrazuje, jak se daný compose dostává do jednotlivých služeb. Za vydání nové verze je zodpovědný tým *Release Configuration Management*, zkráceně *RCM*. RCM se stará o vydávání různých produktů, mezi ně se řadí i nové verze operačních systémů *Red Hat Enterprise Linux* a *Fedora Linux*.[11]

### 4.1.1 Ověření výskytu verze operačního systému na testovacích infrastrukturách

Této části odpovídají následující kroky z diagramu v Obrázku 4.1:

4. Zkontrolovat Beaker
5. Zkontrolovat OpenStack
6. Nahrát compose do AWS
7. Zkontrolovat AWS

Testovacími infrastrukturami se rozumí virtuální, ale i fyzické stroje, na kterých systém průběžné integrace spouští testy. V současné době systém využívá celkem tři služby, které tyto stroje poskytují, a to *OpenStack*, *Beaker* a *AWS* (zobrazeny na schématu popisující komponenty systému průběžné integrace 3.4). Pro tyto služby se také používá označení *cloud*.

Aby se na rezervovaných strojích mohl nainstalovat požadovaný compose, musí se nejprve do každého cloudu nahrát zvlášť.

### 4.1.2 Aktualizace konfigurace

Z diagramu v Obrázku 4.1 tomuto kroku odpovídá:

8. Aktualizovat variables.yaml
9. Nahrát novou větev na GitLab

Jak již bylo zmíněno, aktualizace konfigurace spočívá v modifikaci jediného souboru `variables.yaml`, který se nachází v konfiguračním repozitáři.

Nástroj by měl dle názvu compose správně nahradit požadované řetězce v souboru `variables.yaml`, dále by měl tuto změnu automaticky nahrát do vzdáleného repozitáře a vytvořit pro ní merge request.

### 4.1.3 Spuštění integračních testů

Jako poslední zbývá krok:

10. Spustit testy

V tomto kontextu se integračními testy rozumí testy, které slouží pro ověření, že systém průběžné integrace pracuje správně. Na serveru Jenkins pro tento účel již existuje úloha *integration-tests*, která s vhodně zvolenými vstupními parametry otestuje požadovanou funkčnost systému.

Úloha *integration-tests* umožňuje testovat libovolnou větev konfiguračního repozitáře, z tohoto důvodu došlo k nahrání změn do vzdáleného repozitáře před spuštěním testů.

Nástroj spustí tuto Jenkins úlohu prostřednictvím *REST API*. Nástroj dále také zaznamená výsledky testů v komentáři, který přidá v konfiguračním repozitáři k merge requstu, který byl vytvořen

## 4.2 Nástroj tft-admin

Nástroj tft-admin je program s uživatelským rozhraním v příkazovém řádku (CLI, Command Line Interface), jehož účel spočívá v usnadnění přístupu k různým komponentám patřícím do systému průběžné integrace (věnuje se mu kapitola 3.3.1).

Příkazy, které uživatel zadává na příkazový řádek, zprostředkovávají informace související se systémem průběžné integrace, případně prostřednictvím skriptů vykonávají různé činnosti.

Mezi některé již existující příkazy patří například:

- `tft-admin alerts current` na standardní výstup vypíše seznam aktuálních varování, která souvisí se systém průběžné integrace, a mohou tak narušovat jeho činnost.
- `tft-admin queue running` na standardní výstup vypíše seznam aktuálně vykonávaných testovacích úloh Jenkins serverem.

Jedním z dílčích cílů práce je naimplementovat další příkazy do tohoto nástroje, které budou vykonávat jednotlivé kroky v automatizovaných úlohách. Před tím, než ovšem bude možné tyto dílčí příkazy implementovat, je na místě provést revizi tohoto nástroje a přizpůsobit jeho fungování ve skriptech, které budou příkazy řetězit a v některých případech zpracovávat jejich výstup.

### 4.2.1 Implementační detaily

Tato část popisuje blíže jazyk *Python* a framework *Click*, který byl použit při vývoji programu tft-admin.

#### Jazyk Python

Python je jednoduchý na naučení, silný programovací jazyk. Obsahuje efektivní vysokoúrovňové datové struktury a jednoduchý a efektivní přístup k objektově orientovanému programování. Elegantní syntaxe a dynamické typování dohromady s charakteristikou interpretovaného jazyka tvoří z Pythonu ideální jazyk pro skriptování a rychlý vývoj aplikací v mnoha oblastech na většině platforem.[12]

#### Framework Click

Click (Command Line Interface Creation Kit) je balíček pro Python umožňující vytváření programů s rozhraním v příkazové řádce, programátorům tedy usnadňuje práci při vytváření těchto programů.[2]

- Libovolné zanořování příkazů.
- Automatické vytváření stránek s nápovědou o daném programu.
- Podporuje princip *lazy loading* vnořených příkazů během běhu programu.

Pro ilustraci si ukážeme příklad jednoduchého programu:

```
import click

@click.command()
@click.option('--count', default=1, help='Number of greetings.')
@click.option('--name', prompt='Your name',
              help='The person to greet.')
def hello(count, name):
    """Simple program that greets NAME for a total of COUNT times."""
    for x in range(count):
        click.echo('Hello %s!' % name)

if __name__ == '__main__':
    hello()
```

**Výpis 4.1:** *Ukázka jednoduchého programu ve frameworku Click. Kód převzat z [2].*

Při interpretaci tohoto zdrojového kódu se na standardní výstup vytiskne následující text:

```
$ python hello.py --count=3
Your name: John
Hello John!
Hello John!
Hello John!
```

**Výpis 4.2:** *Výstup programu z Výpisu 4.1.*

### Jazyk YAML

Jazyk *YAML* (rekurzivní akronym *YAML Ain't Markup Language*) je jazyk pro serializaci dat vytvořený okolo běžných datových typů agilních programovacích jazyků.[9] Jazyk YAML se hojně využívá například pro konfigurační soubory nebo internetové komunikační aplikace.

Při vývoji tohoto jazyka byl kladen důraz na to, aby splňoval následující požadavky:

- YAML je snadno čitelný pro člověka.
- Data ve formátu YAML jsou přenositelná mezi programovacími jazyky.
- Formát YAML odpovídá nativním datovým strukturám v programovacích jazycích.
- YAML obsahuje konzistentní model pro podporu generických nástrojů.
- YAML podporuje jednoprůchodové zpracování.

- YAML je expresivní a rozšiřitelný.
- YAML je jednoduchý na implementaci a použití.

#### Konfigurační soubory nástroje `tft-admin`

Nástroj `tft-admin` pro správnou funkčnost vyžaduje soubory `config.yaml` a `session.yaml`. Jak plyne z jejich názvu, jejich formát odpovídá jazyku YAML. Oba soubory se musí vyskytovat v adresáři `/.config/tft-admin/`.

Soubor `config.yaml` obsahuje nezbytné údaje potřebné například pro komunikaci s různými službami nebo cesty k některým lokálním adresářům a souborům. Některé příkazy nástroje například komunikují s Jenkins serverem, proto se v konfiguraci musí vyskytovat autentizační údaje s tímto serverem.

Soubor `session.yaml`, jak plyne z názvu, ukládá uživatelské sezení. Ve výchozím stavu se používá pouze pro zapamatování větve lokálního konfiguračního repozitáře *citool-config*, se kterým nástroj bude pracovat.

## 4.3 Využití Jenkins serveru k automatizaci

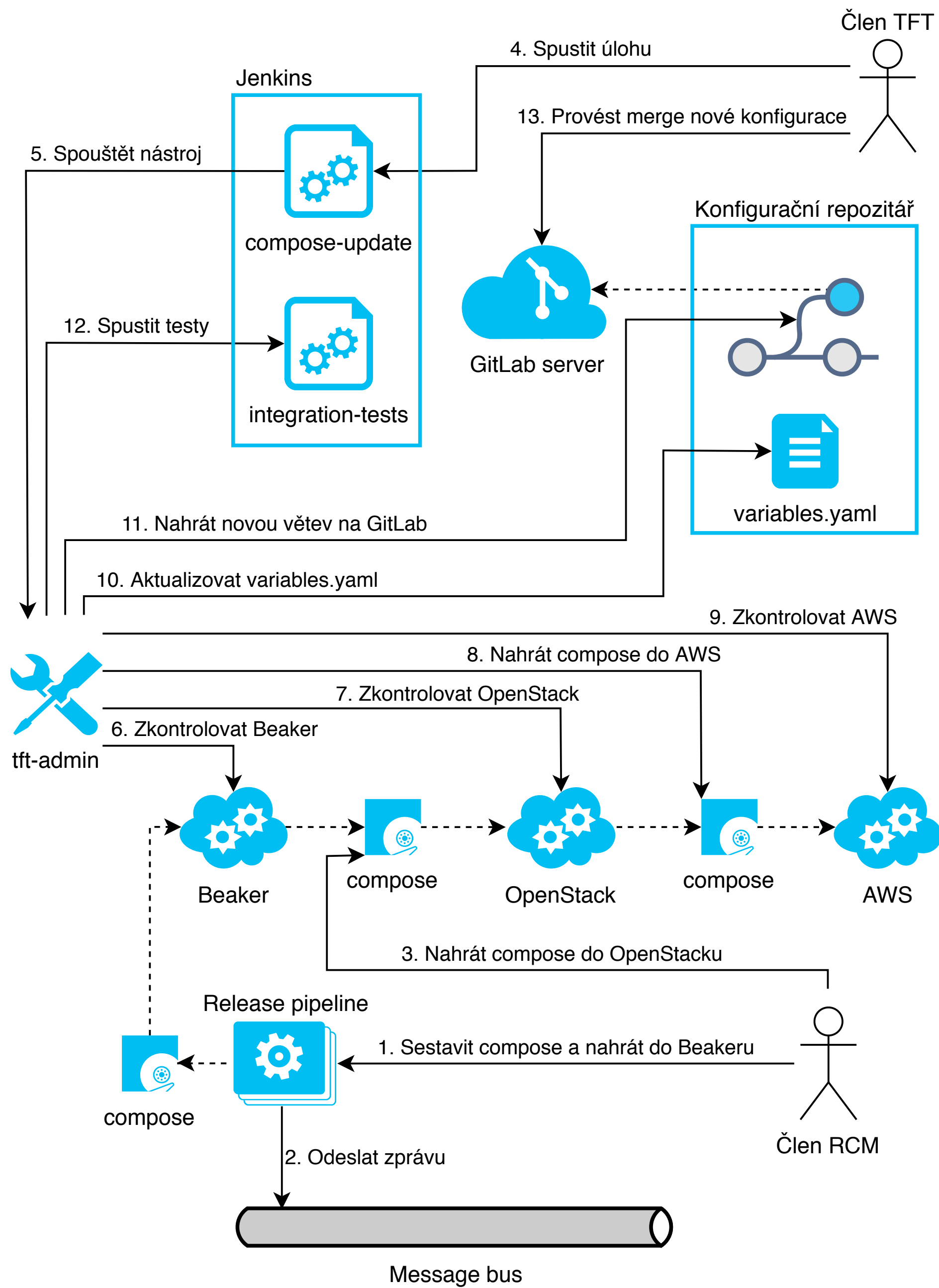
V kapitole 2.2 jsme si představili automatizační server Jenkins a následně v kapitole 3.3.1 rozšířili tyto znalosti o tom, jak jej využívá systém průběžné integrace.

V předchozích sekcích jsou představeny dílčí kroky, které je nezbytné vykonat k aktualizování systému průběžné integrace s novou verzí compose. Tyto dílčí kroky budou umět vykonávat příkazy nástroje `tft-admin`. Poslední částí je vytvořit skript, který spojí tyto dílčí příkazy do jednoho celku, který vykoná tuto úlohu. Tento skript se bude spouštět v rámci Jenkins úlohy.

## 4.4 Výsledný stav

Následující diagram v Obrázku 4.2 zobrazuje výsledný stav, kterého se dosáhne navrženou implementací Jenkins úlohy `compose-update` a příkazů do nástroje `tft-admin`.

Většina operací, které musel vykonávat *člen TFT* se přenese na nástroj `tft-admin`, který bude spouštěný v Jenkins úloze `compose-update`.




**Obrázek 4.2:** *Diagram zobrazující výsledný stav.*

### 4.4.1 Odlišnost od počáteční koncepce práce

Jak je z diagramu v Obrázku 4.2 patrné, na *člena TFT* se přenesly pouze dvě operace, které musí vykonat, aby aktualizoval systém průběžné integrace:

1. Spustit úlohu `compose-update`, která prostřednictvím nástroje `tft-admin` vykoná většinu činností, které ve výchozím stavu musel vykonat člověk.
2. Zkontrolovat výstup úlohy na GitLab serveru, kam se zapíší relevantní výstupy z úlohy `compose-update`. V případě, že úloha proběhla úspěšně, je možné nasadit novou verzi konfigurace systému.

Tato skutečnost se ovšem rozchází s původní myšlenkou práce, která spočívala v automatickém spuštění úlohy v reakci na událost. Dle obrázku 4.2 se nabízí reagovat na zprávu z *message busu*, která vznikne současně s vydáním nového compose týmem *RCM*.

Reakce na tuto událost v současnosti není možná, neboť úlohu lze spustit až poté, co *člen TFT* nahraje daný compose do cloudu OpenStack a obeznámí tím *členy TFT* prostřednictvím komunikačního kanálu IRC, kteří následně mohou spustit úlohu `compose-update`, jak je navržena.

Automatizací tohoto článku by se docílilo stavu, kdy by bylo možné spouštět úlohu automaticky při vzniknutí nového compose týmem RCM.

# Kapitola 5

# Implementace automatizace manuálních úloh

## 5.1 Úpravy nástroje tft-admin

Jak již bylo nastíněno v předchozí kapitole, nástroj `tft-admin` je užitečný pomocník pro lidi starající se o systém průběžné integrace, ovšem nevyužívá všechen svůj potenciál, neboť v současném stavu je určen pro zacházení pouze s uživatelem. Cílem je využít příkazy tohoto nástroje v automatizovaných skriptech v úlohách na serveru Jenkins. Z tohoto důvodu je nezbytné provést několik úprav v tomto nástroji.

### 5.1.1 Rozlišení výpisů na standardním výstupu

Při spuštění některého příkazu začne nástroj vypisovat text na standardní výstup. Tyto výpisy se dají rozdělit do dvou kategorií:

- Výpisy logovacího charakteru, které uživatele zpravidla informují o dění běhu příkazu.
- Užitečná data, která příkazy nasbírají v průběhu vykonávání a zpravidla vypisují na konci běhu příkazu.

Výpisy na standardní výstup, které informují uživatele o průběhu příkazu je nutné přesměrovat na standardní chybový výstup. Mezi tyto výpisy patří logovací zprávy, výzvy pro zadání vstupu uživatelem, grafické prvky indikující průběh, kdy proces čeká například na příchod požadavku ze serveru.

Výstupy programu, které neslouží jako pouhá informace o průběhu, ale obsahují užitečná data (například seznam aktuálně běžících testovacích úloh), budou ponechány na standardním výstupu.

### 5.1.2 Přepínač pro režim strojově čitelného výstupu

Nyní jsou v nástroji rozlišeny dva typy výstupů – užitečná data, která vypisují příkazy před ukončením jejich činnosti, se vypisují na standardní výstup, ostatní výpisy se tisknou na standardní chybový výstup.

Nyní je zapotřebí nabídnout různé formáty pro výpisy na standardním výstupu. Pokud nástroj používá uživatel, nástroj by měl tisknout ve formátu, který je snadno čitelný pro

člověka (např. data zobrazená v tabulce). Pokud se nástroj používá ve skriptech, je potřeba, aby data byla tisknuta ve strojově čitelném formátu (např. *JSON*).

K příkazu nejvyšší úrovně, kterým je `tft-admin` se přidá přepínač `--format TEXT`, kde `TEXT` musí být buď `json` nebo `human`. Tento přepínač určuje, zda se data vytisknou ve strojově čitelném formátu, nebo v tabulkovém:

```
@click.option(
    '--format',
    type=click.Choice(['human', 'json']),
    help='Print command output in human or machine format.'
)
```

**Výpis 5.1:** *Přepínač pro příkaz nejvyšší úrovně, který se používá pro rozlišení mezi různými formáty výstupu.*

V nástroji je již implementovaná funkce `print_table()`, která, jak je zřejmé z názvu, slouží k tisknutí tabulek na standardní výstup. Druhou její funkcionalitou je ovšem tisknutí tabulky ve formátu *JSON* nebo *YAML*. Přepínač tak snadno rozhodne, který z těchto formátů se má zvolit.

### 5.1.3 Výsledný stav

Po splnění těchto dvou bodů je dosaženo stavu, kdy je nástroj přizpůsobený pro použití jak uživatelem, tak v automatizovaných skriptech a snadno se používá s různými unixovými utilitami.

Následující výpisy demonstrují různé způsoby výpisů u jednoho konkrétního příkazu, jemuž se věnuje následující kapitola 5.2.

Následující příkaz vytiskne informace o daném compose na aktuálně nastaveném cloudu v tabulkovém formátu (implicitní formát určený pro uživatele):

```
$ tft-admin cloud compose sync --compose Fedora-Rawhide
[os] looking for 1MT-Fedora-Rawhide first
[os] compose found successfully

+--------------------+--------------------------------------+--------+
| Name               | ID                                   | Status |
+--------------------+--------------------------------------+--------+
| 1MT-Fedora-Rawhide | e5b6f3d2-3c9e-435a-b5c8-4ae09cf501ae | active |
+--------------------+--------------------------------------+--------+
```

**Výpis 5.2:** *Příklad použití příkazu s implicitním tabulkovým výstupem.*

Příkaz vytiskne stejná data, ale ve formátu *JSON* a použitím `2> /dev/null` zahodí výpisy na standardním chybovém výstupu:

```
$ tft-admin --format json cloud compose sync --compose Fedora-Rawhide \
2> /dev/null
[
    {
        "id": "e5b6f3d2-3c9e-435a-b5c8-4ae09cf501ae",
```

```
        "name": "1MT-Fedora-Rawhide",
        "status": "active"
    }
]
```

**Výpis 5.3:** *Příklad použití příkazu se strojově čitelným výstupem.*

Příkaz vytiskne stejná data jako v předchozích dvou příkladech, ale nástrojem `jq` se dotážeme na položku status v daném výstupu:

```
$ tft-admin --format json cloud compose sync --compose Fedora-Rawhide \
2> /dev/null | jq .[0].status
"active"
```

**Výpis 5.4:** *Příklad použití výstupu příkazu.*

## 5.2 Aktualizace a otestování systému průběžné integrace s novým sestavením operačního systému

Systém průběžné integrace, jehož činnost jsme si podrobněji popsali v kapitole 3.3.1, testuje správné fungování balíčků RPM (popsaný v kapitole 3.2.1) na určité verzi operačního systému, kterou nejčastěji bývá nejnovější dostupná. Tato verze operačního systému bývá také nazývána *compose*. Mezi komponenty systému patří repozitář obsahující konfigurační soubory, jeden z těchto souborů obsahuje informaci, který compose má systém zvolit ke spuštění testů.

Cílem této úlohy je modifikovat soubor v konfiguračním repozitáři s novou verzí compose. Tato kapitola se věnuje implementaci dílčích příkazů do nástroje `tft-admin`, které tuto činnost budou schopné vykonat.

### 5.2.1 Rozšíření konfigurace o nutné položky

Jak již bylo vysvětleno v předchozí kapitole 4.2.1, soubor `config.yaml` ukládá data, která jsou nezbytná pro vykonávání většiny příkazů. Ani příkazy určené k vykonávání této úlohy nebudou výjimkou, a tak bude nezbytné přidat některé položky do tohoto konfiguračního souboru.

V první řadě je nutné přidat cesty k programům, které komunikují s cloudy (věnuje se jim kapitola 3.3.1 popisující systém průběžné integrace), tedy pro *OpenStack*, *Beaker* a *AWS*. Pro všechny tyto infrastruktury již existují požadované programy `openstack`, `bkr`, respektive `aws`, které při poskytnutí autentizačních údajů lze využít pro komunikaci s danou službou.

```
openstack_cli: /usr/bin/openstack
beaker_cli: /usr/bin/bkr
aws_cli: ~/.virtualenvs/tft-admin/bin/aws
```

**Výpis 5.5:** *Příklad části konfiguračního souboru, která obsahuje cesty k požadovaným nástrojům pro komunikaci s cloudy.*

Tyto autentizační údaje je tedy nezbytné poskytnout nástroji `tft-admin`. Vzhledem k tomu, že může existovat více instancí těchto služeb, je nezbytné, aby nástroj umožňoval

připojení k libovolné instanci, ke které mu uživatel poskytne autentizační údaje. V současné době se ovšem používá jedna instance pro každou z těchto tří služeb.

Následující výpis ukazuje způsob zápisu autentizačních údajů k jednotlivým instancím těchto služeb v konfiguračním souboru `config.yaml`:

```
clouds:
    os:
        openstack:
            auth_url: ###
            user_domain_name: ###
            identity_api_version: 3
            project_domain_name: ###
            project_name: baseos-jenkins
            username: ###
            password: ###
    bkr:
        beaker:
    aws:
        aws:
            access_key_id: ###
            secret_access_key: ###
            region: us-east-1
            s3_bucket_name: tft-image-import
            tags:
                ServiceOwner: TFT
```

**Výpis 5.6:** *Příklad části konfiguračního souboru, která definuje cloudy. Některé interní údaje byly odstraněny.*

Název `clouds` označuje objekt, pod nímž se nachází definice. Pod tímto objektem se nachází položky `os`, `bkr` a `aws`, které pojmenovávají instance pouze v rámci nástroje `tft-admin`. Na další úrovni se nachází položky, které striktně musí být buď `openstack`, `beaker`, nebo `aws` a udávají typ cloudu.

### 5.2.2 Implementace příkazů nutných k vykonání dílčích kroků

Před tím, než danou verzi operačního systému (compose) bude moci systém průběžné integrace začít využívat, je nezbytné, aby byla přítomná na testovacích infrastrukturách a také aby byla obsažená v konfiguračním souboru `variables.yaml`. K tomuto účelu byly implementovány následující příkazy:

- `tft-admin cloud set` nastaví kontext v rámci nástroje `tft-admin` na některý z nakonfigurovaných cloudů, jenž mohou být typu *OpenStack*, *Beaker* nebo *AWS*.
- `tft-admin cloud compose sync` se dotáže aktuálně nastaveného cloudu, zda se na něm nachází určitý compose. Na cloud typu *AWS* umí nahrát compose, pokud se vyskytuje na cloudu typu *OpenStack*.
- `tft-admin config compose-update` v konfiguračním souboru `variables.yaml` nahradí řetězce novým názvem compose.

- `tft-admin config compose-propose` provedené změny v konfiguračním souboru nahraje do vzdáleného repozitáře a vytvoří merge request.
- `tft-admin config test` spustí integrační testy nad určitou větví konfiguračního repozitáře. Výsledky testu zapíše do merge requestu této větve.

Zbytek této sekce se věnuje jejich podrobnějšímu popisu a implementačních detailů.

#### Příkaz `tft-admin cloud set [NAME]`

Příkaz slouží k přepnutí kontextu mezi jednotlivými infrastrukturami (cloudy). Volitelný argument udává název cloudu, na nějž se má nástroj přepnout. Dostupné cloudy jsou uloženy v konfiguračním souboru `config.yaml` spolu Záznam o aktuálně nastaveném cloudu se ukládá do souboru `session.yaml`. V případě, že argument nebyl zadán, vytiskne se název aktuálně nastaveného cloudu.

#### Příkaz `tft-admin cloud compose sync`

Tento příkaz plní celkem dvě funkce. První, při zadání pouze povinného parametru `--compose NÁZEV` se odešle požadavek na cloud, zda se na něm nachází compose s názvem `NÁZEV`, tento příkaz demonstruje následující výpis:

```
$ tft-admin cloud compose sync --compose Fedora-Rawhide
[os] looking for 1MT-Fedora-Rawhide first
[os] compose found successfully

+--------------------+--------------------------------------+--------+
| Name               | ID                                   | Status |
+--------------------+--------------------------------------+--------+
| 1MT-Fedora-Rawhide | e5b6f3d2-3c9e-435a-b5c8-4ae09cf501ae | active |
+--------------------+--------------------------------------+--------+
```

**Výpis 5.7:** *Příklad výstupu příkazu `tft-admin cloud compose sync`.*

Druhá funkce se aktivuje použitím volitelného přepínače `--from-cloud NÁZEV`. Příkaz stáhne compose daný argumentem přepínače `--compose` z cloudu, jehož jméno je dáno argumentem přepínače `--from-cloud`. Tento compose se posléze nahraje na cloud, kterým je aktuálně nastavený v souboru `session.yaml`.

V současné době je tato funkce omezená na jediný možný případ – stahování compose umožňuje pouze cloud typu *OpenStack* a nahrávání compose podporuje pouze cloud typu *AWS*.

Příkaz provádí nahrávání compose dle následujícího algoritmu:

1. Cloud z aktuálního kontextu nastav jako cílový.
2. Odešli dotaz na cílový cloud, zda se na něm nachází daný compose (podobně jako bez použitého parametru `--from-cloud`). V případě, že ano, vytiskni informace o dostupném compose a ukonči úspěšně program, jinak pokračuj dalším bodem.
3. Z parametru `--from-cloud název` vyhledej v konfigračním souboru cloud s názvem *název* a nastav ho jako zdrojový. V případě, že daný cloud nebyl nalezen, ukonči program neúspěšně, jinak pokračuj dalším bodem.

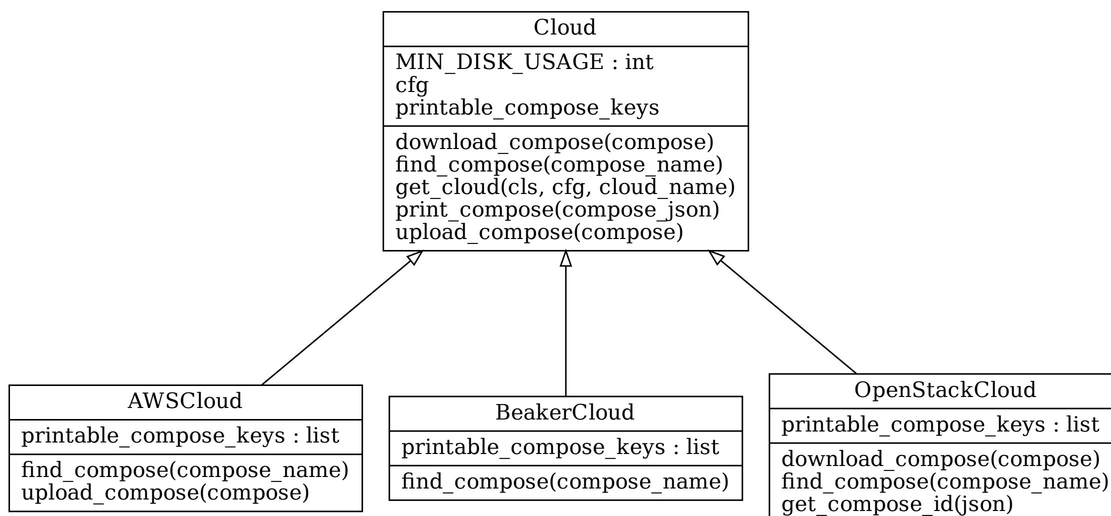
4. Odešli dotaz na zdrojový cloud, zda se na něm nachází požadovaný compose. V případě, že ne, ukonči program neúspěšně, jinak pokračuj dalším bodem.
5. Ze zdrojového cloudu stáhni image daného compose.
6. Pokud je to potřeba, pomocí nástroje `qemu-img` zkonvertuj stažený image do formátu, který cílový cloud podporuje.
7. Uploaduj (zkonvertovaný) image na cílový cloud.
8. Vytiskni informace o nahraném compose a ukonči program úspěšně.

Pro práci s cloudy, kterou jsme si právě popsali, se ve zdrojovém kódu nachází třída `Cloud` a její potomci `AWSCloud`, `BeakerCloud` a `OpenStackCloud`.

Při vytváření instance odpovídající třídy dané typem cloudu se využívá metoda `Cloud.get_cloud(cfg, cloud_name)`, která dle názvu daného parametrem `cloud_name` vyhledá v konfiguračním souboru odpovídající cloud a dle jeho typu vytvoří objekt `AWSCloud`, `BeakerCloud`, nebo `OpenStackCloud`, který nainicializuje a posléze vrátí.

Tedy například při použití metody `cloud = Cloud.get_cloud(cfg)`, kde parametr `cfg` je objekt odvozený z třídy `Configuration`, která poskytuje data z konfiguračního souboru, se vytvoří objekt odvozený z odpovídající třídy. Třídy podporují vlastnost objektově orientovaných jazyků polymorfismus, tedy například pro dotazování se na výskyt určitého compose poskytují všechny třídy stejné rozhraní metodou `cloud.find_compose(název)`.

Vztah mezi jednotlivými třídami zobrazuje také následující diagram tříd v Obrázku 5.1:




**Obrázek 5.1:** *Diagram tříd zobrazující třídu `Cloud` a její potomky.*

### Příkaz `tft-admin config compose-update`

Cílem tohoto příkazu je modifikovat soubor `variables.yaml` dle poskytnutých parametrů a uložit ho.

Soubor `variables.yaml` se nachází v konfiguračním repozitáři a slouží systému průběžné integrace, aby před spuštěním testů vybral vhodnou verzi operačního systému na určité infrastruktuře.

```
RHEL_8_3_0:
  compose: RHEL-8.3.0-20200616.0
  buildroot: BUILDROOT-8.3.0-RHEL-8-20200616.0
  aws:
    image: 1MT-RHEL-8.3.0-20200616.0
  openstack:
    image: 1MT-RHEL-8.3.0-20200616.0
  beaker:
    distro: RHEL-8.3.0-20200616.0
```

**Výpis 5.8:** *Ukázka části konfiguračního souboru `variables.yaml`.*

Příkaz obsahuje následující argument a přepínače:

- Příkaz obsahuje povinný argument `SECTION`, který udává, v jaké sekci se budou řetězce nahrazovat. Sekce je v souboru objekt na nejvyšší úrovni a zpravidla nese název dle verze operačního systému, ve Výpisu 5.8 tomuto odpovídá objekt s klíčem `RHEL_8_3_0`.
- `--compose NAME` v dané sekci nahrazuje hodnoty `compose`, `buildroot`, `aws.image`, `openstack.image`, `beaker.distro`, tedy podle Výpisu 5.8 všechny hodnoty v sekci `RHEL_8_3_0`.
- `--buildroot NAME` v dané sekci přepíše hodnotu danou klíčem `buildroot`.
- `--openstack-image NAME` v dané sekci přepíše hodnotu danou klíčem `openstack.image`.
- `--beaker-distro NAME` v dané sekci přepíše hodnotu danou klíčem `beaker.distro`.
- `--aws-image NAME` v dané sekci přepíše hodnotu danou klíčem `aws.image`.
- `--force-with-dirty-tree` v případě, že lokální repozitář obsahuje necommitnuté změny (výstup příkazu `git diff HEAD` je neprázdný), příkaz na tuto skutečnost upozorní uživatele a vyzve ho, zda si přeje, aby příkaz pokračoval. Tento přepínač obchází toto potvrzení.

Činnost příkazu podrobně popisují následující kroky:

1. Příkazem `git diff HEAD` se zkontroluje, zda konfigurační repozitář obsahuje změny. Pokud ano, uživatel je vyzván, zda si přeje pokračovat.
2. Prostřednictvím modulu `ruamel.yaml` se načte konfigurační soubor `variables.yaml` do slovníku.
3. V načteném slovníku se vyhledá, zda obsahuje sekci danou argumentem příkazu, pokud se v něm tato sekce nenachází, příkaz skončí neúspěšně.
4. V sekci se provedou náhrady řetězců dle zadaných přepínačů.
5. V sekci se vyhledají parametry pro integrační testy, které jsou uloženy v komentářích.
6. Na standardní výstup se vypíše název sekce, která byla modifikována a přepínače pro integrační testy pro tuto sekci, aby později její změny byly otestovány.

### Příkaz `tft-admin config compose-propose`

Příkaz, v případě, že lokální konfigurační repozitář obsahuje změny, vytvoří novou větev, která bude obsahovat tyto změny, dále automaticky vytvoří commit s těmito změnami, nahraje ho do vzdáleného repozitáře, který se nachází na serveru GitLab a pro tuto nově nahranou větev vytvoří merge request, kde cílovou větví bude původní větev, kterou měl lokální repozitář aktuálně nastavenou před spuštěním tohoto příkazu.

Příkaz obsahuje následující přepínače:

- `--branch NAME` Volitelný parametr. Udává název větve, která se nově vytvoří. Pokud je název duplicitní, na konci řetězce se automaticky přidá sufix, aby byla zaručena jeho unikátnost na serveru *GitLab*.
- `--title NAME` Volitelný parametr. Na serveru *GitLab* udává titulek k merge requestu, který se nově vytvoří.

Činnost příkazu je podrobněji vyobrazena v následujícíh bodech:

1. Kontrola, zda lokální repozitář obsahuje změny se provádí příkazem `git diff HEAD`, pokud je standardní výstup tohoto příkazu neprázdný, znamená to, že lokální repozitář obsahuje změny.
2. Využitím knihovny `ogr`[1] se nainicializuje instance třídy `GitlabProject`, která poskytuje aplikační rozhraní ke vzdálenému repozitáři uloženému na serveru *GitLab*. Pro tento účel vznikla funkce `get_gitlab_project()`.
3. Použije se argument parametru `--branch` nebo se vygeneruje název budoucí větve ve formátu `variables-bump-YYYY-MM-DD`, kde `variables-bump` je označení pro aktualizaci souboru *variables.yaml* a `YYYY-MM-DD` je aktuální datum. Pokud by se ve vzdáleném repozitáři již vyskytovala větev se stejným názvem, přidá se na konec řetězce sufix `-N`, kde `N` je takové číslo, aby zajistilo unikátnost vygenerovanáho názvu větve.
4. Uloží se název aktuální větve, kterou vypisuje příkaz `git rev-parse, -abbrev-ref HEAD`.
5. Vytvoří se nová větev s vygenerovaným názvem.
6. Vytvoří se commit se změněným konfiguračním souborem `variables.yaml`, commit se odešle do vzdáleného repozitáře.
7. Metodou `GitlabProject.create_pr()` se vytvoří ve vzdáleném repozitáři merge request, kde zdrojovou větví je nová větev a cílovou větví větev uložená v bodě 4.
8. Na standardní výstup se vypíše název nově vytvořené větve.

### Příkaz `tft-admin config test`

Poté, co do vzdáleného repozitáře byla nahraná nová větev a také byl vytvořen merge request je potřeba spustit na Jenkins serveru integrační testy, abychom si ověřili, že tato nová úprava souboru `variables.yaml` nenarušila chod systému průběžné integrace.

Příkaz obsahuje následující přepínače:

[1] https://github.com/packit-service/ogr

- `--branch NAME` určuje název větve ve vzdáleném repozitáři, nad kterou se mají spustit integrační testy.
- `--test-options OPTIONS` jsou paramtery pro integrační testy, které se mají spustit. Argument umožňuje zadat buď jeden testovací parametr (např. `--test-options test-option1`), nebo několik parametrů najednou ve formátu JSON seznamu (např. `--test-options '["test-option1", "test-option2"]'`)
- `--force-run` příkaz v průběhu vyzve uživatele, zda si skutečně přeje spustit integrační testy. Tento přepínač toto potvrzení obchází a rovnou je spouští. Užitečné v případě, kdy se příkaz používá ve skriptech, aby nebyl nutný zásah člověka.

1. Pomocí funkce `get_gitlab_project()` se nainicilazuje instance třídy `GitlabProject`.
2. Zkontroluje se, zda větev dána argumentem přepínače `--branch` existuje ve vzdáleném repozitáři.
3. Ve vzdáleném repozitáři se vyhledá merge request, ve kterém zdrojová větev je hodnota argumentu `--branch` a cílová je aktuálně nastavená větev v lokálním repozitáři. Pokud se takový merge request nepodaří nalézt, testy se také spustí, jejich výsledek se ovšem nezapíše k merge requestu, který obsahuje změny, které se testují.
4. Pro každou hodnotu ze seznamu argumentu `--test-options` se spustí vlákno. Každé toto vlákno spustí integrační test a vyčká na jeho dokončení.
5. Po ukončení činnosti všech vláken hlavní větev programu vypíše na standardní výstup tabulku s informacemi o běhu těchto integračních testů, jejich unikátní identifikátor v rámci Jenkins úloh, URL na Jenkins server a jejich výsledek.

### 5.2.3 Jenkins úloha pro vykonání skriptu

Nyní již existují příkazy, které umí vykonat dílčí kroky potřebné k automatizaci této úlohy. Jak plyne z návrhu a obrázku 4.2, posledním krokem je vytvořit Jenkins úlohu, která s využitím příkazů nástroje `tft-admin` vykoná potřebné kroky.

Pro tento účel tedy vznikla úloha `compose-update`, která sestává z dvou souborů: `compose-update.yaml` popisující údaje o úloze a `Jenkinsfile` obsahující skript, který vykonává požadovanou úlohu.

Samotná Jenkins úloha `compose-update` obsahuje následující vstupní parametry, aby mohla být spuštěna:

- `compose_name` – Název compose, který bude systém průběžné integrace nově používat.
- `variables_sections` – Volitelný parametr. Názvy sekcí oddělené čárkou v souboru `variables.yaml`, uvnitř kterých má dojít k textové náhradě nového compose. Pokud není zadán, vyvodí se z argumentu `compose_name`.
- `buildroot` – Volitelný parametr. Při náhradě hodnoty `buildroot` převažuje parametr `compose_name`.
- `aws_image` – Volitelný parametr. Při náhradě hodnoty `aws.image` převažuje parametr `compose_name`.

- beaker_distro – Volitelný parametr. Při náhradě hodnoty beaker.distro převažuje parametr compose_name.
- openstack_image – Volitelný parametr. Při náhradě hodnoty openstack.image převažuje parametr compose_name.

### Shell skript uvnitř úlohy

Nejzajímavější částí této Jenkins úlohy je obsažený shell skript, kde většina jeho příkazů se skládá z nově implementovaných příkazů do nástroje tft-admin.

Jelikož se nástroj tft-admin poměrně často mění, zajišťování jeho správné instalace společně s konfigurací na strojích vykonávající Jenkins úlohy při každé změně by byla časově náročná záležitost. Z těchto důvodů existuje v týmu Testing Farm image označovaný názvem *tft-cnc* (zkratka pro *Testing Farm Team – Command and Control*), který v sobě zahrnuje nejaktuálnější instalace nástrojů používaných v rámci týmu.

Nejvhodnějším způsobem, jak spouštět příkazy nástroje tft-admin je tedy uvnitř kontejneru prostřednictvím zmíněného image *tft-cnc*. Ke komunikaci s image uvnitř kontejneru se využívá open-source nástroj *docker*[2].

Uvnitř shell skriptu se nejprve příkazem docker pull získá nejaktuálnější verze image *tft-cnc*. Dále se příkazem docker run spustí kontejner a jeho identifikátor se uloží do proměnné container_id.

Uvnitř kontejneru image *tft-cnc* je před každým příkazem také vhodné spustit skript /entrypoint-wrapper.sh, který nastavuje proměnné prostředí (angl. *environment variables*).

Všechny příkazy nástroje tft-admin se tedy budou spouštět skrz *docker*, např. docker exec $container_id /entrypoint-wrapper.sh tft-admin cloud set.

Následující Výpis 5.9 zobrazuje část skriptu pracující s nástrojem tft-admin obsaženého v souboru Jenkinsfile:

```
tft-admin cloud set bci-beaker
tft-admin cloud compose sync --compose "$compose_name"

tft-admin cloud set bci-psi-x86_64
tft-admin cloud compose sync --compose "$compose_name"

tft-admin cloud set arr-aws
tft-admin cloud compose sync \
    --compose "$compose_name" \
    --from-cloud bci-psi-x86_64

for section in ${sections//,/ }
do
    tft-admin --format json config compose-update "$section" \
        --compose "$compose_name" \
        --buildroot "$buildroot" \
        --aws-image "$aws_image" \
        --beaker-distro "$beaker_distro" \
```

[2] https://www.docker.com/

```
        --openstack-image "$openstack_image" \
        --force-with-dirty-tree \
        | tee compose-update.json
    test_options=$(cat compose-update.json | \
        jq ".[0].test_options + $test_options")
done

tft-admin --format json config compose-propose | tee compose-propose.json

branch="$(cat compose-propose.json | jq -r .[0].branch)"

tft-admin config test \
    --force-run \
    --test-options "$test_options" \
    --branch "$branch"
```

**Výpis 5.9:** *Výňatek z shell skriptu Jenkins úlohy zobrazující využití implementovaných příkazů nástroje `tft-admin`*

# Kapitola 6

# Vyhodnocení přínosnosti automatizace

Tato kapitola shrnuje přínosy implementace automatizace úlohy *„Aktualizace a otestování systému průběžné integrace s novým sestavením operačního systému“* a objektivně posuzuje ušetřený čas oproti výchozímu stavu.

Vytvořená Jenkins úloha `compose-update` je schopná aktualizovat a otestovat systém průběžné integrace jedním novým compose. V současné době vznikají přibližně 2–3 nové compose týdně, kterými je potřeba aktualizovat systém průběžné integrace. Ve výchozím stavu zabrala aktualizace člověku přibližně 30 minut čistého času. Tím se rozumí, že pozornost člověka při aktualizaci je vyžadována po dobu přibližně 30 minut. Do této doby se nezapočítává čas, kdy se vykonávají činnosti bez zásahu člověka.

Nejdelšími úseky, kdy člověk musel čekat na dokončení běhu operace jsou *„6. Nahrát compose do AWS“* a *„10. Spustit testy“*, jak je vyobrazuje diagram v Obrázku 4.1. Nahrání nového image do AWS trvá přibližně 20–30 minut a vykonání integračních testů na Jenkins serveru 30–45 minut. Zahrnutím těchto procesů do jedné úlohy odpadla člověku povinnost sledovat průběh těchto několika procesů. Celkový čas jedné aktualizace je tedy přibližně 90 minut, z toho po dobu 30 minut je vyžadována pozornost člověka.

Dalším přínosem je také redukce množství operací, které člověk vykoná. Jak již bylo zmíněno v diagramu v Obrázku 4.1, úloha se původně skládala z osmi kroků, které musel vykonat *Člen TFT*. Automatizací došlo k redukování počtu kroků na dva – *Člen TFT* musí na začátku pouze zahájit aktualizaci a na konci ji potvrdit.

Nelze opomenout také fakt, že Jenkins úloha `compose-update` obsahuje potenciál do budoucna. V současné době vznikají 2–3 nové compose denně, ovšem aktualizovat systém průběžné integrace je možné přibližně s 2–3 compose týdně. Toto omezení je zapříčiněno dosud neautomatizovaným nahráváním compose do služby OpenStack, jak jej zobrazuje například Obrázek 4.2 v kroku *„3. Nahrát compose do OpenStacku“*. Pokud by došlo k automatizaci tohoto kroku, úlohu `compose-update` by bylo možné napojit na message bus a automatizovaně začít spouštět aktualizace systému průběžné integrace bez zásahu člověka.

Jelikož se v současné době aktualizuje systém průběžné integrace přibližně 2–3 krát týdně, hotové řešení šetří přibližně 1–1,5 hodiny lidské práce týdně.

# Kapitola 7

# Závěr

Tato práce si kladla za cíl zautomatizovat některé úlohy v rámci týmu Testing Farm firmy Red Hat Czech s.r.o. Ačkoliv se povedlo zautomatizovat pouze jednu úlohu, považuji tuto práci za úspěšnou.

Z identifikovaných úloh popsaných v kapitole pojednávající o týmu Testing Farm v kapitole 3.4 byla vybrána úloha *„Aktualizace a otestování systému průběžné integrace s novým sestavením operačního systému“* jako nejvýznamnější v rámci údržby tohoto systému průběžné integrace. Implementace tak šetří čas potřebný k údržbě tohoto systému.

V praktické části práce, která je navržena v kapitole 4 a implementována v kapitole 5 vznikla nová úloha na Jenkins serveru `compose-update`, která je zodpovědná za vykonání vybrané úlohy. Úloha se v současné době spouští ručně, ovšem do budoucna je připravena na situaci, kdy ji bude spouštět příchozí zpráva z message busu. Tohoto stavu bude dosaženo, až se docílí automatizace některých procesů souvisejících s vydáním nové verze operačního systému, které již nespadají do týmu Testing Farm.

Zmíněná úloha k vykonávání dílčích kroků vedoucích k automatizaci tohoto procesu využívá nástroj `tft-admin`, do kterého bylo naimplementováno 5 nových příkazů, které dohromady umožňují vykonat vybranou úlohu. Dále byl nástroj vhodně upraven tak, aby umožňoval použití ve skriptech, což do budoucna usnadňuje automatizaci dalších úloh pomocí tohoto nástroje.

# Literatura


[1] *An Advanced Message Queuing Protocol (AMQP) Walkthrough* [online]. [cit. 2020-24-3]. Dostupné z: https://www.digitalocean.com/community/tutorials/an-advanced-message-queuing-protocol-amqp-walkthrough.

[2] *Click Documentation* [online]. [cit. 2020-06-14]. Dostupné z: https://click.palletsprojects.com/en/7.x/.

[3] *CloudAMQP Documentation* [online]. [cit. 2020-23-3]. Dostupné z: https://www.cloudamqp.com/docs/amqp.html.

[4] *Fedora Project Wiki – Bodhi* [online]. [cit. 2020-23-3]. Dostupné z: https://fedoraproject.org/wiki/Bodhi.

[5] *Jenkins User Documentation* [online]. [cit. 2020-06-22]. Dostupné z: https://jenkins.io/doc/.

[6] *Repozitář projektu DistGit* [online]. [cit. 2020-20-3]. Dostupné z: https://github.com/release-engineering/dist-git.

[7] *RPM Documentation* [online]. [cit. 2020-23-3]. Dostupné z: http://rpm.org/documentation.html.

[8] *Koji – Fedora Project Wiki* [online]. 2018 [cit. 2020-21-3]. Dostupné z: https://fedoraproject.org/wiki/Koji.

[9] Ben Kiki, O., Evans, C. a Net, I. döt. *Specifikace jazyka YAML 1.2* [online]. [cit. 2020-06-14]. Dostupné z: https://yaml.org/spec/1.2/spec.html.

[10] Duvall, P., Matyas, S. a Glower, A. *Continuous integration: Improving software quality and reducing risk.* Addison-Wesley, 2013. ISBN 978-0-321-33638-5.

[11] Fedor, J. *Automatizovaná distribuce vydání software pro partnery.* 2018. Diplomová práce. Masarykova univerzita, Fakulta informatiky, Brno. Dostupné z: https://is.muni.cz/th/z982v/.

[12] Foundation, T. P. S. *The Python Tutorial* [online]. [cit. 2020-06-14]. Dostupné z: https://docs.python.org/3.7/tutorial/index.html.

[13] Hohpe, G. a Woolf, B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley, 2003. ISBN 978-0321200686.

[14] Klusoň, M. *Podpora průběžné integrace v rámci systému Copr.* 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.vutbr.cz/studenti/zav-prace/detail/114915.

[15] McLean, M. *Koji documentation* [online]. 2017 [cit. 2020-21-3]. Dostupné z: https://pagure.io/docs/koji/.

[16] Olsen, K. *ISTQB Foundation Level Syllabus* [online]. 2019 [cit. 2020-21-3]. Dostupné z: https://www.istqb.org/downloads/send/2-foundation-level-documents/281-istqb-ctfl-syllabus-2018-v3-1.html.

# Příloha A

# Obsah paměťového média

- `./src/cli` – Zdrojové soubory nástroje `tft-admin`.
- `./src/jenkinsfile/Jenkinsfile.compose-update` – Zdrojový soubor Jenkins úlohy `compose-update`.
- `./src/jjb/compose-update.yaml` – Zdrojový soubor Jenkins úlohy `compose-update`.
- `./thesis.zip` – Archiv se zdrojovými soubory bakalářské práce v LaTeXu.
- `./xhavli47-BP.pdf` – Práce ve formátu *pdf*.
- `./README.txt` – Textový soubor dokumentující autorství zdrojových kódů a instalaci nástroje `tft-admin` a úlohy `compose-update`.