VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

VČASNÉ TESTOVÁNÍ V PROJEKTU OVIRT/RHV

EARLY TESTING IN OVIRT/RHV PROJECT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **IVANA SARANOVÁ**
AUTHOR

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

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2019/2020

Zadání bakalářské práce

Studentka: **Saranová Ivana**
Program: Informační technologie
Název: **Včasné testování v projektu oVirt/RHV**
Early Testing in oVirt/RHV Project
Kategorie: Softwarové inženýrství

Zadání:

1. Nastudujte techniku agilního programování a metodologii včasného testování (early testing).
2. Prostudujte prostředí oVirt/RHV a způsob vytváření testů v rámci tohoto prostředí.
3. Prostudujte vývojový nastroj JIRA.
4. Navrhnete způsob integrace metodologie včasného testování do aktuálního životního cyklu projektu oVirt/RHV.
5. Navržené řešení implementujte v rámci nástroje JIRA.
6. Zhodnoťte výsledky a diskutujte možná rozšíření.

Literatura:

- Dokumentace projektu oVirt: https://ovirt.org/documentation/
- Metodologie včasného testování: https://www.softwaretestinghelp.com/early-testing/
- Jira software guide: https://www.atlassian.com/software/jira/guides

Pro udělení zápočtu za první semestr je požadováno:

- Body 1-3

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

Vedoucí práce: **Rogalewicz Adam, doc. Mgr., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 31. října 2019

Abstrakt

Cílem této práce je automatizace přípravy testovacího prostředí v procesu manuálního včasného testování komponenty ovirt-web-ui projektu oVirt/RHV a umožnit tak testerovi efektivně využít svůj čas. Při řešení práce byly nastudovány agilní metodiky, princip včasného testování, projekt oVirt/RHV a nástroje Jira, Jenkins a GitHub usnadňující vývoj software. Automatizace byla implementovaná v jazyce Python a zasazena do existující automatizační struktury v GitLabu, přičemž je automaticky spouštěna pomocí nástroje Jenkins. Vytvořené komponenty pro komunikaci s jednotlivými nástroji umožňují nejen běh skriptu, který provádí přípravu prostředí, ale také budoucí získávání dat do databáze. Výsledná práce byla řádně otestována a podařilo se dosáhnout uvolnění až 2 hodin času testera na jiné pracovní aktivity.

Abstract

The objective of this work is to automate the preparation of a testing environment, which is a part of manual early testing of ovirt-web-ui component in project oVirt/RHV and allow the tester to work efficiently. Agile methods, early testing principles, project oVirt/RHV, and software development tools Jira, Jenkins, and GitHub were studied in this thesis. Automation was implemented in Python programming language and the work was incorporated into an existing automation structure in GitLab, while simultaneously being run by the Jenkins tool. All created components communicating with each software development tool are not only used in the preparation script but also will be utilized to gather data into the database in the future. The finished work was thoroughly tested and helped to free up to 2 hours of tester's time for other work-related activities.

Klíčová slova

Automatizace, agilní metodiky, včasné testování, oVirt, RHV, Python, Jira, Jenkins, GitHub, příprava testovacího prostředí

Keywords

Automation, agile methods, early testing, oVirt, RHV, Python, Jira, Jenkins, GitHub, testing environment preparation

Citace

SARANOVÁ, Ivana. *Včasné testování v projektu oVirt/RHV*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Mgr. Adam Rogalewicz, Ph.D.

Včasné testování v projektu oVirt/RHV

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením pana doc. Mgr. Adama Rogalewicze, Ph.D. Další informace mi poskytl Ing. Lukáš Svatý, technický konzultant za firmu Red Hat. Uvedla jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpala.

.....
Ivana Saranová
27. května 2020

Poděkování

Chtěla bych poděkovat svému vedoucímu bakalářské práce doc. Mgr. Adamovi Rogalewiczovi, Ph.D. za odborné vedení, věcné připomínky a vstřícnost, které mi pomohly tuto práci dokončit. Děkuji také Ing. Lukášovi Svatému za technickou kontrolu a cenné rady při vytváření praktické části bakalářské práce.

Obsah

Kapitola 1

Úvod

Testování je důležitou součástí vývoje software, bez které bychom zřejmě dnes neměli tolik zdařilých produktů, jejichž komerční úspěch souvisí především se stabilitou a dlouhodobou podporou. V čím dál rozsáhlejších projektech se však zvyšuje zátěž a nároky na kvantitu i kvalitu testování, a nakonec to jsou právě samotní testeři, kteří jsou před důležitými termíny prací úplně zahlceni. Obzvlášť problematické je testování manuální, které bývá často spojeno s monotónními a zdlouhavými aktivitami jako je například příprava testovacího prostředí. Existují však mnohé metodiky a přístupy, které mohou tuto situaci zlepšit a umožnit tak testerům efektivně využít čas i úsilí.

Konkrétním případem, kde dochází k opakované činnosti spojené s testováním, je včasné testování uživatelského rozhraní projektu oVirt a produktu RHV týmem RHV QE System & Core tools. Poměrně dost práce vyžaduje příprava prostředí, na kterém bude tester kontrolovat funkcionalitu nových vlastností a oprav chyb. Naštěstí díky postupně zaváděným agilním přístupům a využitým nástrojům pro vývoj software je možné tento proces automatizovat a snížit tak jak lidské, tak i časové nároky na testování.

Cílem práce je tak převést manuální činnost na automatickou, která vhodně doplňuje již existující automatizační struktury, a umožnit tak testerovi věnovat se jiné pracovní náplni, aniž by museli ztrácet energii a motivaci na neustále stejném, nudném procesu. Součástí toho je i vytvořit natolik kvalitní práci, aby mohla být dále rozšiřována a aby se na ni dalo v budoucnu navázat jinými projekty s podobnou náplní.

V kapitole 2.1 jsou přiblíženy agilní metodiky, které se v týmu využívají, a v kapitole 2.2 je popsán princip a výhody včasné testování. Následuje kapitola 2.3 pojednávající o samotném testovaném produktu s detailnějším zaměřením na testovanou komponentu uživatelského rozhraní. Kapitoly 2.4, 2.5 a 2.6 jsou o nástrojích, které úzce souvisí s problémem manuálního testování, což je blíže specifikováno v kapitole 3. Samotný návrh a implementace řešení automatizace najdeme v kapitolách 4 a 5. Na závěr jsou v kapitole 6 shrnuty výsledky a přínos práce včetně vyhlídek do budoucna.

Kapitola 2

Teoretická část

2.1 Agilní vývoj software

Agilní vývoj software je označení pro mnoho aplikačních rámců a praktik, jejichž cílem je rychlý a přizpůsobivý vývoj softwaru v prostředí měnících se požadavků. Důraz se klade na jednotlivce a jejich spolupráci – je důležitější, aby spolu všechny strany (programátoři, testeři, projektoví manažeři i zákazníci) pracovaly efektivně a často komunikovaly, než aby se dodržovaly stanovené postupy a nástroje. Jednotlivci by zároveň měli být dostatečně kompetentní pro řešení všech technických aspektů své práce. Obzvlášť komunikace se zákazníkem je jednou z nejdůležitějších částí, neboť formulace požadavků nemusí být jasná ani úplná a zákazník nemusí mít ucelenou představu o výsledném produktu. Úzká spolupráce s klientem tedy napomáhá k celkovému porozumění potřeb zákazníka, a s tím i spojenou včasnou reakcí na změny požadavků během vytváření softwarového systému. Pro agilní vývoj je tedy velmi důležité rychle vytvářet funkční produkt (malý důraz na dokumentaci) a testovat ho průběžně a automatizovaně [1, 2].

Motivace

Určité techniky agilního vývoje softwaru se využívaly již před jeho oficiálním zavedením v únoru 2001 na sjezdu představitelů různých programovacích metodik. Příkladem jsou extrémní programování, Crystal, pragmatické programování a další. Tyto techniky byly reakcí na těžkopádnost a byrokracii spojenou s pevně danými a neměnnými postupy těžkých, rigorózních metodik, které se hodily na problémy s předvídatelným průběhem, dlouhodobým a rozsáhlým plánováním a realizací, ale už příliš ne na dynamické a neustále se měnící prostředí webu, internetového obchodu a e-komerce. Jednotlivé týmy si tak postupně osvojovaly praktiky dnes typické pro agilní programování a různě je přizpůsobovaly konkrétním problémům, na které při vývoji narážely. Tyto vývoj zefektivňující metodiky se brzy dostaly do širšího povědomí ve sféře softwarových organizací a firem, které s nimi začaly experimentovat a vytvářet své nové postupy, což dalo vzniku dnes již známým agilním metodikám jako Scrum, extrémní programování či vývoj řízený užitnými vlastnostmi daného software [1, 6].

Agilní manifest a principy

Zásadním milníkem definování agilního vývoje softwaru je Manifest Agilního vývoje software [6] z roku 2001. Manifest měl ucelit představy o agilním vývoji, které do té doby nebyly

šířeny v konzistentní podobě, a vyzdvihnout podobnosti mezi mnoha různými metodikami. Nejde však o striktně daný postup, jak agilně vytvářet software, nýbrž se jedná o jednoduchá rozumná a flexibilní doporučení, která se pak přizpůsobí jednotlivým projektům. Výsledkem diskuze mezi představiteli různých agilních i náročných rigorózních praktik, z nichž mezi ty nejznámější patří například Kent Beck, Ward Cunningham, Martin Fowler, Dave Thomas, Jim Highsmith a Robert C. Martin, bylo 12 principů agilního vývoje softwaru [2, 6]:

1. *Naší nejvyšší prioritou je vyhovět zákazníkovi časným a průběžným dodáváním hodnotného softwaru.*
2. *Vítáme změny v požadavcích, a to i v pozdějších fázích vývoje. Agilní procesy podporují změny vedoucí ke zvýšení konkurenceschopnosti zákazníka.*
3. *Dodáváme fungující software v intervalech týdnů až měsíců, s preferencí kratší periody.*
4. *Lidé z byznysu a vývoje musí spolupracovat denně po celou dobu projektu.*
5. *Budujeme projekty kolem motivovaných jednotlivců. Vytváříme jim prostředí, podporujeme jejich potřeby a důvěřujeme, že odvedou dobrou práci.*
6. *Nejúčinnějším a nejefektnějším způsobem sdělování informací vývojovému týmu z vnějšku i uvnitř něj je osobní konverzace.*
7. *Hlavním měřítkem pokroku je fungující software.*
8. *Agilní procesy podporují udržitelný rozvoj. Sponzoři, vývojáři i uživatelé by měli být schopni udržet stálé tempo trvale.*
9. *Agilitu zvyšuje neustálá pozornost věnovaná technické výjimečnosti a dobrému designu.*
10. *Jednoduchost–umění maximalizovat množství nevykonané práce–je klíčová.*
11. *Nejlepší architektury, požadavky a návrhy vzejdou ze samo-organizujících se týmů.*
12. *Tým se pravidelně zamýšlí nad tím, jak se stát efektivnějším, a následně koriguje a přizpůsobuje své chování a zvyklosti.*

Agilní metodiky a praktiky

Agilní vývoj software je tedy spíše způsobem myšlení a přístupem k řešení problémů než přesně daný postup, co provést v různých fázích vývoje. Jak na toto aplikovat metodiky, které jsou z principu obecným pracovním postupem [13]? Metodika se považuje za soubor zvyklostí, kterou se tým rozhodne dodržovat, přičemž jiné týmy mohou použít jiné metodiky pro řešení různých problémů a upravovat si je podle svých potřeb. Existuje mnoho agilních metodik i praktik využívaných při vývoji software. Jedná se například o metodiky Crystal, Lean development, Vývoj řízený testy (Test-Driven Development, TDD), Adaptivní vývoj software (Adaptive Software Development, ASD) a další. V následujících podkapitolách si přiblížíme ty, které se při vývoji projektu oVirt a produktu RHV, byť jen částečně, využívají – jedná se o Extrémní programování, Scrum, Kanban a Vývoj řízený vlastnostmi software [1].

Extrémní programování

Extrémní programování [1, 23] (Extreme programming, XP) je agilní praktika pro vývoj software, jejíž cílem je dosáhnout rychlého a postupného dodávání vysoce kvalitních systémů a zároveň udržení kvalitního životního stylu u vývojářů. Cílovou skupinou této praktiky jsou projekty vyvíjené s dynamicky se měnícími požadavky, riskováním spojeným s používáním nových technologií, malým vývojářským týmem nebo technologiemi umožňujícími automatizované jednotkové a funkcionální testování. V praxi se obvykle používají jen některé z postupů obsažených v extrémním programování. Mezi základní hodnoty, které jsou v extrémním programování dodržovány patří komunikace, jednoduchost, zpětná vazba, odvaha a respekt.

Je velmi důležité, aby spolu programátoři, manažeři i klienti komunikovali a veškerá práce na projektu se tomuto přizpůsobuje. Obvyklé jsou diskuze tváří v tvář (***face to face***), párové programování nebo vytváření popisů, jak bude uživatel konkrétně využívat produkt (***user stories***). Zařazují se také role jako kouč, který pomáhá s technickou komunikací vývojářů, a zároveň komunikuje s vyšším managementem, a kontrolor (***tracker***), který prochází relevantní výsledky z různých metrik (např. výsledky testů, známky přepracování atd.) a určuje slabá místa v týmu, kde je třeba provést zlepšení. Respekt v týmu je důležitý pro správnou komunikaci a k jeho udržení přispívá udržování klidného prostředí, kde se mohou všichni členové týmu soustředit na svou práci, a možnost pracujících dopřát si chvíli soukromí.

Další zásadní vlastností extrémního programování je jednoduchost, jíž se docílí děláním pouze těch absolutně nejdůležitějších věcí a vyhýbáním se těm zbytečným. Zaměřuje se pouze na aspekty, které aktuálně ovlivňují vývoj a nesnaží se predikovat budoucnost. Tímto přístupem lze uspořit mnoho času na složité části produktu a jeho možná rozšíření. Rychle implementované nejdůležitější části systému se pak ihned testují a nasazují v provozu, přičemž se získává zpětná vazba. Funkcionalitu hodnotí zákazník a podle jeho názorů jsou pak prováděny změny v systému. Dalším postupem je i 10minutové sestavení (***10-minute build***), které automaticky sestaví celý systém a pustí na něm všechny testy během 10 minut. Pokud sestavení systému trvá déle než 10 minut, není možné ho často pouštět, a to přináší jistá rizika ohledně dostatečně rychlého zachytávání chyb. Zpětná vazba se dá získat i na základě týdenních nebo čtvrtletních cyklů, což jsou iterace, během kterých se tým setkává, plánují se detaily práce a reflektuje se nad jejími výsledky. S tím je spojená i základní hodnota extrémního programování, odvaha – tedy nebát se provést i zásadní změny, která mohou vést ke snížení efektivity týmu a navýšení celkového úsilí. Např. pomocí postupu uvolňování (***slack***) se v případě, že tým nestíhá plnit stanovený plán, odstraňují méně prioritní úkoly.

Scrum

Scrum [1, 14] je poměrně jednoduchá agilní praktika, která umožňuje efektivní spolupráci týmů při vývoji komplexních produktů a umožňuje se přizpůsobit problémům v jednotlivých fázích vývoje, přičemž je práce produktivní a produkty jsou dodávány v nejvyšší možné kvalitě. Tato praktika je sice jednoduchá na pochopení, ale je obtížné ji zvládat na vysoké úrovni. Implementuje empirickou metodu tím, že se nejprve stanoví hypotéza, která se vyzkouší a tým tyto nově nabyté zkušenosti použije k provedení potřebných změn.

Prvním procesem je plánovaní, kdy je stanoven počáteční seznam požadavků (***product backlog***). Jednotlivé požadavky jsou seřazeny např. podle rizik, odhadů časových i lidských zdrojů, přičemž jednotlivé položky odpovídají možným změnám, které mohou být

na produktu provedeny. Není tedy zaručeno, že budou všechny položky z tohoto seznamu implementovány.

Principem je iterativní vývoj, kdy se jednotlivé iterace nazývají sprinty a jejich délka je ideálně od 2 do 4 týdnů. Na začátku iterace se nejprve naplánuje práce, jejíž součástí je výběr položek z seznamu požadavků (***sprint backlog***), na jehož základě se naplánují jednotlivé úkoly pro tým. Vývoj pak spočívá ve splnění těchto úkolů tak, aby byl splněn cíl iterace (***sprint goal***). Pro zajištění efektivní spolupráce a komunikace mezi členy týmu i celými týmy se v průběhu vývoje v jednotlivých sprintech pořádají každodenní 15minutová setkání týmu. Účastní se jí členové týmu či zástupci z jednotlivých týmů, management a Scrum vedoucí (***Scrum master***), což je osoba, která by měla při setkání zajistit dodržování agilních principů a procesů a praktik, na kterých se tým dohodl. Na konci iterace se posuzuje výsledek, kterým je funkční produkt nebo jeho část nazývaný inkrement. Tato část se nazývá posouzení sprintu (***sprint review***) a v jejím rámci také dochází k provedení změn v seznamu požadavků (označení splněných požadavků, přidávání nových, úpravy starých atd.). Během retrospektivy sprintu (***sprint retrospective***) se pak reflektuje nad prací týmu a určují se činnosti, které by bylo třeba zlepšit. Takto identifikovaná vylepšení se pak obvykle přidávají do následujícího sprintu jako položka v seznamu požadavků. Ihned po skončení sprintu začíná další, přičemž počáteční i koncová data jednotlivých sprintů jsou fixní.

Tyto iterace se opakují, dokud výsledný produkt nesplňuje všechna předem dohodnutá kritéria. Výsledky jednotlivých sprintů, inkrementy, se integrují a celý systém je otestován. Připraví se také dokumentace, zaškolují se uživatelé a provádí se akceptační testování.

Kanban

Kanban [1] metodika umožňuje návrh, správu a zlepšení provozu systému s již existujícími pracovními postupy pomocí postupně zaváděných změn. Nejčastěji se využívají různé metody vizualizace stavu práce pomocí interaktivních tabulí (***kanban board***), kde jsou jednotlivé úkoly rozdělovány podle toho, zda se na nich teprve bude pracovat, zda se na nich už pracuje a jestli jsou už hotové. Nicméně se v praxi tyto toky upravují podle potřeb týmu a přidávají se nové kategorie. Tato metodika je obzvlášť výhodná v nepředvídatelných dynamických prostředích s nutností rychlé produkce výsledků.

Kanban definuje několik hodnot a principů, kterými by se mělo řídit. Podobně jako u jiných agilních metodik a praktik se zaměřuje na komunikaci jak mezi manažery a vývojáři, tak i na domluvu se zákazníkem. Prioritou je sdílet informace otevřeně a přímočaře, respektovat názory a postoje ostatních, dojít ke společnému rozhodnutí a domluvě a aktivně zapojovat zákazníkovy požadavky do pracovního toku. Dále se využívá vedení lidí, spolupráce mezi týmy, vyvažování mezi požadavky úkolů a schopnostmi týmu tak, aby byl proces co nejvíce efektivní a rozdělení samotné práce do epizodických částí. Kanban především ale řeší lidskou neochotu něco měnit. Metodika k tomuto přistupuje poměrně šetrně pomocí postupných změn, na kterých je předem dohodnuto, a podporou aktivního vedení v každé fázi vývoje. Po každé zavedené změně se navíc sbírá zpětná vazba, která může napomoci zlepšení celkového procesu.

Na rozdíl od jiných agilních metodik není Kanban rozdělený do vývojových iterací, ale má nepřetržitý tok práce. Svým způsobem lze náznaky iterativních charakteristik nalézt v opakovaném zhodnocení vývoje pomocí čtvrtletních, měsíčních, týdenních i denních zpětných vazeb, přičemž každá se zaměřuje na jinou oblast od strategických, kde se řeší vhodnost poskytovaných služeb, po běžné malé každodenní týmové schůzky (***daily standup***), kdy si jednotliví členové sdělují, na čem zrovna pracovali a co mají v plánu dál.

Feature Driven Development

Vývoj řízený užitnými vlastnostmi daného software [3, 15] (Feature Driven Development, FDD) je iterativní inkrementální agilní metodika, která se zaměřuje na klienta, architekturu a pragmatický vývoj software. FDD umožňuje postupné aktualizace projektu a rychlou identifikaci chyb, navíc mohou být tyto výsledky poskytovány klientům a může se tím předejít nedorozumění a nutnosti přepracování systému.

V této metodice jsou významným aspektem vlastnosti (***features***), kde je vlastnost definována jako malá, klientská funkce vyjádřená ve formě akce, výsledek a objekt. Příkladem může být validace uživatelského hesla, autorizace transakcí apod. Vlastnosti se získávají z požadavků klienta a následně se používají při plánování. Jednotlivé iterace, na jejichž konci jsou výsledkem funkční vlastnosti, mají na rozdíl od praktiky Scrum pouze 2-10 dní. Dalším větším rozdílem je důraz na dokumentaci, v které jsou definovány všechny důležité vlastnosti, které mají být splněny, a podle které se týmy při vývoji orientují a nemusí se proto tak často scházet.

Počátečním procesem je sbíráním dat. Snahou je určit cílovou skupinu, její potřeby a příčiny těchto potřeb. Na základě takto získaných informací se vytvoří celkový model, jakýsi nástin celého systému, který bude postupně doplňován během vývoje, a seznam vlastností (***features list***). Následně se plánuje podle vlastností, kdy se provádí analýzy složitosti jednotlivých vlastností a vytváří se úkoly pro jednotlivé členy týmu. Vlastnosti, které by trvalo vyvíjet déle, než na jakou dobu je naplánována iterace, se rozbijí na více částí, případně se rozplánují na více iterací. Samotný vývojář obvykle pracuje na sadě vlastností (***feature set***), která patří do určité třídy. Správce třídy, kterého určuje hlavní programátor (***chief programmer***), kontroluje práci všech vývojářů přidělených k jeho třídě a zároveň umožňuje mezi-třídní spolupráci. Hlavní programátor také určuje, která vlastnost se bude implementovat, které týmy se na ní budou podílet apod. Na konci vývoje je implementace zhodnocena celým týmem a vytvoří se prototyp vlastnosti k otestování. Pokud je vlastnost schválena, zařadí se do hlavní větve produkce.

2.2 Včasné testování

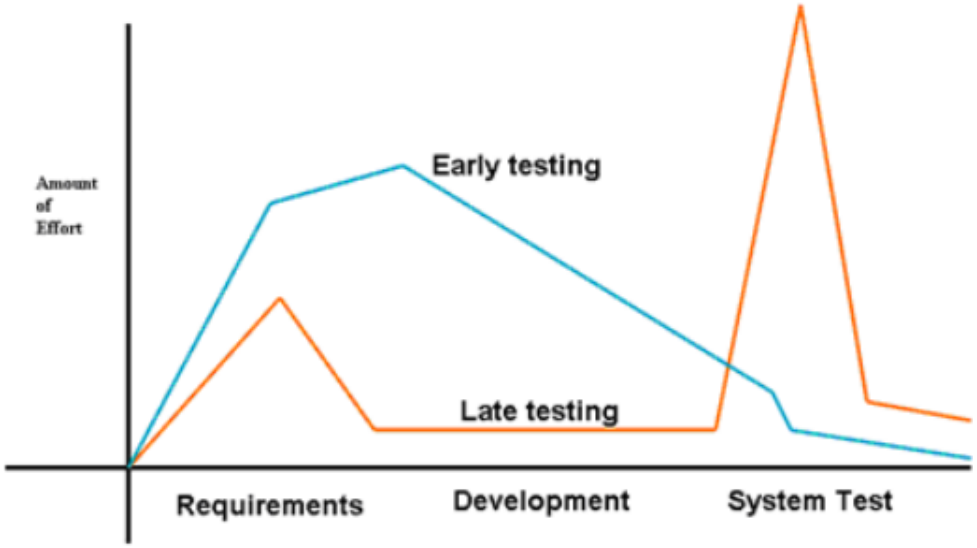
Běžným postupem při vývoji software je úzká spolupráce marketingového oddělení, vývojářů a testerů pod taktovkou produktového manažera. Samotný vývoj se pak řídí určitým časovým rozvrhem, který obvykle umožňuje dodání částí nebo celého vytvářeného produktu QA[1] týmu předtím, než dojde k jeho vydání zákazníkům. Tento způsob je velmi efektivní, co se týče podchycení velkých a kritických problémů, nicméně je velmi časově náročný a v případě nakupení většího množství chyb může vést v lepším případě k oddálení vydání produktu. Další nevýhodou je i fakt, že je velká pozornost směřována především na pro zákazníky nejvýznamnější vyvíjené vlastnosti a na střední či malé problémy není možné uvolnit žádné zdroje, ať už lidské nebo výpočetní. Výsledkem může tedy být produkt splňující žádané vlastnosti, ale s větší mírou menších chyb, které jsou pak nahlašovány uživateli.

Za přetížením testerů obvykle vězí nejasné požadavky, nedostatek finančních prostředků nebo času na testování, nekompletní testy, opakované manuální testování, časté změny v požadavcích, nedostatek motivace a také fakt, že lidé zkrátka dělají chyby. Je tedy vyvíjena snaha předejít nechtěné vysoké zátěži na QA tým po předání produktu k testování. Vytváří

[1] Zajištění jakosti (Quality assurance), způsob předcházení chyb a problémů při poskytování produktu nebo služeb zákazníkům. Obvykle se tak označuje oddělení testerů.

se automatizované testy pro kontrolu základní funkčnosti produktu, tedy je tester nemusí provádět manuálně a stačí pouze kontrolovat výsledek běhu testování. Pořádají se v QA týmech tzv. verifikační dny (***verification days***), kdy se po dobu několika dní jednotliví testeři soustředí pouze na verifikaci již existujících problémů v aktuálně testovaném produktu a ostatní práci (automatizace, porady s vývojáři, příprava test plánů atd.) mohou po tuto dobu ignorovat. K důležitým nově implementovaným vlastnostem jsou ještě před samotným předáním produktu k testování vytvářeny předběžné test plány testerem, které následně vývojář dané vlastnosti zkontroluje a schválí nebo vrátí k předělání.

Technikou, která se v tomto kontextu aplikuje, je včasné testování [22, 16] (***Early Testing***). Jedná se o předběžné testování vyvíjených schválených částí produktu, které jsou dobře zdokumentované a existují dostatečné zdroje a prostředky, které toto testování umožňují. Tento způsob testování, jak již jeho název napovídá, se odehrává ještě před samotným hlavním testování, a tedy před předáním produktu QA týmu. Tester tedy obvykle pracuje se změnami, které ještě nebyly oficiálně zasazeny do hlavního projektu. Velikost takto testovaných aspektů produktu se může různit, a s tím i míra úsilí, která bude do testování vložena. Může se tak stát, že včasné testování bude zahrnovat pouze rychlé a snadné manuální verifikace přijímaných změn, zatímco v jiném případě může jít o psaní rozsáhlých a podrobných test plánů. Zásadní je komunikace QA týmu s vývojáři po celou dobu včasného testování a nastavení vhodného procesu, podle kterého se mohou oba týmy řídit.



Obrázek 2.1: Porovnání testovacího úsilí, John D. McGregor, Clemson University and Luminary Software LLC, U.S.A., převzato z článku [16]

Výhodami včasného testování je mimo jiné snížení zátěže testerů v době testování již téměř hotového produktu (viz Obrázek 2.1). Dále pomáhá dříve odhalit chyby v produktu i napříč komponentami a umožnit tak vývojářům projekt rychleji stabilizovat. V případě velkých podstatných objevených problémů je také stále čas upravit celý plán vývoje produktu, na příklad zvýšeným nasazením testerů a vývojářů v problematické oblasti. Nevýhodou je fakt, že se nejedná o agilní metodu a jako taková se poměrně obtížně do agilního procesu zasazuje [21].

2.3 Projekt oVirt a produkt RHV

Red Hat Virtualization

Red Hat Virtualization (RHV)[2] je virtualizační platforma využívající Red Hat Enterprise Linux (RHEL)[3], díky které může uživatel spravovat celou virtuální infrastrukturu. Jedná se o komerční produkt cílený na podniky. Snadným způsobem nabízí uživatelům možnost poskytovat virtuální servery a pracovní stanice (*workstations*) a efektivně využívat zdroje pro fyzické servery. Mezi klíčové komponenty produktu RHV patří následující [19]:

- **Red Hat Virtualization Manager (RHVM)**: Služba poskytující grafické rozhrání a REST API pro uživatelský přívětivou správu virtuálního prostředí.
- **Hostitelé**: Red Hat Enterprise Linux hosts (RHEL hosts) a Red Hat Virtualization Hosts (obrazoví hypervizoři) jsou podporované typy hostitelů, které používají Kernel-based Virtual Machine[4] (KVM, virtuální stroj založený na jádře) technologie umožňující plnou virtualizaci. Jsou poskytovatelem zdrojů pro běh virtuálních strojů.
- **Red Hat Virtualization Host (RHVH)**: Komerční minimální verze operačního systému RHEL modifikovaného speciálně pro snadnou správu, údržbu a nasazení.
- **Red Hat Enterprise Linux host**: RHEL s potřebnými subskripcemi. Na rozdíl od RHVH je mnohem více přizpůsobitelný.
- **Shared Storage**: Úložiště dat o virtuálních strojích. Podporovanými typy úložných domén jsou datové úložné domény, ISO domény a exportní domény (viz Sekce 2.3, bod Storage Domain).
- **Data Warehouse**: Služba sbírající konfigurační a statistická data z RHVM, která následně zpracovává v databázi. Mezi získávané informace patří data o hostech, virtuálních strojích a úložištích, přičemž se databáze dělí na primární databázi Manažera (konfigurace, stav, výkon) a Data Warehouse databázi (porovnaná dlouhodobě sbíraná data z databáze Manažera).

oVirt

Projekt oVirt[5] je upstream[6] produktu RHV, tedy otestovaný a funkční kód z oVirtu je přímo použit v produktu RHV. Jedná se o otevřený projekt vyvíjený komunitou a má svůj veřejný repositář na GitHubu[7]. Na rozdíl od produktu RHV není sám o sobě cílen na podniky a neposkytuje dlouhodobou podporu a stabilitu. Dalším podstatným rozdílem je absence plnohodnotného testování a zákaznické podpory. Projekt oVirt se tedy hodí spíše pro běžné uživatele a malé projekty, které tyto vlastnosti nevyžadují.

Vzhledem k tomu, že jsou implementace projektu oVirt i produktu RHV totožné, jsou i klíčové komponenty a architektury stejné nebo velmi podobné a ve většině případů se

[2] https://access.redhat.com/products/red-hat-virtualization

[3] https://access.redhat.com/products/red-hat-enterprise-linux/

[4] KVM je technologie virtualizace navržená pro Linux, která umožňuje použít linuxové jádro jako hypervizora. Na hypervizorovi pak lze spustit několik na sobě nezávislých virtuálních strojů. Více na: https://www.redhat.com/en/topics/virtualization/what-is-KVM

[5] https://www.ovirt.org/

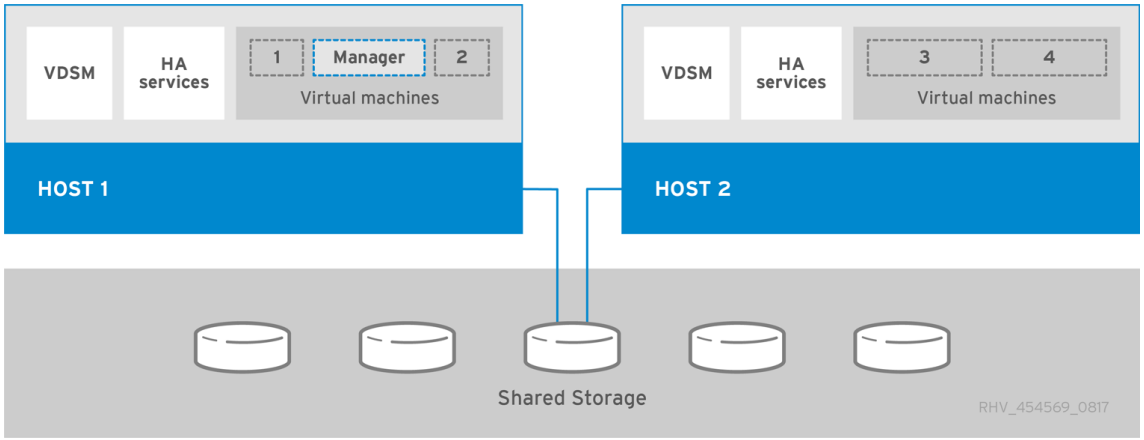
[6] Hlavní vývojová větev produktu

[7] https://github.com/oVirt/

liší pouze v pojmenování. Tedy například místo RHVM se používá název **oVirt engine** a typy hostitelů jsou **Enterprise Linux hosts** a **oVirt Nodes**. Také se místo komerčního operačního systému RHEL používá volně dostupný **CentOS (Community Enterprise Operating Systém)**[8], který je upstreamem k RHELu stejně jako oVirt k RHV [17].

Základní architektury produktu RHV a projektu oVirt

Produkt RHV a projekt oVirt mohou být nasazeny jako **self-hosted engine** (RHVM/oVirt engine běží jako virtuální stroj na specializovaných hostitelích ze stejného spravovaného prostředí) nebo samostatný RHVM/oVirt engine.



Obrázek 2.2: RHV architektura self-hosted engine, Red Hat, Inc., převzato z manuálu [19]

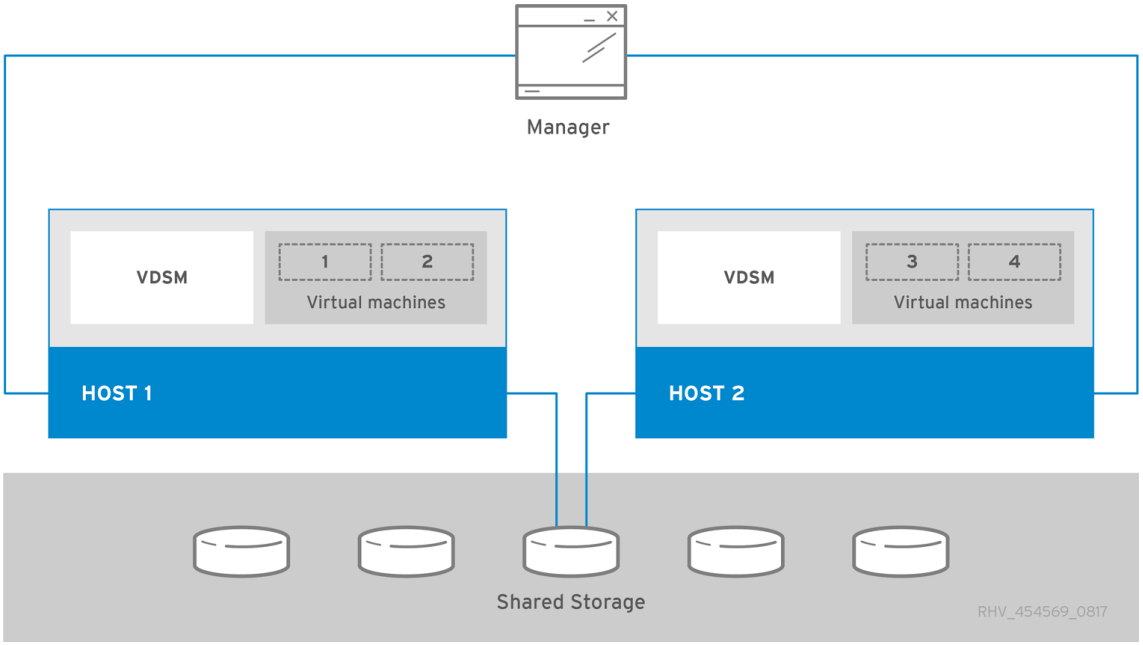
V případě doporučovaného self-hosted engine (viz Obrázek 2.2) je sice potřeba o jeden fyzický server méně, nicméně narůstá administrativa spojená s údržbou a nasazením. Ke spuštění self-hosted engine je tedy třeba mít virtuální stroj s nainstalovaným RHEL/CentOS a v něm minimálně dva specializované hostitele, kteří budou řídit veškerou komunikaci s Managerem. Další nutností je úložiště přístupné všem hostitelům. Tento způsob přináší mimo jiné vysokou dostupnost RHVM/oVirt engine.

Architektura samostatného RHVM/oVirt engine (viz Obrázek 2.3) spočívá v tom, že Manager běží na fyzickém serveru nebo virtuálním stroji z jiného virtuálního prostředí. Pro obě varianty platí, že musí mít nainstalovaný RHEL/CentOS. Na rozdíl od self-hosted engine je jeho administrativa snazší, byť má vyšší nároky na zdroje (fyzický server). Co se týče úložišť, není oproti self-hosted engine žádný rozdíl. Vysokou dostupnost lze zařídit buď dalším produktem (Red Hat High Availability Add-On[9]) nebo větším počtem hostitelů, přičemž minimální počet hostitelů pro dobrou dostupnost engine jsou dva [19, 17].

Důležité pojmy v projektu oVirt a produktu RHV

- **Cluster**: Cluster označuje skupinu hostitelů, kteří sdílí stejnou síť a úložiště a slouží společně jako zdroj pro virtuální stroje a pro migrace virtuálních strojů mezi hostiteli.
- **Data Center**: Datová centra jsou nejvyšší úrovní infrastruktury všech logických i fyzických zdrojů virtuálního prostředí a obvykle označují kolekce clusterů, virtuálních strojů, úložišť a sítí.

[8] https://www.centos.org/
[9] https://www.redhat.com/en/store/high-availability-add



Obrázek 2.3: RHV architektura samostatný engine, Red Hat, Inc., převzato z manuálu [19]

- **Events**: Události informují administrátora o různých aktivitách (varování, upozornění, výpisy) ve virtuálním prostředí a umožňují snadnější monitorování výkonu a stavu zdrojů.
- **HA services (High Availability services)**: Služby spravující vysokou dostupnost Manažera, konkrétně ovirt-ha-agent a ovirt-ha-broker, běží na virtuálních uzlech self-hosted engine.
- **High Availability**: Na rozdíl od taktiky tolerování chyb (*fault tolerance*), kdy se spravují dvě kopie každého zdroje a mohou být vzájemně vyměněny v případě poruchy, umožňuje vysoká dostupnost zachovat stabilitu bez nutnosti zvyšování velikosti virtuálního prostředí jednoduchým restartováním virtuálního stroje v případě, že je jeho proces přerušen na kterémkoliv z hostitelů v clusteru. Nevýhodou může být fakt, že takové restartování způsobí krátkodobou nedostupnost virtuálního stroje.
- **Hostitel**: Hostitel nebo hypervizor je fyzický server umožňující běh virtuálních strojů, přičemž mohou být takové virtuální stroje migrovat mezi hostiteli ve stejném clusteru.
- **Storage Pool Manager (SPM)**: Manažer fondu úložišť je datovým centrem nastavitelná role hostiteli, kdy pak takový hostitel (označovaný jako SPM hostitel) má výhradní autoritu při provádění všech meta-datových změn v datovém centru, např. při vytváření a mazání virtuálních disků.
- **Host Storage Manager (HSM)**: SPM hostitelé mohou být potenciálně zahlcení prováděním dlouhých metadatových operací. Z toho důvodu jsou vyčlenění hostitelé, kteří nejsou SPM, a používají se například pro přesun disku mezi úložnými doménami.
- **Logical Network**: Logická síť je logickou reprezentací sítě fyzické seskupující síťový provoz a veškerou komunikaci mezi Manažerem, hostiteli, úložištěm a virtuálními stroji.

- **Remote Viewer**: Grafické rozhrání pro připojení k virtuálnímu stroji přes síťové připojení.
- **Self-Hosted Engine Node**: Jedná se o hostitele s nainstalovanými balíčky, které mu umožňují hostovat Manažera. Jiní hostitelé mohou být do virtuálního prostředí také připojení, ale nemůže na nich běžet Manažer.
- **Snapshot**: Zachycený stav je náhled obsahující operační systém virtuálního stroje a všechny aplikace v něm zachycený v určitém čase. Může sloužit k uložení původního nastavení před prováděním změn a v případě jejich neúspěchu lze zachycený stav na virtuálním stroji obnovit.
- **Storage Domain**: Úložná doména slouží k ukládání virtuálních disků, ISO obrazů (ISO doména), pro import a export obrazů virtuálních strojů (exportní doména), přičemž se jedná o logický celek se samostatným obrazovým repositářem.
- **Template**: Šablona je snadným a rychlým způsobem, jak vytvořit velké množství virtuálních strojů s předdefinovaným nastavením. Jde o model virtuálního stroje, který obsahuje všechna konfigurační nastavení (paměťové nároky, síť, disky, a další).
- **VDSM**: Agentní služba běžící na hostitelovi komunikující s Manažerem. Poslouchá na TCP portu 54321.
- **Virtual Machine**: Virtuální stroj je virtuální pracovní stanice nebo virtuální server s operačním systémem a aplikacemi, který může být vytvářen, spravován a mazán power uživateli (uživatelé s přístupem do administračního portálu) a ke kterému mohou přistupovat běžní uživatelé. Skupina identických virtuálních strojů může být vytvořena ve fondu.
- **Virtual Machine Pool**: Fond označuje skupinu identických virtuálních strojů vytvořených za určitým účelem (výzkum, marketing, vývoj atd.) a přístupných podle potřeby skupinám uživatelů.

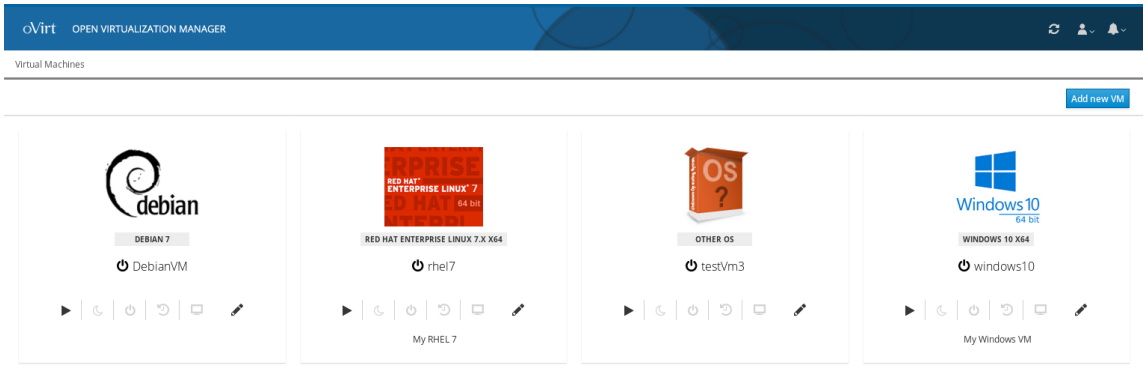
[19]

Portál virtuálních strojů

Portál virtuálních strojů [19, 17] (VM portál) nabízí snadnou a uživatelsky přívětivou správu virtuálních strojů a fondů. Na rozdíl od Správního portálu do něho mají přístup i uživatelé bez administračních práv, ale to, k čemu mohou přistupovat a co mohou upravovat, je ovlivněno sadou dalších administrátory nastavitelných práv a pravidel. Mezi základní funkcionalitu patří spouštění, vypínání, editování a prohlížení detailů virtuálního stroje včetně přímého přístupu k virtuálnímu stroji přes VNC[10] nebo SPICE[11] klienty pomocí konzole. S odpovídajícími právy to pak může být i vytváření a mazání virtuálních strojů, správa disků a síťových rozhraní a správa snapshotů.

[10]VNC je zkratkou pro virtuální síťovou výpočetní techniku (virtual network computing), což je systém sdílející plochu a umožňující tak vzdáleně ovládat takový počítač. Více na http://www.remoteaccess.org/what-is-a-vnc/

[11]SPICE je jednoduchý protokol pro nezávislá vypočetní prostředí (Simple Protocol for Independent Computing Environments). Jedná se o otevřený projekt nabízející vzdálený přístup k virtuálním strojům. Více na https://www.spice-space.org/

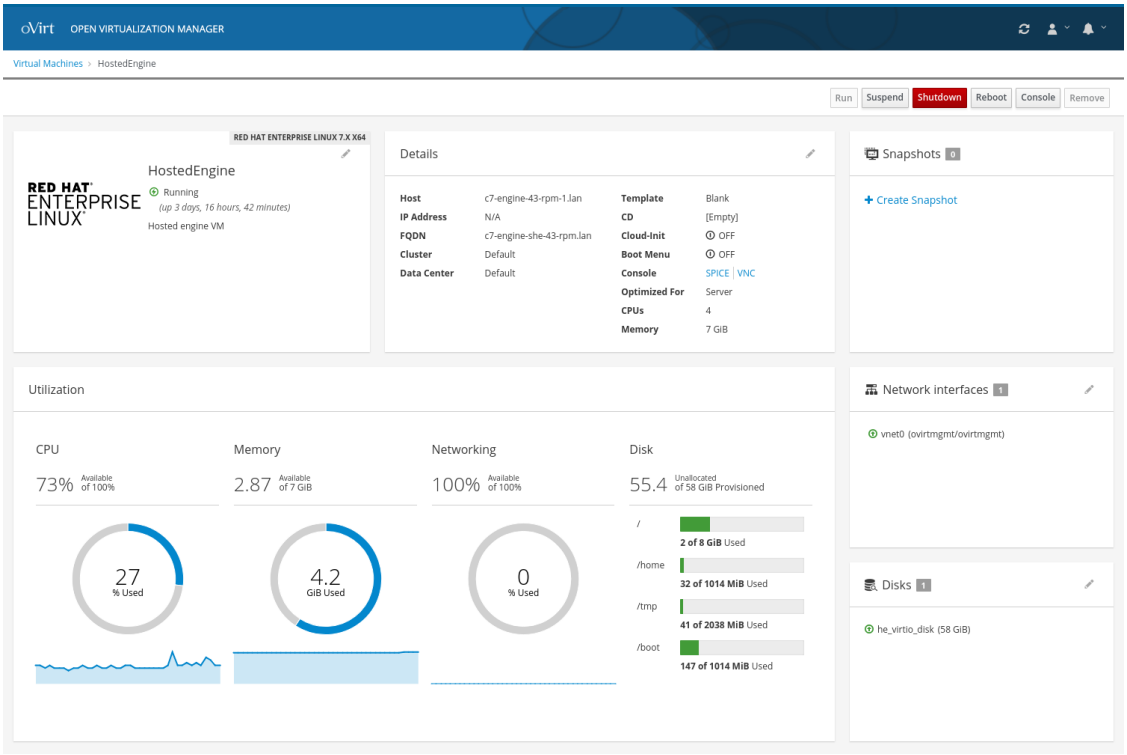


Obrázek 2.4: Portál virtuálních strojů – hlavní náhled, oVirt, převzato z dokumentace [20]

Portál virtuálních strojů se skládá, kromě standardních přihlašovacích dialogů, z hlavního náhledu (viz Obrázek 2.4), ve kterém jsou přehledně pomocí kartiček zobrazeny jednotlivé virtuální stroje a fondy. Karta virtuálního stroje obsahuje jeho název, ikonu, operační systém a aktuální stav. Virtuální stroj lze již zde poměrně pohodlně ovládat pomocí malého menu. Karta fondů je barevně i obsahově odlišná. Kromě společného názvu, ikony a operačního systému vypisuje také kolik může uživatel z daného fondu alokovat virtuálních strojů, kolik jich již má a místo ovládacího menu je pouze tlačítko, kterým se alokují virtuální stroje pro daného uživatele. V záhlaví se také může nacházet tlačítko pro vytvoření nového virtuálního stroje, které uživatele přesune na speciální dialog. Z karty virtuálního stroje se dá přejít na jeho detail, z karty fondů nikoli.

Detail virtuálního stroje (viz Obrázek 2.5) pak nabízí více možností: od správy disků a síťových zařízení, po prohlížení využití zdrojů, editaci nastavení virtuálního stroje a správu snapshotů. Nicméně valná většina těchto rozšiřujících funkcí je pro uživatele bez patřičných práv nedostupná nebo neměnitelná. Standardně ale uživatel může sledovat, jakého hostitele má virtuální stroj přiřazeného, jakou má IP adresu a FQDN[12], do jakého clusteru a datového centra patří, z jaké šablony byl vytvořen atd. Dále se zde nachází kolonka s grafy o využití CPU, paměti, sítě a disků. Správa snapshotů umožňuje uživateli snapshoty vytvářet, mazat a vracet virtuální stroj do zachyceného stavu. Správa síťových rozhraní a disků obdobně zahrnuje vytváření, editaci a mazání příslušných entit. Hlavní funkcionalita a vlastnosti portálu virtuálních strojů se nachází v balíčku s názvem **ovirt-web-ui**. Mezi podporované prohlížeče (na kterých se testuje a vyvíjí) patří nejnovější verze Mozilla Firefox[13], Google Chrome[14] a Microsoft Edge[15]. Veškerý kód je přístupný na GitHubu[16] a veřejnost do něj může přispívat i pomáhat v hledání chyb.

[12] Plně kvalifikované doménové jméno.
[13] https://www.mozilla.org/
[14] https://www.google.com/intl/cs_CZ/chrome/
[15] https://www.microsoft.com/cs-cz/edge
[16] https://github.com/oVirt/ovirt-web-ui



Obrázek 2.5: Portál virtuálních strojů – detail virtuálního stroje, oVirt, převzato z dokumentace [20]

2.4 Jira

Jira[17] [4] je skupina proprietárních produktů pro přehlednou a snadnou správu práce různých odvětí od software týmů, zákaznického centra po finanční oddělení. Jira je vyvíjena společností Atlassian, Inc.[18] v jazyce Java[19] na platformě Java Virtual Machine[20]. Nástroj umožňuje efektivní zapojení agilních technik a metodik ve vývoji a zároveň i propracovanou zákaznickou podporu. Jednotlivé týmy mohou sledovat přidělené úkoly, plánovat a kontrolovat práci a zároveň kolaborovat s jinými odděleními. Mezi stěžejní vlastnosti Jira produktů patří vysoká přizpůsobivost nejrůznějším pracovním prostředím a procesům, propojením s jinými nástroji jako je GitHub nebo Confluence a velmi rychlé nastavení a příprava nástroje k okamžitému využití týmu.

Jira nabízí konkrétně tři produkty: Jira Software, Jira Service Desk a Jira Core. Každý z těchto produktů se zaměřuje na jiná uživatelská využití a nabízí jim přizpůsobené šablony pro jednotnou mezi-týmovou spolupráci. Všechny produkty mohou běžet na cloudu nebo lokálně (spravované samostatně klientem) na serveru či datovém centru. Cloud je doporučován jako nejlepší možnost v případě, že chce zákazník začít s produktem hned pracovat, protože je Jira server nachystaný i hostovaný přímo společností Atlassian. Na druhou stranu, pokud je potřeba jít při nastavení až do detailů a mít důkladnější zabezpečení dat, je lokální

[17] https://www.atlassian.com/cs/software/jira
[18] https://www.atlassian.com/
[19] https://www.java.com/en/
[20] https://docs.oracle.com/javase/9/vm/java-virtual-machine-technology-overview.htm

hostování na vlastním hardware vhodným řešením. Jira produkty jsou navíc kompatibilní s datovými centry v rámci IaaS[21] produktů AWS[22] a Azure[23]. [4]

Jira Software

Jira Software se, jak již název napovídá, specializuje na týmy vyvíjející software a mohou ji tak využít nejen samotní vývojáři, ale také projektoví manažeři a SCRUM vedoucí. Umožňuje svým uživatelům sledovat bugy[24], spravovat úkoly, vývoj, procesy i samotný produkt nebo projekt. V poslední době je tento produkt vyhledáván zejména v souvislosti s přechodem ke agilnímu vývoji software, neboť je mimo jiné možno propojovat Jira úkoly s GitHub či Bitbucket repositáři nebo vytvářet dokumentace procesů a pracovních postupů v Confluence. Nicméně ani to není zdaleka vše, protože se v Jira Portfoliu nachází stovky dalších aplikací s rozšiřující funkcionalitou pro dokonalé přizpůsobení produktu týmu na míru. [4]

Jira Service Desk

Jira Service Desk slouží jako zákaznická linka, kde mohou zákazníci nahlašovat problémy a aktivně je řešit s IT podporou pomocí tiket procesů. Dále se zde dají spravovat nejrůznější služby spojené s podporou, řešením incidentů a změn a komunikací se zákazníky. Uživatelé si mohou prohlížet projekty, vyvíjené a nabízené vlastnosti. [4]

Jira Core

Jira Core je orientována především na obchodní oddělení a tedy týmy, které nejsou technicky zaměřené. Umožňuje správu pracovních procesů a úkolů na vyšší úrovni podnikání, přičemž mají uživatelé přístup ke statistikám, grafům, hlášením atd. [4]

Důležité pojmy

- **Jira issue**: Jedná se o jednotlivou pracovní položku, úkol, různé velikosti, jehož cílem je jeho dokončení. Může se jednat o poměrně velkou nově vyvíjenou vlastnost produktu i běžné to-do.
- **Jira projekt**: Projekt v Jira prostředí označuje skupinu Jira issue se společným původem nebo účelem, např. podle týmu, komponenty, štítků apod.

2.5 Jenkins

Jenkins[25] [11] je komunitou vyvíjený otevřený software vydávaný pod licencí MIT[26] pro automatizaci procesů spojených se sestavením, nasazením a automatickým spouštěním projektů a skriptů pracujícími nad projekty. Jenkins je hodně rozšířený produkt hojně vy-

[21] Iaas, infrastruktura jako služba je počítačová infrastruktura, která je poskytována a spravována po celé internetu. Je jedním z typů cloudových služeb společně se softwarem jako službou (SaaS), platformou jako službou (PaaS) a cloud bez serveru. Více na https://azure.microsoft.com/cs-cz/overview/what-is-iaas/

[22] https://aws.amazon.com/

[23] https://azure.microsoft.com/cs-cz/

[24] Bug je anglické označení pro softwarovou chybu.

[25] https://www.jenkins.io/

[26] https://opensource.org/licenses/MIT

užívaný k automatizaci jak testů v rámci průběžné integrace (continuous integration[27]) u vývojářů, tak i kompletního testování celého produktu v komerční i nekomerční sféře.

Mezi přední nabízené vlastnosti patří podpora průběžné integrace a průběžného dodávání (continuous delivery[28]), snadná instalace pomocí multiplatformních balíčků a správa pomocí webového klienta. Zátěž se dá navíc rozložit na několik strojů, což umožňuje spouštění časově náročných velkoplošných testů. Podobně jako u Jira produktů lze i v projektu Jenkins nastavit nejrůznější detaily a přidávat rozšíření tak, aby byl výsledek co nejblíže potřebám zákazníka. Další velkou výhodou tohoto produktu je i podpora systémů pro správu verzí, přičemž na to může být navázáno i spouštění některých konkrétních instrukcí. [11]

Důležité pojmy

- **Artifact**: Artefakt je soubor vygenerovaný během jednoho z běhů sestavujícím projekt. Soubor zůstává přístupný uživatelům i po skončení sestavování nebo jiné sérii instrukcí.
- **Build**: Výsledek jednoho spuštění projektu.
- **Job**: Úkol, synonymum pro označení projekt, je označení pro uživatelsky definovanou práci prováděnou produktem Jenkins, např. sestavení projektu.

2.6 GitHub

GitHub[29] [8] je bezpochyby jedna z nejoblíbenějších a největších bezplatných platforem pro ukládání a sdílení projektů vytvořených pomocí systému pro správu verzí, Git[30]. Aktuálním majitelem americké společnosti GitHub, Inc., která GitHub vlastní, je Microsoft[31], který cílí především na rozvoj komunit vyvíjejících otevřený software. Mimo uživatelsky vlídné webové rozhraní nabízejícím intuitivní práci se všemi běžnými funkcionalitami systému Git je možné vlastní účet i projekty na GitHubu obohatit o další podpůrná rozšíření, např. vizuální synchronizaci s automatizacemi v Jenkins projektu, projektové dokumentační stránky atd. Zároveň je však možné s GitHubem manipulovat pomocí systému Git klasicky z příkazové řádky.

Uživatelé mohou na GitHubu vytvářet neomezené množství veřejných i privátních **repositářů**, což je místo, kam se ukládají soubory konkrétního projektu. Zatímco obsah veřejného repositáře si mohou prohlížet jak registrovaní, tak i neregistrovaní uživatelé, do privátního repositáře vidí pouze přizvaní registrovaní uživatelé. Kromě úkolů (**issue**), které obvykle obsahují podněty k provedení určitých změn nebo slouží k hlášení problémů a chyb v projektu, lze pohodlně sledovat historii provedených změn a verzí (jednotlivé **commity**) a různé statistiky o přispěvatelích, frekvenci aktivit v repositáři apod. Každé GitHub is-

[27] Continuous integration, také zkracována na CI, je označení pro automatizaci zavádění změn kódu z různých zdrojů do společného software projektu. K docílení tohoto procesu jsou využity automatizační nástroje a prováděny kontroly kódu, včetně ukládání verzí v rámci systému pro správu verzí. Více na https://www.atlassian.com/continuous-delivery/continuous-integration

[28] Continuous delivery je označení pro schopnost zavedení různých změn (nové vlastnosti, konfigurace, opravy, experimenty) do produkčního prostředí nebo přímo k uživatelům bezpečným, rychlým a udržitelným způsobem. Více na https://continuousdelivery.com/

[29] https://github.com/

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

[31] https://www.microsoft.com/cs-cz

sue může mít popisky, mezníky, štítky, reference na změny v kódu a další vlastnosti, které usnadňují vývoj software.

Nad repositáři mohou uživatelé, kteří jej nevlastní, provést větvení (**fork**) a projekt v rámci svého účtu dál upravovat, aniž by zasáhli do původního kódu. Tento přístup napomáhá udržovat a rozvíjet rozmanitost projektů, jelikož mohou vznikat jak úplně nové speciální verze původních projektů, tak i jen lehce pozměněné. Změny lze do původního projektu zavést pomocí žádosti majitelům a správcům repositáře o zavedení změn (**pull request**) . Rozhraní GitHubu umožňuje žádajícím i žádaným efektivně komunikovat, provádět a prohlížet dodatečné změny v rámci toho stejného pull requestu, přiložený kód komentovat, a nakonec i provést přijetí a zavedení změn do původního kódu (**merge**).

Kapitola 3

Aktuální situace v týmu RHV QE System & Core tools

3.1 Využití agilního přístupu

Stejně jako v mnohých jiných organizacích, tak i v týmu RHV QE System & Core tools se přechází na agilní vývoj. Optimální agilní postupy poskytují týmu sadu jednoduchých řešení, která zefektivňují a usnadňují již existující týmové procesy a dodávají pevný základ pro aktivní práci se zákazníky. Dále také přináší nový pohled na to, jak mohou jednotlivé firemní aktivity a celky, jako například podniková architektura, finance, IT operační středisko a další, mnohem více spolupracovat. Tento rámec by měl mimo jiné i popisovat zaměření těchto aktivit a celků a jaké možnosti k tomu lze využít. Zároveň nejsou opomenuty ani kompromisy a speciální případy pro každý takový návrh řešení.

Týmové porady

Pro správný chod týmu jsou důležitá pravidelná setkání, kdy se jednotliví členové mohou dozvědět nové důležité informace o vývoji produktu, hodnotí se dosavadní práce a plánuje se nová. Každý týden proto probíhá 30minutová **„Grooming“ porada**, během které týmový vedoucí (*Team lead*) sdělí nové nebo měnící se důležité termíny, např. datum vydání produktu, poskytnutí nového balíkuS k testování atd. Diskutují se také termíny a postup práce na jednotlivých přidělených úkolech. K této poradě je navíc jednou za 2 týdny připojena speciální 30minutová porada, **„Planning“**, ve které se prochází seznam požadavků (product backlog) sestavený na základě přidělených i nepřidělených bugů v Bugzille[1]. Řeší se také plánování a kontrola již probíhající automatizace, přičemž prioritu má automatizace testů, které kontrolují funkčnost aspektů, které si žádal zákazník.

Jednou za 2 týdny také probíhá i další 30minutová porada. Tentokrát se jedná o retrospektivu (**„Restrospective meeting“**), jejíž cílem je zhodnotit jak podařené (označeno jako „Well Done“), tak i méně dobře nebo úplně špatně provedené („Can be improved“, „No go“) pracovní aktivity a procesy. Vzhledem k tomu, že je cílem se neustále zdokonalovat, jsou z negativně označených hledisek vytvořeny speciální úkoly, které by měly situaci napravit nebo vylepšit.

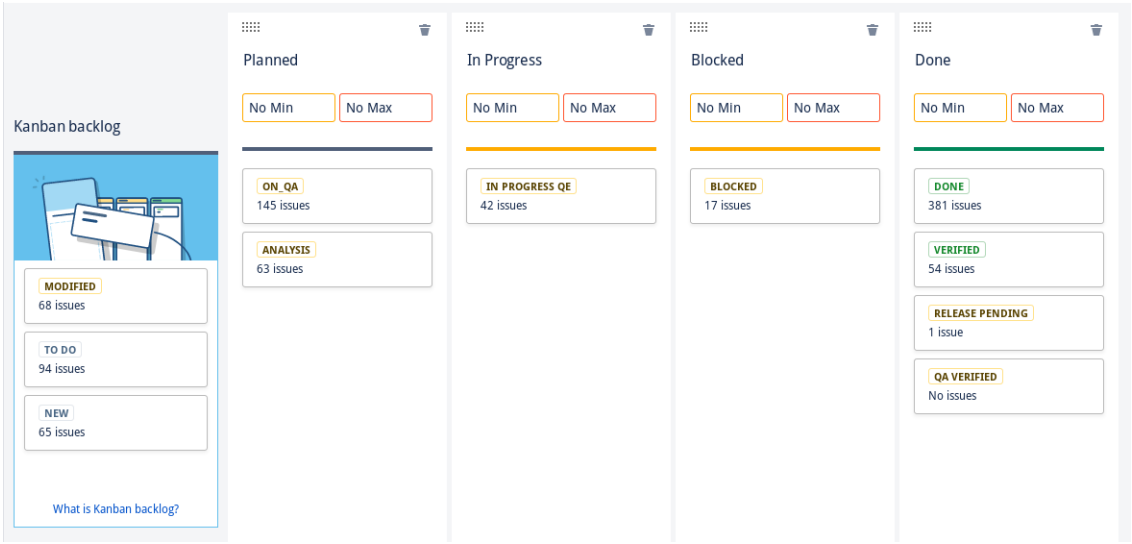
Každodenní **stand-upy** mohou přinášet poměrně velká časová omezení v případě, že je tým velký, proto se v týmu RHV QE System & Core tools přistoupilo k automatizaci tohoto

[1] https://bugzilla.redhat.com/

procesu jednoduchým způsobem. S aktuálním velkoplošným využitím Google chatu[2] pro sdělování jak týmových, tak i mezi-produktových informací, byla vytvořena speciální místnost, ve které jsou všichni členové týmu každý den upozorněni, aby napsali velmi stručně, co od posledního stand-upu dělali a co mají v plánu. Je přitom zohledněno, zda nemá člen týmu například dovolenou nebo není na nemocenské.

V rámci jednotlivých komponent produktu RHV však mohou probíhat nezávislé porady, přičemž se nejčastěji jedná o každodenní stand-upy nebo delší týdenní setkání. Navíc se konají pravidelné periodické (měsíční, čtvrtletní, roční) porady, na kterých je zhodnocena dosavadní práce a přiblížen cíl směřování na úrovni celého produktu, nové ideje, ohlasy zákazníků nebo jsou sdíleny technické znalosti v rámci demo porad. Na některých z těchto setkání může být přítomen Scrum vedoucí (Agile practitioner), který pomáhá v přechodu na agilní procesy, řídí tok konverzace a směřování celé porady tak, aby se měl každý účastník možnost projevit a zároveň se porada neprotahovala.

Kanban board

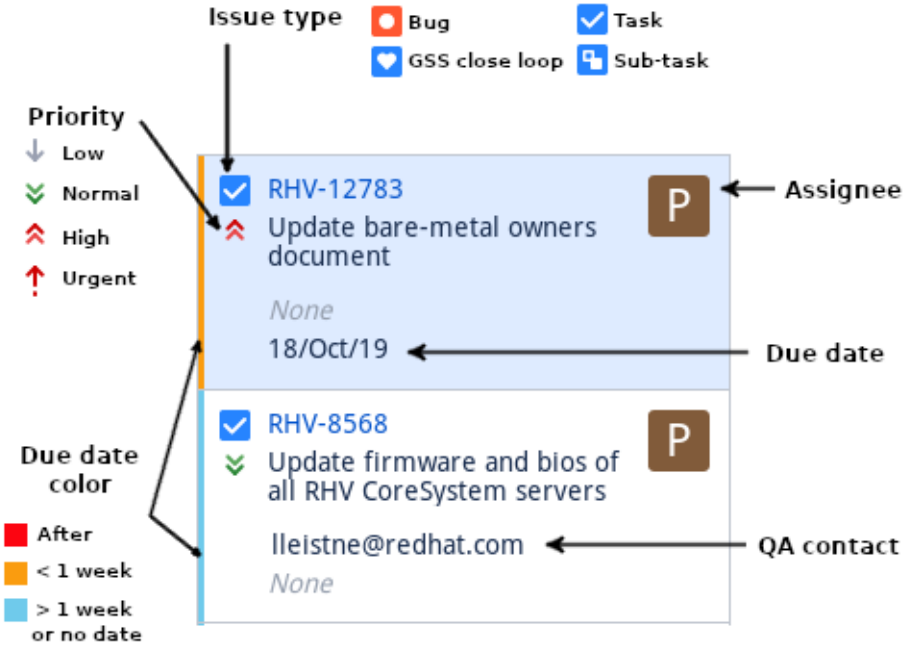


Obrázek 3.1: Kanban board – hlavní náhled, RHV QE System & Core tools, Red Hat, Inc.

Pro zpřehlednění jednotlivých týmových úkolů, plánování automatizací a dalších aktivit je využit Kanban board (viz Obrázek 3.1) poskytovaný softwarovým nástrojem Jira Software (viz Kapitola 2.4). Jednotlivé úkoly vytvořené z bugů na Bugzille a issue na GitHubu jsou automaticky i manuálně přesouvány do jednotlivých sloupečků se specifickým označením: „Kanban backlog“, „Planned“, „In Progress“, „Blocked“ a „Done“. Do „Kanban backlogu“ se ukládají úkoly, které ještě nebyly zpracovány QA týmem, ale jsou v plánu do budoucna. „Planned“ sloupec označuje úkoly, které mají být splněny (jejich stav je „ON_QA“) a je k nim přiřazen konkrétní člen QA týmu. Nicméně práce na takovém úkolu ještě nezačala. Do „In Progress“ kolonky putují úkoly v případě, že na nich začal přiřazený člen týmu pracovat (tedy si i kartičku úkolu sám přesunul do odpovídajícího sloupce). V „Blocked“ se nachází všechny úkoly, které nelze z různých důvodů splnit. Na příklad mohou být blokovány jiným ještě nesplněným úkolem nebo nedostatkem hardware pro testování. „Done“ sloupec slouží pro úkoly, které byly úspěšně dokončeny, tedy v případě,

[2] https://gsuite.google.com/products/chat/

že byl bug na Bugzille verifikován (stav „Verified“) nebo proběhl merge pull requestu na GitHubu po té, co bylo Jira issue označeno jako „QA Verified“.



Obrázek 3.2: Kanban board – detail kartičky, RHV QE System & Core tools, Red Hat, Inc.

Jednotlivé kartičky (viz Obrázek 3.2) s úkoly sledují typ úkolu, jeho prioritu, komu je úkol přiřazen, do kdy má být úkol splněn a samozřejmě číslo a název úkolu, přičemž kartička samotná odkazuje na detail celého úkolu, který může dál obsahovat odkazy na bugy v Bugzille, issue a pull requesty na GitHubu, detailnější popis problému, další úkoly atd.

3.2 Včasné testování komponenty „Portál virtuálních strojů“

Toto testování se odehrává na úrovni upstream, konkrétně v repositáři balíčku **ovirt-web-ui**[3], tedy je nutné, aby měl tester připravené testovací prostředí s oVirt enginem a operačním systémem CentOS, hostiteli a uživatelem, který nemá administrační práva, jelikož ta nejsou pro vstup do Portálu virtuálních strojů nutná, a naopak mohou zabránit zachycení některých problémů.

Proces včasného testování se odehrává v momentě, kdy si vývojáři navzájem odsouhlasí kód, který do projektu přináší novou funkcionalitu nebo nějakým způsobem opravuje nebo nahrazuje tu stávající. Takto přijaté změně v rámci pull requestu v GitHubu je nastaven štítek s označení „needs QE“ nebo „ON_QA“ a issue v Jira je taktéž přesunuto do stavu „ON_QA“.

Úkol je následně přiřazen určenému testerovi vedoucím QA týmu. Tester si naplánuje přibližnou dobu, kdy se bude úkolu věnovat a oznámí to v komentáři v Jira issue. Takto

[3] https://github.com/oVirt/ovirt-web-ui/

pak vývojáři ví, zda se problémem již QA tým zabývá, nebo jestli je třeba zvýšit prioritu a testera na nějakém z naplánovaných setkání urgovat. Jakmile má úkol testerovu pozornost, je třeba udělat několik manuálních kroků:

1. **Najít odkaz ke GitHub pull requestu obsahující změnu k otestování.**: Standardně se odkaz na pull request nachází přímo v Jira issue, nicméně i vývojáři se mohou opomenout, a proto je třeba vyhledat odkaz v referencích odkazovaného GitHub issue. Pokud Jira issue neobsahuje odkazy na issue ani pull requesty na GitHubu, jedná se o chybu ze strany vývojářů, na kterou jsou testerem upozorněni.

2. **Spuštění sestavení balíčku ovirt-web-ui obsahujícího změnu k otestování.**: Tento proces se provádí vytvořením komentáře s textem „ci build please“. K tomu, aby bylo na komentář adekvátně zareagováno, je potřeba speciálních práv k repositáři projektu oVirt. Komentář spustí vzdálené vytvoření balíčku ovirt-web-ui v rámci nástroje Jenkins, přičemž jsou prováděny kontroly kódu. Stav vytváření balíčku a odkazy na build v Jenkinsu jsou zobrazeny pomocí rozšiřující funkce GitHubu pod pull requestem. Je třeba počkat, než projdou všechny kontroly, ale zároveň nenechat vytvořený balíček nevyužitý příliš dlouho, protože časem vyprší a je nutné ho vytvořit znova.

3. **Získání odkazu na balíček v nástroji Jenkins.**: V artefaktech daného Jenkins buildu se vyhledá typ balíčku určeného pro odpovídající operační systém. Tento balíček se pak nainstaluje na virtuální stroj s oVirt enginem a provede se kontrola, že byl úspěšně nainstalován, např. porovnáním verze balíčku v Jenkins buildu a na virtuálním stroji.

4. **Samotné manuální testování.**: V jeho rámci se tester seznamuje s problematikou prováděných změn a následně testuje funkcionalitu a správnost dodaného řešení.

V případě, že je vše otestováno v pořádku, je testerem přidán ke GitHub pull requestu komentář „LGTM“ (looks good to me). Totéž provede na Jira issue, nicméně zde je vhodné, aby tester přidal slovní komentář popisující odůvodnění verifikace a verze balíčků a oVirt engine, na kterých testoval. Jira issue se následně přesune do stavu „QA verified“, načež jsou změny kódu přijaty do hlavního projektu a vývojáři změní stav Jira issue na „Verified“.

Pokud však tester není s kvalitou změn spokojený, je nutné, aby všechny nalezené nedostatky napsal do komentáře ke GitHub pull requestu. Následně je Jira issue testerem vráceno zpět do stavu „Assigned“ s komentářem odkazujícím na komentář v GitHub pull requestu a verzemi, ve kterých se testovalo. Jakmile vývojář chyby opraví, probíhá celý proces znova od začátku.

Kapitola 4

Návrh řešení

4.1 Motivace

Jak již bylo naznačeno (viz Kapitola 3.2), samotná příprava testovacího prostředí je zdlouhavá a neefektivní. Tester se mnohdy musí proklikat několika různými portály, než se vůbec k testovanému balíčku dostane, přičemž může nastat i situace, že balíček není vůbec připravený a je třeba ho nechat vytvořit znovu. Navíc se musí tester neustále kontrolovat, zda je jeho balíček už připravený, případně zda už je nainstalovaný atd. Vzhledem k povaze včasného testování je nutné tento postup opakovat i několikrát za den. Cílem této práce je proto ulehčit a zároveň zefektivnit testerovu práci tak, aby se touto opakovanou činností vůbec nemusel zaobírat a mohl se tak věnovat jiným pracovním aktivitám.

4.2 Požadované vlastnosti

Mimo základní funkcionalitu danou hlavní motivací zautomatizovat proces včasného testování komponenty ovirt-web-ui je pro projekt důležité, aby vhodným způsobem doplnil již existující strukturu v GitLabu propojující různé nástroje využívané týmem RHV QE System & Core tools a mohlo se na něj tak navázat během rozšiřování této struktury a vytváření nových projektů optimalizující nejrůznější aspekty vývoje software.

Další požadovanou vlastností je snadná testovatelnost projektu již existující sadou jednoduchých testů a kontrol formátu kódu, přičemž kvalita kódu bude kontrolována správcem repositáře a dalšími testery. Z důvodu plánovaného navazování dalších projektů na tuto práci (např. propojení s databází) je samozřejmě taky nutné, aby byl kód snadno pochopitelný, udržovatelný a přenositelný.

4.3 Existující GitLab repositář

Komerční alternativou ke GitHubu je GitLab[1] pod společností GitLab Inc., který nabízí kromě většiny společných funkcí pro správu a sdílení projektů podporu pro všechny části softwarového vývoje včetně automatizovaného testování, průběžné integrace a dalších výhod. Kromě plánovacích vlastností může uživatel nastavit i automatické sestavování balíčků, detailně konfigurovat testovací prostředí, sledovat konkrétní metriky v různých aspektech vývoje a také mít GitLab hostovaný přímo na vlastních serverech, což je zejména výhodné v komerčním sektoru [10].

[1] https://about.gitlab.com/

Aktuálně je pro již existující konektory a základní práci s databází vytvořen repositář na privátním GitLabu hostovaném na firemních serverech. Repositář se nazývá `RHV-data` a obsahuje spouštěče a konfigurační soubory pro přípravu virtuálního prostředí, automatické testování a konkrétní skripty. Zdrojové soubory se nachází v adresáři `rhv_db_data`, která se dál dělí na adresáře se soubory pro práci s databází a jednotlivými konektory.

Tato práce by měla tuto již existující strukturu vhodně doplnit konektory pro práci s Jira, GitHub, Jenkins a virtuálními stroji s nainstalovaným oVirt-enginem a možností vytvářet samostatné skripty nezávislé na databázi.

4.4 Git

Celá práce na projektu je spravována pomocí systému pro správu verzí Git[2], což je volně dostupný systém umožňující kontrolovat práci na projektu. Je poskytován pod GNU GPLv2 licencí[3], což z něj dělá otevřený software. Mezi přední vlastnosti patří zejména možnost pracovat ve speciálních nezávislých větvích, které se pak mohou slučovat do větve hlavní, a fakt, že Git je velmi rychlý a poměrně malý nezatěžující systém, který se snadno používá lokálně. Ve spolupráci s mnoha službami jako GitHub, GitLab nebo třeba Azure, které umožňují projekty spravované systémem Git dál sdílet, se jedná o hodně využívaný nástroj [7].

4.5 Python

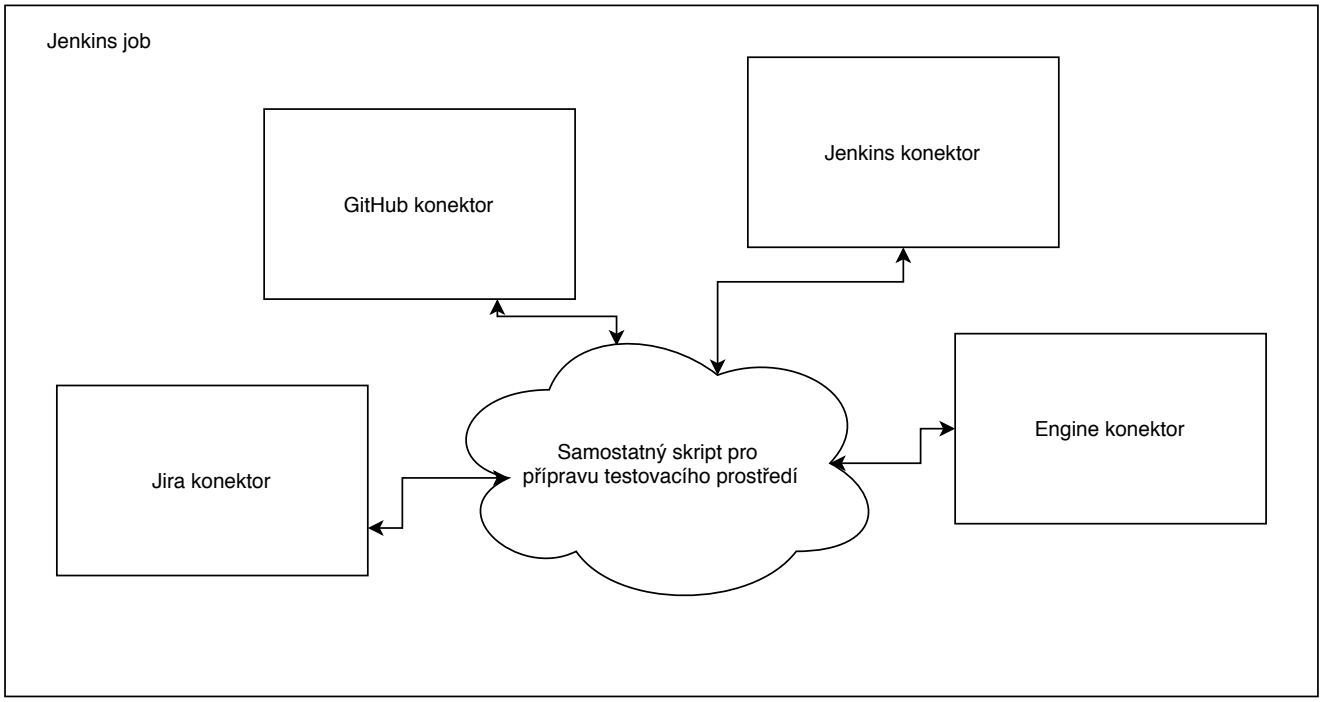
Python[4] je vysokoúrovňový interpretovaný programovací jazyk s širokým využitím. Je vyvíjený Python Software Foundation[5] pod OSI licencí pro otevřený software. Python je multiplatformní jazyk nabízející objektově orientovaný, strukturovaný a funkcionální přístup ve velmi dobře pochopitelné syntaxi, díky které nabývá v poslední době na popularitě. Kromě základních knihoven je možné vybírat z obrovského množství dalších i velmi specifických funkcionalit vyvíjených komunitou i jednotlivci. V této práci je využita poslední aktuální verze Python 3.8.2[6] [18].

4.6 Návrh struktury řešení

Výsledkem práce (viz Obrázek 4.1) bude skupina nových konektorů pro Jira, Github, Jenkins a engine, které jsou dále využity samostatným skriptem vykonávajícím přípravu prostředí pro včasné testování. Celý proces by byl vzdáleně spouštěn pomocí Jenkins jobu na firemních virtuálních strojích, přičemž bude zajištěno jeho opakované spouštění, výpisy logovacích zpráv a případná kontrola jednotlivých Jenkins buildů.

Jednotlivé konektory by měly být ve formě samostatných na sobě nezávislých tříd, které si samy zajišťují přihlášení do odpovídajících nástrojů a poskytují rozhraní pro další práci s nimi pomocí již existujících knihoven v jazyce Python. Všechny funkce by měly být znovupoužitelné i v dalších částech repositáře a mělo by být snadné další funkce do tříd přidávat.

[2] https://git-scm.com/
[3] https://opensource.org/licenses/GPL-2.0
[4] https://www.python.org/
[5] https://www.python.org/psf-landing/
[6] (https://www.python.org/downloads/release/python-382/



Obrázek 4.1: Hrubý návrh práce

Od Jira konektoru se očekává, že umožní sbírat informace o Jira issue a jejich filtrování podle nejrůznějších vlastností. Dále také z takového Jira issue získat všechny komentáře, filtrovat je podle obsahu a mít možnost vytvořit komentář nový.

GitHub konektor by měl především umožňovat získat informace o GitHub issue a pull requestech. Co se týče GitHub issue, je potřeba z něj umět získat reference na pull requesty a případně filtrovat issue podle značek. U pull requestu je naopak nutné, aby se dalo určit, zda byl proveden merge, jaký je stav vytváření Jenkins buildu a vyhledávat odkazy v něm.

S Jenkins konektorem by se měly dát získat jednotlivé buildy a odkazy na balíčky z artefaktů. Engine konektor by měl nabídnout připojení k virtuálnímu stroji a jednoduchou správu instalace balíčku daného odkazem.

Kapitola 5

Implementace

Samotná implementace se skládala ze dvou hlavních částí: implementace hlavního procesu skriptu připravujícího testovací prostřední a implementace jednotlivých konektorů.

5.1 Skript připravující testovací prostředí

Proces, který skript implementuje velmi úzce souvisí s postupem, kterým by se řídil tester, kdyby si prostředí chystal manuálně. Dá se tedy zjednodušit na 4 kroky:

1. Získání relevantních Jira issue
2. Získání informací o pull requestu a virtuálním stroji
3. Seřazení plánovaných příprav prostředí
4. Hlavní řídicí cyklus skriptu

Přímým spouštěčem skriptu je Jenkins job, který se neustále v cyklu sestavuje a provádí všechny výše zmíněné kroky. Nepřímým spouštěčem jsou speciální komentáře v Jira, které ovlivňují chod skriptu a vnitřní rozhodovací procesy.

Jenkins job

Jenkins job slouží k přípravě virtuálního prostředí, na kterém skript poběží, a jeho spouštění v opakovaných 15minutových cyklech. V každém buildu daného jobu se ve speciálním pracovním repositáři na aktuálně přiděleném virtuálním stroji instaluje pomocí shell skriptu Python, Python virtuální prostředí a všechny potřebné Python knihovny dané souborem `requirements.txt`. Zároveň také dojde ke spuštění skriptu.

Popis Jenkins jobu je daný v hromadném YAML[1] souboru (viz Zdrojový kód 5.1) obsahujícím mnoho dalších definic jiných Jenkins jobů, přičemž se řídí předem daným formátem, aby se pak takový job dal automaticky vytvořit v rámci jiného Jenkins jobu nazývaného **Jenkins job builder**[2]. Definice tedy obsahuje název jobu, název shell skriptu pro přípravu prostředí, spouštěče dalších buildů atd. V rámci definice je i odkaz na GitLab repositář, ze kterého se při každém sestavení získá aktuální verze repositáře se zdrojovými kódy projektu.

[1] YAML je uživatelsky přívětivý jazyk pro serializaci dat. Více na: https://yaml.org/

[2] https://docs.openstack.org/infra/jenkins-job-builder/index.html

Zdrojový kód 5.1: Definice Jenkins jobu v YAML souboru

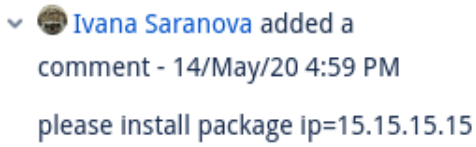
```
- job:
    name: rhv-qe-ovirt-web-ui-env-prep-job
    project-type: freestyle
    # Jednotlivé buildy nemohou běžet současně
    concurrent: false
    description: |
      <h1> Environment preparation for ovirt-web-ui </h1>
    node: 'rhv-flow-node'
    # Specifikace GitLab repositáře
    scm:
      - base-gitlab-scm:
          url: https://cesta_k_repositari/RHV-data.git
          branch: "master"
          basedir: "RHV-data"
    # Odkaz na skript, který se pouští během každého buildu
    builders:
      - shell: !include-raw: shell-scripts/rhv-qe-ovirt-web-ui-env-prep.sh
    # Předdefinované činnosti vykonávané po dokončení buildu
    publishers:
      - archive:
      artifacts: hosts.inventory
          allow-empty: true
      - postbuild-show-username
      - postbuild-catch-traceback
    # Automatický časovaný spouštěč buildů
    # nastavený na spouštění každých 15 minut
    triggers:
      - timed: "*/15 * * * *"
    # Další předdefinované změny běhu buildu
    wrappers:
      - workspace-cleanup
      - timestamps
```

Samotný Jenkins job i jednotlivé buildy lze sledovat ve webovém uživatelském rozhraní, nahlížet do konzolových výpisů, vytvořených souborů a logů a vytvářet nebo rušit další buildy. Historie se podle výchozího nastavení ukládá pro posledních 20 buildů, tedy jeden build má uloženou historii přibližně 5 hodin.

Speciální komentáře

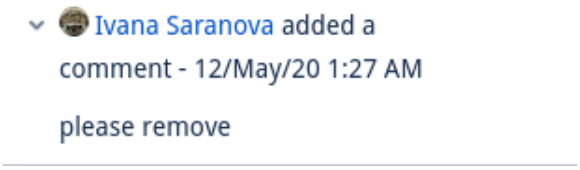
Jako speciální komentáře jsou označovány komentáře v Jira issues, které mají určitý předem daný formát. Slouží především ke snadnému ovládání skriptu uživatelem a zároveň oznamování o úspěšné či neúspěšné přípravě prostředí. V projektu je formát komentářů uložen v konfiguračních proměnných a je možné jejich formát měnit podle potřeby. Projekt rozlišuje 4 typy speciálních komentářů, přičemž dva slouží k ovládání skriptu uživatelem z vnější a dva k oznamování výsledků příprav:

- Ovládací komentáře
  - **REQUEST**: Jedná se o speciální Jira komentář ve tvaru `please install package ip=IP_REGEX [pr=N]` (viz Obrázek 5.2). Vytvoření takového komentáře v Jira issue pro skript znamená, že na virtuální stroj daný IP adresou má být nainstalovaný balíček z `N`-tého pull requestu, který je odkazován v samotném Jira issue, nebo v GitHub issue. Formát je inspirován již existujícím komentářem `ci build please` používaným v GitHub repositáři ovirt-web-ui komponenty.



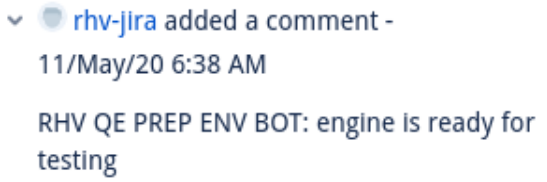
Obrázek 5.2: Příklad REQUEST komentáře

  - **REMOVE**: v případě, že příprava prostředí skončila úspěšně, ale uživatel se rozhodl, že na stejném virtuálním stroji chce provést jinou přípravu a tato ho blokuje, může využít komentář formátu `please remove` (viz Obrázek 5.3). V tomto případě je dané Jira issue v dalším zpracování ignorováno.



Obrázek 5.3: Příklad REMOVE komentáře

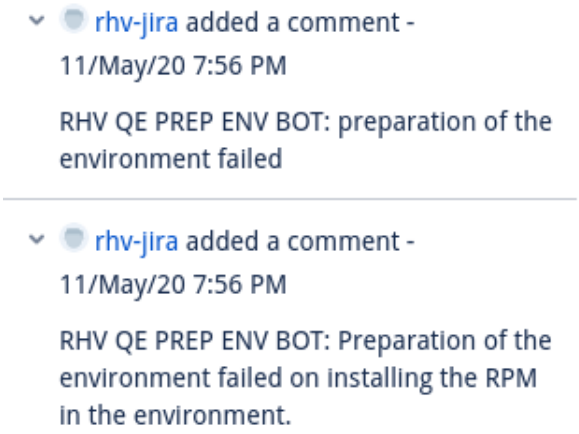
- Oznamovací komentáře
  - **READY**: Pokud příprava prostředí dopadne úspěšně a balíček je nainstalován, je na daném Jira issue vypsáno hlášení o tom, že je prostředí připraveno pro testování (viz Obrázek 5.4). Pokud je virtuální stroj daný IP adresou označený



Obrázek 5.4: Příklad READY komentáře

jako připravený k testování, budou všechny ostatní žádosti o přípravu na stejném stroji ignorovány.

– **FAILED**: Naopak pokud příprava prostředí selže na některém z kroků od hledání odkazů na pull request v Jira issue po instalaci balíčku na virtuálním stroji, je vypsána tato informace do daného Jira issue ve dvou komentářích – první označuje přípravu jako neúspěšnou, druhý specifikuje, kde došlo k problému (viz Obrázek 5.5). Cílem je usnadnit hledání konkrétní příčiny neúspěchu.



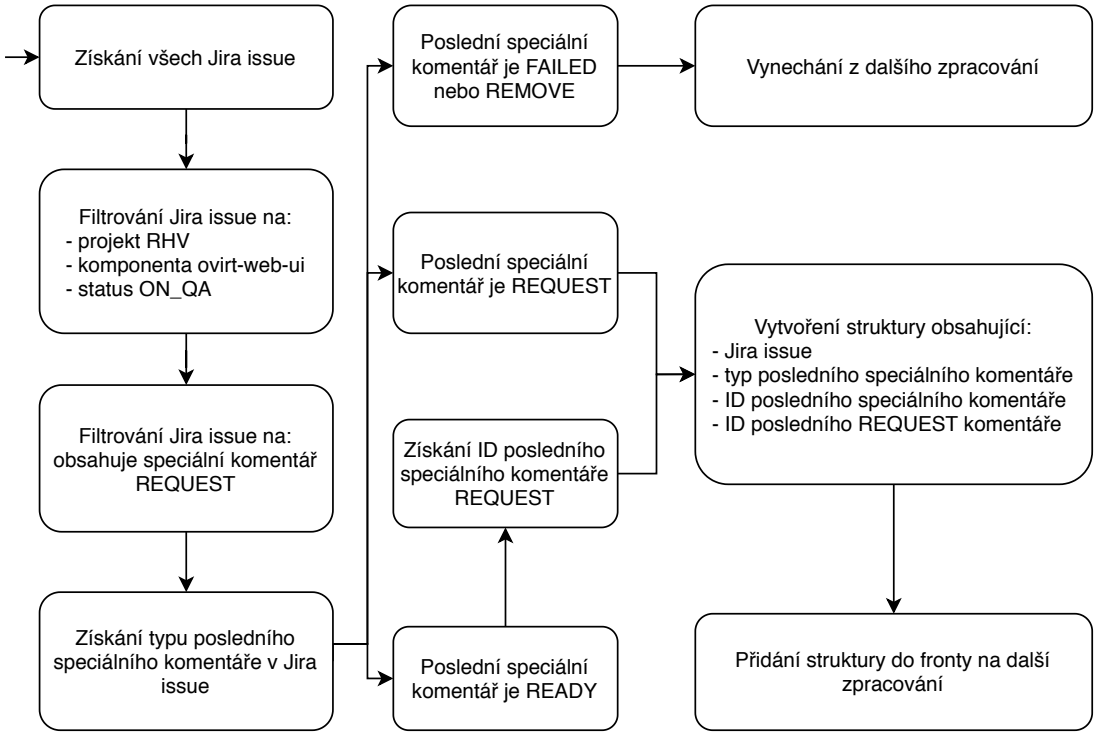
Obrázek 5.5: Příklad FAILED komentářů

Získávání relevantních Jira issue

Metoda `get_filtered_active_issues`, která implementuje proces znázorněný v diagramu (viz Obrázek 5.6), je prvním krokem přípravy testovacího prostředí. Jejím výstupem jsou vyfiltrovaná Jira issue a informace o speciálních komentářích v nich. Zejména v této metodě se nejvíce využívá Jira konektoru zprostředkovávajícího Jira API (viz Kapitola 5.2).

Jako relevantní Jira issue jsou označeny ty issue, které obsahují REQUEST komentář a kontextově souvisí s komponentou ovirt-web-ui v produktu RHV. Dalším důležitým parametrem je aktuální stav issue, kdy jsou vybírána pouze ta ve stavu „ON_QA“. Pokud je issue ve stavu „Assigned“ nebo „QA Verified“, nemá smysl jej řešit, protože zkrátka není určeno v daný moment k testování.

V případě, že byla nalezena některá Jira issue splňující dané podmínky, je nutné zjistit, který typ speciálního komentáře se v issue nachází jako poslední přidaný. To zajišťuje metoda `get_last_comment_action_and_id`, jejíž vstupem je Jira issue a výstupem ID posledního speciálního komentáře a jeho typ. Pokud je posledním komentářem REQUEST, pak se jedná o novou žádost o přípravu prostředí a pro dané issue bude vytvořena výstupní



Obrázek 5.6: Diagram filtrování Jira issue během přípravy testovacího prostředí

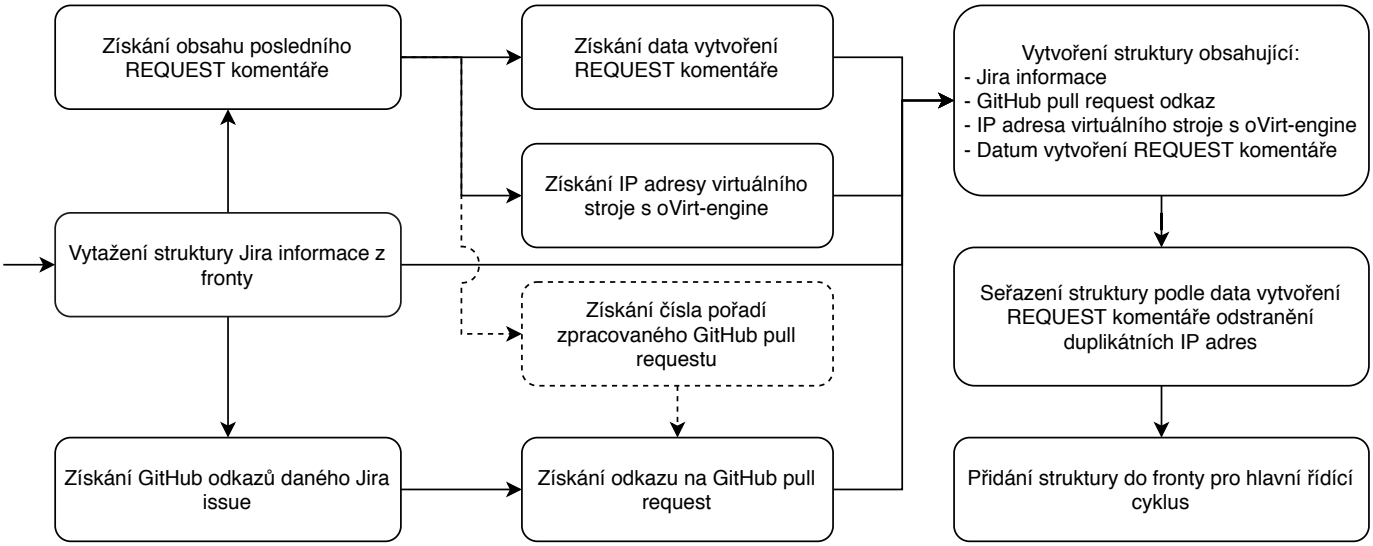
struktura. V případě posledního komentáře typu READY jde o již připravené prostředí, nicméně samotný komentář READY neobsahuje informace o IP adrese virtuálního stroje, proto je nalezen v issue ještě poslední REQUEST komentář a jeho ID přidáno do výstupní struktury. U issue s posledními komentáři FAILED nebo REMOVE se neprovádí již další zpracování a taková issue jsou skriptem ignorována.

Získávání dodatečných informací a seřazení plánovaných příprav

Se strukturami vytvořenými z vyfiltrovaných Jira issue se dále pracuje v metodě `get_pr_engine_info_list` (viz Obrázek 5.7). Získávají se dodatečné informace jako je datum vytvoření REQUEST komentáře, IP adresa virtuálního stroje, na který bude balíček instalován, a odkaz na GitHub pull request. Výstupem je struktura s původními Jira informacemi obohacená o tato nová data.

Z posledního REQUEST komentáře se pomocí regulárních výrazů získá IP adresa a volitelný argument `pr`, kterým lze částečně ovládat výběr GitHub pull requestu nebo issue v případě, že je jich více. Pro READY issue už není kromě IP adresy a data vytvoření posledního REQUEST komentáře nic podstatné a jeho výsledná struktura je vytvořena, zatímco REQUEST issue musí ještě získat odkaz na Github pull request.

Získávání odkazu na pull request se řídí několika pravidly. V případě, že se v Jira issue nachází odkaz nebo odkazy přímo na pull requesty (dá se zkontrolovat pomocí regulárního výrazu na očekávaný formát odkazu), vybere se první z nich, nebo v případě, že byl využit volitelný parametr `pr`, se pak použije jeho hodnota jako index výběru. Pokud v Jira issue nejsou odkazy na pull request, ale pouze odkaz na GitHub issue, získají se pull requesty z referencí v GitHub issue. Chybový je stav, kdy Jira issue neobsahuje jak odkazy na Github



Obrázek 5.7: Diagram získávání dodatečných informací pro přípravu testovacího prostředí a seřazení příprav

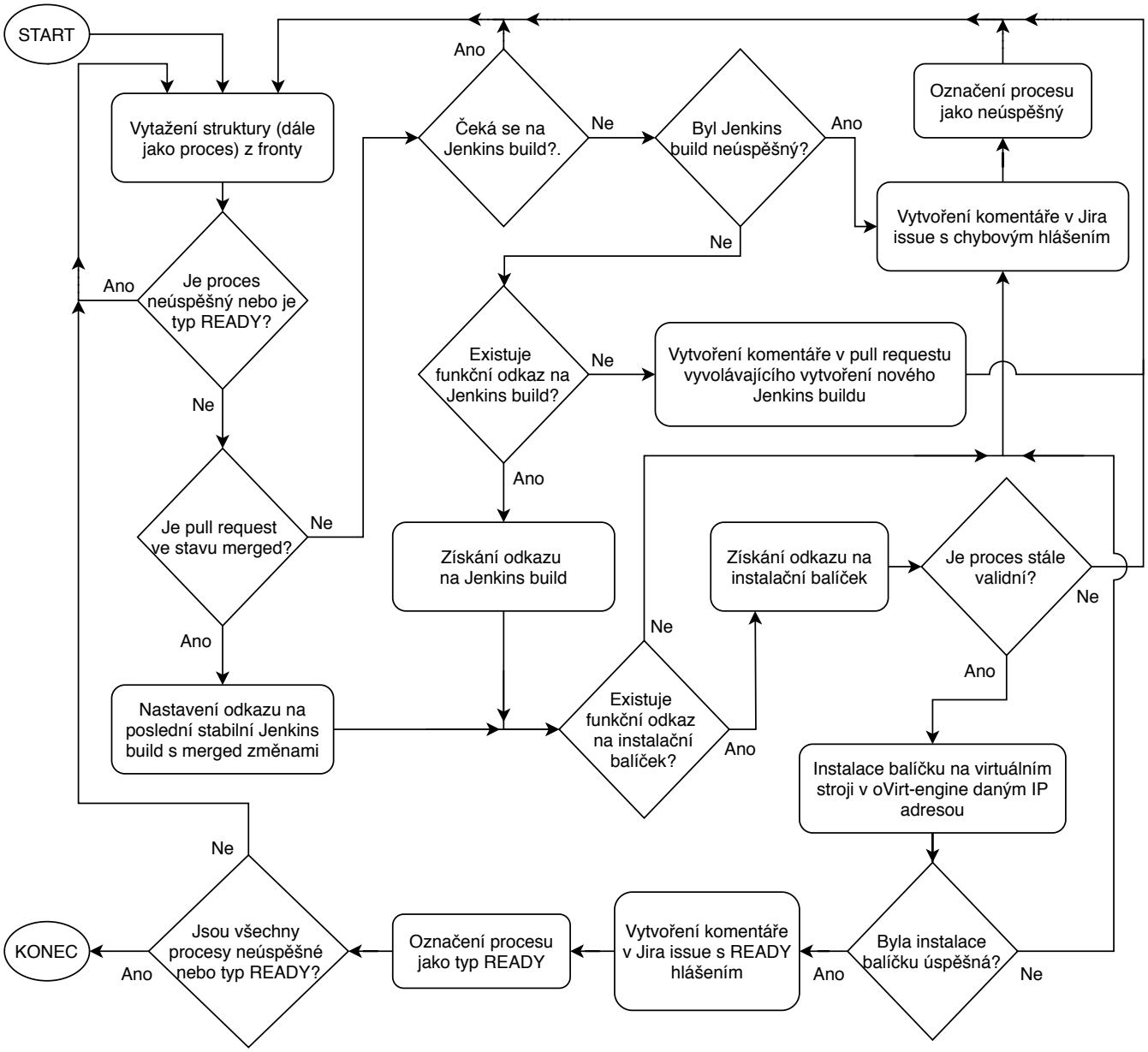
pull requesty, tak na issue. Jakmile k takové chybě dojde, je vytvořen odpovídající FAILED komentář a issue je vyřazeno z dalšího zpracovávání.

Seznam struktur s Jira informacemi, GitHub pull requestem, IP adresou a datem vytvoření je pak předán metodě `filter_pr_engine_info_list_for_unique_engine`, která jednotlivé přípravy seřadí podle data vytvoření a zajistí, že není stejná IP adresa v několika různých přípravách najednou. Přednost při odstraňování duplikátních IP adres mají přípravy ve stavu READY, kdy je potřeba, aby nebylo připravené prostředí přepsáno, dokud není issue verifikováno, nebo uživatelem okomentováno pomocí komentáře REMOVE.

Hlavní řídicí cyklus skriptu

Jádrem celého skriptu je získání odkazu na balíček z Github pull requestu a nainstalování ho na virtuální stroj daný IP adresou. Ač se zdá, že je proces vesměs přímočarý, může nastat mnoho situací, které jeho chod výrazně ovlivňují (viz Obrázek 5.8). Metoda, ve které se vše odehrává, se nazývá `main_cycle_prepare_testing_env`. Jejím vstupem je seřazený seznam se strukturami obsahujícími Jira informace, odkazem na pull request a IP adresu virtuálního stroje. Dále budou tyto struktury označovány jako jednotlivé přípravy. Hlavní cyklus se může opakovat několika minutových cyklech, kdy po zpracování všech příprav následuje ve výchozím stavu aktivní čekání a následně se všechny přípravy prochází znovu. Počet opakování chystání příprav i délka aktivního čekání se dají nastavit v parametrech `cycles` a `active_wait`.

Hned na začátku prvního průchodu příprav je kontrolováno, zda je příprava ve stavu READY. Pokud ano, tak je z dalších úprav vynechána a slouží pouze jako opatření před přeinstalací připraveného prostředí jinými balíčky. Mírné větvení přináší kontrola, zda není už pull request ve stavu „merged“, kdy je potřeba balíček hledat v posledním úspěšném buildu jednoho konkrétního Jenkins jobu, který obsahuje všechny již zanesené změny do hlavní větve komponenty. Balíček nemůže být již získán normální cestou, protože po merge dochází k znemožnění vygenerovat nový build, což je po expiraci odkazu problém.



Obrázek 5.8: Diagram hlavního řídicího cyklu skriptu, samotná příprava testovacího prostředí

Pomocí GitHub API (viz Kapitola 5.2) se dá ověřit status jednotlivých testů pull requestu, mezi kterými je i mimo jiné standardní CI test, který generuje instalační balíček. Pokud test stále probíhá, je příprava přeskočena. Chybové hlášení a nastavení přípravy na neúspěšnou způsobí, pokud test spadl. V opačném případě se pokusí skript získat odkaz na konkrétní Jenkins build, ve kterém test proběhl. Může se stát, že již odkaz vypršel a stránka vrátí chybu 404. V tomto případě se však nejedná o chybový stav, nýbrž potřebu provést test znovu a vygenerovat tak odkaz na nový Jenkins build. To se provede přidáním komentáře `"ci build please"` k pull requestu, načež se příprava opět přeskočí, protože úspěšné dokončení nového buildu není okamžité.

Po získání odkazu na konkrétní Jenkins build, který je dán unikátním číslem v rámci svého Jenkins jobu, se pomocí Jenkins API (viz Kapitola 5.2) prochází jednotlivé artefakty a hledá se balíček určený pro konkrétní operační systém a typ architektury. V případě ovirt-web-ui se jedná o balíčky pro operační systém RHEL a nespecifikovanou architekturu noarch. I zde může nastat chybná situace, že nebyl odpovídající balíček vytvořen a je nutné přípravu označit za neúspěšnou.

Před instalací balíčku na virtuální stroj je zkontrolována validita Jira issue, podle kterého je příprava vedena, aby se zamezilo duplikátním instalacím v případě nechtěného souběžného běhu (např. při lokálním ladění) nebo přípravě prostředí, které již není třeba chystat, v metodě `is_still_valid`. Zkontroluje se, že je Jira issue stále ve stavu „ON_QA“ a že posledním komentářem je skutečně REQUEST. Poté se provede instalace balíčku na virtuální stroj daný IP adresou. Instalace se v případě neúspěchu opakuje celkem třikrát, a pokud ani pak nedojde k úspěchu, je opět nutné vyvolat chybové hlášení v Jira issue a nastavení přípravy na nezdařilou. Správně nainstalovaný balíček je znamením pro skript, že se příprava povedla a je možné ji převést do stavu READY, který je také nahlášen komentářem v Jira issue.

5.2 Konektory

Jednotlivé konektory slouží ke komunikaci testery využívanými nástroji jako jsou například Bugzilla, Jira, GitHub, Google Spreadsheet[3] apod. Jejich cílem není poskytnout wrapper[4] nad REST API[5] nástrojů, ale naopak využít již existující knihovny poskytující takové wrappery a poskytnout více specifické metody a funkcionalitu včetně přihlašování a dalších formalit.

V rámci práce bylo navázáno na již existující konektory pro Bugzilla a Google Spreadsheet dotvořením dalších konektorů pro Jira, GitHub, Jenkins a práci s virtuálními stroji (Engine konektor). Všechny konektory jsou vytvořeny pomocí návrhového vzoru Jedináčka (***Singleton***), jehož cílem je poskytnout pouze jeden jediný objekt určitého typu. Výhodou je snížení časové náročnosti při opakovaném vytváření nových instancí tříd a snadné řízení přístupu k jediné instanci. Návrhového vzoru Jedináčka se využívá v kombinaci s dekorátorem `@property`, Python alternativou k zapouzdření pomocí metod `getter` a `setter` třídní proměnné, kdy lze k proměnné přistupovat obvyklým způsobem, ale ve skutečnosti po jejím zavolání probíhají určené metody. V konektorech se tedy v property proměnných využívá přístupu Jedináčka, kdy je proměnná inicializována pouze jednou v rámci instance. To je obzvlášť výhodné při časově zdlouhavějších přihlašováních k nástrojům pomocí API.

Všechny metody konektorů obsahují také krátké logovací zprávy, které usnadňují sledování běhu programu a jeho případné ladění.

Jira konektor

Pro vzdálenou komunikaci s Jira serverem existuje Jira REST API[6], které poskytuje všechnu základní funkcionalitu (jako například správa issue a komentářů) včetně mnoha doplnitel-

[3] https://www.google.com/sheets/about/

[4] Datová struktura nebo software, který obaluje jiná data nebo software tak, aby byla použitelná v jiném systému. Více na: https://www.pcmag.com/encyclopedia/term/wrapper

[5] REST API je programové aplikační rozhraní (API), které používá HTTP metody GET, PUT, POST a DELETE k práci s daty.

[6] https://developer.atlassian.com/server/jira/platform/rest-apis/

ných rozšíření. Pro komunikaci se používá formát JSON[7] a standardní HTTP metody GET, PUT, POST a DELETE, přičemž jednotlivé zdroje a cíle jsou dány speciálním URI[8] formátem [5].

Pro snadnou práci s Jira REST API v jazyce Python byla zvolena wrapper knihovna **jira-python**, také označována pouze jako jira[9]. Knihovna je vyvíjena a udržována komunitou jako otevřený projekt na GitHubu[10], kam může přispívat i veřejnost. Vzhledem k podpoře od společnosti Atlassian v podobě integračního testování a dlouhodobého kvalitního vývoje byla knihovna vhodným kandidátem pro Jira konektor.

V konektoru je využita HTTP BASIC autentizace[11], přičemž se využívá speciálně vytvořený Jira uživatel s potřebnými právy k prohlížení a upravovaní issue. Konektor dále nabízí metody jako získání všech issue, které odpovídají všem předaným parametrům (`get_issues`), filtrování issue podle specifických komentářů daných regulárním výrazem (`filter_issues_with_specific_comment`), získávání odkazů daných regulárním výrazem z issue (`get_links_from_issue`) a další.

GitHub konektor

Podobně jako Jira konektor i GitHub konektor využívá rozsáhlého GitHub REST API [12], které také využívá pro komunikaci formát JSON a HTTPS metody GET, POST, PATCH, PUT a DELETE. Pomocí API lze spravovat repositáře, issue, pull requesty, vytvářet komentáře, provádět kontrolní testy, provádět akce spojené s commity, větvemi nebo referencemi [9].

Pro GitHub REST API existuje hned několik Python knihoven, na které odkazuje přímo GitHub vývojářská příručka. V práci byla nakonec zvolena knihovna **PyGithub**[13] vzhledem k její uživatelské přívětivosti a poměrně rozsáhlé dokumentaci. Knihovna je vyvíjená na GitHubu[14] jako otevřený projekt.

Konektor využívá autentizaci pomocí GitHub tokenu, což je speciální řetězec, který lze vygenerovat v uživatelském rozhraní na GitHubu. Obdobně jako u Jira konektoru bylo nutné i zde vytvořit speciálního GitHub uživatele, který má přístup k privátnímu obsahu ovirt-web-ui repositáře. Samotný konektor obsahuje dvě třídy – `GithubRepository` a `GithubPR`, kde jednotlivé instance `GithubPR` se ukládají v property proměnné třídy `GithubRepository` podle návrhového vzoru Objektový fond (***Object pool***). Tento návrhový vzor se používá v případě, kdy se opakovaně přistupuje ke stejnému objektu, ale při každém přístupu by jinak bylo nutné jej znova vytvořit. Místo toho si třída `GithubRepository` jednotlivé přístupy a vytvořené objekty ukládá a při dalším volání je pouze vytáhne ze seznamu uložených objektů. Veškeré metody z `GithubPR` mají tedy ekvivalent v `GithubRepository` s tím rozdílem, že přijímají parametr rozlišující mezi jednotlivými `GithubPR` objekty.

Mezi nabízené funkcionality konektoru patří vyhledání referencí na pull requesty v GitHub issue (`find_all_pr_references_in_issue`), filtrování pull requestů podle značek (`filter_pr_links_by_label`), kontrola stavu testů daného pull requestu

[7] JSON je uživatelsky přívětivý jazyk pro serializaci dat, často využívaný jako vstup i výstup komunikace přes REST API. Více na: https://www.json.org/json-en.html

[8] Uniformní identifikátor zdrojů (URI) je kompaktní sekvence znaků, které identifikují abstraktní nebo fyzický zdroj. Více na: https://tools.ietf.org/html/rfc3986

[9] https://pypi.org/project/jira/

[10] https://github.com/pycontribs/jira

[11] Přihlašování pomocí jména a hesla, která jsou přidávána k jednotlivým HTTP požadavkům.

[12] https://developer.github.com/v3/

[13] https://pypi.org/project/PyGithub/

[14] https://github.com/PyGithub/PyGithub

(`check_if_pr_waiting_for_new_build`, `check_if_build_for_pr_failed`), generování nového Jenkins buildu pro daný pull request (`generate_new_build_for_pr`) a další.

Jenkins konektor

Jenkins poskytuje svým uživatelům API pro vzdálený přístup[15] ve třech verzích: v komunikačním formátu XML[16], JSON nebo pomocí Pythonu. Nejedná se o opravdové REST API, protože data přístupná z jednoho zdroje, ale naopak lze ke každému jednotlivému webu, na kterém je hostovaná instalace Jenkins, jednoduše přidat řetězec `/api/`. Uživatel pak může přistupovat k jednotlivým informacím o Jenkins komponentách (job, build, artefakty,...), spouštět nové buildy, spravovat joby pomocí klasických HTTP metod [12].

Konektor využívá právě jednu z Python možností, jak k Jenkins API přistupovat. Jedná se o knihovnu **python-jenkins**[17], poskytovanou pod BSD licencí[18], která je vyvíjená pro automatizaci Jenkins serverů. Tento wrapper nad Jenkins API usnadňuje práci s vytvářením, mazáním a aktualizováním běhů a sestavování projektů a zejména přináší pohodlný způsob jak o jednotlivých objektech získat co nejvíc informací. Knihovna má zdrojové kódy volně přístupné na opendev[19], což je platforma pro sdílení a vývoj otevřeného software.

V konektoru nebylo zapotřebí využívat autentizace, neboť Jenkins server, ke kterému skript přistupuje, je veřejný, a proto nebylo potřeba ani vytvářet speciálního uživatele. Nicméně se tato vlastnost dá v budoucnu do konektoru snadno implementovat, pokud bude nutné přistupovat i do privátních Jenkins serverů. Vzhledem k požadavkům hlavního skriptu byla implementována metoda (`get_rpm_link`), která získá z odkazu na konkrétní Jenkins build odkaz na instalační balíček podle toho, na kterou architekturu a operační systém cílí. Odkaz na balíček je získán z artefaktů, přičemž bylo nutné ošetřit případ končících řetězcem „lastStableBuild“ (poslední stabilní sestavení projektu), kdy není v odkazu na Jenkins build poskytnuto číslo, podle kterého ho lze v Jenkins API vyhledat (`get_build_from_link`).

Engine konektor

Poslední z implementovaných konektorů slouží k instalaci balíčku na virtuální stroj. Za tímto účelem je použita knihovna **python-rrmngmnt**[20], která umožňuje vzdálený přístup a správu linuxových virtuálních strojů přes protokol SSH pomocí různých rozhraní od řízení hostitelů, operacemi nad souborovými systémy nebo síťovými a systémovými službami po instalační manažery. Knihovna je poskytována pod licencí GPLv2 a vybrána byla proto, že je vyvíjená speciálně pro účely automatizace produktu RHV jako otevřený projekt na GitHubu[21].

Konektor se na virtuální stroj daný jeho IP adresou nebo známým doménovým jménem připojí pomocí SSH protokolu, přičemž se předpokládá, že se jedná o uživatele s root[22] právy a předem daným heslem. Metoda pro instalaci balíčku (`install_rpm`) se pak pokusí balíček daný jeho jménem nebo odkazem nainstalovat na virtuální stroj. Vzhledem k tomu,

[15] https://wiki.jenkins.io/display/JENKINS/Remote+access+API

[16] XML je jazyk pro serializaci dat, velmi často používaný v HTTP komunikaci jako vstupní i výstupní formát.

[17] https://pypi.org/project/python-jenkins/

[18] https://opensource.org/licenses/BSD-3-Clause

[19] https://git.openstack.org/cgit/openstack/python-jenkins

[20] https://pypi.org/project/python-rrmngmnt/

[21] https://github.com/rhevm-qe-automation/python-rrmngmnt

[22] Root je uživatel, který má práva pro provádění všech příkazů úprav všech souborů na linuxových operačních systémech.

že se chyby spojené se špatnou IP adresou nebo špatným prostředím na virtuálním stroji projevují až při samotné instalaci, je nutné tyto výjimky odchytávat až zde.

5.3 Spuštění práce

Vzhledem k povaze externího firemního zadání, které počítá s prací s privátním přístupem k nástrojům a speciálními právy, je nemožné projekt spustit mimo specializované prostředí. Podmínky pro správné spuštění a běh projektu jsou:

- VPN přístup do sítě společnosti Red Hat
- Přihlašovací údaje k speciálním účtům s adekvátními právy na serverech Jira a GitHub
- Nainstalovaný Python 3.8.2

Do vybrané složky rozbalíme repositář `RHV-data` se všemi zdrojovými soubory a upravíme soubor `rhv-data/rhv_db_data/creds.py` tak, aby obsahoval všechny korektní přihlašovací údaje. Jednou ze závislostí pro Python knihovnu psycopg2[23] jsou linuxové balíčky, které lze nainstalovat pomocí příkazu (určený pro operační systémy CentOS, Fedora, RHEL):

```
yum -y install python3-devel postgresql-devel
```

Před samotným spuštěním skriptu je nutné vytvořit Python virtuální prostředí a aktivovat jej:

```
python3 -m venv venv
source venv/bin/activate
```

Dále se musí nainstalovat všechny potřebné Python závislosti:

```
pip install -r RHV-data/requirements.txt
```

Nyní je možné pustit jeden běh skriptu:

```
python3 RHV-data/runner.py -ovirt_web_ui_prep_env
```

5.4 Testování

Automatizované testování bylo umožněno v nástroji GitLab, kde se při každém commitu provádí sada testů pro kontrolu dodržování formátu PEP8[24], obecné funkčnosti metod a produkční testy. Kontrola formátu PEP8 se realizuje pomocí nástroje flake8[25]. Při testování metod se všechny implementované metody ve všech konektorech kontrolují, zda je lze pustit s výchozím nastavením tak, aby nedošlo k výjimce. Pro metody, které mají parametry bez výchozích hodnot, je v plánu vytvořit speciální sadu testů a je na to vytvořeno GitLab issue. Produkční testy v GitLabu jsou nahrazeny spouštěním skriptu v samostatném Jenkins jobu, který si automaticky aktualizuje zdrojové soubory podle sledované větve v GitLab

[23] https://pypi.org/project/psycopg2/

[24] PEP8 je příručka pro dodržování jednotného programovacího stylu Python kódu. Více na: https://www.python.org/dev/peps/pep-0008/

[25] https://pypi.org/project/flake8/

repositáři. Pokud během skriptu dojde k výjimečnému stavu, build končí ve stavu „Failed“ (neúspěšný) a v konzoli je díky logovacím zprávám u každé metody konektoru určit přibližné místo chyby a tu následně lokálně zreprodukovat a opravit.

Samotné testování projektu probíhalo jak pomocí automatizovaných testů pro kontrolu dodržování formátu PEP8 kódu a sady unit a produkčních testů, tak především manuálně. Kontrolovala se reakce skriptu na různé scénáře:

- Úspěšná instalace balíčku po zveřejnění komentáře REQUEST na Jira issue.
  - Variace 1: Jira issue neobsahuje odkazy na pull request. => Odkaz na pull request je získán z referencí v Github issue.
  - Variace 2: Odkaz na Jenkins build pro testy u pull requestu již vypršel. => Vygenerování nového Jenkins buildu pomocí komentáře, vyčkání na jeho úspěch a získání odkazu.
  - Variace 3: Několik REQUEST komentářů se stejnou IP adresou zveřejněných na různých Jira issue. => Instalace proběhne s balíčkem z toho Jira issue, na kterém byl daný REQUEST komentář zveřejněn nejdříve.
- Neúspěšná instalace balíčku po zveřejnění komentáře REQUEST na Jira issue v případě, že:
  - Jira issue neobsahuje žádné odkazy na GitHub issue ani pull requesty.
  - Jira issue obsahuje odkazy pouze na GitHub issue, ale to neobsahuje žádné reference na pull requesty.
  - IP adresa v REQUEST komentáři není správná (nedá se na ni připojit, nelze na stroji, na který odkazuje, nainstalovat balíčky apod.).
- Úspěšné nepřeinstalování balíčku na virtuálním stroji daném IP adresou po zveřejnění READY komentáře v případě, že existuje REQUEST komentář na stejnou IP adresu u jiného Jira issue.
- Úspěšné přeinstalování balíčku na virtuálním stroji daném IP adresou po zveřejnění REMOVE komentáře v případě, že existuje REQUEST komentář na stejnou IP adresu u jiného Jira issue.
- Nedochází k opakované instalaci balíčku v případě, že byla daná příprava označená jako neúspěšná po zveřejnění komentáře FAILED.

Dalším aspektem testování bylo i zjišťování, kolik času automatizace přípravy testovacího prostředí skutečně ušetřila. Za tímto účelem bylo stopkami změřeno, kolik času zabere příprava prostředí manuálně. Se sbíráním dat pomáhali i další testeři, kteří mají na starost testování komponenty ovirt-web-ui. Doba měření započala při otevření konkrétního Jira issue a skončila při úspěšné instalaci balíčku na virtuálním stroji. Stopky se dočasně přerušily v momentě, kdy se čekalo na nový Jenkins job, pokud ho bylo nutné vygenerovat. Celkem se podařilo takto otestovat 23 Jira issues za měsíc za jednoho testera, přičemž průměrná doba, kterou zabrala jedna příprava testovacího prostředí byla 6 minut a 25 sekund.

Dále byli testeři také dotázáni, jak využili čas ušetřený automatizací. Mezi odpověďmi zaznělo především prostudování informací o změnách v pull requestu, automatizace testů uživatelského prostředí, testování problémů nesouvisejících s komponentou ovirt-web-ui, samostudium v certifikačních kurzech, komunikace s členy týmu a vývojáři.

Aktuálními problémy, které je potřeba v blízké době vyřešit, je používání hesel v konfiguračních souborech a absence testů pro metody s parametry bez výchozích hodnot. Pro první z překážek se nabízí využití modulů nástroje Ansible[26], konkrétně ansible-vault[27], což je modul, kterým lze šifrovat citlivá data. V druhém případě je nutné napsat větší sadu speciálních testů. Všechny problémy, nedostatky i požadavky na vylepšení jsou dokumentovány v GitLab repositáři v sekci issues.

[26] https://www.ansible.com/
[27] https://docs.ansible.com/ansible/latest/user_guide/vault.html

Kapitola 6

Závěr

Cílem práce bylo automatizovat proces přípravy testovacího prostředí v rámci manuálního včasného testování komponenty ovirt-web-ui v projektu oVirt a produktu RHV. To se podařilo a práce je nyní plně využívána testery v týmu RHV QE System & Core tools.

V teoretické části byly nastudovány a shrnuty agilní metodiky a metodologie včasného testování. Neméně pozornosti bylo věnováno projektu oVirt a produktu RHV, zejména pak testovanému uživatelskému rozhraní Portál virtuálních strojů. Následovala problematika nástrojů Jira, GitHub a Jenkins. V návrhu byla zohledněna aktuální situace v týmu RHV QE System & Core tools, stejně tak jako využití agilních metodik a včasného testování komponenty ovirt-web-ui. Implementace proběhla na základě návrhu a její nasazení bylo otestováno a výsledky zhodnoceny.

Práce byla zasazena do již existující automatizační struktury na GitLabu a implementovaná v jazyce Python. Byly použity wrappery v podobě Python knihoven pro REST API jednotlivých nástrojů a celá práce je automaticky spouštěna v Jenkins jobu, který umožňuje sledovat průběh skriptu. Ovládání práce je pomocí jednoduchých komentářů v nástroji Jira, přičemž uživatel je o výsledcích příprav prostředí informován. Celkem se díky práci podařilo testerům ušetřit v průměru 2 hodiny a 27 minut měsíčně, kdy se mohou testeři věnovat jiné užitečnější aktivitě.

Práce mi rozšířila obzory zejména v oblasti propojení agilních metod a vývojových nástrojů a práci s nimi. Taktéž jsem velmi ocenila nové programátorské zkušenosti s jazykem Python a navrhováním vlastního Jenkins jobu.

V práci bych chtěla pokračovat tak, že bude možné nahrávat výsledky z nově implementovaných konektorů do databáze a generovat tak mnohem rychleji data nová, která půjde využít i mimo samostatné Python skripty. Zároveň by se v práci dalo pokračovat nachystáním sady již existujících virtuálních strojů určených pouze pro účely včasného testování, a tím by odpadla nutnost uživatele spravovat vlastní virtuální stroj.

Literatura

[1] Agile Alliance. *Agile Alliance* [online]. Agile Alliance, 2020 [cit. 2020-05-14]. Dostupné z: https://www.agilealliance.org/.

[2] Ambler, S. W. *Examining the Agile Manifesto* [online]. Scott W. Ambler, 2014 [cit. 2020-05-14]. Dostupné z: http://www.ambysoft.com/essays/agileManifesto.html.

[3] Ambler, S. W. *Feature Driven Development (FDD) and Agile Modeling* [online]. Scott W. Ambler, 2018 [cit. 2020-05-14]. Dostupné z: http://www.agilemodeling.com/essays/fdd.htm.

[4] Atlassian. *Jira: Issue & Project Tracking Software* [online]. Atlassian, 2020 [cit. 2020-05-14]. Dostupné z: https://www.atlassian.com/software/jira.

[5] Atlassian. *Jira Server Developer* [online]. 2020 [cit. 2020-05-14]. Dostupné z: https://developer.atlassian.com/server/jira/platform/rest-apis/.

[6] Beck, K., Highsmith, J., Cockburn, A., Cunningham, W., Fowler, M. et al. *Manifesto for Agile Software Development* [online]. Ward Cunningham, 2001 [cit. 2020-05-14]. Dostupné z: http://agilemanifesto.org/iso/cs/manifesto.html.

[7] Chacon, S. *About* [online]. Git community, 2020 [cit. 2020-05-14]. Dostupné z: https://git-scm.com/about.

[8] GitHub, Inc. *Build software better, together* [online]. GitHub, Inc., 2020 [cit. 2020-05-14]. Dostupné z: https://github.com/about/.

[9] GitHub Inc.. *GitHub API v3* [online]. 2020 [cit. 2020-05-14]. Dostupné z: https://developer.github.com/v3/.

[10] GitLab Inc.. *The first single application for the entire DevOps lifecycle* [online]. GitLab Inc., 2020 [cit. 2020-05-14]. Dostupné z: https://about.gitlab.com/.

[11] Jenkins. *Jenkins* [online]. Jenkins, 2020 [cit. 2020-05-14]. Dostupné z: https://www.jenkins.io/.

[12] Jenkins. *Remote access API - Jenkins* [online]. 2020 [cit. 2020-05-14]. Dostupné z: https://wiki.jenkins.io/display/JENKINS/RemoteaccessAPI.

[13] Katedra aplikované geoinformatiky a kartografie, Přírodovědecká fakulta Univerzita Karlova. *Metoda, metodika, metodologie* [online]. 2020 [cit. 2020-05-14]. Dostupné z: https://www.natur.cuni.cz/geografie/geoinformatika-kartografie/studium/bakalarske-studium/pravidla-pro-bakalarske-prace/metoda-metodika-metodolige.

[14] Ken Schwaber, J. S. *What is Scrum?* [online]. Scrum.org, 2020 [cit. 2020-05-14]. Dostupné z: https://www.scrum.org/resources/what-is-scrum.

[15] Lynn, R. *What is FDD in Agile?* [online]. Planview, Inc., 2020 [cit. 2020-05-14]. Dostupné z: https://leankit.com/learn/agile/fdd-agile/.

[16] McGregor, J. D. Test early, test often. *The Journal of Object Technology*. AITO - Association Internationale pour les Technologies Objets. 2007, roč. 6, č. 4, [cit. 2020-04-08], s. 7. Dostupné z: http://dx.doi.org/10.5381/jot.2007.6.4.c1.

[17] oVirt. *Documentation* [online]. 2020 [cit. 2020-05-14]. Dostupné z: https://ovirt.org/documentation/.

[18] Python Software Foundation. *Welcome to Python.org* [online]. Python Software Foundation, 2020 [cit. 2020-05-14]. Dostupné z: https://www.python.org/.

[19] Red Hat Virtualization Documentation Team, R. *Product Guide Red Hat Virtualization 4.3* [online]. 2020 [cit. 2020-05-14]. Dostupné z: https://access.redhat.com/documentation/en-us/red_hat_virtualization/4.3/html/product_guide.

[20] Sheremeta, G. a Libra, M. *OVirt ovirt-web-ui documentation screenshots* [online]. GitHub, Inc., 2020 [cit. 2020-05-27]. Dostupné z: https://github.com/oVirt/ovirt-web-ui/tree/master/doc/screenshots.

[21] Singhal, V. *Benefits of Early Testing* [online]. HelpingTesters, 2018 [cit. 2020-05-14]. Dostupné z: http://www.helpingtesters.com/benefits-early-testing/.

[22] Vijay, S. *What is Early Testing and Why to Start Testing Early in SDLC (Practical)* [online]. SoftwareTestingHelp, Apr 2020 [cit. 2020-05-14]. Dostupné z: https://www.softwaretestinghelp.com/early-testing/.

[23] Wells, D. *Extreme Programming: A gentle introduction* [online]. Don Wells, 2013 [cit. 2020-05-14]. Dostupné z: http://www.extremeprogramming.org/index.html.

Příloha A

Obsah přiloženého paměťového média

- `/rhv-data/*` – RHV-data GitLab repositář obsahující vytvořené konektory a skript pro přípravu testovacího prostředí
- `/demovideo.mp4` – Demo video s praktickou ukázkou práce
- `/text/*` – zdrojový kód této práce
- `/xsaran02.pdf` – finální verze této práce
- `/README.txt` – detailnější popis jednotlivých adresářů a souborů na paměťovém médiu