VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

PODPORA PRŮBĚŽNÉ INTEGRACE V RÁMCI BUILD SYSTÉMU COPR

CONTINUOUS INTEGRATION SUPPORT FOR COPR BUILD SYSTEM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. MARTIN KLUSOŇ**
AUTHOR

VEDOUCÍ PRÁCE **doc. Mgr. ADAM ROGALEWICZ, Ph.D.**
SUPERVISOR

BRNO 2018

Zadání diplomové práce/21261/2017/xkluso01

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Klusoň Martin, Bc.**
Obor: Počítačová grafika a multimédia
Téma: **Podpora průběžné integrace v rámci systému Copr**
Continues Integration Support for Copr Build System
Kategorie: Analýza a testování softwaru

Pokyny:

1. Prostudujte problematiku průběžné integrace (Continues Integration, CI).
2. Prostudujte systémy pro vytváření instalačních balíků v Linuxu se zaměřením na systém Copr.
3. Prostudujte architekturu frameworků citool a beakerlib užívaných pro CI.
4. Navrhněte propojení systému Copr s frameworkem citool pro zajištění CI.
5. Navržené řešení implementujte.
6. Implementované řešení otestujte a připravte pro zařazení do up-streamu projektu Copr.

Literatura:

- ISTQB Foundation Level Syllabus: http://www.istqb.org/downloads/send/2-foundation-level-documents/3-foundation-level-syllabus-2011.html
- Dokumentace k projektu Copr: http://copr-backend.readthedocs.io
- Dokumentace k projektu Koji: https://pagure.io/docs/koji/
- Dokumentace k projektu Open Build Service: http://openbuildservice.org

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 4.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Rogalewicz Adam, doc. Mgr., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

L.S.

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Práce se zabývá implementací průběžné integrace pro build systém Copr. Implementace využívá framework Citool a jeho moduly, které se již používají pro průběžnou integraci build systému Koji.

Výsledný systém umožňuje automaticky spustit testování pro nový balíček v build systému Copr a otestovat ho v prostředí virtuálního stroje. Práce ukazuje způsob, jakým je možné realizovat průběžnou integraci pro build systém Copr.

Abstract

This thesis deals with implementation of continuous integration for build system Copr. The implementation uses framework Citool and its modules, which are already used for continuous integration of build system Koji.

The outcome system can run the tests for the new package from the build system Copr and test it on virtual machine. This thesis shows way how to implement continuous integration for build system Copr.

Klíčová slova

Testování softwaru, Průběžná integrace, Build systém, Copr, Koji, Citool, Gluetool, Jenkins, Jenkins job builder, Openstack, Beakerlib, Ansible

Keywords

Software testing, Continuous integration, Build system, Copr, Koji, Citool, Gluetool, Jenkins, Jenkins job builder, Openstack, Beakerlib, Ansible

Citace

KLUSOŇ, Martin. *Podpora průběžné integrace v rámci build systému Copr*. Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Mgr. Adam Rogalewicz, Ph.D.

Podpora průběžné integrace v rámci build systému Copr

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana doc. Mgr. Adama Rogalewicze, Ph.D. Další informace mi poskytl technický konzultant pan Mgr. Miroslav Vadkerti. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Klusoň
22. května 2018

Poděkování

Na tomto místě bych rád poděkoval společnosti Red Hat Czech s.r.o., technickému konzultantovi panu Mgr. Miroslavu Vadkertimu a vedoucímu práce panu doc. Mgr. Adamu Rogalewiczovi, Ph.D., za jejich přispění ke vzniku této práce.

Obsah

1 Úvod **3**

2 Popis současného stavu **4**
2.1 Přehled build systémů 4
2.1.1 Open Build Service 4
2.1.2 Build systém Koji 5
2.1.3 Build systém Copr 6
2.2 Úvod do testování softwaru 10
2.2.1 Úrovně testování 11
2.2.2 Typy testování 12
2.2.3 Statické testování 13
2.2.4 Vztah testování a build systému 13
2.2.5 Knihovna Beakerlib 14
2.3 Úvod do průběžné integrace 18
2.4 Nástroje pro průběžnou integraci 21
2.4.1 Automatizační server Jenkins 21
2.4.2 Frameworky Citool a Gluetool 24
2.4.3 Další používané technologie 27
2.5 Průběžná integrace build systému Brew 30
2.5.1 Architektura průběžné integrace build systému Brew 30
2.5.2 Implementace Jenkins úloh 33

3 Návrh průběžné integrace build systému Copr **37**
3.1 Schéma Jenkins úloh 37
3.2 Analýza dostupných modulů 38
3.3 Modul Copr 40
3.4 Kontext artefaktu 40
3.5 Úloha Copr dispatcher 41
3.6 Úloha Test in Openstack (Copr) 42
3.7 Export výsledků 43

4 Implementace průběžné integrace build systému Copr **45**
4.1 Implementace modulu Copr 45
4.2 Poskytování kontextu 48
4.3 Instalace testovaného balíčku 49
4.3.1 Modul install-copr-build 50
4.4 Implementace Jenkins úloh 51
4.5 Implementace předávání výsledků testování 51

5 Závěr **52**

Literatura **53**

Kapitola 1

Úvod

Diplomová práce se zabývá problematikou testování softwaru, konkrétně přístupem průběžné integrace. Cílem je implementovat průběžnou integraci pro build systém Copr. Zadání této práce vzniklo ve spolupráci se společností Red Hat Czech s.r.o., kde se již průběžná integrace používá pro testování balíčků z build systémů Brew. Úkolem je tedy upravit stávají řešení tak, aby umožnilo testování balíčků z obou systémů. Hotové řešení bude nasazeno do interního firemního provozu a zároveň poskytnuto do upstream projektu Copr.

Kapitola 2 nejprve představí všechny nástroje a technologie potřebné k pochopení fungování průběžné integrace a na závěr popíše i současně používanou implementaci. Dozvíme se zde více o testování softwaru, průběžné integraci, build sytému Copr, automatizačním serveru Jenkins a také o frameworcích Citool a Gluetool. V druhé polovině, v kapitolách 3 a 4, se práce věnuje návrhu průběžné integrace pro build systém Copr a implementaci tohoto návrhu.

Diplomová práce navazuje na stejnojmenný semestrální projekt a přebírá z něj obsah kapitoly 2, konkrétně podkapitoly: *Přehled build systémů* (2.1), *Úvod do testování softwaru* (2.2), *Úvod do průběžné integrace* (2.3) a *Nástroje pro průběžnou integraci* (2.4).

Kapitola 2

Popis současného stavu

Tato kapitola shrnuje všechny poznatky, které jsou nezbytné k pochopení výchozího stavu diplomové práce.

2.1 Přehled build systémů

K distribuci softwaru se v linuxových distribucích obvykle používají tzv. *balíčkovací systémy*. Balíčky zjednodušují a zrychlují proces instalace, protože software v nich je již přeložen do binární podoby, takže odpadá nutnost jeho kompilace. Zmenšují velikost, protože neobsahují zdrojové kódy. Pomocí podepisování balíčků ověřují původ softwaru. V neposlední řadě poskytují pohodlný způsob jak instalovat (včetně závislostí), aktualizovat a odebírat software v rámci operačního systému.

Nejznámější dva reprezentanti balíčkovacích systémů jsou *DEB* a *RPM*. DEB se používá v distribucích Debian, Ubuntu a dalších z nich odvozených. RPM najdeme ve Fedoře, SUSE a jejich odnožích. Vzhledem k zaměření práce se dále budeme zabývat pouze balíčkovacím systémem RPM.

Rozlišujeme dva typy balíčků tzv. *zdrojový balíček* a *předkompilovaný balíček*. Zdrojový balíček obsahuje zdrojové kódy programu a tzv. *specfile*, jenž obsahuje metadata o programu (verze, licence, popis apod.). I když je možné ho přímo nainstalovat, většinou se používá k vytvoření předkompilovaného balíčku. [14]

Předkompilovaný balíček je již připraven k instalaci. K jeho vytvoření pomáhá tvůrcům tzv. *build systém*. Build systém vytváří předkompilované balíčky ze zdrojových kódů, nebo zdrojových balíčků. Výsledek tohoto vytváření se nazývá *build*.

Dále se budeme věnovat build systémům pro balíčkovací systém RPM.

2.1.1 Open Build Service

Jako první si představíme *Open Build Service*[1] (OBS). Zmínka o něm se zde nachází především kvůli možnosti jeho porovnání se zbylými build systémy, dále se s ním již nesetkáme. Podkapitola čerpá informace z *Open Build Service Beginner's Guide* [4].

OBS vznikl v roce 2005, umí vytvářet balíčky pro celou řadu operačních systémů (SUSE, Debian, Ubuntu, Red Hat Enterprise Linux, Windows) a hardwarových architektur (x86, AMD64, z Systems, POWER). OBS je používán v linuxové distribuci SUSE.

K vytvoření balíčku v OBS je zapotřebí *build recipe*, který obsahuje:

[1] http://openbuildservice.org

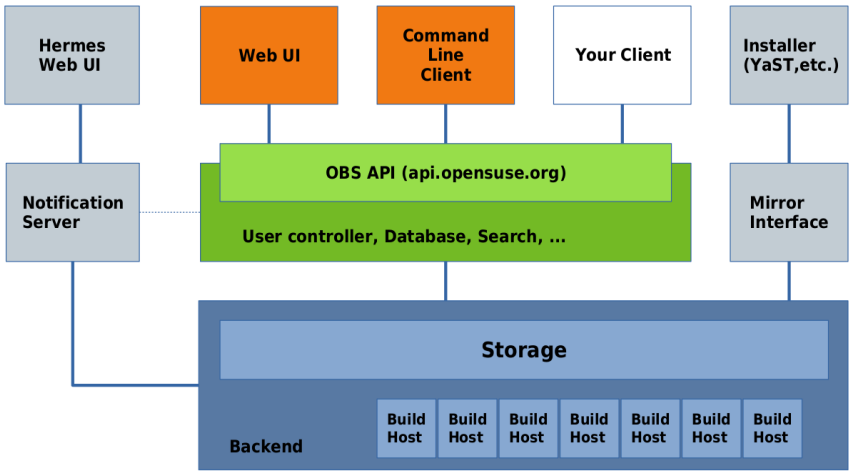
- *Metadata* – musí obsahovat název a popis balíčku. Další informace o balíčku jako verze, licence a url projektu jsou volitelná.
- *Requirements* – jsou závislosti potřebné ke správné funkci balíčku. Dělí se na dvě skupiny. Závislosti vyžadované během překladu balíčku (tzv. *build requirements*) a závislosti nutné k instalaci výsledného balíčku (tzv. *installation requirements*).
- *Package List* – je seznam souborů patřících balíčku. Používá je správce balíčků při instalaci a odinstalaci balíčku.

Systémy používající balíčkovací systém RPM mají tyto údaje uvedeny ve svém specfile.

Architektura OBS je znázorněna na obrázku 2.1. Centrální OBS server (na obrázku zeleně) poskytuje webové uživatelské rozhraní (na obrázku oranžově) a rozhraní příkazové řádky (též oranžově). Build probíhá na backend systému (na obrázku modře). Pro každý balíček se vytvoří izolovaný tzv. *sandbox* s příslušnou distribucí a architekturou. V závislosti na build recipe jsou nainstalovány další vyžadované balíčky. Následně jsou provedeny všechny kroky překladu a pokud byl překlad úspěšný, nainstalují se do sandboxu všechny soubory patřící balíčku. Z těchto souborů se pak vytvoří RPM balíček a uloží se.

OBS poskytuje rozhraní příkazové řádky pomocí nástroje `osc`.

Balíčky jsou v OBS organizovány v *projektech*. Jeden projekt musí obsahovat jeden či více balíčků a může obsahovat podprojekty. Projekty a podprojekty se používají k vytváření struktur balíčků. Projekt definuje přístupová práva, související repozitáře a cíl buildu (operační systém a architekturu).



Obrázek 2.1: Architektura Open Build Service (zeleně – OBS server, oranžově – webové uživatelské rozhraní a rozhraní příkazové řádky, modře – backend s build uzly)

2.1.2 Build systém Koji

Další build systém, který si představíme, je *Koji*. Informace v této podkapitole pochází z dokumentace projektu [15].

Koji[2] je používán v linuxové distribuci Fedora. V textu se budeme setkávat také s názvem Brew. Brew je instance build systému Koji používaná k vytváření balíčků v linuxové distribuci *Red Hat Enterprise Linux* (RHEL).

Protože pojem *balíček* je poměrně přetížený a někdy nemusí být zcela zřejmé, o co konkrétně se jedná, zavádí Koji následující pojmenování:

- *Balíček (Package)* – je jméno zdrojového rpm balíčku (soubor s příponou .src.rpm). Pojmenovává balíček obecně. Neobsahuje verzi a číslo vydání. Například *glibc*.
- *Build* – označuje konkrétní build balíčku, zahrnuje všechny architektury a podbalíčky. Například *glibc-2.3.4-2.19*.
- *RPM* – specifikuje jeden konkrétní rpm soubor s konkrétní architekturou a buildem. Například *glibc-2.3.4-2.19.i686*.

Koji používá Mock[3] k vytvoření prostředí pro build. Architektura Koji zahrnuje centrální server, uživatelské rozhraní a build uzly. Konkrétně se Koji skládá z následujích komponent:

- *Koji-hub* – je středem všech operací v Koji. Je to XML-RPC[4] server, který pouze přijímá XML-RPC volání a čeká dokud některý z build uzlů, nebo jiná komponenta nezahájí komunikaci. Je to jediná komponenta s přímým přístupem do databáze a jedna ze dvou komponent, která má práva přístupu na souborový systém.
- *Kojid* – je program, který ovladá build uzly. Tyto stroje mohou být přidávány dynamicky, a tím lze škálovat výkon build systému.
- *Koji-web* – je webové rozhraní. Poskytuje náhled na data a možnost zadávání příkazů (např. zrušení buildu).
- *Koji* – je klientská aplikace implementována v jazyce Python. Umožňuje uživateli zadávat dotazy na build systém a spouštět překlady.
- *Kojirepod* – je démon, který udržuje aktuální repozitář pro build.

Build systém Koji používá k organizaci balíčků *tagy*. K balíčku jsou přiřazeny tagy a k tagu se vážou například vlastnictví balíčku. Každý tag má seznam validních balíčků.

Koji rozeznává dva typy buildů – *produkční* a *scratch* build. Produkční je standardní build, který může být vydán a distribuován. Scratch build vydán být nemůže a slouží pouze pro testování.

2.1.3 Build systém Copr

Posledním představeným, v kontextu této práce nejdůležitějším, build systémem je *Copr*[5]. Informačním zdrojem této podkapitoly je uživatelská a vývojářská dokumentace projektu [6] [2].

Copr je nejmladším přírůstkem do rodiny build systémů operačního systému Fedora. Původním záměrem jeho tvůrců nebylo vytvořit další build systém, chtěli pouze zjednodušit přidávání repozitářů s balíčky třetích stran. Bylo zvažováno i použití Open build servise

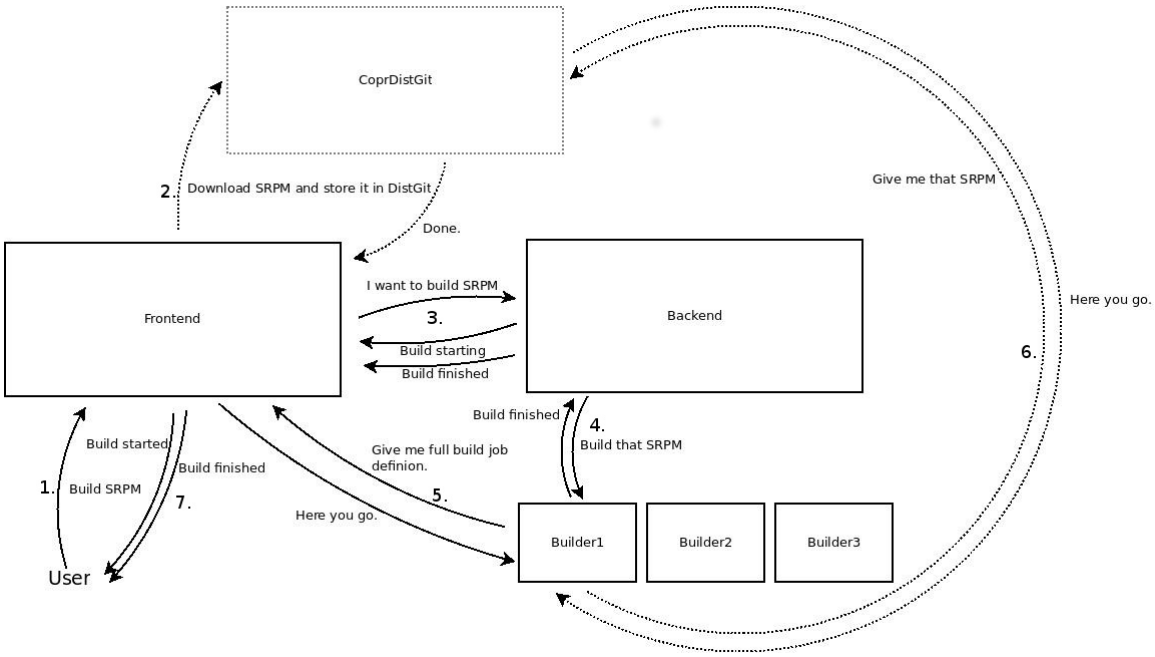
[2] https://pagure.io/docs/koji
[3] https://github.com/rpm-software-management/mock/wiki
[4] http://www.xmlrpc.com
[5] https://pagure.io/copr/copr

(viz podkapitola 2.1.1), tímto problémem se zabývá například diplomová práce Bc. Josefa Stříbrného – *Open Build Service migration to Fedora* [19].

Po konzultacích s vývojáři Koji (podkapitola 2.1.2) byla však shledána potřeba vytvoření nového, lehkého, jednoduše udržovatelného build systému. Název *Copr* vznikl jako zkratka anglického *Cool Other Package Repositories*, což s nadsázkou popisuje jeho účel.

Copr umožňuje vývojářům jednoduše vytvářet nové balíčky a poskytuje repozitáře k jejich distribuci. Je vhodný k buildu upstream projektů, testování balíčků, průběžné integraci a k distribuci softwaru třetích stran, protože balíčky v jeho repozitářích neprochází review, jako je tomu v těch standardních.

Build systém Copr se skládá z frontendu, backendu, build uzlů a úložiště CoprDistGit. Obrázek 2.2 zobrazuje jejich úlohu v procesu vytváření balíčku. O nezbytnosti úložiště se v komunitě vedou spory, a proto je na obrázku znázorněno tečkovanými čarami. Dále je popsán samotný proces vytvoření rpm balíčku.



Obrázek 2.2: Proces vytváření balíčku v build sytému Copr

1. Uživatel požádá frontend o build balíčku a poskytne zdrojové rpm. Uživatel je informován o zahájení buildu.
2. Frontend uloží zdrojové rpm na úložiště.
3. Frontend pověří backend vytvořením balíčku.
4. Backend vybere jeden z build uzlů a pověří ho vytvořením balíčku.
5. Build uzel si vyžádá kompletní popis buildu od frontendu.
6. Build uzel si stáhne zdrojové rpm z úložiště.
7. Po dokončení buildu, build uzel předá tuto informaci přes backend a frontend uživateli.

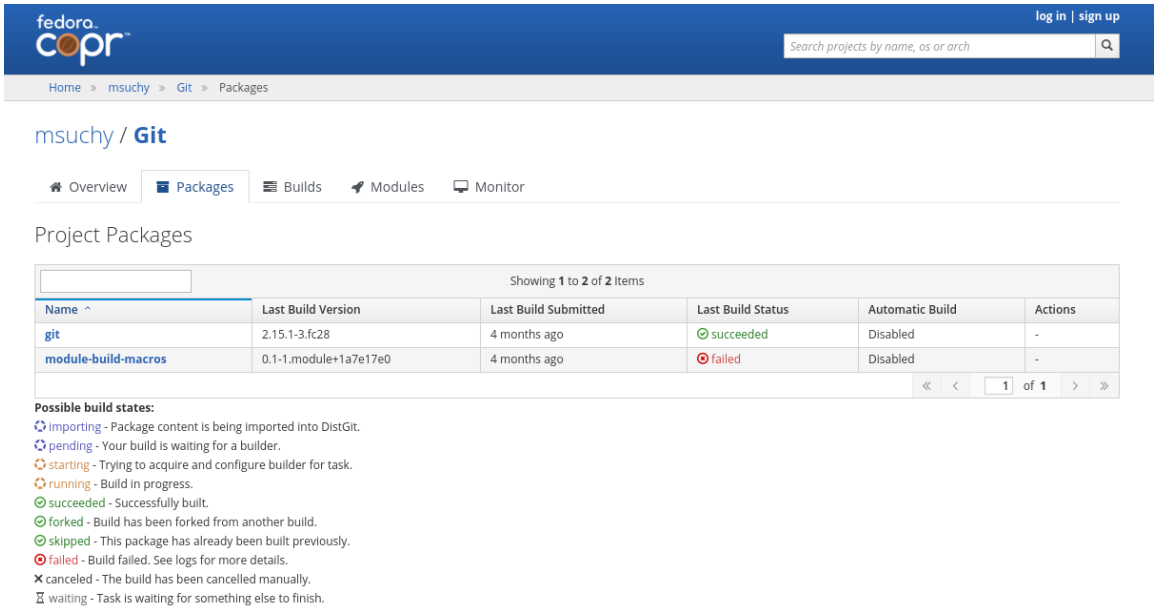
Copr po dokončení buildu o této události informuje na IRC kanál a na tzv. *message bus* (komunikační kanál založený na protokolu AMQP, který je popsán v části 2.4.3).

Program `copr-cli` umožňuje Copr plnohodnotně ovládat z příkazové řádky.

Aby mohl Copr vytvořit balíček, musí mu být poskytnuto zdrojové rpm, případně zdrojový kód doplněný o specfile. Toto poskytnutí může probíhat následujícími způsoby:

- *Direct Upload* – zdrojové rpm, nebo specfile je nahrán přímo přes webové rozhraní nebo `copr-cli`.
- *URL* – zdrojové rpm, nebo specfile je nejprve uloženo na server s veřejnou url, která je následně poskytnuta Copru. Tímto způsobem lze zadat více buildů najednou.
- *SCM* – Copru je poskytnuta adresa repozitáře (Git, DistGit, SVN). Repozitář musí mít veřejnou url, umožňovat klonování a obsahovat specfile.
- *PyPI* – provede build Python projektu přímo z `https://pypi.python.org/pypi`.
- *RubyGems* – je stejný jako předchozí bod, ale zdroje se stahují z `https://rubygems.org`.

Varianta *SCM* nabízí další funkcionalitu – tzv. *webhooks*. V okamžiku provedení nové změny v repozitáři se automaticky spustí nový build zahrnující tuto změnu. Tato funkce je krok směrem k průběžné integraci popisované v části 2.3.



Obrázek 2.3: Ukázka uživatelského rozhraní build systému Copr

K organizaci balíčků slouží *projekty*. Projekt obsahuje jeden nebo více balíčků, poskytuje repozitář k jejich instalaci, má svého majitele a definuje přístupová práva. Jednoznačnou identifikací projektu je kombinace jeho názvu a jména majitele. Každý uživatel si může vytvořit neomezený počet projektů.

Skupina rpm souborů se stejnou verzí vytvořená pro různá prostředí (Fedora 27, Fedora 28 apod.) z jednoho zdrojového rpm souboru se označuje *build*. Každý build má svoje

jedinečné id. Cílová prostředí jsou označena jako *chroots*. Část buildu odpovědná za vytvoření rpm pro konkrétní chroot se nazývá *build task* a k jeho jedinečné identifikaci slouží kombinace chroot a id buildu.

Na obrázku 2.3 vidíme ukázku uživatelského rozhraní Copru. Je zde zobrazen detail projektu uživatele „msuchy“ s názvem „Git“, který obsahuje dva balíčky – „git“ a „module-build-macros“.

Veřejné informace poskytuje Copr ve formátu json přes aplikační programové rozhraní. Je umístěno na adrese `[server_url]/api_2`, kde `[server_url]` zastupuje adresu domovské stránky webového rozhraní. Nacházejí se zde informace o projektech, buildech a build tascích.

2.2 Úvod do testování softwaru

Tato podkapitola čerpá z publikace *ISTQB Foundation Level Syllabus* [13].

Proč se zabýváme testováním? Protože lidé, v kontextu vývoje softwaru vývojáři, dělají chyby. Na vině může být nedostatek času, komplexnost kódu, nebo nová technologie. Ve chvíli, kdy se taková chyba projeví, může způsobit neočekávané chování, případně selhání programu. Pro úplnost musíme dodat, že takové chování může být způsobeno i vnějšími vlivy, které působí na hardware. Například přítomnost radiace, elektrického nebo magnetického pole.

Obecně můžeme testování považovat za proces hledání chyb, proto aby mohly být opraveny. Za testování jsou však považovány i aktivity, které vlastnímu vykonávání testů předcházejí a následují ho. Mezi ně patří plánování testů, definování podmínek testů a návrh testů, následuje sběr výsledků a jejich vyhodnocování.

Jeden test se označuje *test case*, jeho vykonávání může být manuální nebo automatizované. Test case může být zařazen do jednoho či více tzv. *test plan*. Test plan je seznam několika test case.

Myšlení potřebné k testování se od myšlení používatého při vývoji softwaru liší. I když zkušený vývojář je schopen, díky znalosti svého kódu, testovat vlastní program, oddělení těchto dvou rolí, *vývojáře* a *testera*, přináší výhody. Například se eliminuje předpojatost autora. Zkušený tester je efektivnější v hledání chyb a problémů. Toto rozdělení lze klasifikovat do následujících stupňů:

1. Testy vytváří sám autor programu.
2. Testy vytváří jiný vývojář než autor programu.
3. Testy vytváří člověk z jiné organizační složky firmy, než je autor programu (typicky tester).
4. Testy vytváří člověk z jiné firmy (outsourcing).

Během více než čtyřicetileté historie testování softwaru byla definována celá řada jeho zásad. Následujících sedm nejdůležitějších nám může pomoci pochopit podstatu správného testování.

1. *Testing shows presence of defects* – testování může dokázat přítomnost chyb, ale nemůže dokázat jejich nepřítomnost. Testování snižuje pravděpodobnost přítomnosti chyb, ale i když žádná chyba není nalezena, není to důkaz správnosti programu.
2. *Exhausive testing is impossible* – ani pro jednoduché test case není možné vyzkoušet všechny kombinace vstupů. Proto se musí používat analýza rizik, která ukáže, na co se má testování zaměřit.
3. *Early testing* – testování má být zahájeno, jakmile je to možné.
4. *Defect clustering* – zpravidla malé množství modulů programu obsahuje většinu chyb, nebo způsobuje většinu selhání.
5. *Pesticide paradox* – pokud je test nebo množina testů používána stále dokola, přestane nalézat nové chyby. Proto musí být testy pravidelně kontrolovány, aktualizovány a vytvářeny nové.

6. *Testing is context dependent* – testování se provádí odlišně v různých situacích. Například testování kritického bezpečnostního softwaru bude probíhat jinak než testování komerční webové stránky.

7. *Absence-of-errors fallacy* – hledání o opravy chyb nepomohou, pokud je program nepoužitelný a neplní požadavky uživatele.

2.2.1 Úrovně testování

Postupně si projdeme několik různých rozdělení testování. První z nich dělí testování do úrovní, které reflektují proces vývoje softwaru.

Testování komponent

Testování komponent je také označováno jako testování modulů, programů, nebo jako *unit testování*. Hledá chyby a verifikuje funkcionalitu v nejmenších blocích (modulech, programech, objektech, třídách), jež jsou samostatně testovatelné bez zbytku systému. Vyžaduje vytvoření generátorů, které simulují vstupy.

Testování komponent zahrnuje funkcionální, nefunkcionální a strukturální testování. Tyto pojmy budou vysvětleny v části 2.2.2. Testy jsou odvozeny ze specifikace, návrhu nebo datového modelu. Nejčastěji jsou vytvářeny vývojářem programu. Objevené chyby jsou okamžitě opravovány a nejsou nijak formálně dokumentovány.

Speciálním druhem testování komponent je přístup, kdy se testy vytvoří ještě před zahájením vývoje kódu programu. Pak mluvíme o tzv. *test-driven development*.

Integrační testování

Integrační testování testuje rozhraní mezi komponentami, spolupráci s různými částmi systému a rozhraní mezi systémy. Nemělo by se zabývat testováním samotných komponent, to by již mělo být řešeno v rámci unit testování. Integrační testování může probíhat na dvou úrovních.

1. Integrační testování komponent, které předtím byly otestovány jednotlivě pomocí unit testů.

2. Integrační testování systému. Do této kategorie spadá i testování spolupráce mezi hardwarem a softwarem, nebo mezi systémy různých organizací.

Integrační testování by mělo být inkrementální a moduly přidávány postupně, pak se nalezená chyba snadno lokalizuje. Ideálně by měl systém přidávání kopírovat architekturu programu nebo tok dat.

Integrační testy jsou většinou funkcionální, ale v konkrétních případech mohou být i nefunkcionální (například výkonnostní testy).

Systémové testování

Systémové testování se zabývá chováním celého systému (někdy též produktu). Testování by mělo probíhat podle hlavního plánu (*Master test plan*) v prostředí nejlépe shodném s produkčním, aby se minimalizovalo riziko, že nebudou odhaleny chyby spojené s prostředím.

Testování by mělo být založeno na odhadu rizik, specifikaci požadavků, firemních procesech, případech použití, případně na jiných písemných požadavcích. Mělo by kontrolovat funkcionální a nefunkcionální požadavky na systém.

Systémové testování je obvykle prováděno nezávislým týmem testerů.

Akceptační testování

Akceptační testování je často odpovědností zákazníka nebo uživatele, případně jiné zainteresované strany. Cílem je získat důvěru ve funkčnost systému, část systému, nebo některou charakteristickou nefunkcionální vlastnost systému. Cílem není nálézt nové chyby. Akceptační testování může sloužit k vyhodnocení, zda je systém připraven k nasazení do produkce a používání. Nemusí to být poslední stupeň testování, v případě rozsáhlých systémů ho může následovat další kolo integračního testování.

2.2.2 Typy testování

Testování můžeme dále dělit podle toho, na co se zaměřuje.

Funkcionální testování

Funkcionální testování kontroluje *co* program dělá. Kontroluje funkce a vlastnosti programu (ty jsou buď zdokumentovány, nebo je tester musí sám pochopit). Sleduje vnější chování programu (používá black-box přístup) a může být použito ve všech úrovních testování.

Speciálním druhem funkcionálního testování je bezpečnostní testování (zaměřuje se na hledání hrozeb) a interoperabilní testování (testování schopnoti interakce s jinými programy).

Nefunkcionální testování

Nefunkcionální testování kontroluje *jak* dobře program pracuje. Patří sem testy výkonnosti, použitelnosti, přenositelnosti, spolehlivosti nebo testy zátěžové. Nefunkcionální testy zpravidla měří charakteristiky testovaného systému a kvantifikují získané výsledky. Tento typ hodnotí vnější chování programu (používá black-box přístup). Používá se ve všech úrovních testování.

Strukturální testování

Strukturální testování (neboli white-box přístup) využívá znalosti architektury programu. Zavádí se zde pojem *pokrytí*. Pokrytí je poměr mezi prvky, které jsou otestovány, a těmi které nejsou. Cílem je získat pokrytí sto procent. Typickými prvky, u nichž se měří pokrytí, jsou větve nebo příkazy programu. Používají se nástroje, které pokrytí jak měří, tak i zobrazují.

Strukturální testování se používá ve všech úrovních.

Opětovné a regresní testování

Poté, co je opravena chyba, by měl být program přetestován, aby se oprava potvrdila. *Regresní* testování je opětovné spuštění těchto potvrzovacích testů ve chvíli, kdy byla v programu provedena změna. Tím se povrdí, že se opravou neprojevila jiná již známá chyba.

Test case pro regresní testování by měl být ideálně automatizovaný, protože se bude vykonávat při každé změně programu.

Regresní testování se používá jak ve všech úrovních tak i ve všech předchozích typech testování.

Podmnožinou regresních testů jsou tzv. *smoke* a *sanity* testy. Jejich úkolem je rychle zkontrolovat zakladní funkcionalitu, ideálně po kazde změně programu.

Smoke test ověřuje hlavní funkcionalitu programu. Jejich název pochází z vývoje hardwaru. Když se po prvním zapnutí neobjeví kouř, znamená to, že součástka alespoň nějak funguje. Smoke test používá vývojář před předáním programu testerům. [9]

Sanity test, oproti smoke testu, testuje pouze jednu funkcionalitu a více do hloubky. Sanity testy používá tester. V případě nedostatku času mohou nahradit regresní testování. [8]

2.2.3 Statické testování

Statické testování je přístup, při němž, narozdíl od dynamického testování, nedochází ke spuštění kódu. Patří sem *review* a *statická analýza*.

Review je manuální kontrola čtenářem odlišným od autora. Používá se jak na dokumenty týkající se programu, tak i na samotný kód. V takovém případě hovoříme o tzv. *code review*. Review může proběhnout mnohem dříve než dynamické testování, protože nevyžaduje spustitelný kód. Opravit chyby nalezené během review je jednoduší a levnější, než kdyby byly nalezeny až během vykonávání kódu. Review může probíhat zcela manuálně, ale je vhodné používat některý z nástrojů pro podporu efektivity práce. Takový nástroj by měl umět zobrazovat dokument a umožnit k němu přidávat komentáře. Služby pro správu verzí (například *Github*[6] nebo *Gitlab*[7]) takové nástroje již mají integrované.

Statická analýza je automatizovaná kontrola kódu specializovanými nástroji. Ty jsou založeny na vyhledávání chyb a zranitelností (například *Coverity Scan*[8]), případně využívají umělou inteligenci (například *Diffblue*[9]). Mohou také kontrolovat dodržovaní stylu kódu (například *Coala*[10]).

2.2.4 Vztah testování a build systému

Schéma na obrázku 2.4 ukazuje vztah mezi build systémy, probíranými v části 2.1, a jednotlivými typy testování, jež byly představeny v této podkapitole.

Build systém vytváří balíčky tzv. *komponent*. Komponenta je buď část operačního systému, nebo program, který může být doinstalován. Vstupem build systému je zdrojový kód, jehož součástí jsou i unit testy pro jeho jednotlivé podčásti a specfile. Unit testy jsou na kód aplikovány v průběhu buildu v build systému. Výstupem úspěšného buildu je rpm balíček.

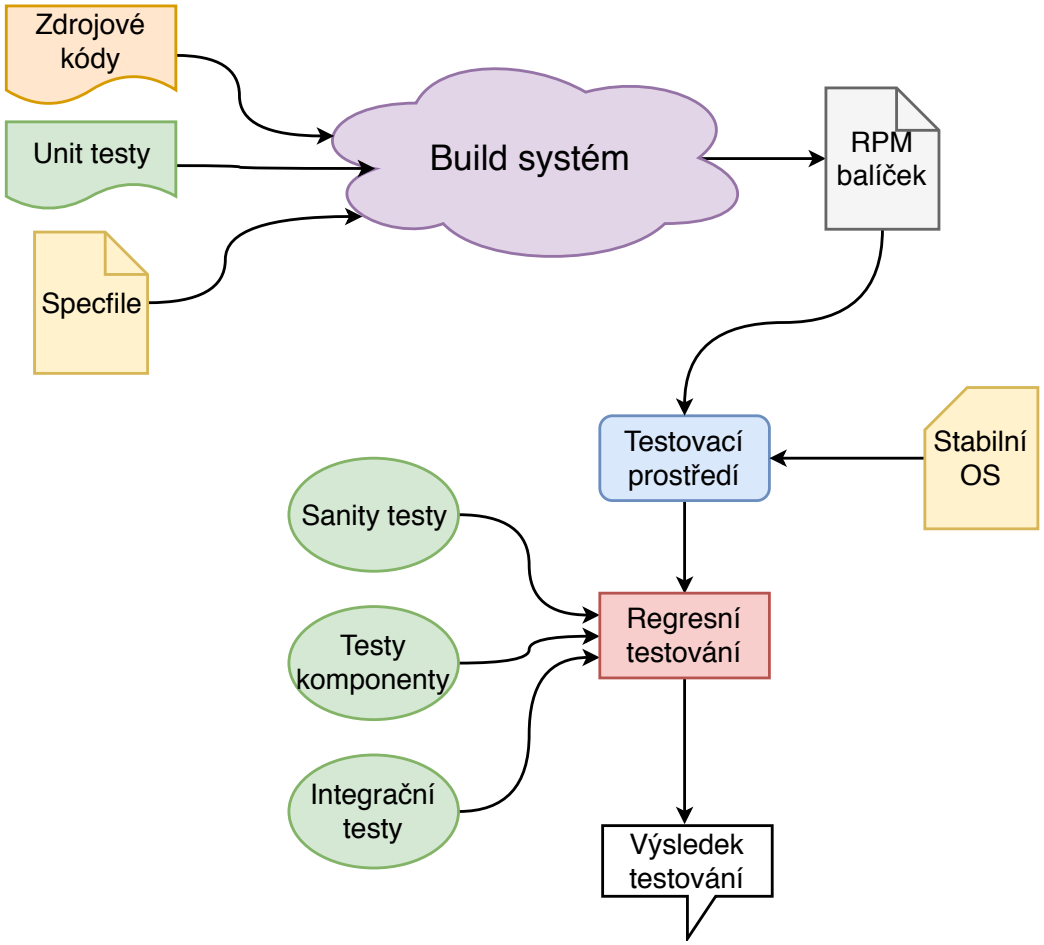
Tento balíček se nainstaluje do testovacího prostředí, které tvoří operační systém. Již v tomto kroku může být odhalen problém, a to v případě, že se nepodaří balíček do systému nainstalovat.

Následně se balíček zkontroluje testem, který verifikuje opravovanou chybu, společně se sadou regresních testů. Ty jsou tvořeny z sanity testů, testů komponenty a integračních testů. Testování je dynamické a používá jak funkcionálními tak nefunkcionálními testy.

Testování je považováno za úspěšné, pokud verifikační test potvrdí opravu chyby, a zároveň žádný z regresních testů neobjeví zanesení jiné chyby.

[6] https://github.com
[7] https://about.gitlab.com
[8] https://scan.coverity.com
[9] http://diffblue.com
[10] https://coala.io

Komponenty, které úspěšně prošly touto úrovní, pokračují do systémového testování, kde jsou do testovacího prostředí nainstalovány společně s dalšími (tuto úroveň už obrázek nezachycuje).



Obrázek 2.4: Schéma vztahu testování a build systému

2.2.5 Knihovna Beakerlib

Již několikrát bylo zmíněno, že test case může být automatizován. Následující část popisuje jeden z nástrojů pro automatizaci testování – knihovnu Breakerlib. Podkapitola přebírá informace z manuálové stránky projektu [16].

Beakerlib je knihovna, která do linuxového shellu integruje funkcionalitu psaní testů. Poskytuje funkce, které usnadňují psaní, běh a analýzu integračních a black-box testů. Beakerlib byl původně vyvíjen jako součást projektu *Beaker*.[11] Obsahuje následující základní funkce:

- *Journal* – poskytuje jednotný systém logování, ukládání výsledků ve formátu vhodném k porovnávání a generování hlášení.

[11] https://beaker-project.org

- *Phases* – poskytuje uspořádání testovacích kroků do logických celků.
- *Asserts* – poskytuje kontrolu průběhu testu (kontrola návratových kódů, zda soubor existuje, zda má správný obsah atd.).
- *Helpers* – poskytuje funkce ke správě služeb systému, zálohování a obnovu dat.

Beakerlib dělí testy do více částí tzv. *fází*. Fáze může být definována uživatelem. Nejčastěji se však používá struktura obsahující následující tři fáze, jež jsou v knihovně předdefinované:

- *Setup* – slouží k přípravě prostředí pro test, vytvoření dočasných souborů, zálohování, spouštění služeb.
- *Test* – slouží k provedení vlastního testu.
- *Cleanup* – slouží k obnovení systému do stavu před testem.

Beakerlib test je shell skript, který kromě běžných shell příkazů (například `ls`, `pwd`) používá také pomocné funkce knihovny Beakerlib. Těmi definuje začátky a konce fází a kontroluje průběh testu. Následuje popis několika nejpoužívanějších pomocných funkcí.

- `rlRun()` – spustí příkaz z prvního parametru a porovná jeho návratovou hodnotu s druhým parametrem.
- `rlAssertExists()` – zkontroluje, zda zadaný soubor existuje.
- `rlAssertDiffer()` – porovná obsah dvou souborů.
- `rlAssertRpm()` – zkontroluje, zda zadaný balíček je nainstalován.

Výpis 2.1 ukazuje jednoduchý test case implementovaný knihovou Beakerlib. Vidíme, že test je rozdělen do třech předdefinovaných fází. V části *setup* se vytvoří dočasný adresář k provádění testu. V části *test* je nejprve vytvořen soubor a zkontrolováno jeho vytvoření, následně je soubor smazán a opět zkontrolováno jeho odstranění. Závěrečná fáze *cleanup* opustí a smaže dočasný adresář.

Výpis 2.2 zachycuje část výstupu testu z výpisu 2.1. Části odpovídající jednotlivým fázím jsou odděleny návěštími s jejich jmény. Řádky obsahují tag a komentář. Tag má jednu z hodnot: `PASS`, `WARN`, `FAIL` nebo `LOG` a zobrazuje výsledek příkazu. Komentář pak slovně popisuje náplň příkazu. Funkce `rlRun()` nemá jasně definované chování, proto se jako komentář použije řetězec v jejím třetím parametru.

```
# Include the BeakerLib environment
. /usr/share/beakerlib/beakerlib.sh

# Set the full test name
TEST="/examples/beakerlib/Sanity/phases"

# Package being tested
PACKAGE="coreutils"

rlJournalStart
    # Setup phase: Prepare test directory
    rlPhaseStartSetup
        rlAssertRpm $PACKAGE
        rlRun 'TmpDir=$(mktemp -d)' 0 'Creating tmp directory' # no-reboot
        rlRun "pushd $TmpDir"
    rlPhaseEnd

    # Test phase: Testing touch, ls and rm commands
    rlPhaseStartTest
        rlRun "touch foo" 0 "Creating the foo test file"
        rlAssertExists "foo"
        rlRun "ls -l foo" 0 "Listing the foo test file"
        rlRun "rm foo" 0 "Removing the foo test file"
        rlAssertNotExists "foo"
        rlRun "ls -l foo" 2 "Listing foo should now report an error"
    rlPhaseEnd

    # Cleanup phase: Remove test directory
    rlPhaseStartCleanup
        rlRun "popd"
        rlRun "rm -r $TmpDir" 0 "Removing tmp directory"
    rlPhaseEnd
rlJournalEnd

# Print the test report
rlJournalPrintText
```

Výpis 2.1: Příklad automatizovaného test case, implementovaného pomocí knihovny Beakerlib

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: [ LOG ] :: Setup
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [ PASS ] :: Checking for the presence of coreutils rpm
:: [ PASS ] :: Creating tmp directory
:: [ PASS ] :: Running 'pushd /tmp/tmp.IcluQu5GVS' # no-reboot
:: [ LOG ] :: Duration: 0s
:: [ LOG ] :: Assertions: 3 good, 0 bad
:: [ PASS ] :: RESULT: Setup

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: [ LOG ] :: Test
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [ PASS ] :: Creating the foo test file
:: [ PASS ] :: File foo should exist
:: [ PASS ] :: Listing the foo test file
:: [ PASS ] :: Removing the foo test file
:: [ PASS ] :: File foo should not exist
:: [ PASS ] :: Listing foo should now report an error
:: [ LOG ] :: Duration: 1s
:: [ LOG ] :: Assertions: 6 good, 0 bad
:: [ PASS ] :: RESULT: Test

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: [ LOG ] :: Cleanup
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [ PASS ] :: Running 'popd'
:: [ PASS ] :: Removing tmp directory
:: [ LOG ] :: Duration: 1s
:: [ LOG ] :: Assertions: 2 good, 0 bad
:: [ PASS ] :: RESULT: Cleanup
```

Výpis 2.2: Příklad výstupu Beakerlib testu

Test Harness

Dalším nástrojem k automatizaci testování je tzv. *Test Harness*. Je to síťová aplikace, která zajišťuje vzdálené provádění automatizovaných test case. Na testovacích strojích běží klientské aplikaci, jež přijímají instrukce od serveru. Serverem je počítač, který testování spouští.

Příkladem takového softwaru je *Restraint*[12]. Příkaz `restraint` spustí seznam úloh (testů) popsaný v xml formátu na zadaných strojích a vrátí výsledky testování.

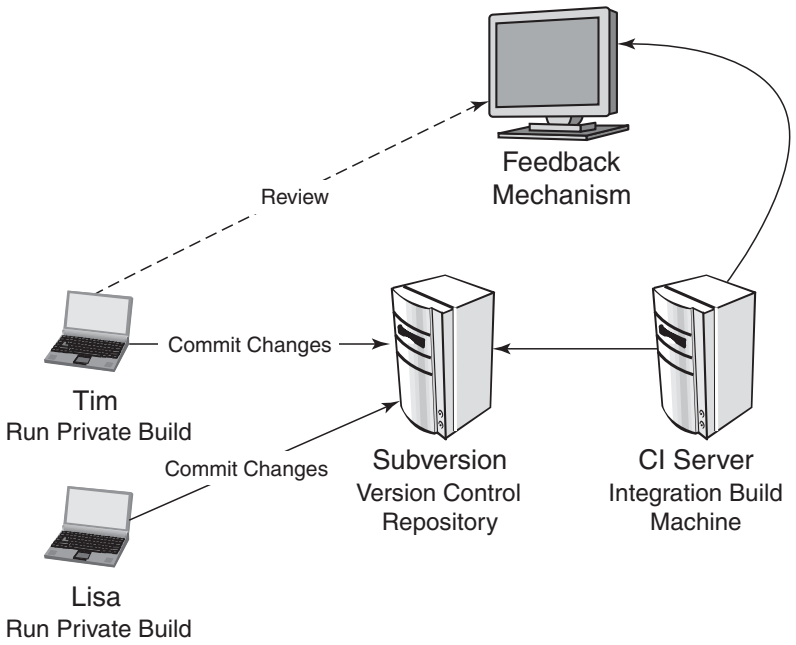
[12] https://restraint.readthedocs.io

2.3 Úvod do průběžné integrace

Průběžnou integraci najdeme v literatuře spíše pod anglickým názvem *continuous integration*, nebo pod zkratkou *CI*. Průběžnou integrací se zabývá například *Continuous integration: improving software quality and reducing risk* [11], tato podkapitola však vychází z článku Martina Fowlera – *Continuous integration* [12].

Definice říká: „Průběžná integrace je postup vývoje softwaru, kde členové vývojového týmu začleňují své změny často, obvykle každý alespoň jednou denně – to vede k vícenásobnému začleňování během dne. Každé začlenění je ověřeno automatizovaným překladem (zahrnujícím testy), aby se chyby spojené s integrací odhalily, jakmile to je možné. Hodně týmů shledalo, že tento přístup vede k výraznému snížení problémů spojených s integrací a umožňuje týmu vyvíjet soudržný kód rychleji."

Typický průběh implementace nové funkce programuza použití přístupu průběžné integrace vypadá následovně. Vývojář si nejprve vytvoří pracovní kopii nejnovějšího kódu z centrálního úložiště. Poté implementuje změny kódu, a také, protože průběžná integrace předpokládá velkou míru využití unit testů, upraví i testy.



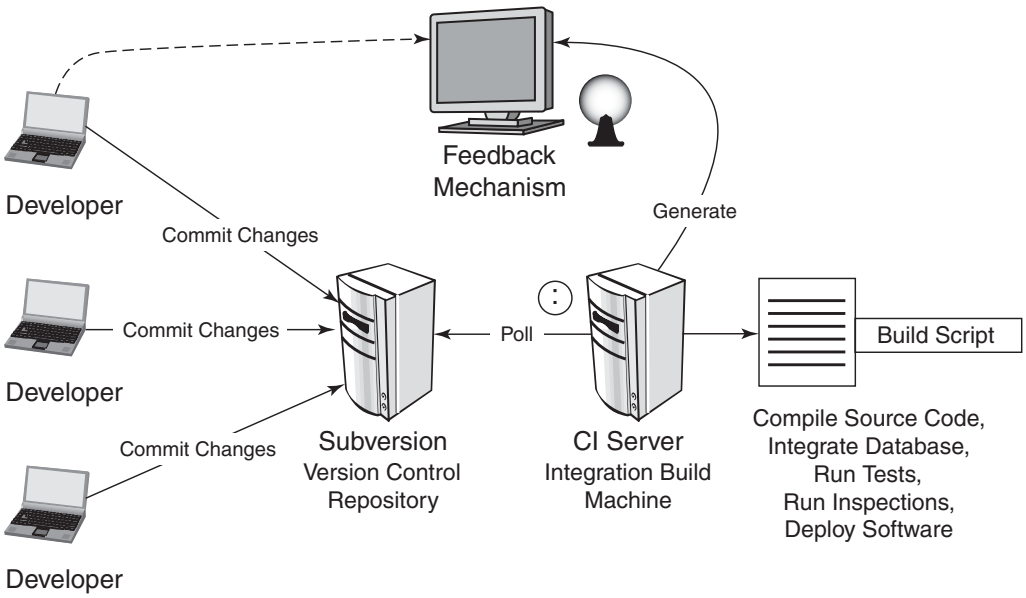
Obrázek 2.5: Schéma průběžné integrace – *Private build* spouští každý vývojář na svém počítači, *Integrační build* verifikuje nejnovější verzi kódu v centrálním úložišti (obrázek je převzat z [11])

Vývojář následně zkusí na svém počítači přeložit svůj pozměněný program, během kterého se zároveň spustí i unit testy. Tento proces se označuje *private build*. Jestliže private build proběhl úspěšně, mohou se změny začlenit do centrálního úložiště. Kód v centrálním úložišti však mohl být v průběhu práce s pracovní kopií změněn dalšími spolupracovníky, proto si vývojář musí aktualizovat svou pracovní kopii, vyřešit případné konflikty a spustit nový private build. Ve chvíli, kdy je i tento build úspěšný, může vývojář začlenit změny ze své pracovní kopie do centrálního úložiště.

Práce vývojáře ale nekončí odesláním změn. Ve chvíli začlenění změn se spustí další build – tzv. *integrační build*. Ten probíhá na serveru průběžné integrace a je založen na nejnovějším kódu z centrálního úložiště. Úspěch tohoto integračního buildu není zaručen. Například pokud vývojář nezahrnul do aktualizace centrálního úložiště všechny změny, dojde k selhání. Až ve chvíli kdy vývojář zkontroluje výsledek integračního buildu, je jeho práce na implementaci změny dokončena.

Pokud dva vývojáři vytvoří konfliktní změny, problémy by měly být odhaleny v okamžiku, kdy se druhý z nich pokusí začlenit své změny do centrálního úložiště. V případě, že problémy z nějakého důvodu neodhalí druhý vývojář, odhalí je integrační build. Ve správné realizaci průběžné integrace by integrační build neměl zůstat ve stavu selhání příliš dlouho.

Výše popsaný postup je zachycen na obrázku 2.5.



Obrázek 2.6: Základní prvky průběžné integrace (obrázek je převzat z [11])

Obrázek 2.6 ukazuje architekturu průběžné intergace a nezbytné prvky k jejímu fungování. Dříve než softwarový projekt začne používat průběžnou integraci, je nutné splnění několika požadavků.

Prvním předpokladem je, že projekt používá centrální úložiště souborů. To je typicky systém pro správu verzí programu (dnes je rošířený program `git`[13]). Takový systém uchovává historii změn a umožňuje vývoj dělit do více vývojových *větví*. Průběžné intergraci stačí jedna *hlavní větev*. V centrálním úložišti by mělo být přítomno vše nezbytné k provedení buildu, aby i nový uživatel byl schopen po stažení pracovní kopie provést build. Samotné výsledky buildů ale do centrálního úložiště nepatří.

Druhým předpokladem je automatizovaný překlad. I když překlad může být komplexní činnost, většinou se jedná o stále se opakující postup, který může a měl by být automatizován. Správný automatizační skript by měl jedním příkazem provést vše nezbytné a měl by si vystačit se soubory obsaženými v centrálním úložišti.

[13] https://git-scm.com

Dalším požadavkem je, aby testy, které testují nový build, byly automatizované. Tyto testy mohou být v případě integračního buildu doplněny o techniky statické analýzy, které byly popsány v části 2.2.3.

Poslední nezbytnou součástí je server průběžné integrace. Ten monitoruje úložiště a v případě změny zdrojového kódu inicializuje integrační build. Posléze poskytne vývojáři zpětnou vazbu v podobě výsledků buildu. Příkladem takového serveru je Jenkins (posaný v části 2.4). Integrační build by měl probíhat v prostředí, které je co nejvíce podobné produkčnímu.

Hlavní myšlenkou průběžné integrace je mít neustále (nebo jen s krátkými výpadky) funkční a otestovaný software vytvořený z poslední verze zdrojového kódu. Přirozeným důsledkem této snahy je tzv. *continuous delivery*, někdy též *continuous deployment* (zkráceně *CD*), volně přeloženo jako průběžné aktualizování. Je to nasazování nejnovější funkční verze softwaru do produkčního prostředí automatizovaně a s minimem potřebné manuální práce. K dobrému fungování CD je nezbytné, aby do testování byly zahrnuty i systémové a akceptační testy.

2.4 Nástroje pro průběžnou integraci

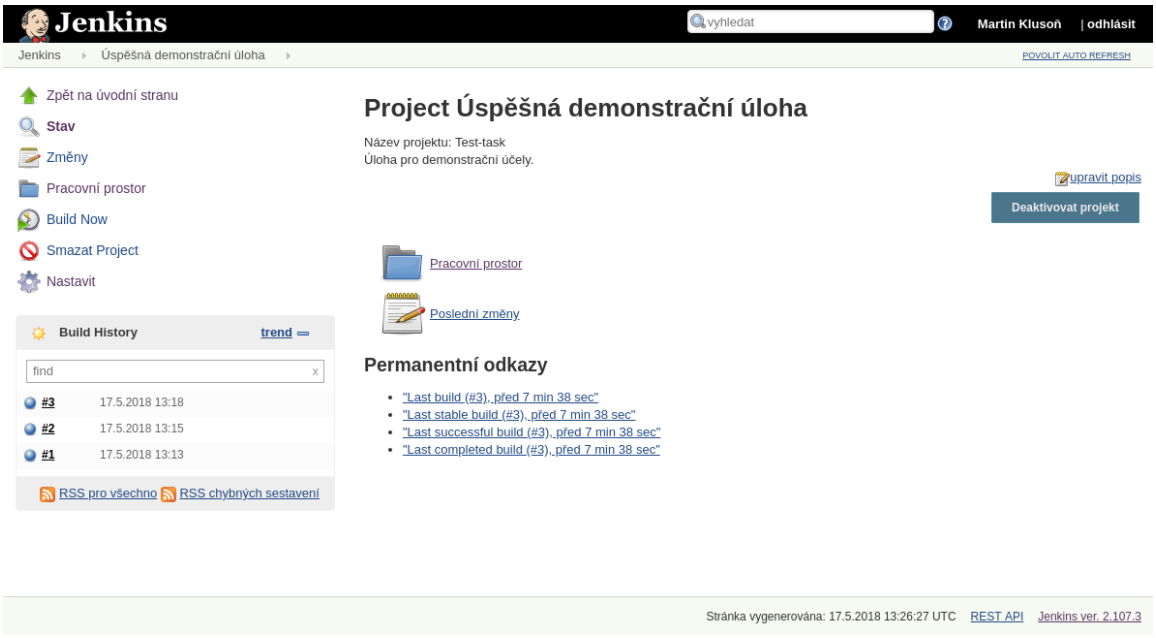
V tuto chvíli by bylo nejvhodnější přistoupit přímo k popisu průběžné integrace pro build systém Brew (v části 2.5.1), aby ho bylo možné porovnat s teoretickými znalostmi z části 2.3. Bohužel než tak učiníme, musíme se ještě seznámit s pojmy jako Jenkins, Gluetool, Citool, Openstack, Ansible a AMQP.

2.4.1 Automatizační server Jenkins

Podkapitola 2.3 věnovaná úvodu do průběžné integrace se několikrát zmiňuje o serveru průběžné integrace. Implementacemi takového serveru jsou například *Travis CI*[14], *TeamCity*[15] nebo *Jenkins*[16]. Posledně zmíněnému bude věnován následující text.

Kniha *Jenkins: The Definitive Guide* [18], ze které tato podkapitola čerpá informace, uvádí, že je Jenkins nejrozšířenějším CI řešením, což potvrzují i současná data společnosti Datanyze[17].

Historie projektu sahá do roku 2004, kdy zaměstnanec firmy Sun, Kohsuke Kawaguchi, začal ve svém volném čase vyvíjet open source CI řešení Hudson. Postupně ho začaly používat i další vývojové týmy firmy Sun. Proto se další vývoj Hudsonu stal v roce 2008 hlavní pracovní náplní jeho zakladatele. Jenkins se odštěpil od projektu Hudson v roce 2010 z důvodu neshod mezi jeho vývojáři a firmou Oracle, jež koupila firmu Sun.



Obrázek 2.7: Ukázka zobrazení Jenkins úlohy ve webovém uživatelském rozhraní

Úlohy Základním prvkem jenkins serveru jsou *úlohy* (anglicky *build jobs*). Úloha je předpis, který se po spuštění vykoná. Typickým úkolem je provést build programu po detekci

[14] https://travis-ci.org
[15] https://www.jetbrains.com/teamcity
[16] https://jenkins.io
[17] https://www.datanyze.com/market-share/ci/jenkins-market-share

změny v systému pro správu verzí a informovat o výsledcích. Proto se úloha dělí do tří hlavních částí.

Tzv. *trigger* je zodpovědný za spuštění úlohy, pokud je splněna vstupní podmínka. Jeho obvyklá implementace se periodicky dotazuje úložiště, zda-li nenastala změna. Parametry dotazování mají syntaxi shodnou s linuxovým programem `cron`. Úloha může být také spuštěna ručně buď z uživatelského rozhraní, nebo vzdáleně dotazem na speciální url adresu.

Část *builder* provádí konkrétní náplň úlohy. Zpočátku to byl především překlad softwaru pomocí nástrojů *Ant*[18] a *Maven*[19], nyní to může být také spuštění unit testů, měření pokrytí testů, generování dokumentace nebo, pokud je server nainstalován v linuxovém operačním systému, spuštění příkazu v shellu.

Poslední je tzv. *post-build* sekce. Ta se stará o informování uživatelů. Typickým prostředkem k informování je e-mail, ale je podporována i celá řada dalších kanálů (sms zprávy, IRC atd.). Post-build sekce může také definovat, které úlohy mají být dále spuštěny.

Úloha může být podobně jako běžný program parametrizována a parametr použit v rámci její definice.

Příklad zobrazení úlohy ve webovém rozhraní vidíme na obrázku 2.7. Úloha je difinována svým jménem, které se používá v url adrese a nemělo by obsahovat mezery. K zobrazení může být nastaveno odlišné jméno a detailnější popisek.

Builds V levém sloupci se nachází seznam *Build History*. Build vznikne provedením úlohy. Barevná tečka před jeho názvem signalizuje, zda skončil úspěšně, či nikoliv. Modrá tečka (ve starších verzích zelená) značí úspěch buildu, červená neúspěch. Během buildů mohou vznikat soubory. Pakliže je úloha nakonfigurována tak, aby je uchovávala, uloží se jako tzv. *artefakty* buildu. U úlohy je zobrazena složka s artefakty posledního úspěšného buildu (na obrázku pojmenována jako *Pracovní prostor*). Jelikož uchovávání artefaktů všech buildů může být prostorově náročné, lze Jenkins nakonfigurovat, aby ukládal pouze několik nejnovějších.

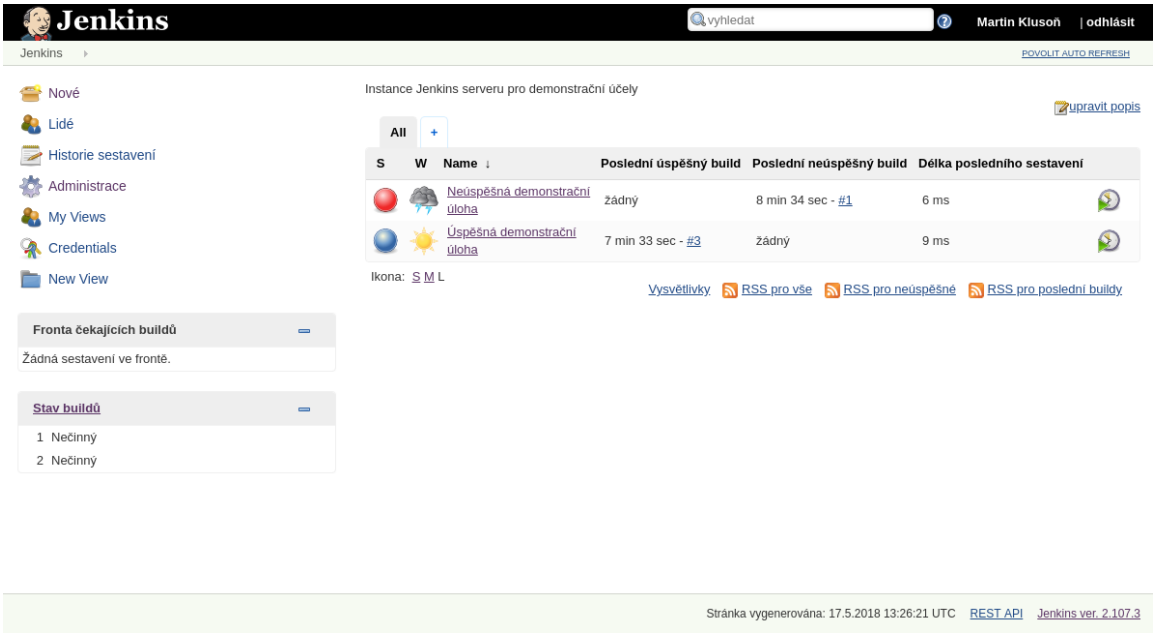
Každá úloha má množinu permanentních odkazů. Ty vedou na poslední, poslední úspěšný, poslední neúspěšný build atd. V případě, že úloha má nastaveno uchovávat pouze omezený počet posledních buildů a není mezi nimi žádný úspěšný, poslední úspěšný nebude smazán.

Uživatelské rozhraní Obrázek 2.8 zachycuje domovskou stránku webového rozhraní Jekins serveru. Významnou část zabírá seznam úloh. U každé je zobrazena informace o datu jejího posledního úspěšného a neúspěšného buildu, době trvání a v podobě ikonky se symbolem počasí poměr posledních úspěšných a neúspěšných buildů.

V levé části je zobrazen seznam buildů, které čekají na zpracování. Pod ním je seznam build serverů, jež úlohy zpracovávají. Takový server se označuje jako *slave* a provádí vlastní vykonání úloh. Každý slave má své označení. Úloha může vyžadovat, aby byla spuštěna pouze na serveru s patřičným označením. Na obrázku jsou oba zmíněné seznamy prázdné.

Pluginy Funkčnost Jenkins serveru je možné rozšiřovat *pluginy*. Už základní instalace obsahuje celou řadu pluginů a další je možné doinstalovat. Pro další výklad je zajímavý především *JMS Messaging Plugin*[20], který přidává podporu komunikace protokolem AMQP (viz 2.4.3). Přidává do sekce trigger možnost spouštět úlohu v reakci na přijetí zprávy a do sekce post-build možnost odeslat notifikace AMQP protokolem.

[18] https://ant.apache.org
[19] https://maven.apache.org
[20] https://wiki.jenkins.io/display/JENKINS/JMS+Messaging+Plugin



Obrázek 2.8: Uživatelské rozhraní Jenkins serveru

Jenkins job builder

Výklad o Jenkins serveru by nebyl kompletní bez zmínky o nástroji *Jenkins job builder*, který přidává možnost ovládat Jenkins z příkazové řádky (příkazem `jenkins-jobs`). Umí vytvářet, aktualizovat a mazat úlohy na serveru Jenkins. Ke své činnosti vyžaduje konfigurační soubor a popis Jenkins úloh ve formátu *yaml* nebo *json*. Konfigurace úloh je tak uložena v dobře čitelné podobě, může být verzována a posloužit k obnovení konfigurace Jenkins serveru do základního stavu. [3]

Výpis 2.3 ukazuje popis jednoduché Jenkins úlohy ve formátu yaml. Úloha se jmenuje „test-project“, ale zobrazovat se bude spolu s popisem název „Test projekt“, nevychází z žádné šablony (poject-typu: freestyle), její vykonávání není zakázáno (disabled: false), může běžet pralelně (concurrent: true) pouze na uzlu s označením „TestingNode“ a během svého vykonávání vypíše řetězec „Hello world!“.

```
- job:
    name: test-project
    project-type: freestyle
    description: 'A basic example of a jjb'
    disabled: false
    display-name: 'Test projekt'
    concurrent: true
    node: TestingNode
    builders:
        - shell: |
            echo 'Hello world!'
```

Výpis 2.3: Příklad Jenkins job builder šablony ve formátu yaml

2.4.2 Frameworky Citool a Gluetool

Následující podkapitola se věnuje popisu frameworku Gluetool a jeho derivátu Citool. Podkapitola čerpá informace z dokumentace projektu Gluetool [17].

Framework Gluetool

Gluetool je framework jazyka Python pracující v linuxovém shellu. Jeho hlavní myšlenkou je rozdělit logiku programu do samostatných oddělených bloků – tzv. *modulů*. Moduly jsou na příkazové řádce řazeny za sebe, v pořadí v jakém se budou vykonávat. Z toho plyne jeho hlavní výhoda – chování programu lze jednoduše modifikovat přidáváním, odebíráním nebo náhradou modulů ve volaném řetězci.

Framework zajišťuje postupné vykonávání modulů. Poskytuje modulům jednotný styl logování výstupů s podporou více úrovní detailu. Rozlišuje stupně: info, warn, debug a verbose. V případě vyvolání výjimky framework nabízí možnost jejího odeslání do služby *Sentry*[21]. Sentry zobrazuje výjimky přehledně seskupené podle jejich druhu a doplněné o časové a další údaje.

Výpis 2.4 ukazuje hypotetický příklad implementace zálohování souborů. Příkazu gluetool jsou jako parametry předány jména modulů, které bude postupně vykonávat a které dohromady tvoří zálohování do cloudového úložiště. Nejprve modul `find-files` určí, které soubory budou zálohovány. Následně modul `gzip-compress` provede kompresi a nakonec modul `dropbox-upload` odešle soubory na vzdálené úložiště.

```
> gluetool find-files gzip-compress dropbox-upload
```

Výpis 2.4: Příklad použití frameworku Gluetool

Výpis 2.5 ukazuje obdobný příklad jako 2.4 s tím rozdílem, že se soubory zálohují na vzdálený ftp server. Protože se uvedené dva příklady liší pouze implementací odeslání dat, stačí nahradit modul `dropbox-upload` modulem `ftp-upload`. Díky modularitě frameworku mohou být využity první dva moduly z předchozího příkladu. Samozřejmě, že k této záměně může dojít jen v případě, že moduly `dropbox-upload` a `ftp-upload` používají stejné rozhraní.

```
> gluetool find-files gzip-compress ftp-upload
```

Výpis 2.5: Další příklad použití frameworku Gluetool

Moduly Všechny funkce aplikace by měly být umístěny v *modulech*. Modul implementuje vždy pouze jednu samostatnou část programu. Je definován jako třída, která dědí z třídy `gluetool.glue.Module` a je umístěna v samostatném souboru.

Modul musí definovat alespoň jedno jméno v proměnné `name`, dále by měl poskytnout svůj popis v proměnné `description`.

Modul může pracovat s parametry jako každý jiný program. Parametry se definují jako slovník uložený v proměnné `options`. Všechny parametry musí být pojmenované. Atribut `required_options` obsahuje seznam jmen parametru, jež jsou pro modul nezbytné.

[21] https://sentry.io

Struktura modulu se skládá z metod sanity, execute a destroy. Může obsabovat i jiné metody a pomocné funkce.

- *Sanity* –je první metoda, kterou framework v modulu invokuje. Obvykle se používá k validaci vstupů nebo kontrole zdrojů nezbytných pro běh modulu.
- *Execute* – je hlavní metoda modulu. Framework ji zavolá po dokončení execute metody předcházejícího modulu. Pořadí volání execute method odpovídá pořadí modulů na příkazové řádce.
- *Destroy* – je volána po dokončení execute metod všech modulů i v případě, že během vykonávání nastala chyba. Destroy metody jsou invokovány v opačném pořadí, tedy první je volána destroy metoda modulu, jež je na příkazové řádce poslední. Typickým použitím je zavírání souborů nebo dealokace prostředků.

Rozhraní mezi moduly Moduly mezi sebou mohou komunikovat pomocí tzv. *sdílených funkcí*. Modul může sdílet některou ze svých metod s dalšími moduly tím, že její jméno uloží do seznamu v proměnné `shared_functions`. Taková funkce je pak dostupná všem následně prováděným modulům. Modul, který chce sdílenou funkci použít, tak učiní pomocí své metody `shared`, které jako první parametr předá název sdílené funkce. Typ návratové hodnoty sdílené funkce není nijak omezen. V případě, že více modulů sdílí funkci se shodným jménem, novější definice předefinuje tu předešlou.

Výpis 2.6 obsahuje minimální strukturu modulu frameworku Gluetool. Modul nese název „myapi“, během svého vykonávání vypíše řetězec „hello world“ a poskytuje sdílenou funkci `getapi()`, která vrací hodnotu třídní proměnné `api_version`.

```
class MyApiModule(Module):
    name = 'myapi'
    api_version = 2

    shared_functions = ['getapi']

    def getapi(self):
        return self.api_version

    def execute(self):
        self.info('hello world')
```

Výpis 2.6: Příklad jednoduchého modulu frameworku Gluetool

Konfigurace modulů Jak již bylo řečeno, každý modul může definovat své parametry. Hodnota parametru modulu může být zadána třemi způsoby. Tyto způsoby se uplatňují i na samotný framework. Pořadí, v jakém jsou uvedeny, odpovídá prioritám (hodnota s větší prioritou přepisuje tu s nižší).

1. Defaultní hodnota uvedená u parametru v proměnné `options`.
2. Hodnota nastavená v *konfiguračním souboru* modulu.

3. Hodnota zadaná na příkazovém řádku za jménem modulu.

Samotný framework a každý modul může definovat zakladí hodnoty svých parametru ve svém *konfiguračním souboru*, který obsahuje dvojice *parametr modulu = nastavená hodnota*. Konfigurační soubor má stejný název jako modul. Všechny konfigurační soubory jsou uloženy ve společném adresáři.

Koncept konfiguračních souborů přináší několik výhod. Hodnoty parametru jsou odděleny od kódu aplikace. Přenastavení aplikace, například z testovacího do produkčního režimu, je provedeno pouze změnou konfiguračního adresáře. Konfigurační soubory mohou být uloženy v systému pro správu verzí a těžit z jeho výhod (historie změn, verzování).

Výpis 2.7 ukazuje definici parametrů (včetně defaultních hodnot) hypotetického modulu `dropbox-upload`. Výpis 2.8 ukazuje jak by mohl vypadat jeho konfigurační soubor. Hodnoty z něj se použijí, pokud bude modul zavolán bez parametrů. V případě z výpisu 2.9 se pro parametr username použije hodnota z konfiguračního souboru a pro parametr password hodnota zadaná na příkazové řádce.

```
options = {
        'username': {
            'help': 'Dropbox user name',
            'default': 'default_user'
        },
        'password': {
            'help': 'Dropbox password',
            'default': 'default_passwd'
        }
```

Výpis 2.7: Příklad jednoduchého modulu frameworku Gluetool

```
[default]
username = batman
password = passwd1234
```

Výpis 2.8: Příklad jednoduchého modulu frameworku Gluetool

```
gluetool dropbox-upload --password 123passwd
```

Výpis 2.9: Příklad jednoduchého modulu frameworku Gluetool

Jména modulu Již bylo uvedeno, že modul je idenitifikován svým jménem – řetězcem uloženým v atributu `name`. Tato proměnná může obsahovat také seznam řetězců, modul pak má více jmen. Takový modul bude interpretován frameworkem Gluetool jako více modulů. Ty sice budou sdílet stejný kód, ale pro každý z nich bude použit jejich vlastní konfigurační soubor.

Stejného chování může být docíleno i během volání modulu z příkazové řádky použitím následující syntaxe: `[nové_jméno]:[existující_modul]`. Výpis 2.10 ukazuje takový postup na upraveném příkladu z výpisu 2.5. Modul `personal-ftp-upload` vykonává stejný kód jako `ftp-upload`, ale používá konfiguraci ze svého konfiguračního souboru.

```
> gluetool find-files gzip-compress personal-ftp-upload:ftp-upload
```

Výpis 2.10: Příklad přejmenování modulu na příkazové řádce

Framework Citool

Gluetool je generický framework, který neimplementuje žádné aplikačně orientované funkce. Pokud je potřeba takové funkce do frameworku přidat, vznikne jeho nadstavba. Prvním takový rozšířením je *Citool*.

Zaměřuje se na implementaci testování softwaru a průběžnou integraci. Přidává navíc následující funkce, které usnadňují:

1. Práci s výsledky testování (třída `libci.results.TestResult`).
2. Komunikaci a správu virtuálních strojů (třída `libci.guest.Guest`).

2.4.3 Další používané technologie

Následující podkapitola popisuje další nástroje a technologie, které je nutné znát, avšak jejich použití není svázáno pouze s průběžnou integrací.

Openstack

Openstack[22] je open source platforma, která umožňuje spravovat velké množství harwarových prostředků (výpočetních uzlů, úložišť, síťových prvků). Tyto prostředky poskytuje dále v podobě virtuálních strojů, které jsou spravovány z webového rozhraní. [5]

Ansible

Ansible[23] je open source nástroj k vzdálené správě konfigurace počítačů. Narozdíl od svých konkurentů (např. *Puppet*[24]) nevyžaduje přítomnost svého agenta na vzdáleném stroji, postačí mu funkční ssh spojení a přítomnost interpretu programovacího jazyka Python. [7]

Seznam počítačů je uložen v tzv. *inventory*. Inventory je seznam počítačů, které mají být spravovány. Počítač je identifikován ip adresou nebo názvem. Počítače mohou být sdružovány do skupin.

K popisu akcí, jež mají být na vzdáleném stroji provedeny, používá Ansible tzv. *playbooky*. Playbook je soubor ve formátu yaml, který má strukturu naznačenou ve výpisu 2.11. Vidíme, že obsahuje jednu sekci `hosts` pro počítače ze skupiny „webservers“. Na nich se pod uživatelem „root“ provedou dva úkoly – zajistí se přítomnost nejnovějšího balíčku „httpd“ a vytvoří se konfigurační soubor.

[22] https://www.openstack.org
[23] https://www.ansible.com
[24] https://puppet.com

```
---
- hosts: webservers
  remote_user: root

  tasks:
  - name: ensure apache is at the latest version
    yum:
      name: httpd
      state: latest
  - name: write the apache config file
    template:
      src: /srv/httpd.j2
      dest: /etc/httpd.conf
```

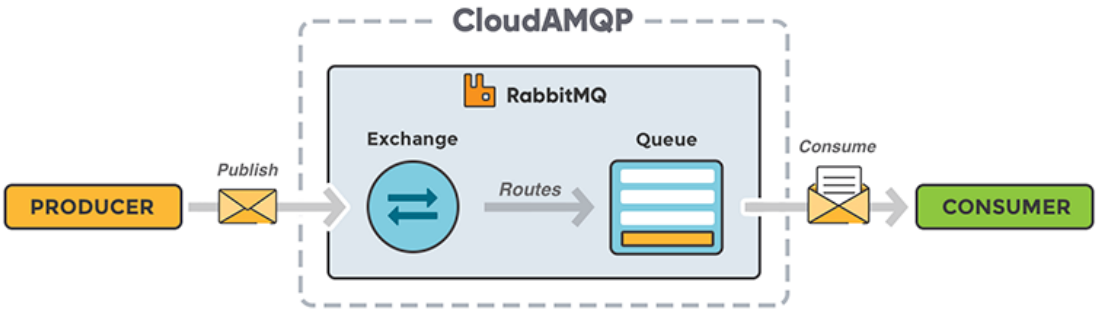
Výpis 2.11: Příklad Ansible playbooku

Advanced Message Queuing Protocol

Advanced Message Queuing Protocol[25] (AMQP) je otevřený protokol k přenos zpráv mezi systémy. Podkapitola čerpá informace z článku *An Advanced Message Queuing Protocol (AMQP) Walkthrough* [10].

Obrázek 2.9 znázorňuje schéma přenosu zpráv. *Odesílatel* (producer) posílá zprávy na *broker* (na obrázku je znázorněna přímo jedna z jeho implementací – *RabbitMQ*). Broker zprávu zpracuje a odešle *příjemci* (consumer). Broker se skládá z dalších podčástí:

- *Exchange* – přijímá zprávy od odesílatele.
- *Queue* (fronta) – sdružuje zprávy jednoho příjemce.
- *Bindings* – jsou pravidla pro přesměrování zpráv z exchange do queue.



Obrázek 2.9: Schéma posílání zpráv pomocí AMQP (obrázek převzat [1])

V rámci protokolu je definováno několik typů přesměrování.

[25] https://www.amqp.org

- *Direct Exchange* – zpráva je přiřazena do front podle svých parametrů. Fronta definuje, jaké parametry musí zpráva mít, aby do ní mohla být přiřazena.
- *Fanout Exchange* – zpráva je přiřazena do všech front nezávisle na parametrech.
- *Topic Exchange* – zpráva je přiřazena do front podle svých parametrů. Fronta definuje, jaké šabloně musí parametr zprávy odpovídat, aby do ní mohla být přiřazena.
- *Headers Exchange* – zpráva je přiřazena do front podle položek své hlavičky.

2.5 Průběžná integrace build systému Brew

Z předchozích podkapitol již víme, co je build systém Brew (2.1.2) i jaké jsou principy průběžné integrace (2.3), ale stále není zcela zřejmé, co je *průběžná integrace build systému Brew*.

V rámci vývoje operačního systému *Red Hat Enterprise Linux* (RHEL) jsou vyvíjeny jednotlivé jeho komponenty. Pro každou komponentu platí následující tvrzení:

- Zdrojové kódy komponenty jsou uloženy v systému pro zprávu verzí.
- Build komponenty do podoby rpm balíčku probíhá automatizovaně v build systému Brew.
- Existují pro ni automatizované test case implementované knihovnou Beakerlib (2.2.5).

Každá z komponent tak splňuje podmínky pro zavedení průběžné integrace. Zároveň je průběh testování všech komponent téměř shodný (znázorňuje ho obrázek 2.4). Bylo by nevýhodné řešit průběžnou integraci každé komponenty odděleně jako samostatný projekt. Proto vznikl jeden systém, který zajišťuje průběžnou integraci pro všechny komponenty. Dále ho budeme nazývat *průběžná integrace build systému Brew*.

2.5.1 Architektura průběžné integrace build systému Brew

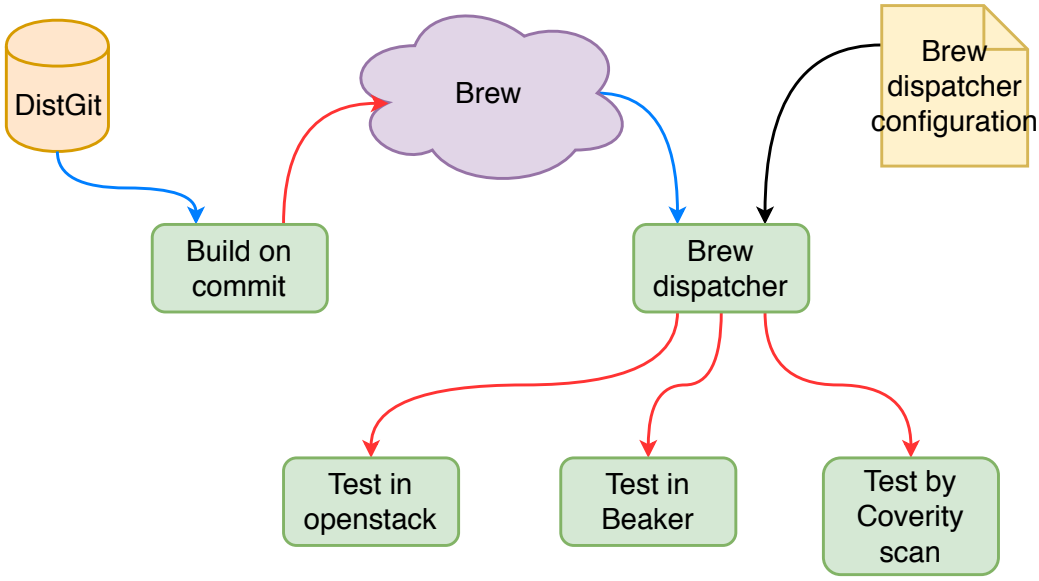
Průběžná integrace balíčků z build systému Brew je realizována pomocí serveru Jenkins (věnuje se mu podkapitola 2.4.1). Schéma používaných Jenkins úloh a dalších prvků, se kterými spolupracují, je zachyceno na obrázku 2.10. Úlohy jsou na obrázku zobrazeny zeleně. Prvky mezi sebou mohou komunikovat pomocí tzv. *message busu* (komunikační kanál založený na protokolu AMQP, který je popsán v části 2.4.3), tato komunikace je značena modrými šipkami. Červené šipky znamenají, že úloha může spustit další úlohu nebo build balíčku.

Úloha Brew dispatcher

Pokaždé, když Brew dokončí build nového balíčku, sdělí tuto informaci zasláním zprávy na message bus. JMS Messaging Plugin Jenkins serveru zprávu zachytí a spustí úlohu *Brew dispatcher*. Ta podle id buildu ze zprávy zjistí podrobnější informace o balíčku. Pokud pro takový balíček nalezne záznam v konfiguračním souboru testování, spustí daný záznam.

Úloha Test in Openstack

Jeden ze způsobů testování realizuje Jenkins úloha *Test in Openstack*. Ta je spuštěna Brew dispatcherem a jako parametry dostane id buildu, který má otestovat, id test plánu, který má použít, a seznam uživatelů, kterým má odeslat e-mail s výsledky testování. Úloha získá od služby Openstack (viz 2.4.3) virtuální stroj s odpovídajícím operačním systémem a nainstaluje do něj testovaný balíček. Následně spustí vykonávání testů na virtuálním stroji a po jejich dokončení vyhodnotí výsledky.



Obrázek 2.10: Schéma Jenkins úloh (zeleně), které jsou použity pro průběžnou integraci build systému Brew (fialově)

Úloha Test in Beaker

Další ze způsobů testování je podobný předchozímu s tím rozdílem, že stroj pro testování neposkytuje Openstack ale služba *Beaker*[26]. Realizuje ho úloha *Test in Beaker*.

Úloha Test by Coverity Scan

Poslední možností testování je úloha *Test by Coverity Scan*, která provádí statickou analýzu (viz 2.2.3) pomocí služby *Coverity Scan*. Pracuje se stejnými vstupy jako předchozí úlohy, s tím rozdílem, že předmětem testování je zdrojový rpm balíček.

Úloha Build on commit

Pro průběžnou integraci je nejtypičtější úloha, která reaguje na změnu kódu v centrálním úložišti a spouští build softwaru. Najdeme ji i v této implementaci. Úloha *Build on commit* dělá právě toto. Narozdíl od standardní implementace změna není zjišťována pollingem úložiště, ale zprávou poslanou přes message bus Jenkins serveru. Náplň úlohy je spustit build balíčku v build systému Brew.

Specifika oproti teorii průběžné integrace

Nyní se dostáváme k největší odlišnosti této implementace od teoretické průběžné integrace popsané dříve. Úloha Build on commit spouští build balíčku pouze pro změny provedené v tzv. *staging* větvích systému pro správu verzí, nikoliv pro tu hlavní. Použitím staging větví se implementuje private build, neboť jeho provedení na lokálním stroji je přinejmenším nepraktické, možná i nerealizovatelné. Staging větví může mít každý vývojář více a používat je k testování svých změn.

[26] https://beaker-project.org

Tím se ztrácí rozdíl mezi private buildem a integračním buildem, tudíž jeho opakování pozbývá smysl. Takový build lze však stále vytvořit, ale ne za použití úlohy Build on commit.

Některé vývojové týmy zatím krok build on commit přeskakují, vytvoření balíčku řeší ve vlastní režii a průběžnou integraci používají až od kroku Brew dispatcher.

2.5.2 Implementace Jenkins úloh

Z předcházejícího popisu Jenkins úloh vidíme, že i když realizují odlišné úkoly, ve všech se opakují stejné kroky. Například stále se opakující komunikace s Brew serverem za účelem získání podrobnějších informací o balíčku. Stejně tak e-mailové notifikace v úlohách, které provádí některý druh testování, jsou stále stejné.

Taková situace je ideální pro použití frameworku Citool. Abychom byli přesní, framework Citool vznikl jako reakce na tuto situaci. Později z něj byla vyčleněna obecná část – Gluetool.

Logika jednotlivých úloh je rozdělena do samostatných logických celků – modulů frameworku Citool tak, aby bylo možné jeden modul používat ve více úlohách. Implementace úlohy je volání těchto modulů v konkrétním pořadí a se správnými parametry. Úlohy používají v sekci builder volání shell příkazu, v něm je uložena sekvence gluetool modulů. Úloha je parametrizovatelná a parametry se předávají jednotlivým modulům.

Ke správě úloh se používá Jenkins job builder (viz 2.3), každá úloha je uložena ve vlastním souboru. Používá se formát yaml.

Implementace pomocí sekvencí modulů je dále vysvětlena na úlohách Brew dispatcher a Test in Openstack.

Jenkins úloha Brew dispatcher

Již bylo vysvětleno, *co* tato úloha dělá, nyní bude popsáno *jak*. Seznam vysvětluje účel jednotlivých modulů sekvence tak, jak jdou v úloze za sebou.

Pokud popis říká, že modul něco *poskytuje*, myslí se tím, že obsahuje sdílenou funkci, jež má danou entitu jako návratovou hodnotu.

- `brew` – poskytuje informace o Brew buildu. Trigger úlohy mu předá id buildu, které získal ze zprávy od build systému. Modul provádí dotazy na api Brew serveru a podle id získá podrobné informace, které dále poskytuje.
- `testing-thread` – poskytuje unikátní id úlohy. Id modul vytvoří jednocestnou funkcí z proměných spojených s aktuálním Jenkins buildem.
- `jenkins` – poskytuje funkce pro komunikaci s Jenkins serverem.
- `brew-build-name` – nastaví popisek aktuálního Jenkins buildu, který se zobrazí ve webovém uživatelském rozhraní.
- `publisher-umb-bus` – poskytuje funkci k odesílání zpráv na message bus.
- `pipeline-state-reporter` – pošle zprávu na message bus o zahájení úlohy (v metodě `execute`) a o ukončení úlohy (v metodě `destroy`).
- `test-batch-planner` – v konfiguračním souboru najde relevantní záznamy pro testovaný Brew build a poskytne je.
- `brew-dispatcher` – spustí záznamy, které získal od `test-batch-planner`. Tím spustí další úlohy.

Nyní se ještě zastavíme u konfiguračního souboru. To je soubor ve formátu yaml, ve kterém je uloženo, jaké typy testování a s jakými parametry se mají pro daný balíček spustit. Je rozdělen do oddílů, které obsahují nastavení pro jeden operační systém. Každý oddíl je rozdělen do následujících sekcí:

- `Rule` – obsahuje regulární výraz, který definuje, pro které operační systémy je daný oddíl určen.
- `Default` – obsahuje seznam modulů a parametrů, které se použijí, pokud není nalezena přesnější konfigurace.
- `All` – obsahuje seznam modulů a parametrů, které se použijí u všech balíčků bez ohledu na další konfiguraci.
- `Packages` – obsahuje seznam jednotlivých balíčků a pro každý balíček seznam modulů a parametrů, které se použijí k testování. Nahrazuje seznam ze sekce `Default`.

Příklad konfiguračního souboru testování zobrazuje výpis 2.12. Obsahuje pouze jeden oddíl pro operační systémy RHEL.

Pro komponentu `grep` se vybere řetězec `openstack-job`. To je opět modul frameworku Citool, jehož jediným úkolem je spustit Jenkins úlohu *Test in Openstack* a předat ji svoje parametry. Jenkins úloha pak předá tyto parametry jednotlivým modulům. Parametr `wow-options` je předán jako parametr modulu `wow`, parametr `notify-recipients-options` je předán jako parametr modulu `notify-recipients` atd.

```
---

- rhel:

  rule: BUILD_TARGET.match('(rhel-[6-7])(.[0-9]+)?')

  default:

  all:

  packages:

    grep:
      - openstack-job --wow-options="--plan=5494"
                      --notify-recipients-options="
                      --restraint-add-notify=mkluson"
```

Výpis 2.12: Ukázka konfiguračního souboru testování

Jenkins úloha Test in Openstack

Test in Openstack je jedna z úloh, kterou může spustit Brew dispatcher. Úloha je parametrizovatelná. Hodnoty parametrů jsou buď nastaveny v konfiguračním souboru testování, nebo je úloze předá Brew dispatcher.

Následuje seznam modulů, které úlohu implementují.

- `notify-recipients` – poskytne seznam příjemců e-mailové zprávy.
- `beah-xunit` – poskytuje funkci k serializaci Restraint výsledků.

- `testing-results` – poskytuje objekt k uložení a následnému zpracování výsledků.
- `testing-thread` – je shodný s předchozí úlohou, avšak v tomto případě je thread-id odvozeno z thread-id dispatcheru, jenž tuto úlohu spustil.
- `brew` – je shodný s předchozí úlohou. Build id předá této úloze dispatcher.
- `jenkins` – je shodný s předchozí úlohou.
- `brew-build-name` – je shodný s předchozí úlohou.
- `publisher-umb-bus` – je shodný s předchozí úlohou.
- `pipeline-state-reporter` – je shodný s předchozí úlohou.
- `build-dependencies` – rozšíří seznam balíčků v modulu `brew` o další závislosti, které jsou explicitně uvedené v konfiguraci testování. Používá se v případě některých balíčků, jejichž závislosti zatím nejsou součástí operačního systému.
- `ansible` – poskytuje funkci, která na vzdáleném stroji vykoná Ansible playbook.
- `restraint` – poskytuje funkci, která obaluje volání příkazu `restraint`.
- `guest-setup` – poskytuje funkci, která na testovacím stroji udělá nezbytná nastavení.
- `notify-email-beah-formatter` – poskytne tělo e-mailové zprávy.
- `wow` – poskytne metadata o testech, které mají být vykonány.
- `beah-result-parser` – poskytuje funkci, která zpracovává výsledky z restraintu.
- `openstack` – poskytuje přístup k api Openstacku, které umožňuje vytvoření virtuálního stroje.
- `guess-product` – poskytuje informaci o produktu podle detailů z modulu `brew` a mapovacího souboru.
- `guess-beaker-distro` – poskytuje informaci o distribuci podle detailů z modulu `brew` a mapovacího souboru.
- `guess-openstack-image` – poskytuje informaci o image podle detailů z modulu `brew` a mapovacího souboru.
- `restraint-scheduler` – získá seznam testů z modulu `wow`, pomocí předcházejících `guess-*` modulů zjistí produkt, distribuci a jméno image. Tyto informace použije k získání virtuálního stroje k testování z modulu `openstack`. Poskytne adresy strojů a seznam restraint úloh, které na nich mají být provedeny.
- `restraint-runner` – získá informace od předcházejícího modulu a pomocí funkce z modulu `restraint` je interpretuje. Ke zpracování výsledků používá funkci z modulu `beah-result-parser` a k jejich serializaci funkci z modulu `beah-xunit`.
- `notify-email` – odešle e-mailovou zprávu s výsledky testování. Příjemce mu poskytne modul `notify-recipients` a tělo zprávy poskytne modul `notify-email-beah-formatter`.

Z výše uvedeného seznamu modulů vidíme, že většina z nich poskytuje funkci pro následující modul. To sice umožňuje měnit implementaci těchto funkcí záměnou modulu za jiný, který poskytuje stejnou funkci, ale zároveň to v některých ohledech implementaci činí méně přehlednou.

Kapitola 3

Návrh průběžné integrace build systému Copr

Předcházející kapitola 2 popsala podrobně současný stav a zasadila ho do širšího kontextu. Nyní se na současnou situaci podíváme z pohledu vývojáře, který pracuje na svém softwaru a potřebuje pro něj vytvářet předkompilované instalační balíčky.

K tomuto účelu má k dispozici dva build systémy – Copr a Brew. K vytvoření produkční verze balíčku bude samozřejmě použit build systém Brew. Před tím se však vytváří balíčky zkušební a u nich volba nemusí být zcela zřejmá.

Výhodou Copru je, že vývojář může vytvářet projekty a přidávat do nich různé balíčky bez zásadních omezení. Výhodou Brew, které neposkytuje podobnou volnost v přidávání zdrojů, je, že je na něj napojena průběžná integrace, jež vytvořený balíček otestuje.

Vhodným řešením by bylo, kdyby i na build systém Copr byla napojena průběžná integrace. Ta by měla vycházet z průběžné integrace build systému Brew a sdílet s ní základ implementace, ale zároveň by mělo být možné ji nasadit zcela nezávisle. Tak nebude docházet k replikaci kódu a výsledky testování balíčků z obou build systémů budou lépe porovnatelné.

Návrh řešení výše popsaného úkolu je náplní této kapitoly.

3.1 Schéma Jenkins úloh

Lze předpokládat, že uživatelé průběžné integrace build systému Copr zároveň používají i průběžnou integraci build systému Brew. Proto není vhodné vytvářet pro Copr jinou strukturu úloh než se používá pro Brew.

Schéma na obrázku 3.1 ukazuje úlohy, které jsem navrhl. Můžeme si všimnout, že schéma je obdobou toho z obrázku 2.10, které popisuje průběžnou integraci systému Brew. Jsou ale vynechány některé úlohy.

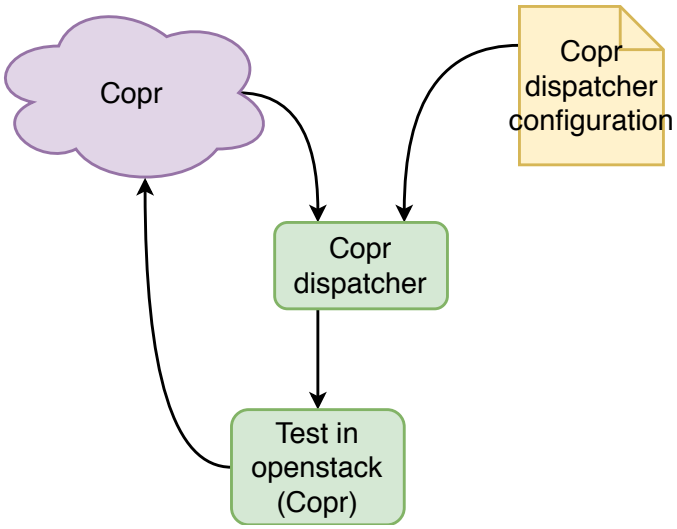
Protože build systém Copr poskytuje tzv. *webhooks* (spustí build v případě nové změny v repozitáři), není třeba implementovat úlohu *Build on commit*, která dělá totéž.

Jelikož se jedná o pilotní projekt průběžné integrace build systému Copr, je z testovacích úloh navržena pouze úloha *Test in Openstack (Copr)*. Další úlohy mohou být přidány později s využitím principů, které přináší tato práce.

Na obrázku je navíc vyznačena šipka z testovací úlohy do Copru. Ta naznačuje, že je navrženo předávání výsledků testování zpět do build systému, aby zde mohly být dále zpracovány.

Úloha *Copr dispatcher* plní v návrhu stejnou funkci jako *Brew dispatcher* v současné implementaci. Citool moduly, které implementují Brew dispatcher, budou zobecněny tak, aby mohly být použity i v úloze Copr dispatcher.

Stejný postup se uplatní i v případě modulů, které implementují současnou úlohu *Test in Openstack*, aby mohly tvořit úlohu *Test in Openstack (Copr)*.



Obrázek 3.1: Schéma Jenkins úloh (zeleně), které budou použity pro průběžnou integraci build systému Copr (fialově)

3.2 Analýza dostupných modulů

Návrh zahájíme analýzou dostupných modulů a jejich možnostmi použití v implementaci průběžné integrace build systému Copr.

Začneme moduly, které implementují Brew dispatcher.

- `brew` – nemůže být použit, jelikož zajišťuje napojení na build systém Brew. Poskytuje objekt, jehož atributy reprezentují vlastnosti Brew buildu. Dále tento objekt budeme nazývat *artefakt*. Tento modul bude nahrazen novým modulem `copr`, který bude reprezentovat balíček z Copru a také bude poskytovat artefakt.
- `testing-thread` – může být použit, ale vyžaduje úpravu implementace. Současný stav očekává konkrétní atributy artefaktu.
- `jenkins` – může být použit beze změny.
- `brew-build-name` – musí být zobecněn, nebo nahrazen jiným modulem.
- `publisher-umb-bus` – může být použit beze změny.
- `pipeline-state-reporter` – může být použit, ale vyžaduje úpravu implementace. Současný stav očekává konkrétní atributy artefaktu.

- `test-batch-planner` – může být použit, ale vyžaduje úpravu implementace. Současný stav očekává konkrétní atributy artefaktu. Jeho vstupem bude nový konfigurační soubor testování pro balíčky z build systému Copr.
- `brew-dispatcher` – i když obsahuje řetězec „brew“ ve svém názvu, jeho funkcionalita s Brew nijak nesouvisí. Bude přejmenován a použit.

Následují moduly, které implementují úlohu *Test in Openstack*. Z výčtu jsou vynechány ty moduly, které jsou již zahrnuty výše.

- `notify-recipients` – může být použit, ale vyžaduje úpravu implementace. Současný stav očekává konkrétní atributy artefaktu.
- `beah-xunit` – může být použit beze změny.
- `testing-results` – může být použit beze změny.
- `build-dependencies` – jeho funkce je blízce navázána na modul `brew`. Nebude použit.
- `ansible` – může být použit beze změny.
- `restraint` – může být použit beze změny.
- `guest-setup` – může být použit beze změny.
- `notify-email-beah-formatter` – může být použit beze změny.
- `wow` – může být použit, ale vyžaduje úpravu implementace. Současný stav očekává konkrétní atributy artefaktu.
- `beah-result-parser` – může být použit beze změny.
- `openstack` – může být použit beze změny.
- `guess-product` – může být použit beze změny. Potřebné mapování musí být doplněno do mapovacího souboru.
- `guess-beaker-distro` – může být použit beze změny. Potřebné mapování musí být doplněno do mapovacího souboru.
- `guess-openstack-image` – může být použit beze změny. Potřebné mapování musí být doplněno do mapovacího souboru.
- `restraint-scheduler` – z modulu musí být vyčleněna do samostatného modulu část, která se stará o instalaci balíčku z Brew na testovací stroj. Tato vyčleněná část bude pro Copr nahrazena vlastním modulem, který zajistí instalaci balíčku z Copru.
- `restraint-runner` – může být použit beze změny.
- `notify-email` – může být použit, ale vyžaduje úpravu implementace. Současný stav očekává konkrétní atributy artefaktu.

Nakonec se zaměříme na modul, který nefiguruje v implementaci žádné z úloh, ale používá se v konfiguraci testování.

- `openstack-job` – spouští Jenkins úlohu *Test in Openstack* a předává ji své parametry. Může však spustit i jinou Jenkins úlohu, stačí mu jméno takové úlohy předat jako parametr.

Můžeme ho tedy využít i ke spuštění nové úlohy *Test in Openstack (Copr)*. Konfigurační soubory testování obou průběžných integrací tak budou používat stejné položky. Jediným rozdílem bude, že konfigurační soubor pro Copr bude mít u svých položek navíc parametr se jménem nové úlohy.

Tím ale také vzniká omezení pro implementaci nové úlohy. Musí používat stejné parametry jako úloha *Test in Openstack*.

3.3 Modul Copr

Z předchozí analýzy vyplývá nutnost vytvořit nový modul ve frameworku Citool – `copr`. Ten poskytuje zbytku modulů v sekvenci artefakt, stejně jako to ve stávající průběžné integraci dělá modul `brew`. artefakt je objekt, který nese informace o testovaném balíčku.

Hlavní úlohou tohoto modulu je získat informace z build serveru a poskytnout je. K tomu může používat například program příkazové řádky `copr-cli`, který nejen informace poskytuje, ale umí i zadávat příkazy build systému. Protože se jedná o program, který běží na klientské stanici, je k jeho použití nutné, aby zde byl nainstalován, nakonfigurován a posléze spravován.

Praktičtější alternativou je použití programového rozhraní serveru. První jeho výhodou je, že není vyžadována přítomnost zvláštního programového vybavení, protože ke komunikaci s api stačí http dotazy. Druhou výhodou je, že data jsou poskytnuta ve formátu json a usnadňují tak jejich strojové zpracování. Všechny argumenty svědčí ve prospěch použití programového rozhraní.

Modul ke své inicializaci potřebuje id buildu a název prostředí (chroot name), pro které byl balíček vytvořen. Modul během vykonávání provede dotaz na následující části programového rozhraní.

- *Build_tasks* – na základě build id a chroot name získá modul informaci o výsledku build tasku, tedy zda bylo vytvoření balíčku úspěšné či nikoliv.
- *Builds* – na základě build id modul zjistí název a verzi balíčku a také jméno uživatele, jenž překlad inicializoval. Nachází se zde také url adresa projektu, pod který build spadá.
- *Project* – poskytne modulu jméno projektu a jméno jeho vlastníka.

Modul tyto informace uloží do artefaktu a poskytne ho dalším modulům tak, že ho předá jako návratovou hodnotu své sdílené funkce. Ta má stejné jméno jako obdobná sdílená funkce modulu `brew` (jmenuje se *primary_task*).

3.4 Kontext artefaktu

Dalším zjištěním, které vyplývá z předchozí analýzy (podkapitola 3.2), je, že podstatná část modulů přistupuje přímo k jednotlivým atributům artefaktu. To může způsobit potíže v případě, kdy Copr artefakt nedisponuje stejnými parametry jako Brew artefakt, či naopak. Například Brew artefakt má atribut „scratch“, který indikuje, zda se jedná o scratch build či

nikoliv. V Copru se nic takového nevyskytuje. Na druhou stranu Copr artefakt má například atribut „project“, který nedává smysl v rámci build sytému Brew.

Nabízí se jednoduché řešení. Sloučit atributy všech artefaktů. Každý artefakt pak bude poskytovat tuto množinu. Atributy, které u něj nemají smysl, budou mít hodnotu `None`. Nebo povolíme použití jen těch atributů, které jsou pro všechny artefakty společné. Ani jedno z popsaných řešení není ideální a do budoucna by přineslo spíše zněpřehlednění implementace zbylých modulů.

Při dalším zkoumání modulů, které přistupují přímo k atributům artefaktu, bylo zjištěno, že většina z nich je používá následovně. Hodnoty atributů uloží do slovníku, kde je klíčem název proměnné. Tento slovník dále použijí jako tzv. kontext během renderování šablon, nebo během vyhodnocování pravidel. To nás už vede k přijatelnému řešení.

Slovník svých atributů vytvoří přímo modul, který poskytuje artefakt (`copr` nebo `brew`) a poskytne ho sdílenou funkcí ostatním modulům. Tím se zjednoduší implementace modulů, které kontext artefaktu používají. Tento poskytnutý slovník budeme dále označovat jako *kontext artefaktu*.

Novým povinným atributem artefaktu, a zároveň povinnou položkou kontextu artefaktu je `artefakt_type`, který definuje typ artefaktu. Pro Copr artefakt má hodnotu „copr-build“.

Zbylé moduly, které přistupují přímo k atributům artefaktu, jsou buď blízce spjaty s fungováním modulu `brew` a pro `copr` se nepoužijí, nebo musí být upraveny tak, aby kontext artefaktu podporovaly.

Bližšímu popisu této problematiky se věnuje podkapitola 4.2.

3.5 Úloha Copr dispatcher

Jenkins úloha *Copr dispatcher* je spuštěna JMS Messaging Pluginem na základě zprávy přijaté přes message bus. Tuto zprávu odešle Copr v okamžiku, kdy je dokončen build task. JMS Messaging plugin uloží obsah zprávy do proměnné prostředí a spustí builder část úlohy. Ta obsahuje následující sekvenci modulů.

- `trigger-message` – stávající modul frameworku Citool, který zatím nebyl zmíněn v rámci popisu jiné úlohy. Získá hodnotu proměnné prostředí, kterou tam předtím uložil JMS Messaging plugin, a poskytne ji.
- `copr` – nový modul, který poskytuje ostatním artefakt a kontext artefaktu. K inicializaci použije build id a chroot name, které získá ze zprávy poskytnuté předešlým modulem.
- `testing-thread` – je upraven. Jako parametr dostane seznam názvů proměnných z kontextu artefaktu, které jsou vstupem jednosměrné funkce. Dříve byly použité proměnné zanesené v jeho kódu.
- `jenkins` – je převzat beze změn.
- `copr-build-name:jenkins-build-name` – je ukázka přejmenování modulu na příkazové řádce (viz 2.4.2). Modul `jenkins-build-name` je přejmenovaný a přepracovaný modul `brew-build-name`. Jméno, které buildu nastavuje, už není pevně nastaveno v modulu, ale předává se modulu parametrem. Hodnota tohoto parametru může být šablona a obsahovat proměnné z kontextu artefaktu. Díky přejmenování na příkazové řádce se používá konfigurace pro modul `copr-build-name`.

- `publisher-umb-bus` – je převzat beze změn.
- `pipeline-state-reporter` – je upraven. Nově používá konfigurační soubor, který definuje, jaké proměnné z kontextu artefaktu má zahrnout do zprávy v závislosti na typu artefaktu.
- `test-batch-planner` – pro správný chod celé úlohy je zde nutné udělat několik modifikací. Věnuje se jim odstavec následující po tomto výčtu.
- `copr-dispatcher` – je další jméno modulu `task-dispatcher`, který vznikl přejmenováním původního modulu `brew-dispatcher`. Jinak je modul beze změn.

Modul `test-batch-planner` používá k vyhledávání odpovídajícího záznamu v konfiguraci testování hodnotu atributu `component` artefaktu. Ten obsahuje název balíčku. Takové řešení funguje s build systémem Brew, kde jsou názvy balíčku unikátní. Ale v případě Copru, kde více projektů může obsahovat balíček se stejným názvem, je nedostačující. K jednoznačné identifikaci balíčku v build systému Copr je potřeba jméno majitele projektu, jméno projektu a název balíčku.

Řešení je následující. Každý artefakt musí mít atribut `component_id`, který obsahuje řetězec, jenž identifikuje balíček v rámci konfiguračního souboru testování. Modul `test-batch-planner` pak během vyhledávání používá hodnotu z atributu `component_id` místo z atributu `component`.

Dalším rozdílem této úlohy oproti úloze *Brew dispatcher* je, že je spuštěna pro všechny dokončené build tasky, ne jen pro úspěšně dokončené. Testování balíčku, který se nepodařilo vytvořit, postrádá smysl, a proto by žádné testování nemělo být naplánováno.

Proto Copr artefakt nabízí atribut `testable`. Ten je nastaven na hodnotu `true`, pokud build proběhl v pořádku, a na hodnotu `false` pokud ne. Modul `test-batch-planner` tuto hodnotu zkontroluje před plánováním testů a v případě, že je záporná, neposkytne žádné testy. Zbytek modulů v sekvenci se dál vykoná, ale bez efektu.

3.6 Úloha Test in Openstack (Copr)

Nyní si ukážeme, z jakých modulů se skládá Jenkins úloha *Test in Openstack (Copr)*. Úloha je spouštěna modulem `openstack-job`, který je invokován úlohou *Copr dispatcher*.

- `notify-recipients` – je upraven. Nově používá kontext artefaktu.
- `beah-xunit` – je převzat beze změn.
- `testing-results` – je převzat beze změn.
- `testing-thread` – je shodný s předchozí úlohou.
- `copr` – je shodný s předchozí úlohou.
- `jenkins` – je převzat beze změn.
- `copr-build-name:jenkins-build-name` – je shodný s předchozí úlohou.
- `publisher-umb-bus` – je převzat beze změn.
- `pipeline-state-reporter` – je shodný s předchozí úlohou.

- `ansible` – je převzat beze změn.
- `restraint` – je převzat beze změn.
- `guest-setup` – je převzat beze změn.
- `notify-email-beah-formatter` – je převzat beze změn.
- `install-copr-build` – nový modul poskytující funkci, která zajistí instalaci nově vytvořeného balíčku v build systému Copr na testovací stroj.
- `wow` – je upraven. Nově používá kontext artefaktu.
- `beah-result-parser` – je převzat beze změn.
- `openstack` – je převzat beze změn.
- `guess-product` – je převzat beze změn.
- `guess-beaker-distro` – je převzat beze změn.
- `guess-openstack-image` – je převzat beze změn.
- `restraint-scheduler` – je upraven. Byla z něj odstraněna část, která pomocí Beakerlib knihovny instaluje balíček z Brew na testovací stroj.
- `restraint-runner` – je převzat beze změn.
- `notify-email` – je upraven. Nově používá kontext artefaktu.

V předcházejícím výčtu můžeme pozorovat výhody použití frameworku Citool. Většina modulů, které vyžadovaly úpravu, je již zpracována v rámci návrhu předchozí úlohy.

Nejdůležitější úpravy v této úloze je instalace testovaného balíčku na testovací stroj. Původní implementace používá k instalaci Restraint a test case implementovaný knihovnou Bekaerlib (viz 2.2.5). Tato úloha k tomu používá nástroj Ansible (viz 2.4.3), který se pro takový úkol zdá vhodnější.

Změny se týkají modulu `restraint-scheduler` a nového modulu `install-copr-build`. Podrobně se těmto změnám věnuje podkapitola 4.3.

3.7 Export výsledků

Poslední část ze schématu na obrázku 3.1, která zatím nebyla popsána je export výsledků testování zpět do build systému Copr. Tato část nemá svůj vzor v průběžné integraci build systému Brew. Jedná se o pilotní projekt.

O předání výsledků testování se bude starat program běžící na straně build systému Copr.

Výsledky testování jsou uloženy v artefaktech buildů Jenkins úlohy *Test in Openstack (Copr)*. Lze předpokládat, že Jenkins server nebude uchovávat všechny buildy. Ke zpracování artefaktů úlohy by musel program periodicky provádět dotazy na Jenkins server. Zjistit číslo posledního buildu úlohy, porovnat ho s číslem buildu, který naposledy zpracoval a zpracovat všechny buildy, které vznikly v době od posledního dotazu.

Druhou možností je využít zpráv, které odesílá modul `pipeline-state-reporter`. Ten po skončení úlohy odešle zprávu ma message bus, která obsahuje výsledek testování. Navíc, díky tomu že modul implementuje podporu kontextu artefaktu, je možné pomocí jeho konfiguračního souboru nastavit, které údaje o artefaktu budou přidány do zprávy. Implementace programu, který získává výsledky tímto způsobem, je podstatně jednodušší. Jedná se o příjemce zpráv protokolu AMQP (viz 2.4.3). Výsledky testování jsou jednoduše, díky použitému formátu json, získány z přijaté zprávy.

Kapitola 4

Implementace průběžné integrace build systému Copr

Předchozí kapitola 3 popsala, jaké úpravy bylo nutné provést v modulech frameworku Citool k tomu, aby mohly implementovat průběžnou integraci build systému Copr. Tato kapitola se věnuje detailnějšímu popisu implementace těchto změn.

4.1 Implementace modulu Copr

Hlavní náplní nového modulu `copr` je získat informace o konkrétním build tasku a poskytnout je. K získávání informací používá programové rozhraní build serveru Copr. To je umístěno na adrese `/api_2`. Například pro Copr build systém distribuce Fedora se jedná o adresu `https://copr.fedorainfracloud.org/api_2/`. Další text bude uvádět všechny adresy programového rozhraní bez úvodní url adresy serveru.

Copr modul obdrží jako vstup build id (například 692690) a chroot name (například „epel-7-x86_64“). Z nich získá všechny potřebné údaje následovně.

Z adresy `/api_2/build_tasks/692690/epel-7-x86_64` získá modul data zobrazená ve výpisu 4.1. Data jsou ve formátu json a obsahují jak informace o build tasku (položka „build_task“) tak odkazy na příslušný build a projekt (položka „_links“). Modul si uloží informaci, jak build task dopadl (položka „state“), a adresu složky s výsledky (položka „result_dir_url“).

```
{
    "build_task": {
        "build_id": 692690,
        "result_dir_url":"https://copr-be.cloud.fedoraproject.org/
                          results/msuchy/Git/epel-7-x86_64/00692690-git/",
        "ended_on": 1514911241,
        "started_on": 1514909799,
        "git_hash": "4bacec3381a1ee67376125a60e0652bfae4f1930",
        "chroot_name": "epel-7-x86_64",
        "state": "succeeded"
    },
    "_links": {
        "project": {
```

```
            "href": "/api_2/projects/17643"
        },
        "build": {
            "href": "/api_2/builds/692690"
        },
        "self": {
            "href": "/api_2/build_tasks/692690/epel-7-x86_64"
        }
    }
}
```


Výpis 4.1: Ukázka informací o build tasku poskytnutých programovým rozhraním Copru


```
{
    "build": {
        "package_version": "2.15.1-3.fc28",
         "submitted_on": 1514909786,
         "submitter": "msuchy",
         "enable_net": false,
         "built_packages": [{
             "name": "git",
             "version": "2.15.1"
         }],
         "id": 692690,
         "source_type": "link",
         "package_name": "git",
         "repos": [],
         "state": "succeeded",
         "source_metadata": {
            "url": "https://kojipkgs.fedoraproject.org/packages/
                   git/2.15.1/3.fc28/src/git-2.15.1-3.fc28.src.rpm"
         }
    },
    "_links": {
        "self": {
            "href": "/api_2/builds/692690"
        },
        "project": {
            "href": "/api_2/projects/17643"
        },
        "build_tasks": {
            "href": "/api_2/build_tasks?build_id=692690"
        }
    }
}
```


Výpis 4.2: Ukázka informací o buildu poskytnutých programovým rozhraním Copru

Z adresy `/api_2/builds/692690` získá modul data zobrazená ve výpisu 4.2. Data popisují build, v rámci kterého vznikl předchozí build task. Důležité položky, jež si modul uloží jsou „submitter“, „package_name“ a „package_version“. Dále použije adresu ze sekce „project“ k získání dalších informací o projektu.

```
{
    "project": {
        "instructions": "",
        "disable_createrepo": false,
        "build_enable_net": false,
        "owner": "msuchy",
        "id": 17643,
        "is_a_group_project": false,
        "contact": "",
        "repos": [],
        "description": "",
        "name": "Git"
    },
    "_links": {
        "self": {
            "href": "/api_2/projects/17643"
        },
        "builds": {
            "href": "/api_2/builds?project_id=17643"
        },
        "chroots": {
            "href": "/api_2/projects/17643/chroots"
        },
        "build_tasks": {
            "href": "/api_2/build_tasks?project_id=17643"
        }
    }
}
```

Výpis 4.3: Ukázka informací o projektu poskytnutých programovým rozhraním Copru

Z adresy `/api_2/projects/17643` získá modul data zobrazená ve výpisu 4.3. Data popisují projekt, v němž vznikl předcházející build. Důležité položky, které si modul uloží jsou, „name“ a „owner“.

Získané položky „package_name“, „name“ a „owner“ se oddělené lomítky vloží jako řetězec do atributu `component_id` artefaktu, který slouží k identifikaci balíčku v konfiguračním souboru testování. S daty z předchozího příkladu bude mít atribut hodnotu „msuchy/Git
/git“.

Atribut `id` artefaktu je řetězec skládající se z build id a chroot name. K jejich oddělení v řetězci se použije dvojtečka. V tomto případě bude mít atribut hodnotu „692690:epel-7-x86_64“.

V prvním kroku získal modul url adresu složky s výsledky build tasku. V ní jsou uloženy výstupní předkompilované rpm balíčky a také soubor se záznamy o průběhu překladu. Modul z těchto záznamů pomocí regulárního výrazu získá jména rpm balíčku a uloží si jejich url adresy do atributu `rpm_urls` artefaktu. Tento atribut bude následně použit k instalaci balíčků na testovací stroj.

4.2 Poskytování kontextu

Největší změnou, kterou tato práce přinesla do implementace Citool modulů, je koncept předávání kontextu artefaktu. Jeho princip a přínos bude demonstrován na změně implementace modulů `notify-email` a `jenkins-build-name`.

Nejprve se podíváme na to, jak takový kontext artefaktu vypadá. Výpis 4.4 ukazuje jeho obsah pro Copr artefakt. Vrací ho sdílená funkce `artifact_context`. Objekt `primary_task` je artefakt a lze ho též získat sdílenou funkcí `primary_task`.

```
{
    'ARTIFACT_TYPE': primary_task.ARTIFACT_NAMESPACE,
    'BUILD_TARGET': primary_task.target,
    'NVR': primary_task.nvr,
    'PRIMARY_TASK': primary_task,
    'TASKS': self.tasks()
}
```

Výpis 4.4: Kontext Copr artefaktu

Ve výpisu 4.5 vidíme změnu implementace v modulu `notify-email`. Podobná změna proběhla v celé řadě dalších modulů, které stejně jako tento používají šablony a potřebují k jejich vyplnění dodat kontext.

Část kódu nad pomlčkami ukazuje původní implementaci a část pod ní novou využívající kontext artefaku.

```
rules_context = {'BUILD_TARGET': self.shared('primary_task').target,
                 'PRIMARY_TASK': self.shared('primary_task'),
                 'TASKS': self.shared('tasks'),
                 'RECIPIENTS': recipients
-------------------------------------------------------------------------
rules_context = gluetool.utils.dict_update(self.shared('artifact_context'),
                                           {'RECIPIENTS': recipients})
```

Výpis 4.5: Změna implementace modulu notify-email před (horní část) a po použití kontextu artefaktu (spodní část)

Modul už nepřistupuje přímo k atributu `target`. Nyní, pokud artefakt nemá atribut `target`, nedojde k vyvolání výjimky, ale pouze nebude doplněn do šablony, protože tato položka nebude ani v kontextu artefaktu. V takovém případě stačí upravit pouze používanou šablonu, nikoliv kód modulu.

Podobný slovník si vytvářelo více modulů, čímž docházelo k replikaci kódu, která se nyní odstranila. Nová implementace také ukazuje, jak je možné do kontextu přidat další položky (funkce `gluetool.utils.dict_update` spojí oba slovníky).

Druhý příklad ve výpisu 4.6 ukazuje změnu implementace modulu `jenkins-build-name`, který měl původně v kódu napevno nastavenou strukturu proměnné `name`.

V nové implementaci využívající kontext artefaktu je struktura proměnné `name` definována šablonou, kterou modul dostane jako parametr (hodnota se získá voláním `self.option ('name')`). Kontext se doplní o další hodnotu a šablona se vyplní (funkce `render_template` obaluje volání knihovny `jinja2`).

```
task = self.shared('primary_task')
name = task.short_name
thread_id = self.shared('thread_id')
name = '{}:{}'.format(thread_id, name)
-------------------------------------------------------------------------
context = {}
context.update(self.shared('artifact_context')
context.update({
               'THREAD_ID': self.shared('thread_id')
               })

name = gluetool.utils.render_template(jinja2.Template(self.option('name')),
                                      logger=self.logger,
                                      **context)
```

Výpis 4.6: Změna implementace modulu jenkins-build-name před (horní část) a po použití kontextu artefaktu (spodní část)

4.3 Instalace testovaného balíčku

Nedílnou součástí testování je instalace testovaného balíčku do testovacího prostředí. Tento úkol byl zahrnut v Citool modulu `restraint-scheduler`, který připravuje seznam restraint úloh a jako první takovou úlohu zařadil instalaci balíčku z Brew. Abychom mohli tento modul použít i v implementaci průběžné integrace build systému Copr, bylo nutné přidávání této instalační úlohy z modulu odstranit. Instalace byla vyčleněna do samostatného modulu `install-brew-build`.

Rozhraní těchto dvou modulů je zajištěno následovně. Modul `restraint-scheduler` pro každý virtuální stroj, který získá pro testování, spustí funkci, jež na stroji provede prvotní nastavení. Tato funkce je sdílená, jmenuje se `setup_guest()` a poskytuje ji modul `guest-setup`. Modul `install-brew-build` poskytuje také funkci `setup_guest()`, a protože je v sekvenci modulů umístěn až za modulem `guest-setup`, použije se sdílená funkce modulu `install-brew-build`. Ta jako první zavolá sdílenou funkci `setup_guest()` z modulu `guest-setup`, a poté nainstaluje na testovací stroj testovaný balíček.

Touto reorganizací se zatím nijak nezměnilo chování původní sekvence modulů, pouze se vytvořil prostor k jednoduché změně implementace instalace testovaného balíčku. Modul

`install-brew-build` může být nahrazen jiným, který dodržuje výše popsané rozhraní. Příklad takového modulu je `install-copr-build`.

4.3.1 Modul install-copr-build

Modul `install-copr-build` poskytuje funkci, která na vzdálený stroj nainstaluje testovaný balíček.

Copr nabízí pro každý svůj projekt repozitář, z něhož je možné balíčky projektu instalovat. Tak lze však nainstalovat pouze poslední verze balíčků, nikoliv balíček z konkrétního build tasku. Proto jsou instalovány přímo z url adres, které poskytne Copr artefakt v atributu `rpm_urls`. Popis získání těchto adres je uveden v podkapitole 4.1.

Jak již bylo vysvětleno v podkapitole 3.7, tento modul používá nástroj Ansible (viz 2.4.3). Ansible sice disponuje pluginem pro práci s balíčky pomocí nástroje `yum`, ale jeho možnosti jsou značně omezené. Umí zajistit pouze přítomnost a nepřítomnost daného balíčku v systému, což je pro naše účely nedostačující, protože tím nelze implementovat nahrazení balíčku.

Je nutné použít volání shell příkazů, které budou obsahovat volání programu `yum` s různými parametry. Není možné balíček jednoduše odinstalovat a poté znovu nainstalovat, protože v případě systémových komponent to správci balíčků nedovolí operační systém.

K zajištění přítomnosti balíčku z build tasku je nutné provedení následující posloupnosti příkazů.

1. `yum reinstall` – pokrývá případ, kdy je již nainstalován balíček ve stejné verzi.
2. `yum install` – pokrývá případ, kdy balíček není nainstalován.
3. `yum update` – pokrývá případ, kdy je již nainstalován balíček v nižší verzi.
4. `yum downgrade` – pokrývá případ, kdy je již nainstalován ve vyšší verzi.

Tato volání jsou součástí Ansible playbooku, který si na testovací stroj aplikuje. Url adresy balíčků, jež se mají nainstalovat, jsou playbooku předány jako parametr. Výpis 4.7 zachycuje implementaci jednoho z volání. Na jeho konci se nachází příkaz `ignore_errors: "yes"`, který nastaví, že se playbook neukončí po provedení první neúspěšné úlohy. Implementace předpokládá, že z výše popsaných volání skončí úspěšně pouze jedno a ostatní skončí chybou, proto je nutné nastavit ignorování chyb.

```
- name: try reinstall package
  shell: yum -y reinstall "{{ item }}"
  args:
    warn: "False"
  with_items:
    - "{{ PACKAGE_URLS }}"
  ignore_errors: "yes"
```

Výpis 4.7: Ukázka úlohy z Ansible playbooku, který instaluje testovaný balíček

Výsledný Ansible playbook se nachází v souboru `install-build-packages.yaml`

4.4 Implementace Jenkins úloh

V rámci této práce vznikly dvě nové Jenkins úlohy – *Copr dispatcher* a *Test in Openstack (Copr)*. Z jakých Citool modulů se skládají, už popisují podkapitoly 3.5 a 3.7. K jejich vlastnímu zaznamenání byly použity šablony Jenkins job builderu (viz 2.3) ve formátu yaml.

Tyto šablony nesou názvy `ci-copr-dispatcher.yaml` a `ci-openstack-copr.yaml`. Implementace modulů, které používají, se nachází ve složce `gluetool-modules`.

Pro úlohu *Copr dispatcher* vznikl nový konfigurační soubor testování s názvem `copr-dispatcher.yaml`.

4.5 Implementace předávání výsledků testování

Příjem informací na straně build systému Copr zajišťuje program s názvem `copr_listener.py`. Jedná se o skript implementovaný v jazyce Python, který s využitím knihovny *Stomp*[1] přijímá zprávy z message busu.

Přijaté zprávy ve formátu json jsou zpracovány a jsou z nich získány podstatné informace: build task id, výsledek testování a url adresa buildu Jenkins úlohy, která testování provedla. Program tyto údaje zobrazí, případně je může poskytnout build sytému například k zobrazení ve webovém uživatelském rozhraní.

Tento program jako jediný z této práce nevyužívá framework Gluetool, protože režie spojená s jeho údržbou na build serveru by nevyvážila výhody, které by jeho použití přineslo.

[1] https://stomp.github.io

Kapitola 5

Závěr

Výsledkem této diplomové práce je implementace průběžné integrace build systému Copr. Systém je v současné době nasazen v testovacím prostředí a je plánováno jeho používání v produkčním prostředí v rámci interní infrastruktury ve společnosti Red Hat Czech s.r.o.

Během práce vznikly nové Jenkins úlohy *Copr dispatcher* a *Test in Openstack (Copr)*. Byl vytvořen nový modul frameworku Citool, který zajišťuje komunikaci s build systémem Copr. Práce dále přináší nový způsob sdílení informací mezi moduly – *kontext artefaktu*. Ten zjednodušuje a zobecňuje jejich implementaci. Testovaný balíček je do testovacího prostředí nově instalován pomocí Ansible playbooku. V neposlední řadě byl také navržen a implementován způsob předávání výsledků testování build systému Copr.

Provedené změny výrazně usnadní vytvoření průběžné integrace pro případný další build systém.

Implementace může být rozšířena o další testovací Jenkins úlohy. Příkladem je dynamické testování na strojích poskytnutých službou Beaker, nebo některá z technik statické analýzy. Navázat lze na tuto práci i vizualizací výsledků testování ve webovém uživatelském rozhraní build systému Copr.

Literatura

[1] *AMQP*. [Online; navštíveno 11.05.2018].
URL https://www.cloudamqp.com/docs/amqp.html

[2] *Developer Documentation*. [Online; navštíveno 18.03.2018].
URL https://docs.pagure.org/copr.copr/developer_documentation.html

[3] *Jenkins job builder documentation page*. [Online; navštíveno 07.05.2018].
URL https://docs.openstack.org/infra/jenkins-job-builder

[4] *Open Build Service Beginner's Guide*. [Online; navštíveno 12.01.2018].
URL http://openbuildservice.org/files/manuals/obs-beginners-guide.pdf

[5] *Openstack documentation page*. [Online; navštíveno 10.05.2018].
URL https://docs.openstack.org

[6] *User Documentation*. [Online; navštíveno 18.03.2018].
URL https://docs.pagure.org/copr.copr/user_documentation.html

[7] *What is Ansible?* [Online; navštíveno 10.05.2018].
URL https://networklore.com/ansible

[8] *What is Sanity testing? Advantages, disadvantages & differences*. [Online; navštíveno 05.05.2018].
URL http://istqbexamcertification.com/what-is-sanity-testing

[9] *What is smoke testing? When to use it? Advantages and Disadvantages*. [Online; navštíveno 05.05.2018].
URL http://istqbexamcertification.com/what-is-smoke-testing-when-to-use-it-advantages-and-disadvantages-2

[10] *An Advanced Message Queuing Protocol (AMQP) Walkthrough*. 2013, [Online; navštíveno 11.05.2018].
URL https://www.digitalocean.com/community/tutorials/an-advanced-message-queuing-protocol-amqp-walkthrough

[11] Duvall, P. M.; Matyas, S.; Glover, A.: *Continuous integration: improving software quality and reducing risk*. RR Donnelley, 2007, ISBN 978-0-321-33638-5, 275 s.

[12] Fowler, M.: *Continuous Integration*. [Online; navštíveno 07.05.2018].
URL https://www.martinfowler.com/articles/continuousIntegration.html

[13] ISTQB: *Foundation Level Syllabus*. 2011.
URL http://www.istqb.org/downloads/send/2-foundation-level-documents/3-foundation-level-syllabus-2011.html

[14] Klos, T.: *Linuxové balíčkovací systémy 1*. [Online; navštíveno 21.02.2018].
URL https://www.linuxexpres.cz/praxe/linuxove-balickovaci-systemy-1

[15] McLean, M.; B, M.; Gilmore, D.; aj.: *Welcome to Koji's documentation!* 2017, [Online; navštíveno 11.01.2018].
URL https://pagure.io/docs/koji

[16] Muller, P.; Hudlicky, O.; Hutar, J.; aj.: *BeakerLib Github man page*. [Online; navštíveno 10.01.2018].
URL https://github.com/beakerlib/beakerlib/wiki/man

[17] Prchlík, M.; Vadkerti, M.: *Welcome to gluetool's documentation!* [Online; navštíveno 21.02.2018].
URL https://gluetool.readthedocs.io

[18] Smart, J. F.: *Jenkins: The Definitive Guide*. O'Reilly, 2011, ISBN 978-1-449-30535-2.

[19] Stříbný, J.: *Open Build Service migration to Fedora*. Diplomová práce, Masarykova Univerzita, Brno, 2013.