VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

SYSTÉM PRE SPRÁVU VÝSLEDKOV TESTOV DOPLŇUJÚCI NÁSTROJ TMT

TEST RESULTS MANAGEMENT SYSTEM COMPLEMENTING THE TMT TOOL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. ONDREJ DUBAJ**
AUTHOR

VEDOUCÍ PRÁCE **prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2020/2021

Zadání diplomové práce

Student: **Dubaj Ondrej, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Informační systémy
Název: **Systém pro správu výsledků testů doplňující nástroj tmt**
Test Results Management System Complementing the tmt Tool
Kategorie: Analýza a testování softwaru
Zadání:

1. Seznamte se se stávající testovací infrastrukturou firmy Red Hat využívající nástroj Nitrate a s plány na jeho náhradu nástrojem tmt.
2. Shromážděte uživatelské požadavky na ukládání výsledků testování a identifikujte funkce nástroje Nitrate, které nejsou podporovány tmt.
3. Vyberte vhodný nástroj doplňující funkcionalitu tmt a navrhněte jeho nastavení a případné změny tak, aby co nejlépe pokryl požadavky uživatelů.
4. Implementujte potřebné funkce, které nejsou nástrojem pokryty a pokuste se alespoň některé z nich nabídnout jako vylepšení do "upstreamu".
5. Získejte zpětnou vazbu od uživatelů, shrňte dosažené výsledky a diskutujte o dalších možnostech rozšíření.

Literatura:

- Red Hat, Inc..: Nitrate -- Test Case Management System, 2013--20. [online, cit.: 10.9.2020]. Dostupné z https://nitrate.readthedocs.io/.
- Šplíchal, P.: fmf, 2015 [online, cit.: 10.9.2020]. Dostupné z https://fmf.readthedocs.io/.
- Red Hat, Inc..: tmt, 2019 [online, cit.: 10.9.2020]. Dostupné z https://tmt.readthedocs.io/.
- Rossel S.: Continuous Integration, Delivery, and Deployment. Packt Publishing, 2017. ISBN 978-1-78728-661-0

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání a alespoň část bodu třetího.

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

Vedoucí práce: **Vojnar Tomáš, prof. Ing., Ph.D.**
Konzultant: Šplíchal Petr, Mgr., RedHatCZ
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 19. května 2021
Datum schválení: 11. listopadu 2020

Abstrakt

Diplomová práca sa zoberá oblasťou testovania softvéru, presnejšie témou správy výsledkov testov. Cieľom práce je nájdenie, nastavenie a implementácia systému dopĺňajúceho chýbajúcu funkcionalitu nástroja TMT, ktorý má v spoločnosti Red Hat nahradiť nástroj Nitrate ako systému pre správu testov a ich výsledkov. Obsahom tejto práce je základné predstavenie nástrojov Nitrate, TMT a technológií používaných v spoločnosti Red Hat. Ďalej je v práci predstavený súčasný stav testovacej infraštruktúry a zozbierané požiadavky užívateľov na nový systém pre správu výsledkov testov. Následne je predstavený nástroj ReportPortal ako systém pre správu výsledkov testov a definovaná chýbajúca funkcionalita. Zvyšok práce sa venuje samotnému nastaveniu systému a implementácii chýbajúcej funkcionality spolu s implementáciou infraštruktúry potrebnej na importovanie výsledkov testovania do ReportPortalu. V práci je taktiež popísaný spôsob nasadenia systému do užívania a spätná väzba od užívateľov. Nasadený systém je vyhodnotený a sú diskutované jeho ďalšie možnosti rozšírenia.

Abstract

This diploma thesis deals with the area of software testing, more precisely with the topic of managing test results. The aim of this work is to find, set up and implement a system that complements the missing functionality of the TMT tool, which is going to replace the Nitrate tool in Red Hat as a test management system. The content of this work is a basic introduction to the tools Nitrate, TMT and other technologies used in Red Hat. Furthermore, the work presents the current state of the test infrastructure and collected user requirements for a new system for managing test results. Subsequently, the ReportPortal tool is introduced as a system for test results management and the missing functionality is defined. The rest of the work is devoted to setting up the system itself and implementing the missing functionality, along with implementing the infrastructure needed to import test results into ReportPortal. The work describes the method of deploying the system in use and feedback from users. The deployed system is evaluated and its further possible improvements are discussed.

Kľúčové slová

Nitrate, Test Management Tool, Flexible Metadata Format, testovanie softvéru, správa výsledkov testov, ReportPortal, AMQ 7, BeakerLib, Bugzilla

Keywords

Nitrate, Test Management Tool, Flexible Metadata Format, software testing, test results management, ReportPortal, AMQ 7, BeakerLib, Bugzilla

Citácia

DUBAJ, Ondrej. *Systém pre správu výsledkov testov doplňujúci nástroj TMT*. Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Tomáš Vojnar, Ph.D.

Systém pre správu výsledkov testov doplňujúci nástroj TMT

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána prof. Ing. Tomáša Vojnara Ph.D. a odborným vedením Mgr. Petra Šplíchala zo spoločnosti Red Hat. Ďalšie potrebné informácie mi poskytol Mgr. Miroslav Vadkerti, taktiež zo spoločnosti Red Hat. Uviedol som všetky literárne pramene, publikácie a dalšie zdroje, z ktorých som čerpal.

.....
Ondrej Dubaj
10. mája 2021

Poďakovanie

Rád by som poďakoval vedúcemu svojej práce prof. Ing. Tomášovi Vojnarovi Ph.D. a odborným konzultantom Mgr. Petrovi Šplíchalovi a Mgr. Miroslavovi Vadkertimu za odbornú pomoc poskytnutú pri realizácii tejto práce.

Ďalej by som rád poďakoval svojmu otcovi a súrodencom za mentálnu a materiálnu podporu počas celého štúdia, svojim priateľom za spoločnú cestu vysokou školou a množstvo nezabudnuteľných zážitkov a v neposlednom rade svojej snúbenici, ktorá mi bola veľkou oporou počas všetkých ťažkých chvíľ.

Obsah

1 Úvod **3**

2 Systém pre správu testov Nitrate a súvisiace technológie **5**
2.1 Nitrate 5
2.1.1 Funkcionalita Nitrate 6
2.1.2 Nitrate Statusy 6
2.2 Unified Message Bus 7
2.3 Bugzilla 9
2.4 BeakerLib 11
2.5 Jenkins 12
2.5.1 Jenkins Pipeline 12

3 Test Management Tool **14**
3.1 FMF – Flexible Metadata Format 14
3.1.1 Popis formátu 15
3.1.2 Vlastnosti formátu 15
3.2 TMT – Test Management Tool 16
3.2.1 Špecifikácia metadát 17
3.2.2 Funkcionalita TMT 21

4 Zhromaždenie požiadaviek a súčasný stav **23**
4.1 Súčasný stav 23
4.2 Zhromaždenie požiadaviek 25

5 Výber vhodného nástroja a návrh zmien **28**
5.1 ReportPortal 29
5.1.1 Štruktúra systému 30
5.1.2 Funkcionalita 31
5.2 Chýbajúca funkcionalita a návrh zmien 36
5.3 Nastavenie systému 38
5.4 Návrh infraštruktúry pre komunikáciu s UMB 40

6 Implementácia infraštruktúry a chýbajúcej funkcionality **42**
6.1 Nástroj ReportPortal 43
6.1.1 Implementácia stavov Untested a Running 43
6.1.2 Úprava zobrazenia histórie a implementácia aditívnych filtrov 44
6.1.3 Implementácia možnosti importu nového formátu XML 45
6.1.4 Implementácia nových kategórii zlyhaní 46

6.1.5 Nastavenie štruktúry testov, uloženie metadát a úpravy užívateľského rozhrania . . . 46
6.2 Testovacia infraštruktúra . . . 47
6.2.1 Implementácia modulu listener pre zachytávanie správ z UMB . . . 47
6.2.2 Implementácia modulu watcher a perzistentnej fronty . . . 48
6.2.3 Implementácia modulu parser a skriptov pre import dát . . . 48

7 Nasadenie systému, testovanie, výsledky a možnosti rozšírenia 51
7.1 Nasadenie systému a testovanie . . . 51
7.1.1 Nasadenie a testovanie prototypu . . . 51
7.1.2 Finálne nasadenie . . . 52
7.2 Výsledky a možnosti rozšírenia . . . 53

8 Záver 55

Literatúra 57

A Obsah priloženého pamäťového média 59

B Štandardizovaný formát ReportPortal Xunit 60

Kapitola 1

Úvod

Testovanie je v súčasnosti dôležitým prvkom akejkoľvek oblasti vývoja softvéru zameranej na dodanie výsledného produktu zákazníkovi. Proces testovania slúži na kontrolu kvality softvérového diela, ktorého cieľom je dosiahnutie požadovanej úrovne z hľadiska funkčnosti, spoľahlivosti, výkonnosti, použiteľnosti a podporovateľnosti.

Veľké spoločnosti kladú na testovanie a kvalitu ich produktov veľký dôraz, čo logicky vyúsťuje do potreby použitia širokej testovacej sady rozdelenej do viacerých kategórii. Tieto kategórie sú kombinované do rôznych testovacích sád, ktoré sú použité v rôznych scenároch a fázach vývoja koncového produktu. Pre správu testov a ich kategórii existujú systémy zacielené pre uchovanie, filtrovanie a manipuláciu s nimi.

Pri potrebe testovania vo veľkom rozsahu vzniká požiadavka na efektívne vyhodnocovanie a užívateľsky prívetivú manipuláciu s výsledkami testovania. Z tohto dôvodu vznikajú popri systémoch pre správu testov nástroje zameriavajúce sa na správu výsledkov testovania. Poskytujú prívetivý spôsob skúmania najmä zlyhaných testov spolu s možnosťami zobrazenia historického chovania testu na rôznych verziách softvéru.

V súčasnej dobe začína počet týchto nástrojov na trhu rásť, avšak veľké spoločnosti si z dôvodu špecifických požiadaviek implementovali vlastné systémy pre správu výsledkov testov, ktorým mnohokrát chýba potrebná funkcionalita vzhľadom na zložitosť problému zastrešenia rôznych testovacích scenárov. Tento prístup vedie k potrebe robiť kompromisy na úkor času testerov a vývojárov, ktorý trávia skúmaním neefektívnej interpretácie výsledkov testovania. V konečnom dôsledku by kvalitný systém s potenciálnou automatizáciou vyhodnocovania typov chýb dokázal ušetriť množstvo času a prostriedkov, čo by viedlo k efektívnejšiemu využitiu času softvérových inžinierov a zníženiu nákladov spoločnosti.

Ideálnou kombináciou pre tvorbu takéhoto systému by bolo využitie voľne dostupného nástroja poskytujúceho potrebné množstvo funkcionality spojeného s úpravou a prípadnou implementáciou jej chýbajúcej časti. Takýto nástroj musí byť dostatočne flexibilný, aby dokázal pokryť špecifické požiadavky onej spoločnosti a ponúkal možnosti rozšírenia pre potenciálny vývoj nových testovacích scenárov.

Témou tejto práce bude hľadanie vhodného nástroja pre správu výsledkov testov, jeho nastavenia a implementácie chýbajúcich častí pre potreby spoločnosti *Red Hat*, ktorý bude dopĺňať funkcionalitu aktuálne vyvíjaného a v *Red Hatu* nasadzovaného nástroja pre správu testov – **Test Management Tool**.

V úvodnej časti práce bude predstavený aktuálne používaný nástroj pre správu testov **Nitrate**, ktorý momentálne slúži aj ako systém pre správu výsledkov testov. Spolu s týmto nástrojom budú predstavené technológie priamo ale aj nepriamo podieľajúce sa na problematike testovania, ako napríklad v *Red Hatu* interne používaný testovací framework

BeakerLib, systém pre zaznamenávanie chýb softvéru **Bugzilla**, komunikačná zbernica **Unified Message Bus (UMB)** a nástroj **Jenkins** používaný pre *Continuous Integration (CI)* a *Continuous Delivery (CD)*.

V nasledujúcej časti sa presunieme k popisu budúceho nástroja pre správu výsledkov **TMT** spolu s definíciou jeho funkcionality a nástroja **FMF**, ktorý popisuje a spravuje metadáta využívané nástrojom **TMT**.

Následne sa pozrieme na súčasný stav a požiadavky užívateľov na systém pre správu výsledkov testov, kde tieto požiadavky budú použité pre nájdenie ideálneho kandidáta. Funkcionalita vybraného nástroja sa porovná s požiadavkami, bude vymedzená jej chýbajúca časť a definované nastavenie tohto systému tak, aby spĺňalo potreby spoločnosti *Red Hat*.

Na základe aktuálneho stavu, získaných uživateľských požiadaviek a ďalšieho potenciálneho smerovania vývoja testovacej infraštruktúry v spoločnosti *Red Hat* bude spomedzi kandidátov vybraný nástroj **ReportPortal** ako systém pre správu výsledkov testov, ktorého vlastnosti budú predstavené. Vzhľadom k jeho aktuálne dostupnej funcionalite budú definované návrhy zmien a implementácie chýbajúcich častí pre účel pokrytia užívateľských požiadaviek. Rovnako bude definovaný návrh infraštruktúry pre komunikáciu s **UMB**, z ktorej bude nástroj ReportPortal získavať dáta o výsledkoch testovania.

Ďalej budú v tejto práci predstavené implementačné detaily zmien finálnej verzie nástroja **ReportPortal**, ako aj testovacej infraštruktúry, kde prototypiálna verzia tohto systému bude vopred nasadená do experimentálneho použitia za účelom overenia správnej funkcionality a doplnenia užívateľských požiadaviek.

V závere bude popísané reálne nasadenia nástroja **ReportPortal** do používania v spoločnosti *Red Hat*. V rámci finálneho zhrnutia sa predstavia výsledky tejto práce spolu so spätnou väzbou od užívateľov a taktiež ďalšie plánované možnosti rozšírenia tohto nástroja pre iné typy testovania alebo zabudovania umelej inteligencie do vyhodnocovania zlyhaných testov.

Kapitola 2

Systém pre správu testov Nitrate a súvisiace technológie

V tejto kapitole sa budeme venovať predstaveniu systému pre správu testov **Nitrate**, ktorý bude v budúcnosti nahradený nástrojom **TMT**, k čomu má aj táto práca dopomôcť. Pokúsime sa popísať jeho funkcionalitu a statusy výsledkov testovania, ktoré ponúka. Ďalej sa budeme venovať komunikačnej zbernici **Unified Message Bus**, ktorý je v spoločnosti *Red Hat* využívaný pre komunikáciu jednotlivých prvkov testovacej infraštruktúry. Následne si predstavíme nástroj **Bugzilla**, ktorý slúži ako vyspelý systém pre zaznamenávanie chýb softvéru ale aj testov. Jeho funkcionalita je využívaná aj pre zaznamenávanie výsledného statusu testovania a prípadne aj ako priestor pre oficiálnu komunikáciu medzi vývojárom a testerom.

Bude tu taktiež rozobraný aj testovací framework **BeakerLib** používaný pre implementáciu väčšiny testov v spoločnosti *Red Hat* a následne popísaný systém **Jenkins** slúžiaci pre implementáciu a integráciu *Continuous Integration (CI)* a *Continuous Delivery (CD)*.

2.1 Nitrate

Nitrate je open-source systém určený primárne pre správu testov, testovacích plánov a ich jednotlivých spustení. Pre implementáciu bol využitý jazyk Python v spolupráci s webovým frameworkom *Django*.

Jeho výhoda je najmä v podpore veľkého spektra funkcionality. Ako príklad je možné uviesť široké možnosti vytvárania jednotlivých testov, správy životného cyklu testov a testovacích plánov a možnosti detailnej analýzy spustení a výsledkov jednotlivých testovacích plánov, nehovoriac o efektívnej podpore filtrovania vyššie spomenutých objektov. Za zmienku stojí aj prepracovaná kontrola prístupu v spolupráci s konfigurovateľnými autentifikačnými logikami. Je pripravená aj možnosť použitia systému pre zaznamenávanie interných alebo externých chýb konkrétnych testov alebo testovacích plánov. Spomenutá funkcionalita je prístupná pomocou *XMLRPC* aplikačného rozhrania.

V súčasnosti systém podporuje použitie rôznych typov databáz, avšak medzi odporúčané patria *MariaDB*, *MySQL* a *PostgreSQL*, kde konkrétne podporované verzie je možné nájsť v priloženej dokumentácii [18]. Pre chod systému Nitrate je potrebné použitie interpreta jazyka *Python* o minimálnej verzii 3.6 a frameworku *Django* verzie 2.x [18].

2.1.1 Funkcionalita Nitrate

Ako bolo už vyššie spomenuté, Nitrate ponúka široké spektrum funkcionality. Okrem možností správy užívateľov a prostredia pre beh testov budú pre naše účely podstatné objekty vzťahujúce sa k správe testovania. Medzi základné patria [18]:

- **Test-case** – predstavuje jeden konkrétny test definovaný jednoznačným identifikátorom, ktorému je možné priraďovať atribúty a relevanciu pre jednotlivé distribúcie. Jeho výhoda spočíva v možnosti využitia naprieč testovacími plánmi, avšak je nutné zadefinovanie metadát nanovo.
- **Test-plan** – predstavuje jednoznačne identifikovanú skupinu testov tvoriacich plán pre testovanie. Nitrate umožňuje vytvárať stromovú štruktúru testovacích plánov, kde zároveň ponúka funkcionalitu pre ich správu, exportu do formátu XML a možnosť klonovania ako pri testoch.
- **Test-run** – predstavuje konkrétny beh testovacieho plánu pre definovanú verziu OS a architektúry, kde každý test má priradený výsledok z množiny podporovaných statusov testov. Konkrétne statusy testov a ich graf prechodov budú predstavené v nasledujúcej časti.

Medzi jednotlivými objektami je možné efektívne filtrovať, pridávať atribúty, sledovať a upravovať metadáta. Bežným prípadom použitia je aj preskúmanie zlyhaných testov, určenie o aký typ zlyhania sa jedná, pridanie komentáru a prípadne aj vytvorenie ticketu v trackovacom systéme a priloženie odkazu na neho.

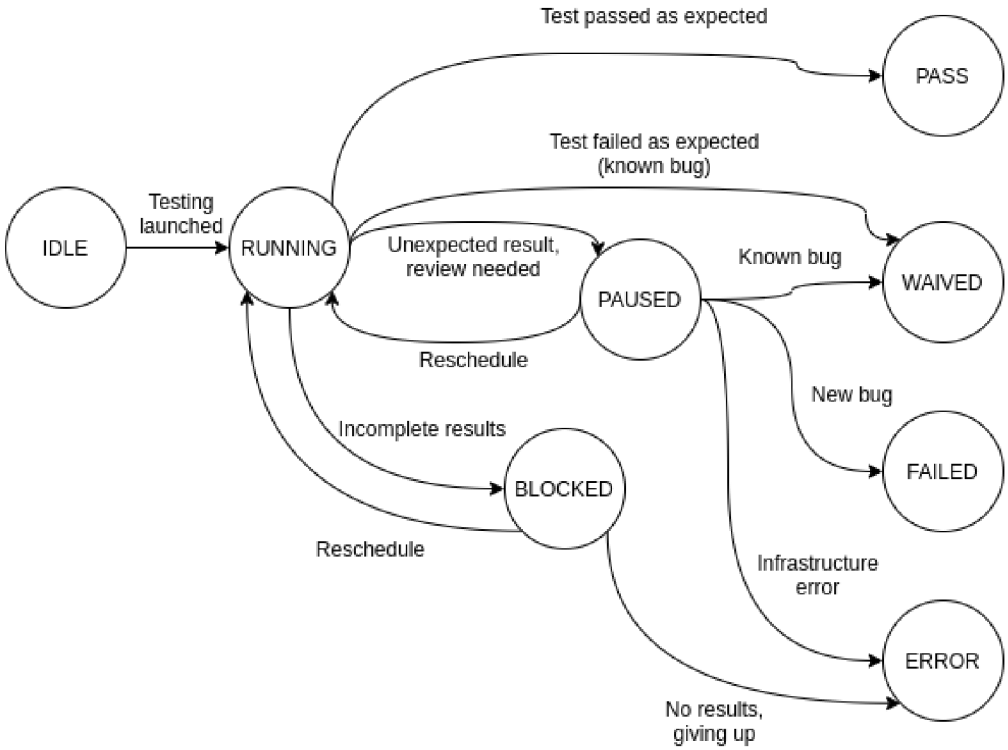
V súčasnosti je Nitrate využívaný ako hlavný systém pre správu CI testov a ich výsledkov v spoločnosti *Red Hat* v rámci RHEL produktov, avšak už nespĺňa požiadavky užívateľov, ktorí volajú po zmene. Touto zmenou by mal byť v budúcnosti nástroj TMT, ktorý je momentálne vo vývoji a v postupnom nasadzovaní pre rôzne komponenty za účelom ujasnenia požiadaviek a overenia funkcionality.

2.1.2 Nitrate Statusy

Ako bolo v predchádzjúcej časti spomenuté, každý test vykonaný v rámci jedného behu testov má priradený status, ktorý popisuje aktuálny stav testovania. Statusy su používané ako pre manuálne, tak aj pre automatické testy a delíme ich na koncové a prechodové [4], kde prechody medzi jednotlivými stavmi testov sú graficky znázornené na obrázku 2.1:

- **IDLE** – iniciálny stav testov pred spustením testovania;
- **RUNNING** – exekúcia testov prebieha;
- **PAUSED** – test skončil v neočakávanom stave a je potrebné preskúmanie testerom;
- **BLOCKED** – exekúcia testu neprebehla celá, výsledky nie sú kompletné, napríklad z dôvodu nedodania stroja s potrebnou architektúrou;
- **PASS** – koncový stav, test skončil úspechom;
- **WAIVE** – koncový stav, test skončil neúspechom, avšak je to nepodstatné zlyhanie alebo známy problém;
- **ERROR** – koncový stav, neúspech z dôvodu zlyhania testovacej infraštruktúry;

- **FAIL** – koncový stav, test skončil neúspechom z dôvodu chyby v teste alebo komponente.



Obr. 2.1: Prechodový diagram statusov nástroja Nitrate

Problémom u pevne definovaných statusov je nízka miera flexibility pri vytváraní a priraďovaní špeciálnych typov chýb v prípade neúspechu. Ako príklad je možné uviesť rôzne typy testov pre grafické rozhranie alebo nízkoúrovňové komponenty systému ako napríklad jadro OS Linux. Tieto tímy musia používať rovnakú sémantiku statusov, čo nie je v každom prípade správne a jednoznačné a následne vedie k potrebe tvorby kompromisov. Cieľom nového systému bude taktiež možnosť definovať určité tímovo špecifické označenie stavov testov, aby bola interpretácia výsledkov užívateľsky prívetivejšia.

2.2 Unified Message Bus

Unified Message Bus je interný názov pre komunikačnú zbernicu pre zasielanie správ medzi uzlami. Je to praktické využitie middlewaru **Red Hat AMQ** dodávaného spoločnosťou *Red Hat* pre komunikáciu jednotlivých prvkov testovacej infraštruktúry a prvkov verifikácie dodávaného softvéru. Samotný middleware sa skladá z niekoľkých častí, ktoré spolu tvoria funkčný komunikačný systém [2].

AMQ Broker je vysoko výkonná komunikačná zbernica vychádzajúci z open-source projektu *ActiveMQ Artemis*. Poskytuje rýchle, jednoduché a bezpečné prostredie pre zasielanie správ medzi web-based aplikáciami. Predstavme si komponenty AMQ ako nástroje vo vnútri určitej sady. Tieto nástroje môžu byť použité spoločne alebo oddelene na vytvorenie

aplikácie na odosielanie správ a protokol AMQP je lepidlo, ktoré ich spája. Komponenty AMQ zdieľajú spoločnú konzolu pre správu, takže ich môžeme spravovať z jedného rozhrania.

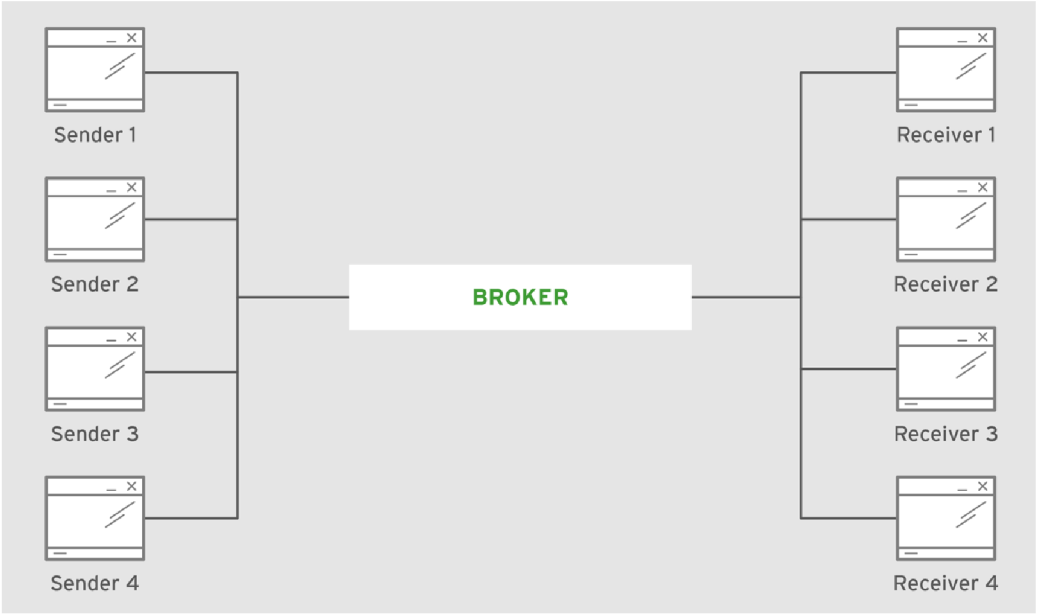
AMQ Interconnect využíva protokol AMQP na distribúciu a smerovanie správ medzi koncovými bodmi, vrátane klientov, zberníc a služieb, kde tieto koncové zariadenia AMQ poskytujú aplikačné rozhranie pre zasielanie a prijímanie správ. Jediným pripojením do siete si môže klient vymieňať správy s akýmkoľvek iným koncovým bodom pripojeným k sieti.

AMQ Interconnect pre zabezpečenie dostupnosti nepoužíva klastre typu master-slave, ale je spravidla nasadený v topológiách viacerých smerovačov s redundantnými sieťovými cestami, ktoré používa na zabezpečenie spoľahlivého pripojenia. Dokáže taktiež distribuovať pracovné zaťaženie prostredníctvom správ a dosiahnuť nové úrovne rozsahu s veľmi nízkou latenciou.

AMQ Klient je koncový bod danej toplógie poskytujúci aplikačné rozhranie pre prijímanie a odosielanie správ pre rôzne jazyky a platformy. Zahŕňa podporu nového API založeného na udalostiach, ktoré umožňujú integráciu do existujúcich aplikácií.

Poskytuje flexibilné smerovanie správ medzi koncovými bodmi s povoleným AMQP, vrátane klientov, maklérov a samostatných služieb. Jediným pripojením do siete smerovačov AMQ Interconnect si môže klient vymieňať správy s akýmkoľvek iným koncovým bodom pripojeným k sieti.

Red Hat AMQ ponúka viacero využiteľných topológii pre stavbu komunikačného systému. My si predstavíme topológiu využívanú pre chod UMB [6] vyobrazenú na obrázku 2.2 zahŕňajúca jeden centrálny prvok (Broker), ktorý distribuuje správy medzi jednotlivými koncovými bodmi. Vzhľadom na to, že sa jednotlivé koncové prvky nachádzajú na jednej sieti, nie je potrebné riešiť zbytočne komplikované návrhy topológií, ich udržovateľnosť a taktiež ani množstvo pripojených koncových bodov. Jediný kritický element v tejto zostave je centrálny prvok, na ktorý sa sústreďuje úsilie údržby tak, aby bol k dispozícii klientom.



Obr. 2.2: Topológia AMQ využitá pre implementáciu Unified Message Bus [2]

2.3 Bugzilla

Bugzilla je robustný, inovatívny a vyspelý trackovací systém umožňujúci vývojárom efektívne a prehľadné sledovanie a zaznamenávanie chýb, problémov a požiadaviek. Jednoduché funkcie na sledovanie chýb sú často zabudované do integrovaných prostredí na správu zdrojových kódov, akým je napríklad *GitHub* alebo iné webové a lokálne nainštalované ekvivalenty. V prípade, že požiadavky na zaznamenávanie chýb prerastajú možnosti týchto systémov, mnohé spoločnosti sa obracajú na externé trackovacie systémy. Príkladom vyspelých požiadaviek môže byť workflow manažment, správa prístupu užívateľov k jednotlivým chybám alebo možnosť tvorby vlastných položiek formulárov. Narozdiel od svojich priamych konkurentov, je Bugzilla open-source, kde vďaka širokej komunite ponúka väčšie množstvo praktickej funkcionality, ktorá je vyžadovaná priamo od vývojárov z praxe. Jedna z mála nevýhod tohto systému je absencia verejne dostupných inštancií na webe a je nutné ju mať pre správne používanie nainštalovanú na vlastnom serveri [16].

V rámci spoločnosti *Red Hat* je vytvorený vlastný workflow manažment pre správu chýb, ktorý je založený primárne na kľúčových slovách vkladaných do daného trackeru, jeho stave, prioritou, prítomnosti indikátorov a nastavením **ITM** (*Internal Target Milestone*) a **ITR** (*Internal Target Release*).

Indikátory v Bugzille slúžia na potvrdenie určitej schopnosti vývojárov. Najpoužívanejšie sú dva základné typy:

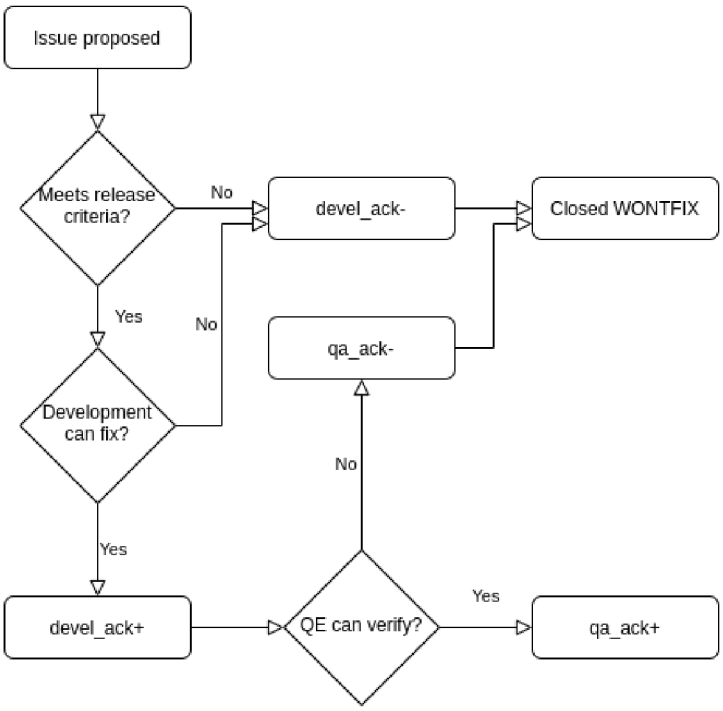
- **devel_ack** – vývojár potvrdzuje, že je schopný daný problém vyriešiť v rámci konečného termínu pre daný produkt;
- **qa_ack** – tester potvrdzuje, že je schopný chybu a jej opravu verifikovať v rámci konečného termínu pre daný produkt.

Dôležitým prvkom pre nastavovanie indikátorov je priorita danej požiadavky (Low, Medium, High, Urgent). Na základe nej sa produktový manažment a vývojár rozhodujú, ktorým požiadavkám vyhovejú a ktorým nie. Pre ďalšie vyhodnotenie a plánovanie práce sa využíva **ITR** a **ITM**, ktorých význam v tejto práci nemôže byť vysvetlený, nakoľko sa jedná o interný detail spoločnosti *Red Hat*. Prehľad nastavovania indikátorov je možné vidieť na priloženom obrázku 2.3.

Kľúčové slová určujú o aký typ chyby sa jedná (Security bug, Enhancement, Bugfix) a taktiež ukazujú potrebu implementácie automatického testu, resp. jeho prítomnosť. Na základe kľúčových slov a testu je vývojár schopný reprodukovať a opraviť daný problém, aby následne mohla prebehnúť oficiálna verifikácia.

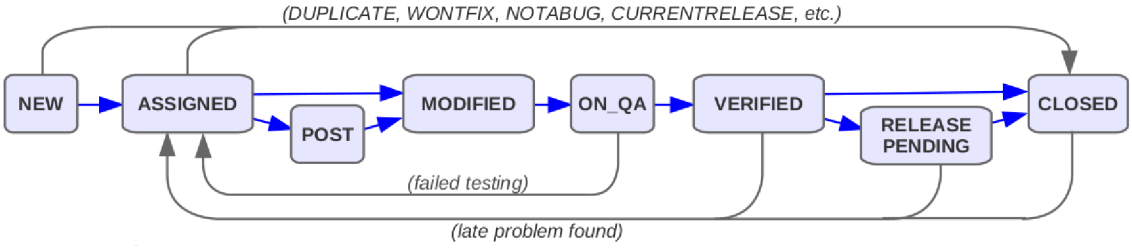
Stav trackeru určuje v ktorej fáze vývoja/opravy sa problém aktuálne nachádza. Používané sú nasledujúce stavy [1], kde ich prechodový diagram je zobrazený na obrázku 2.4:

- **NEW** – bol vytvorený tracker pre problém, ktorý musí byť preskúmaný vývojárom;
- **ASSIGNED** – tracker bol pridelený konkrétnemu vývojárovi, je dobre popísaný, avšak nie vyriešený;
- **POST** – daný problém má konkrétne riešenie, čaká však na kontrolu od zákazníka alebo iného vývojára;
- **MODIFIED** – problém bol vyriešený a otestovaný jednotkovými testami;



Obr. 2.3: Workflow nastavenia devel_ack a qa_ack

- **ON_QA** – problém je dostupný testerom pre oficiálnu verifikáciu pokročilými testami;
- **VERIFIED** – problém bol úspešne verifikovaný;
- **RELEASE_PENDING** – problém je pripravený na dodanie zákazníkovi;
- **CLOSED** – problém uzavretý, kde dôvodom môže byť nevôľa riešiť (WONTFIX), neexistencia problému (NOTABUG), duplikovaný problém (DUPLICATE), problém vyriešený (ERRATA) a mnoho iných.



Obr. 2.4: Prehľad možných prechodov jednotlivých stavov

2.4 BeakerLib

BeakerLib je knižnica pre testovanie poskytujúca komfortné funkcie, ktoré zjednodušujú písanie, beh a analýzu integračných a jednotkových testov v prostredí shellu. Je využívaná spoločnosťou *Red Hat* ako jeden z hlavných prostriedkov v rámci verifikácie [13].

Knižnica bola vyvinutá skupinou testerov a vývojárov v spoločnosti *Red Hat* a bola širokej komunite sprístupnená ako open-source softvér pre potenciálne zapojenie *Fedora* komunity do tvorby testov a ich následným využitím pre skvalitnenie distribúcie *Red Hat Enterprise Linux*. Medzi základné vlastnosti patrí [17]:

Žurnálovanie Jednotný mechanizmus logovania, kde logy a výsledky testov sú uložené vo flexibilnom XML formáte, čo umožňuje ľahké generovanie správ a porovnanie výsledkov rozšírené o informačné správy. Journaling podporuje taktiež metrické informácie o teste, ako napríklad spotreba pamäte, čas behu a podobne.

Fázovanie Možnosť logického zoskupenia testovacích príkazov do fáz. Jednotlivé fázy nie sú pevne dané a je možné implementovať vlastné rozdelenie fáz podľa potreby, avšak štandardne sú používané 3 základné fázy - *Setup*, *Test* a *Cleanup*.

Kontroly Kontrola návratových hodnôt, existencie súboru alebo jeho obsahu alebo prítomnosti nainštalovaných balíkov. Existuje tu možnosť vytvorenia manuálnych assertov pre účely manuálneho testovania.

Funkcie riadenia Pohodlné funkcie pre bežné operácie, ako správa služieb a socketov, zálohovanie, obnova stavu, reštart systému, inštaláciu potrebných balíkov, synchronizáciu procesov alebo behu serveru.

Vo výpise 2.1 je uvedený príklad jednoduchého testu implementovaného v knižnici BeakerLib, ktorý pripraví prostredie, spustí testovací skript a porovná výsledky. Na záver vráti prostredie do pôvodného stavu vymazaním dočasného adresáru.

```
1  rlJournalStart
2      rlPhaseStartSetup
3          rlAssertRpm $PACKAGE
4          rlRun "TmpDir=\‘mktemp -d\‘" 0 "Creating tmp directory"
5          rlRun "pushd $TmpDir"
6      rlPhaseEnd
7
8      rlPhaseStartTest
9          rlRun "echo ’perror foo’ > b.exp; runtest a.exp b.exp" 0
10     rlAssertGrep "PASS: bar" testrun.sum
11     rlPhaseEnd
12
13     rlPhaseStartCleanup
14         rlRun "popd"
15         rlRun "rm -r $TmpDir" 0 "Removing tmp directory"
16     rlPhaseEnd
17 rlJournalEnd
```

Výpis 2.1: Ukážka testu vo frameworku BeakerLib

2.5 Jenkins

Jenkins je samostatný a automatizovaný server s otvoreným zdrojovým kódom použiteľný na automatizáciu rôznych úloh súvisiacich s vývojom, testovaním a dodávkou alebo nasadením softvéru. Vďaka jeho implementácii v jazyku *Java* ponúka široké možnosti inštalácie a behu pomocou natívnych systémových balíkov, *Docker*, *Kubernetes* alebo akéhokoľvek stroja s nainštalovaným prostredím *JRE*. Príkladom môže byť beh v prostredí Java servlet kontajnerov ako *Apache Tomcat* alebo *Glassfish*. Výhodou je taktiež veľké spektrum možností inštalácie zásuvných modulov pre rozšírenie funkcionality [7].

2.5.1 Jenkins Pipeline

Jenkins Pipeline je základnym kameňom funkcionality systému Jenkins. Je to sada zásuvných modulov podporujúca implementáciu a integráciu continuous delivery pipeline do Jenkinsu.

Continuous delivery pipeline predstavuje automatický beh procesu pre zostavenie, verifikáciu softvéru a jeho možnosť dodania zákazníkovi v každom okamihu vývoja. Pipeline procesy sú zväčša zložené z viacerých krokov, ktoré užívateľom ponúkajú možnosti modelovať akýkoľvek druh automatizačného procesu. Krok procesu si je možné predstaviť ako príkaz, ktorý vykoná akciu. Keď je krok úspešný, prejde na ďalší krok. Ak sa niektorý krok nepodarí správne vykonať, pipeline zlyhá. Ak sú všetky kroky procesu úspešne dokončené, považuje sa pipeline za úspešne vykonanú [12].

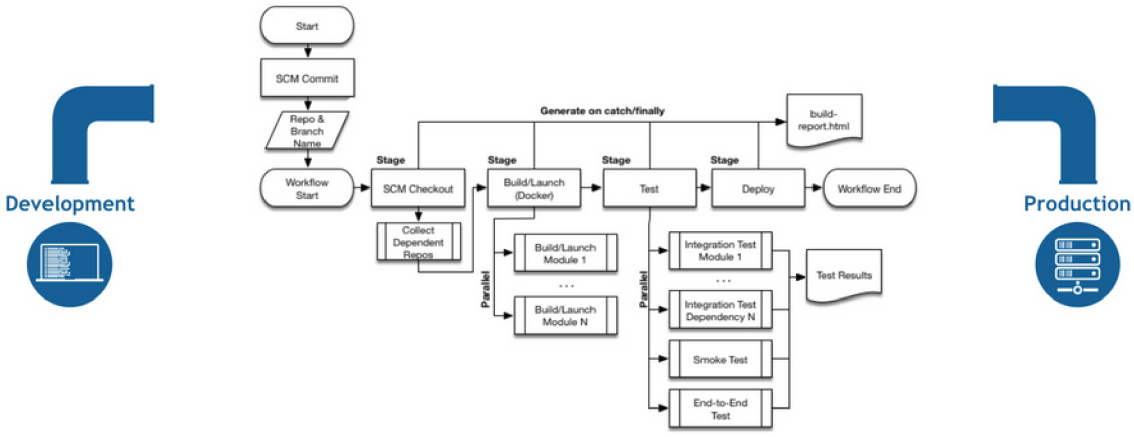
Každá zmena zdrojového kódu softvéru prechádza zložitým procesom na ceste k vydaniu. Tento proces zahŕňa spoľahlivú a opakovanú kompiláciu kódu ako aj viacfázové testovanie pred automatickým sprístupnením zmien zákazníkom, kde takýto jednoduchý proces je možné vidieť na obrázku 2.5. Medzi popredné výhody Jenkins Pipeline patria:

- implementácia a nastavenie procesov priamo v repozitároch zdrojového kódu,
- odolnosť voči plánovaným aj neplánovaným reštartom,
- pozastaviteľnosť v prípade potreby zmeny počas behu pipeline procesu,
- univerzálnosť v zmysle podpory jednoduchých, ale aj zložitých continuous-delivey procesov,
- rozšíriteľnosť v podobe možnosti využitia, ale aj vlastnej implementácie zásuvných modulov.

Jenkins Pipeline poskytuje rozšíriteľnú sadu nástrojov na modelovanie zložitých doručovacích procesov. Je definovaná zvyčajne v textovom súbore zvanom Jenkinsfile, ktorý je uložený v repozitároch zdrojových kódov projektu. Tento prístup umožňuje aplikovať princíp **“Pipeline-as-code”**, čo znamená, že je s ňou zaobchádzané ako so súčasťou zdrojového kódu [7].

Vytvorenie konfiguračného súboru Jenkins Pipeline a jeho pridanie do repozitáru k zdrojovému kódu prináša hneď niekolko benefitov:

- automatické vytváranie pipeline procesu pre všetky vetvy repozitára,
- revízia kódu a audit,
- „Single source of Truth“ pre daný pipeline proces, ktorý je prístupný všetkým vývojárom projektu.



Obr. 2.5: Príklad behu jednoduchého pipeline procesu [14]

Jenkinsfile je možné implementovať v rôznych programovacích jazykoch (Ruby, Go, Python, Java,...) ako aj v dvoch syntaktických štýloch – **deklaratívnom** a **skriptovacom**. Funkcionalitou sa takmer nelíšia, avšak v súčasnosti je preferovaný novší deklaratívny syntax, ktorého príklad je možné vidieť vo výpise 2.2.

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                make build
            }
        }
        stage('Test') {
            steps {
                make test
            }
        }
        stage('Deploy') {
            steps {
                make deploy
            }
        }
    }
}
```

Výpis 2.2: Príklad jednoduchého konfiguračného súboru Jenkinsfile

Kapitola 3

Test Management Tool

V nasledujúcej kapitole bude predstavený novovyvíjaný nástroj TMT, ktorý má v budúcnosti nahradiť nástroj Nitrate ako systém pre správu testov, spolu s nástrojom FMF poskytujúcim flexibilný formát pre definíciu metadát, ktorý TMT využíva v rámci svojej funkcionality. Budú popísané ich vlastnosti a funkcionalita aj s príkladmi.

3.1 FMF – Flexible Metadata Format

FMF je open-source nástroj príkazového riadku implementujúci flexibilný formát pre definíciu metadát v textových súboroch, ktoré je možné ukladať blízko zdrojového kódu testovaných aplikácií. Je vyvíjaný vývojármi spoločnosti *Red Hat* najmä pre použitie s nástrojom **TMT**, ktorý bude detailnejšie popísaný v nasledujúcich kapitolách. Pre dizajn formátu boli použité tieto základné princípy [20]:

- správa FMF súborov vo verzovacích systémoch,
- hierarchizácia,
- zabránenie duplikácie,
- metadáta blízko kódu aplikácie aj testov,
- možnosť využitia v rámci open-source komunity,
- zameranie sa na základné a najzvyčajnejšie prípady použitia.

Vzhľadom na neustále narastajúce množstvo testovaného kódu, s tým súvisiace množstvo testov a efektívne vykonávanie testov je nevyhnutné udržiavať zodpovedajúce metadáta definujúce aspekty vykonávania testu. Hierarchická štruktúra metadát podporuje možnosti dedenia, ako ho poznáme z objektovo-orientovaného programovania, čo znižuje redundanciu, zefektívňuje udržateľnosť a dovoľuje vytvárať rozsiahle a usporiadané dokumenty.

Aj keď bol formát pôvodne navrhnutý podľa požiadaviek užívateľov sústredených okolo problematiky testovania, je všeobecný a preto je možné ho použiť aj v širších scenároch, napríklad pre mapovanie pokrytia kódu testami. Pomocou tohto prístupu je možné kombinovať metadáta vykonania testu a taktiež informácie o pokrytí testu. Za zmienku stojí aj možnosť modifikácie metadát na základe aktuálneho kontextu. Je možné si to predstaviť ako vyhodnotenie relevantnosti testu a jeho následná úprava počas behu [20].

3.1.1 Popis formátu

FMF je formát slúžiaci pre serializáciu metadát vo forme, ktorá je pre človeka ľahko čitateľná. Vychádza štandardizovaného formátu YAML a obsahuje primitíva ako zoznamy a asociatívne polia.

Základným kameňom je súbor *main.fmf*, ktorý funguje podobne ako súbor *index.html* v rámci webových stránok. Definuje totiž metadáta najvyššej úrovne pre daný adresár. Formát FMF, podobne ako YAML, nedefinuje názvoslovie atribútov, čím umožňuje prispôsobiť formát akémukoľvek projektu. Jediná výnimka je už spomínaný súbor *main.fmf* [20].

Metadáta formujú súborový strom, v ktorom je možné uplatniť dedičnosť. Koreň je definovaný adresárom *.fmf/* podobne ako adresár *.git/*, ktorý musí obsahovať minimálne súbor *version* obsahujúci číslo verzie formátu.

Názvy uzlov sú v rámci stromu metadát jedinečné a je ich možné použiť ako identifikátory pre lokálne odkazovanie v rámci daného stromu. Aby bolo možné odkazovať na vzdialené FMF uzly z iných stromov, je úplny identifikátor FMF definovaný ako slovník obsahujúci kľúče s následujúcim významom [20]:

- **url** – úložisko git repozitára obsahujúce strom metadát. Je možné použiť ľubovolný formát akceptovaný príkazom *git clone*. Ak nie je poskytnutá url, predpokladá sa lokálny strom.
- **ref** – vetva, tag alebo commit špecifikujúca požadovaný stav repozitára. Je potrebná pre vykonanie príkazu *git checkout* v rámci repozitáru, avšak predvolenou variantou je vetva *default*.
- **path** – cesta ku koreňu v strome metadát. Mala by byť relatívna ku koreňu stromu ak je poskytnutá URL git repositára, inak musí byť absolútna vzhľadom k súborovému systému.
- **name** – názov uzla definovaný hierarchiou v strome metadát.

3.1.2 Vlastnosti formátu

V tejto časti predstavíme základné vlastnosti formátu FMF, ktoré sú aktívne využívané pri práci s nástrojom TMT v rámci spoločnosti *Red Hat* [20] a sú prakticky znázornené vo výpise 3.1.

Jednoduchosť Základom je jednoduchosť čitateľnosti a modifikácie pre užívateľa.

Hierarchizácia Je definovaná adresárovou štruktúrou a explicitným zanorovaním využívajúcim atribúty s predponou /. Je možné definovať metadáta pre viacero testov v jednom *.fmf súbore.

Dedičnosť Metadáta sa môžu dediť od rodičovských objektov.

Zlučovanie Pri dedení hodnôt od rodičovského objektu je možné použiť špeciálne prípony atribútov na zlúčenie hodnoty potomka s údajmi rodiča. Je možné použiť operátory spojenia (+) aj odobrania (-).

Elasticita Je možné použiť jeden súbor alebo rozptýliť metadáta v celej hierarchii podľa toho, čo je pre projekt prínosnejšie.

Virtualizácia Použitie rovnakého testovacieho skriptu pre testovanie rôznych scenárov môže byť ľahko implementované použitím metadát uložených v listoch dediacich od rovnakého rodiča.

Prispôsobivosť Je možné upraviť hodnoty atribútov na základe aktuálneho kontextu, ako napríklad deaktivovať test, ak nie je relevantný pre dané prostredie.

```
# rodicovsky objekt FMF metadat (koren FMF stromu)
tester: Petr Splichal <psplicha@redhat.com>
tags: [TierSecurity]
test: runtest.sh
time: 3 min

# hierarchizacia a dedicnost objektov
# elasticita spociva v rozdeleni metadat do viac suborov v hierarchii
/download-fast: # virtualizacia
    description: Check basic download options (quick smoke test)
    environment: MODE=fast
    tags: [Tier1]
/download-full: # virtualizacia
    description: Check basic download options (full test set)
    environment: MODE=full
    tags: [Tier2]
    time: 30 min
/recursion:
    description: Check recursive download options
    time+: 3 # zlucovanie
    enabled: true
    adjust: # prisposobivost
        enabled: false
        when: distro < fedora-33
        because: the feature was added in Fedora 33
```

Výpis 3.1: Príklad použitia vlastností formátu FMF

3.2 TMT – Test Management Tool

Test Management Tool je nástroj pre správu testov a testovacích plánov, ktorý je nezávislý od externých systémov pre správu testov a ich výsledkov. Tento nástroj poskytuje užívateľsky príjemný spôsob vytvárania, ladenia a ľahkého spúšťania testov v rôznych prostrediach. Umožňuje tiež ľahkú konverziu starých metadát, filtrovanie dostupných testov a ich overenie podľa špecifikácie Level 1 (L1), ktorá bude bližšie vysvetlená v nasledujúcej podsekcii.

Plány sa používajú na zoskupenie súvisiacich testov a konfiguráciu jednotlivých krokov testu definovaných v špecifikácii Level 2 (L2). Opisujú, akým spôsobom sa majú testy

vykonávať, ako zabezpečiť prostredie, ako ho pripraviť na testovanie alebo ako by by mali byť interpretované výsledky testov.

Požiadavky užívateľov pre daný nástroj sú zaznamenané vo forme *user-stories*, ktoré sú definované podľa špecifikácie Level 3 (L3). Je možné ich využiť pre sledovanie implementácie, testovania a pokrytia dokumentácie pre jednotlivé funkcie alebo požiadavky. Vďaka tomu môžeme sledovať všetko na jednom mieste, vrátane priebehu implementácie projektu [21].

3.2.1 Špecifikácia metadát

Špecifikácia metadát definuje spôsob uloženia matadát, potrebných pre beh testov, v git repozitároch blízko zdrojového kódu testovaných komponent alebo testovacieho kódu. Pre definíciu metadát pre TMT sa využíva formát FMF predstavený v predchádzajúcej časti. TMT týmto metadátam dodáva sémantiku a definuje niekoľko úrovní špecifikácii metadát [21].

Level 0

Popisujú základné obecné atribúty ako popis alebo poradie jednotlivých fáz, ktoré je možné použiť na všetkých úrovniach metadát. Nazývame ich tiež aj **core-metadata**. Vychádzajú z požiadaviek užívateľov mať k dispozícii všeobecné atribúty, ktoré je možné použiť na všetkých úrovniach [21]. Patria medzi ne primitíva ako atribút **adjust** pre dynamickú zmenu metadát za behu, poradie testov (**order**), popis, alebo alebo stručné zhrnutie. Príklad použitia týchto metadát je možné vidiet vo výpise 3.2.

```
1 tester: Petr Splichal <psplicha@redhat.com>
2 description: Check basic download options (quick smoke test)
3 summary: A brief summary of the test.
4 adjust:
5     enabled: false
6     when: distro ~< fedora-33
7     because: the feature was added in Fedora 33
```

Výpis 3.2: Ukážka použitia Level 0 metadát

Level 1

Predstavujú metadáta a informácie úzko súvisiace s jednotlivými testami, ako napríklad spôsob spustenia testovacieho skriptu (**test**), cesta k adresáru (**path**), maximálna doba trvania alebo tagy, ktoré sú uložené priamo s kódom testu. Pre každý atribút je štandardizované jednoznačné a unikátne označenie, ktoré vyplynulo z užívateľských požiadaviek. Ďalej definujú nastavenie premenných prostredia pre beh testu (**environment**), použitý testovací framework (**framework**), závislosti (**require**), interpretáciu výsledku testu (**result**) a informáciu o tom, či je test manuálny alebo automatizovaný (**manual**) [21]. Väčšina vyššie uvedených vlastností je znázornená vo výpise 3.3.

```
1 component: automake
2 test: ./runtest.sh
3 path: /rebuild-srpm
4 framework: beakerlib
```

```
require:
    - url: https://github.com/beakerlib/distribution.git
        name: /RpmSnapshot
        ref: master
    - bash
    - bzip2
    - coreutils
tag:
     - dependencies
duration: 3h
result: respect
manual: false
```

Výpis 3.3: Ukážka použitia Level 1 metadát

Level 2

Táto úroveň predstavuje testovacie plány zložené z jednotlivých testov, ktoré popisujú, ako zabezpečiť a pripraviť prostredie pre bezproblémový priebeh testovacieho plánu ako celku. Ponúka možnosť definovania premenných prostredia (**environment**), ktoré budú platiť pre všetky testy zaradené do daného plánu, podmienky pre gating balíku (**gate**), ktoré umožnia ďalší posun aktualizácie v rámci *continuous delivery* a taktiež kontext (**context**), ktorý popisuje všetky štandardné dimenzie kontextu pre beh testov v rámci testovacieho plánu. V rámci kontextu je zadefinovaná použitá distribúcia operačného systému pre testovanie, architektúru, alebo variantu testovacieho balíka [21], kde názorný príklad použitia je vidieť vo výpise 3.4.

```
context:
    distro: fedora-33
    variant: Workstation
    arch: x86_64
environment:
    KOJI_TASK_ID: 42890031
    RELEASE: f33
gate:
    - merge-pull-request
    - add-build-to-update
    - add-build-to-compose
```

Výpis 3.4: Ukážka použitia Level 2 metadát

Level 3

Popisujú predstavy a plány užívateľov pre definovanie očakávaných funkcií projektu. Slúžia tiež na sledovanie plánovanej, implementovanej, testovanej a zdokumentovanej funkcionality. Majú obvykle formu user-stories s ďalšími pridanými atribútmi popisujúcimi stav danej požiadavky, čo je možné názorne vidieť vo výpise 3.5 . Obsahujú krátky popis (**description**), informáciu o dokumentácii (**documented-by**), jednoduchý príklad použitia (**example**), pokrytie testami (**verified-by**), stav implementácie (**implemented-by**), prioritu a samotnú user-story (**story**) [21].

```
story:
    As a user I want to see more detailed information for
    particular command.
description:
    Different verbose levels can be enabled by using the
    option several times.
example:
  - tmt test show -v
  - tmt test show -vvv
  - tmt test show --verbose
link:
  - implemented-by: /tmt/cli.py
  - documented-by: /tmt/cli.py
  - verified-by: /tests/core/dry
priority: must
```

Výpis 3.5: Ukážka použitia Level 3 metadát

Kroky testovania

Definícia metadát o krokoch testovania spadá pod kategóriu *Level 2*, avšak je špecifická tým, že tieto údaje bývajú definované priamo pri zdrojovom kóde komponent a udávajú z ktorých miest má nástroj **TMT** zbierať dáta o testovaní. Týmto spôsobom sú zhromaždené informácie pre stavbu testovacieho plánu, ktoré sú ďalej exekuované.

Jasné oddelenie fáz testovania dáva používateľom kontrolu nad ich individuálnymi aspektmi, kde každý krok objasňuje, čo a ako je možné v konkrétnej etape procesu ovplyvniť. Tento prístup tiež umožňuje spustiť podľa potreby iba vybrané kroky. Napríklad spustením kroku **discover** je možné vidieť, ktoré testy sa vykonajú. Takisto je možné preskočiť kroky **provision** a **prepare** pre rýchle testovanie na lokálnom stroji. Každý krok môže byť podporený viacerými implementáciami definovanými špeciálnym kľúčovým slovom **how**. Medzi základné kroky patria [21]:

Discover Zhromaždí relevantné testy a informácie o nich pre daný beh testov. Poskytuje metódy *tests()* vracajúce zoznam objavených testov a *requires()* vracajúce zoznam všetkých požadovaných balíkov agregovaných z atribútu **require** jednotlivých metadát testu, tak ako je to uvedené vo výpise 3.6. Pre zhromaždenie je možné použiť FMF metadáta alebo manuálne definovaný testovací plán skladajúci sa zo zoznamu adresárov obsahujúcich meno testu a testovací skript.

```
discover:
    how: fmf
    url: https://github.com/psss/tmt
    ref: master
    path: /metadata/tree/path
    test: [regexp]
    filter: tier:1
    require: gcc
```

Výpis 3.6: Ukážka zadefinovania fázy Discover

Provision Definuje prostredie, ktoré by malo byť zabezpečené pre testovanie. Poskytuje všeobecný a rozšíriteľný spôsob zapisovania základných hardvérových a softvérových požiadaviek (príklad dostupný vo výpise 3.7). Je možné využiť prostredie Beaker pre poskytnutie virtuálneho stroja, pripojenie sa na predom definovaný virtuálny stroj alebo kontajner, využitie lokálneho virtuálneho stroja alebo aj samotného lokálneho počítača. Je taktiež možné definovať hadvérové požiadavky pre každú variantu.

```
provision:
    how: container
    image: fedora:latest
    memory: 8 GB
    disk: 500 GB
    cpu:
        model_name: =~ .*AMD.*
        processors: 2
        cores: 16
        model: 37
```

Výpis 3.7: Ukážka zadefinovania fázy Provision

Prepare Príprava a konfigurácia dodaného systému pre testovanie. Môže zahrňovať inštaláciu prídavných balíkov, riešenie konfliktov a prídavné nastavenia serverov. Pre prípravu prostredia sa využíva **ansible-playbook**, kde je možné deklarovať viacero prostredí, ktoré budú aplikované postupne, atribút **install** pre definovanie dodatočných balíkov pre inštaláciu alebo využitie shellu pre exekúciu potrebných príkazov pre prípravu. Názornú ukážku zadefinovanie fázy prepare je možné vidieť vo výpise 3.8.

```
prepare:
    - how: ansible
      playbook:
          - common.yml
          - rhel7.yml
    - how: install
      package:
          - tmp/RPMS/noarch/tmt-0.15-1.fc31.noarch.rpm
          - tmp/RPMS/noarch/python3-tmt-0.15-1.fc31.noarch.rpm
```

Výpis 3.8: Ukážka zadefinovania fázy Prepare

Execute Vykoná testy v definovanom prostredí za pomoci vybraného exekútora. Pre každý test sa vytvorí samostatný adresár na ukladanie artefaktov súvisiacich s vykonaním testu. Jeho cesta je zostavená z názvu testu a je uložená v adresári *execute/data*, ktorý obsahuje súbor *metadata.yaml* s agregovanými L1 metadátami použiteľnými pre prípadné riešenie problémov. Implicitným exekútorom je nástroj TMT, avšak je možné definovať aj spustenie testov ako atomickú dávku pomocou shell skriptu vykonávaného v zabezpečenom

a vzdialenom prostredí (atribút **detach**) ako je to uvedené vo výpise 3.9, čo môže byť užitočné v prípade pomalého propojenia k hostovi, alebo v prípade dlhého behu testov, kedy je žiaduce odpojiť osobný počítač a nechať testy bežať na pozadí.

```
execute:
    how: detach
    isolate: true
```

Výpis 3.9: Ukážka zadefinovania fázy Execute

Finish Definuje akcie, ktoré sa majú vykonať po dobehnutí testu. Je to prakticky protiklad kroku *Prepare* užitočný pre návrat systému do pôvodného stavu, uvoľnenie alokovaných zdrojov a spracovanie výsledkov testovania. Vo výpise 3.10 je uvedený príklad fázy *Finish*, v ktorej sa logovacie dáta v závere testovania uložia do zvoleného úložiska pomocou skriptu *upload-logs.sh*.

```
finish:
    how: shell
    script: upload-logs.sh
```

Výpis 3.10: Ukážka zadefinovania fázy Finish

3.2.2 Funkcionalita TMT

Funkcionalitu nástroja **TMT** je možné odvodiť od metadát popisujúcich kroky testovania definovaných v predchádzajúcej kapitole. Samotné kroky ale predstavujú až samotné parametre exekúcie istej podmnožiny funkcionality, ktorá je primárne orientovaná na správu testov, testovacích plánov a ich exekúciu. V tejto kapitole sa zameráme na nástroj TMT ako taký a definujeme si základné príkazy, ktoré podporuje [21], kde ich praktickú ukážku je možné vidieť vo výpise 3.11:

Init Nástroj umožňuje inicializáciu stromu FMF metadát. Vytvorí skrytý adresát *.fmf/* a voliteľne aj základný testovací plán podľa vybranej šablóny.

Tests Príkaz **test** slúži na správu a preskúmavanie testov, pričom pre svoju prácu využíva L1 metadáta. Umožňuje filtrovanie testov na základe špecifikovaných kritérií, funkciu lint pre kontrolu správnosti formátu L1 metadát, vytváranie nových testov a možnosť použitia externých knižníc pre účely testovania.

Plans Príkaz **plan** slúži na správu, preskúmavanie a vytváranie nových testovacích plánov. Vlastnosti plánov sú špecifikované v rámci L2 metadát, ktoré sú týmto príkazom využívané.

Stories Umožňuje efektívne vytvárať, zobrazovať a filtrovať user-stories v rámci L3 metadát. Zaujímavosťou je, že dovoľuje sledovať pokrytie požiadaviek dokumentáciou, implementáciou a testami.

Run Poskytuje možnosť exekúcie testov, kde využíva najmä metadáta o krokoch testovania. Umožňuje tiež filtrovanie plánov podľa definovaných kritérii, výber krokov testovania pre exekúciu a možnosti detailného debugovania testov.

```
# vyfiltrovanie tier0 testov
$ tmt test ls --filter 'tier: 0'
---
# vytvorenie noveho testu
$ tmt test create /tests/smoke
---
# vytvorenie noveho testovacieho planu
$ tmt plan create --template mini /plans/smoke
---
# zobrazenie pokrytia kodu deokumentaciou
$ tmt story coverage
---
# spustenie vybranych krokov
$ tmt run discover provision prepare
---
# spustenie testov v interaktivnom mode pre potreby debugovania
tmt run --all execute --how tmt --interactive
```

Výpis 3.11: Výpis použitia dostupnej funkcionality nástroja TMT

Kapitola 4

Zhromaždenie požiadaviek a súčasný stav

V následujúcej kapitole sa budeme venovať priblíženiu a podrobnému opisu používaných technológii v spoločnosti *Red Hat*. Predstavíme si ako funguje workflow opravy, testovania a následnej aktualizácie balíka na čom vysvetlíme, ktorým častiam sa v tejto práci budeme venovať. Následne sa zameráme na zber užívateľských požiadaviek na nový systém pre správu výsledkov testov a vyberieme podmnožinu, ktorej sa budeme ďalej venovať.

4.1 Súčasný stav

V tejto časti sa pre lepšie pochopenie problematiky návrhu systému pre správu testov budeme zaoberať aj nástrojmi priamo súvisiacimi s danou témou, avšak tieto nie sú predmetom diplomovej práce a slúžia primárne pre doplnenie obrazu čitateľa.

Spoločnosť *Red Hat* spravuje veľké množstvo balíkov primárne používaných pre produkt *Red Hat Enterprise Linux* a pre vedľajší produkt *Fedora*, z ktorej RHEL priamo vychádza. Výhoda spočíva v otestovaní a stabilizácie funkcinality na OS Fedora a následné prebratie tohto kódu pre OS RHEL [5]. Kód jednotlivých komponent tvoriacich samotný OS je uschovaný v git repozitároch, ktoré sú voľne dostupné (Fedora), alebo interné (RHEL). Pre každý z interných repozitárov existuje sesterský repozitár, v ktorom je uložený zdrojový kód testov pre jednotlivé komponenty. Nakoľko testovanie v rámci Fedory nie je momentálne na úrovni, kde by sa dalo s výsledkami systematicky pracovať [9], budeme sa zaoberať iba testovaním pre RHEL, avšak so zachovaním princípov, ktoré pri úspechu bude možné plnohodnotne aplikovať aj pre Fedoru.

Pre zaznamenávanie chýb jednotlivých komponent alebo ich testov, či pre RHEL alebo Fedoru, slúži nástroj **Bugzilla** detailne popísaný v kapitole 2. Sú do neho zapisované aj výsledky testovania, prípadne problémy, poznámky a všetky informácie, ktoré priamo súvisia s dodaním softvéru koncovému užívateľovi. V prípade nahlásenia problému vývojár štandardne pracuje na oprave a je zvykom vytvoriť aj test. Pokiaľ je to možné, je použitý framework **BeakerLib**, kde test je po schválení testerom uložený do prislúchajúceho repozitáru a skompilovaný do RPM balíku uloženého v systéme pre správu testov.

Pre správu testov a testovacích plánov je využívaný nástroj **Nitrate**, ponúkajúci možnosť využiť skompilované RPM testovacie balíky uložené v nástroji **Beaker** a pridať ich do konkrétnych testovacích plánov. Každý tento testovací balík, ďalej iba test, má priradené prislúchajúce atribúty a relevanciu pre verzie jednotlivých OS, inými slovami metadáta.

Nitrate umožňuje tieto plány uložiť, ako aj zobraziť výsledky exekúcie plánov, avšak bez možnosti zložitejšieho filtrovania alebo porovnávania s predchádzajúcimi behmi. Rozoznávame niekoľko základných druhov testovaní používaných v rámci subsytému, pre ktorý sú plánované zmeny:

- **CI testy** – základné jednotkové a integračné testy pre komponenty;
- **Errata testy** – pokročilé jednotkové a integračné testy pre komponenty potvrdzujúce správnosť aktualizácie pred dodaním zákazníkovi;
- **Tier testy** – pokročilé hromadné integračné testovanie komponent systému, napríklad pred vydaním novej verzie distribúcie.

Z technického hľadiska medzi týmito testami nie je rozdiel, jedná sa skôr označenie sád testov, kde každá sada je spustená v inom okamihu vývoja a má rôzneho zadávateľa. **CI testy** sú spustené zadávateľom **Jenkins CI** okamžite po úspešnej kompilácii kódu komponenty do RPM balíka. Testy sú vykonávané v prostredí poskytnutom nástrojom Beaker, ktorý dodáva virtuálne stroje pre samotný beh testov. Beaker reportuje stav testovania zadávateľovi, ako napríklad začiatok alebo koniec testovania, a ten tieto správy reportuje na **UMB** (Unified Message Bus). Na UMB sú dostupné informácie aj ohľadom kompilácie kódu komponent do RPM balíkov, avšak tieto nie sú pre naše potreby momentálne dôležité. Týmto spôsobom je možné získať informácie o testovaní a pracovať s výsledkami testov prakticky v akejkoľvek aplikácii. Bohužiaľ nástroj Nitrate informácie z UMB nevyužíva a všetky druhy výsledkov sú do neho ukladané pomocou externých skriptov, čo je jedna z nevýhod systému, ktorú je potrebné zmeniť.

Po úspešnom dokončení CI testov je nový RPM balík manuálne vložený do systému pre správu aktualizácií **Errata** [3], kde testeri vykonávajú **Errata testing**. Zadávateľ tohto Errata testingu je **EWA** (*Errata Workflow Automation*), ktorá taktiež využíva prostredie dodávané nástrojom Beaker, avšak EWA nie je schopná reportovať výsledky získané z Beakeru na UMB, čiže sa do Nitrate dostávajú iba pomocou už spomínaných externých skriptov.

Tier testing je integračné testovanie komponent systému u ktorého je definovaná úroveň určujúca špecifickosť testov. Spúšťa sa pred vydaním určitej verzie distribúcie, kde cieľom je overiť, že daná verzia ako celok funguje. Pre nás je podstatné, že výsledky tohto testovania taktiež nie sú reportované zadávateľom na UMB, ale vkladané do nástroja Nitrate externými skriptami.

Vzhľadom na budúcnosť by bolo ideálne, aby boli výsledky všetkých typov testovaní reportované na UMB, následne zachytávané, ukladané a zobrazované nejakým nástrojom na jednom mieste v užívateľsky prívetivom formáte. Z tohto dôvodu sa pre účely tejto práce budeme venovať iba výsledkom dostupným na UMB, nakoľko je v pláne pripraviť pre tento princíp postupne celú infraštruktúru a všetky typy testovania. Správy na UMB sú vo formáte JSON obsahujúci metadáta o testovaní, kde výsledky testov s detailnými logmi sú uložené v položke **xunit** vo formáte XML [19]. Formát XML je využívaný najmä z dôvodu, že sú v ňom reportované výsledky testov využívajúce framework BeakerLib, ktorý je v spoločnosti v rámci testovania RHEL produktov najvyužívanejší.

Momentálne prebieha vývoj a nasadenie nového nástroja pre správu testov TMT, ktorý ako zadávateľa využíva Jenkins CI, čo umožňuje dostupnosť informácií o testovaní na UMB. Tento nástroj v spolupráci s FMF by mal v budúcnosti úplne nahradiť nástroj Nitrate a vyradiť ho z používania.

Z popisu z predchádzajúcich kapitol je zrejmé, že po nasadení nástroja TMT, ktorý má nahradiť nástroj Nitrate ako systém pre správu testov bude chýbajúcou časťou skladačky pokročilý systém pre správu výsledkov testov. TMT dokáže plnohodnotne nahradiť funkcionalitu nástroja Nitrate, ktorá sa týka správy testov a testovacích plánov, avšak absentuje tu možnosť efektívneho zaznamenávania a prehliadania výsledkov testov. Pre túto podmnožinu funkcionality je potrebné vytvoriť nezávislý nástroj, ktorý bude určený pre správu výsledkov testov, aby v prípade potreby mohol byť rozšírený alebo upravený bez toho aby to ovplyvnilo funkcionalitu správy testov. Jeden z problémov nástroja Nitrate bol totiž v nemožnosti nahradiť istú funkcionalitu iným nástrojom, nakoľko bola centralizovaná do jedného nástroja s nemožnosťou úpravy. Predstava je, aby v budúcnosti jednotlivé prvky celého systému fungovali ako nezávislé moduly, využívajúce určité rozhranie, ktoré je možné izolovane nahradiť bez ovplyvnenia funkcionality iných modulov.

Tu sa dostávame k samotnej téme tejto práce, kde z vyššie uvedeného vyplýva, že je potrebné navrhúť nástroj, ktorý bude efektívne zaznamenávať a zobrazovať dáta o výsledkoch testov, ktoré získa z UMB. Momentálne existuje v spoločnosti viacero nástrojov pre zobrazovanie výsledkov, avšak žiadny z nich nespĺňa všetky požiadavky užívateľov a neexistuje jednotný kanál, v ktorom by boli všetky výsledky reportované. Ako príklad je možné uviesť **CI Dashboard**, ktorý dokáže čítať správy z UMB a zobrazovať výsledky testovania, avšak má veľmi obmedzenú funkcionalitu. Naopak Nitrate poskytuje podmnožinu požadovanej funkcionality pre zobrazovanie výsledkov, avšak nedokáže čítať dáta z UMB.

4.2 Zhromaždenie požiadaviek

Zhromažďovanie požiadaviek vo forme user-stories pre daný projekt prebiehalo formou organizovaných stretnutí s užívateľmi v niekoľkých iteráciach. Takýchto stretnutí prebehlo 8, kde na každom sa zúčastnilo približne 15 až 20 užívateľov zainteresovaných do danej problematiky. Mimo oficiálnych stretnutí prebiehali neformálne konzultácie s vybranými užívateľmi, ktorí mali v rámci stretnutí zaujímavé postrehy, ohľadom doplnenia informácií.

Na prvých stretnutiach prebiehal zber všeobecných požiadaviek a ich porovnávanie s používanými nástrojmi ako sú **CI Dashboard** alebo **Nitrate**, z ktorých vychádzajú najmä prioritné požiadavky. Postupom času a so štúdiom existujúcich externých projektov sme na posledných stretnutiach dokázali prioritné požiadavky aj priamo predvádzať užívateľom na prototype pre lepšie pochopenie a predstavu. Tento krok bol náležite ocenený zo strany užívateľov a taktiež neskôr po nasadení aj využitý pre odladenie implementačných chýb daného nástroja.

V rámci stretnutí prebehla aj prioritizácia požiadaviek, aby bolo možné jasne oddeliť, čo je potrebné mať k dispozícii pre základný chod systému a ktorú funkcionalitu je možné doplniť do aplikácie neskôr. Požiadavky boli rozdelené do okruhov na základe funkcionality, ku ktorej sa viažu. V rámci diskusií prebehla aj zhoda na type testov, ktoré by mali byť v danom systéme spracovávané a zobrazované. Diskutovalo sa nad vyššie uvedenými možnosťami **CI testov**, **Errata testov** a **Tier testov**.

V rámci diskusie a následného štúdia bolo objasnené, že výsledky spomenutých testov sú reportované rôzne, majú iných exekútorov, testy bežia za pomoci rôznych zadávateľov a za pomoci rôznych testovacích frameworkov, čo bolo detailne popísané v predchádzajúcej sekcii. So zameraním sa najmä na nástroj TMT, sú pre nás podstatné výsledky testov, ktoré sú dostupné na UMB a to sú momentálne iba CI testy. V prípade, že pracujeme s plánom, kde v budúcnosti aj ostatné typy testovania budú dostupné na UMB a prejdu na používanie nástroja TMT ako exekútora a tým aj zadávateľa Jenkins CI, budú ich výsledky pridané

automaticky do aplikácie pre zobrazovanie výsledkov testov, ktorá bude navrhnutá tak, aby tieto výsledky dokázala uložiť a interpretovať v potrebnom formáte. Z tohto dôvodu sa v súčasnom stave a v rámci tejto práce budeme zaujímať primárne CI testami.

Funkcionálne požiadavky na systém boli v rámci stretnutí definované vo forme user-stories, rozdelené do okruhov podľa dotknutej funkcionality a boli im definované priority. Následne bola vybraná podmnožina požiadaviek s ktorou budeme pracovať. Priority požiadaviek boli na základe konzultácii rozdelené do troch stupňov ako **vysoká** (HP), **stredná** (MP), **nízka** (LP), kde nižšie je možné vidieť požiadavky, ktorým sa v rámci tejto práce budeme aktívne venovať.

Základné požiadavky

- (HP) ako tester/vývojár, chcem mať možnosť zobraziť výsledky konkrétneho behu testov;
- (HP) ako tester/vývojár, chcem mať možnosť zmeniť stav výsledkov testov manuálne;
- (HP) ako tester/vývojár, chcem mať možnosť zobraziť zmeny výsledkov testov, ako napríklad manuálna zmena stavu, špecifikácia chyby alebo komentár;
- (HP) ako tester/vývojár, chcem mať možnosť zobraziť detailné logy behu testov;
- (HP) ako tester/vývojár, chcem mať odkaz na zdrojový kód testu priamo pri výsledkoch;
- (HP) ako tester/vývojár, chcem mať možnosť pridávať výsledkom testov rôzne atribúty;
- (HP) ako tester/vývojár, chcem mať možnosť ukladať výsledky manuálneho testovania;
- (HP) ako tester/vývojár, chcem mať možnosť uložiť a zobraziť metadáta vykonaných testov;
- (HP) ako tester/vývojár, chcem mať možnosť filtrovať podľa mena vlastníka testov;
- (HP) ako tester/vývojár, chcem mať možnosť definovať rôzne typy zlyhaní testov;
- (MP) ako tester/vývojár, chcem mať pokryté všetky stavy výsledkov testov, ktoré boli dostupné v Nitrate;
- (MP) ako tester/vývojár, chcem aby systém automaticky vyhodnocoval typy chýb na základe výsledkov predchádzajúcich behov.

Požiadavky na zobrazenie histórie

- (HP) ako tester/vývojár, chcem mať možnosť porovnať výsledky testov jednotlivých behov;
- (HP) ako tester/vývojár, chcem mať možnosť zobraziť výsledky behov konkrétneho testu na jednej distribúcii alebo naprieč viacerými distribúciami;
- (HP) ako tester/vývojár, chcem mať možnosť porovnať výsledky fáz jednotlivých behov testov s predchádzajúcimi behmi;

- (HP) ako tester/vývojár, chcem mať možnosť zobraziť históriu výsledkov testu naprieč architektúrami;
- (MP) ako tester/vývojár, chcem mať možnosť zobrziť výsledky behu testu naprieč testovacími plánmi.

Požiadavky na filtrovanie

- (HP) ako tester/vývojár, chcem mať možnosť zobraziť výsledky iba testov, ktoré skončili neúspešne, alebo v inom ľubovoľnom stave;
- (HP) ako tester/vývojár, chcem mať možnosť filtrovať podľa komponent, testov, testovacích plánov, užívateľov, dátumu a rôznych FMF atribútov;
- (HP) ako tester/vývojár, chcem mať možnosť uložiť a zdieľať zložité filtre;
- (HP) ako tester/vývojár, chcem mať možnosť prezerať iba manuálne/automatické testy.

Požiadavky na agregáciu

- (HP) ako tester/vývojár, chcem mať možnosť zobraziť agregované výsledky testov, ktoré je možné si rozbaliť na úroveň architektúr a logov;
- (HP) ako tester/vývojár, chcem mať možnosť zobraziť štatistiky agregovaných výsledkov konkrétnej podvetvy.

Ostatné požiadavky

- (MP) ako tester/vývojár, chcem mať možnosť využiť tento nástroj aj pre ukladanie výsledkov testov komponent Fedory;
- (LP) ako tester/vývojár, chcem mať možnosť zobraziť výsledky testov, hneď po ich dobehnutí, aj keď ešte nie je ukončený beh daného plánu testovania;
- (LP) ako tester/vývojár, chcem mať možnosť zobraziť kostru testov, ktorá bude k dispozícii pred spustením testovania;
- (LP) ako tester/vývojár, chcem mať možnosť zobraziť stav testovania pre konkrétnu komponentu.

Vyššie uvedené požiadavky boli získané od užívateľov a boli vzaté do úvahy v rámci výberu a návrhu nového systému pre správu výsledkov testov. Vzhľadom ale na neustále prebiehajúcu implementáciu testovacej infraštruktúry v spoločnosti *Red Hat*, nebolo možné všetky požiadavky implementovať v rámci tejto práce. Dôvodom sú najmä nedostatočné nasadenie nástroja TMT do používania, nedostupnoť niektorých potrebných informácií na UMB ale aj veľké množstvo potrebných zmien, ktoré by ďaleko prevyšovali rozsah diplomovej práce. V rámci tejto práce ale prebehne predpríprava a overenie možnosti implementácie chýbajúcej funkcionality, ktorú nie je možné v súčasnom stave implementovať. Touto funkcionalitou sa budeme zaoberať v rámci kapitoly popisujúcej možné rozšírenia systému.

Kapitola 5

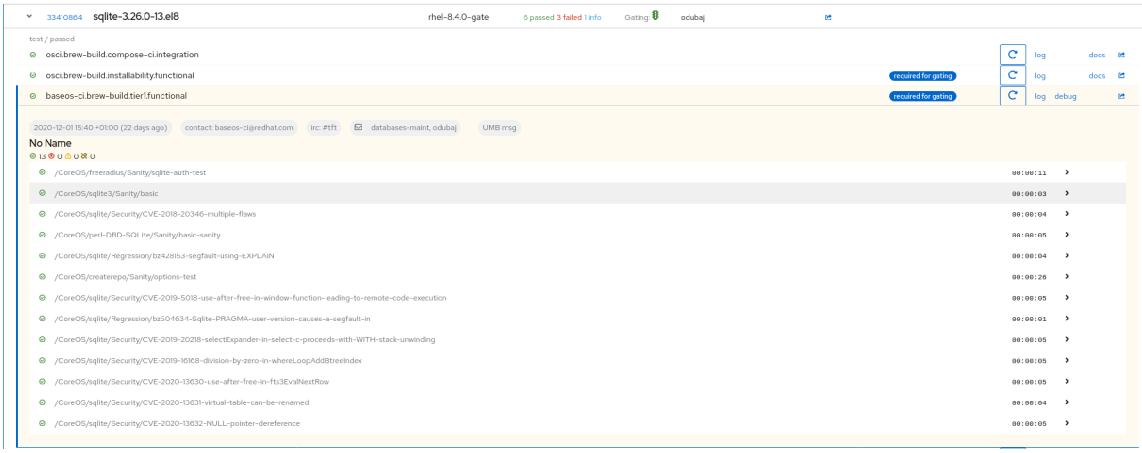
Výber vhodného nástroja a návrh zmien

V tejto kapitole sa budeme venovať výberu vhodného nástroja pre pokrytie chýbajúcej funkcionality a užívateľských požiadaviek, ktoré boli definované v predchádzajúcej časti. Ako už bolo spomenuté, zameriavame sa na výber nástroja pre správu výsledkov testov, nakoľko nástroj **TMT** zabezpečuje funkcionalitu pre správu testov a testovacích plánov. Novovyvinutý nástroj spolu s **TMT** by mal posktytovať plný obraz ohľadom správy testov a zároveň aj ich výsledkov, čo by viedlo k dlhodobému cieľu, ktorým je nahradenie nástroja **Nitrate**. Zároveň by tieto nástroje dodržiavali princíp oddelených modulov a jasne oddelovali funkcionalitu pre správu testov a ich výsledkov. Týmto spôsobom je možné nasledujúci koncept využiť v budúcnosti aj v rámci projektu *Fedora*.

Vzhľadom na množstvo potrebnej funkcionality a rôznorodosti požiadaviek by implementácia systému pre správu výsledkov testov od základov bola enormne náročná a táto alternatíva riešenia bola hneď v úvode zamietnutá. V súčasnej dobe je ale na trhu množstvo existujúcich nástrojov poskytujúcich potrebnú funkcionalitu. Jedno z riešení je vybrať nástroj pokrývajúci istú podmnožinu funkcionality, doimplementovať chýbajúcu a nastaviť danú aplikáciu pre potreby používania. Takýto nástroj by mal byť open-source, aby bolo v budúcnosti možné jeho rozšírenie pre testovanie *Fedory*.

Jedným z možných nástrojov je interne vyvíjaný **CI Dashboard** (ukážka dostupná na obrázku 5.1), ktorý mimo iného umožňuje zobraznie výsledkov CI testovania a taktiež má implementované rozhranie pre prácu s UMB. V jeho prospech hovorí taktiež používanie v rámci spoločnosti a tým pádom aj zvyk užívateľov na dané prostredie. Najväčšou nevýhodou je jeho oklieštená funkcionalita, kde by v prípade použitia bolo nutné implementovať dôležité prvky ako zobrazovanie histórie, efektívne filtrovanie alebo automatické vyhodnocovanie chýb testovania. Prakticky by okrem implementácie rozhrania pre prácu s UMB a základného zobrazovania výsledkov bolo nutné implementovať všetko. Z tohto dôvodu nie je CI Dashboard ideálnym kandidátom pre tento projekt.

Ďalším sľubným kandidátom bol nástroj **Polarion** (ukážka dostupná na obrázku 5.2) ponúkajúci široké spektrum potrebnej funkcionality. Je aktívne využívaný určitými tímami spoločnosti *Red Hat*, čo umožňuje získať spätnú väzbu na jeho používanie. Pokrýva značnú časť funkcionality systému **Nitrate**, ale bohužial nepridáva veľa potrebných noviniek navyše. Taktiež spätná väzba užívateľov na jeho používanie nie je pozitívna a stretli sme sa najmä s negatívnymi vyjadreniami na jeho adresu, čo je veľmi dôležitý faktor pri tvorbe nového systému. Zároveň ponúka množstvo funkcionality pre správu testov a testovacích plá-



Obr. 5.1: CI Dashboard

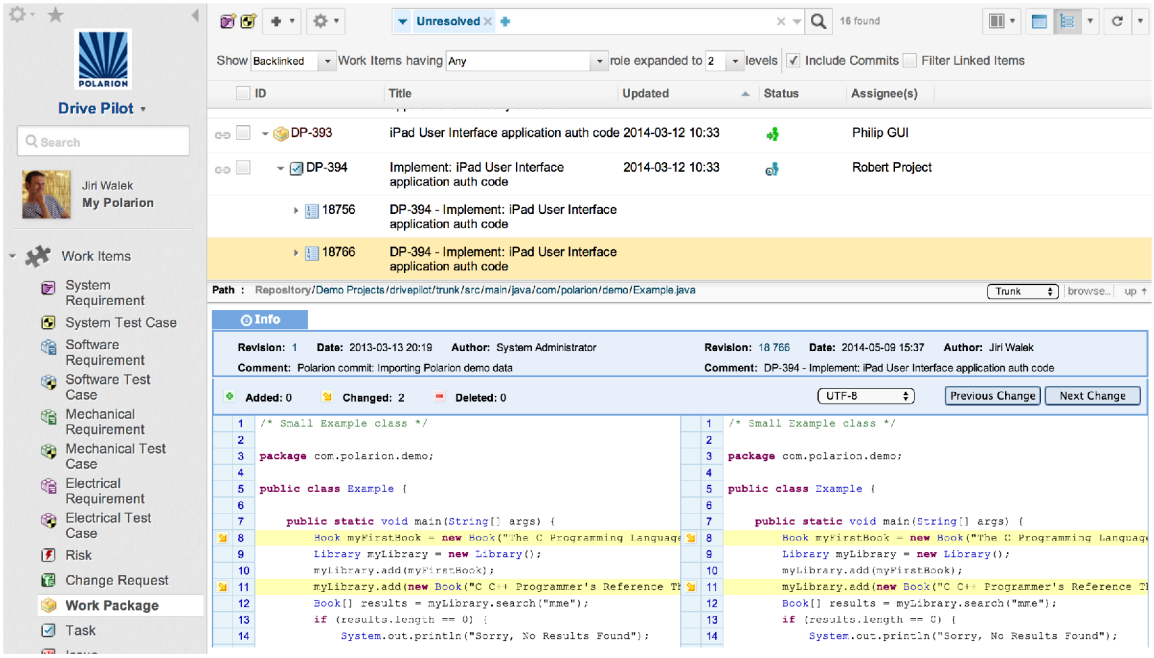
nov, ktorá v našom prípade nie je potrebná. Rovnako nástroj **Polarion** nie je open-source, čo neumožňuje jeho využitie v rámci testovania *Fedory*. Z týchto dôvodov bol Polarion taktiež vylúčený z výberu [10].

Kandidát, ktorý nás najviac oslovil svojou flexibilitou, jednoduchosťou a množstvom funkcionality bol **ReportPortal**. Je to systém poskytujúci funkcionalitu čisto pre správu výsledkov testov a zdal sa byť ideálnym nástrojom pre naše potreby. Z tohto dôvodu mu bude venovaná nasledujúca sekcia, v ktorej bude detailne priblížená jeho funkcionalita a popísané dôvody, prečo bol spomedzi kandidátov vybraný pre daný projekt [8].

5.1 ReportPortal

ReportPortal je open-source projekt, vyvíjaný spoločnosťou sídliacou v Minsku, poskytujúci pokročilé možnosti správy výsledkov testov. Pre jeho implementáciu sú použité štandardné technológie ako *REST Web Service* pre aplikačnú logiku implementovanú v jazyku *Java*, *ReactJS* pre implementáciu UI a *PostgreSQL* pre prácu s dátovou vrstvou [8].

ReportPortal je služba, ktorá poskytuje široké možnosti na urýchlenie reportovania a analýzy výsledkov testov pomocou zabudovaných analytických funkcií, čo skvele dopĺňa funcionalitu CI procesu. Medzi jeho charakteristické vlastnosti patrí real-time reportovanie výsledkov a aktuálneho stavu testovania a zároveň možnosť reportovania výsledkov vo formáte *XML* ako výsledok využitia frameworku *Junit5* [11]. Pre stavbu štruktúry výsledkov testov poskytuje dostatočnú flexibilitu kopírujúcu zapúzdrenosť testovacích plánov, do ktorej je možné uložiť prílohy, referencie alebo logovacie a binárne súbory. Pre nás veľmi dôležitým prvkom je uchovanie, zobrazenie a porovnanie aktuálnych výsledkov s historickými dátami, poprípade vyhodnocovanie trendu výsledkov podľa špecifikovaných kritérii alebo filtrov. Za zmienku stojí taktiež možnosť manuálnej analýzy výsledkov a na to nadväzujúca funkcionalita pre automatickú analýzu a vyhodnotenie výsledkov na základe predošlých manuálnych vyhodnotení užívateľmi. Ako množstvo iných nástrojov, aj ReportPortal ponúka možnosť integrácie s mainstream platformami ako sú *Jenkins*, *Jira* alebo najpoužívanejšími testovacími frameworkmi ako *Junit5* [11].



Obr. 5.2: Polarion [10]

5.1.1 Štruktúra systému

Pre efektívne využitie prostriedkov a load-balancing tvorí tento komplexný systém pre správu výsledkov testov niekoľko navzájom spolupracujúcich služieb [8], kde ich vzájomné previazanie je graficky znázornené na obrázku 5.3:

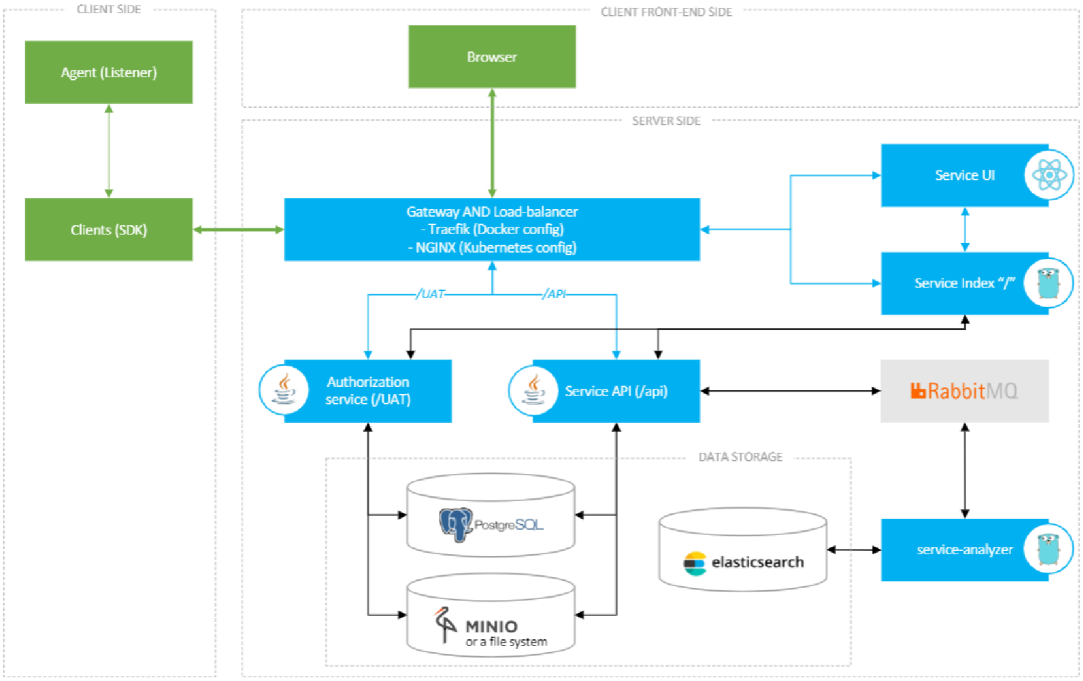
- **API Service** – poskytuje hlavné aplikačné rozhranie systému;
- **UI Service** – tvorí užívateľské rozhranie;
- **Authorization Service** – zabezpečuje autentifikáciu a autorizáciu užívateľov;
- **Analyzer Service** – zhromažďuje a spracúva informácie o výsledkoch testov a posiela ich do *ElasticSearch*;
- **Index Service** – zodpovedá za spracovanie požiadaviek, ktoré nespĺňajú kritéria pre iné služby, ďalej zhromažďovanie informácií o službách, riešenie chýb.

Tieto služby využívajú pre svoju prácu zároveň aj iné externé aplikácie [8]:

- **PostgreSQL** – je využitá ako databáza dát;
- **ElasticSearch** – engine pre efektívne full-textové vyhľadávanie pre účely automatickej analýzy výsledkov;
- **MinIO** – vysoko výkonný distribuovaný server pre ukladanie súborov;
- **RabbitMQ** – komunikačná zbernica pre komunikáciu služby pre automatickú analýzu výsledkov s hlavným aplikačným rozhraním;
- **Traefik/NGINX** – určujú Gateway ako hlavný východzí bod služieb a zabezpečujú smerovanie HTTP požiadaviek na správne služby a load-balancing.

Pre nasadenie systému sa využívajú najmä kontajnerizačné platformy ako *Docker*, kde každá služba alebo externá aplikácia beží vo svojom kontajneri, čo umožňuje efektívnu správu jednotlivých častí a izolovaný beh systému.

Druhou možnosťou je nasadenie na *Kubernetes cluster* za pomoci *Helm package manager*. V oboch prípadoch je dostupná detailná dokumentácia a image, čo užívateľom uľahčuje celý proces. V prípade nasadenia aplikácie pomocou nástroja *Docker* sa ako Gateway používa *Traefik*, v prípade využitia *Kubernetes* je ním *NGINX*. Existuje taktiež možnosť nasadenia systému priamo na operačný systém, ktorá nie je veľmi odporúčaná, nakoľko je náročná na správu, môže prinášať problémy s load-balancingom a zároveň môže byť samotný systém ovplyvnený lokálnymi nastaveniami užívateľa [8].



Obr. 5.3: Štruktúra systému ReportPortal [8]

5.1.2 Funkcionalita

Systém ReportPortal vzhľadom na jeho modulárnu štruktúru ponúka široké spektrum funkcinality a veľa možností pre jej rozšírenie. V nasledujúcej sekcii si danú funkcionalitu bližšie popíšeme aj so zodpovedajúcimi ukážkami [8].

Výsledky a štruktúra testovacích behov Každý testovací beh je zobrazený ako samostatná položka v zozname patriaca určitému projektu, tak ako je to vidieť na obrázku 5.4. **Projekt** je možné si predstaviť ako úplne oddelenú časť systému, ktorá nijak neinteraguje s inými projektami. Môže napríklad obsahovať výsledky testov o ktoré sa zaujíma jeden konkrétny tím. Položka môže obsahovať štruktúru definovanú vstupnými dátami, kde jej základ tvorí nasledujúca hierarchia:

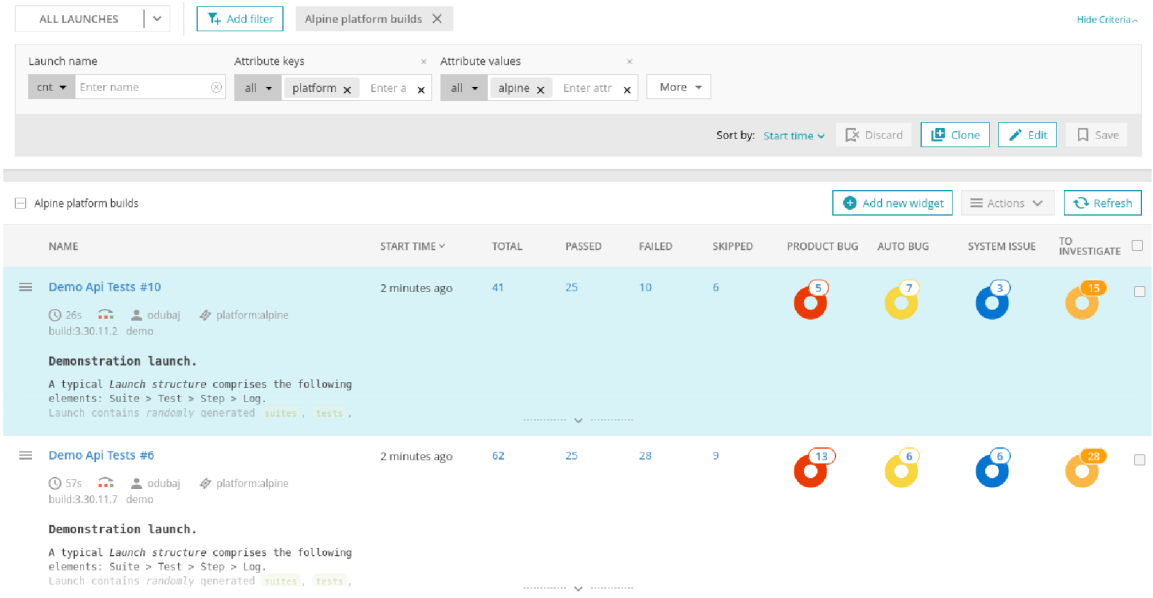
$$Launch > Suite > Test > Step$$

Launch definuje testovací beh, **Suite** plán alebo podplán, **Test** samotný test a **Step** jednotlivé kroky alebo asserty. Flexibilita systému spočíva v možnosti definovať niekoľko objektov Suite za sebou, čo umožňuje vytvoriť niekoľkostupňovú hierarchiu a rôzny pohľad na granularitu výsledkov.

Uloženie a zobrazenie logovacích súborov, príloh V každom stupni hierarchie je možné danému objektu priradiť logy z testovania. Objektom **Test** je možné priradiť aj prílohy vo forme súborov, čo môže byť užitočné v prípade manuálneho testovania GUI.

Atribúty a parametre testov Každému objektu v hierarchii je možné priradiť určité atribúty, čo umožňuje zobrazovať metadáta testov priamo s ich výsledkami. Atribúty je možné pridávať alebo meniť vzhľadom na potreby užívateľa. U testov je možné k atribútom definovať aj parametre spustenia.

Filtrovanie Na základe vyššie spomenutých atribútov, je možné efektívne filtrovať medzi jednotlivými objektami na každom stupni hierarchie. Filtrovanie je možné aj na základe mien, času behu, užívateľov alebo výsledku. Výhodou je, že definované filtre je možné ukladať a zdieľať medzi užívateľmi, čo je obzvášť prospešné pri práci viacerých užívateľov na jednom projekte. Na obrázku 5.4 je možné vidieť aplikovaný filter, ktorý filtruje testovacie behy, ktoré obsahujú atribút *platform* s hodnotou *alpine*.



Obr. 5.4: Zobrazenie testovacieho behu s atribútmi a aplikovaným filtrom

Možnosti importu výsledkov Výsledky testovania je možné importovať dvoma rôznymi spôsobmi. Prvou možnosťou je využiť aplikačné rozhranie, kde sa pomocou HTTP požiadaviek budú systému predkladať informácie o začatí, skončení a výsledku testu, kde predtým bude pomocou týchto požiadaviek vystavaná potrebná štruktúra. Tento spôsob umožňuje zobrazovať aktuálny stav výsledkov testov ešte počas behu testovacieho plánu,

avšak prináša veľkú náročnosť vzhľadom na spracovanie nemalého množstva požiadaviek pre každý jeden test.

Druhou možnosťou je import výsledkov testov vo formáte XML po dobehnutí testovacieho plánu. Táto možnosť je menej náročná, nakoľko systém naraz spracuje dáta a zobrazí ich v aplikácii. Jeho nevýhoda je v nemožnosti zobraziť stav testovania aktuálne bežiaceho plánu.

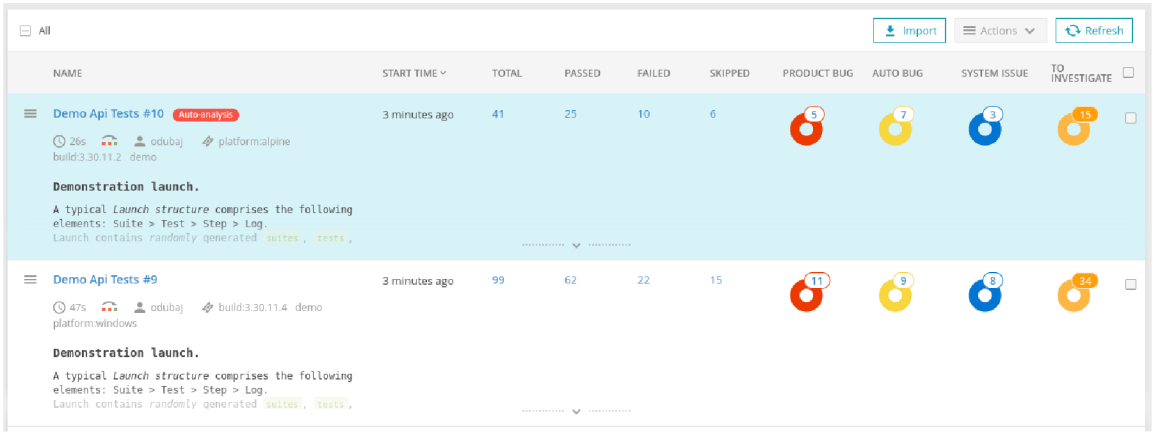
Manuálna analýza chýb V prípade neúspechu určitého testu, je možné manuálne analyzovať príčinu chýb pomocou logov a definovať kategóriu zlyhania. Medzi základné patria:

- **Product bug** – chyba testovaného produktu;
- **System issue** – chyba systému počas testovania;
- **Automation bug** – chyba automatizácie;
- **To Investigate** – neanalyzovaná chyba, predvolená pred začiatkom analýzy;
- **No defect** – chyba nenastala, false negative.

K základným typom je možné definovať vlastné podtypy chýb, aby bolo možné nastaviť celý proces podľa požiadaviek užívateľov.

Automatická analýza chýb a analýza na základe vzorov Okrem manuálnej analýzy dokáže systém automaticky analyzovať chyby na základe predchádzajúcich výsledkov a rozhodnutí pri použití princípov Machine Learning. Pomocou logov z predchádzajúcich výsledkov a využití ElasticSearch hľadá podobnosti a vyhodnocuje dané výsledky, čo dokáže výrazne ulahčiť prácu vývojárom. V prípade zapnutia automatickej analýzy je možné v hlavičke testovacieho behu vidieť upozornenie na aktuálne prebiehajúcu analýzu tak, ako je to vidieť na obrázku 5.5.

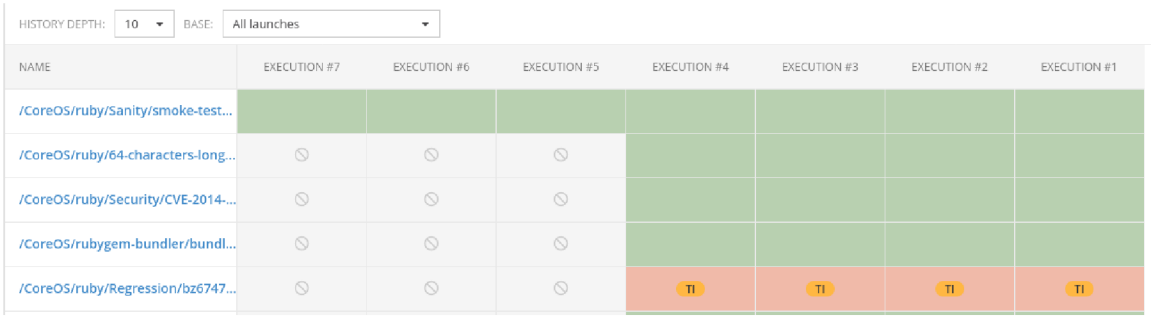
Ďalšou možnosťou je aktivácia analýzy na základe predom definovaných vzorov, ktorá prebieha na základe hľadania reťazcov v logoch na základe definovaného regulárneho výrazu. Táto funkcia je obzvášť prospešná pri vyhodnocovaní veľkeho množstva neúspešných testov, kde je možné všetky testy spoločne vyhodnotiť.



Obr. 5.5: Typy chýb a analýza

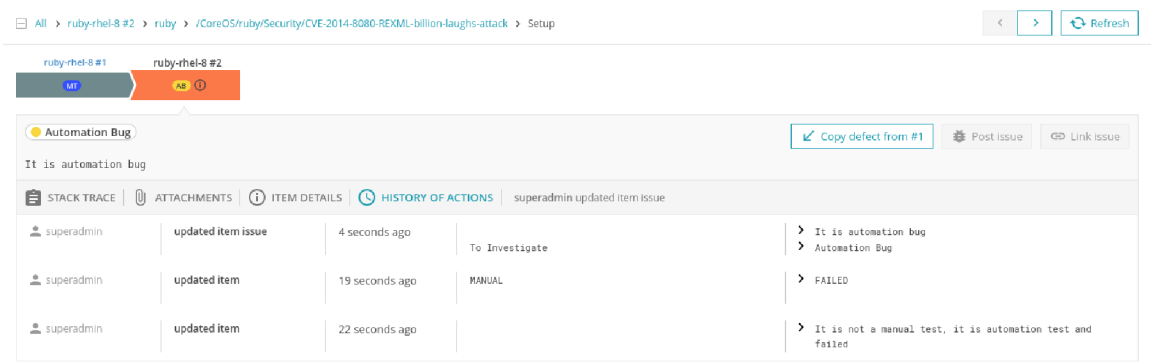
Identifikátor testov V ReportPortale rozlišujeme dva indetifikátory. Prvým je identifikátor konkrétneho výsledku testu (**Test ID**), ktorý je pre každý beh testu automaticky generovaný a druhým je identifikátor testu (**Test Case ID**), ktorý definuje samotný test. Test Case ID je možné systému definovať pomocou parametru pri vkladaní výsledkov testov, čo umožňuje mať rovnaký identifikátor v rámci systému pre správu testov a aj systému pre správu výsledkov testov. V prípade, že Test Case ID nie je poskytnutý, je automaticky generovaný z referencie na kód a parametrov. V prípade, že nie je poskytnutá ani referencia na kód, je generovaný z hierarchie rodičov, jeho mena a parametrov.

História Aplikácia umožňuje zobrazovať históriu výsledkov testov naprieč rôznymi testovacími behmi na ktorejkoľvek úrovni hierarchie. Je možné medzi zobrazovanými položkami testov filtrovať avšak nie je možná filtrácia medzi konkrétnymi testovacími behmi, ktoré je možné iba jednoducho obmedziť na počet v rámci histórie. Zobrazenie histórie je v tomto prípade neprehľadné a neflexibilné, nakoľko užívateľ nevie, ktorému testovaciemu behu patria jednotlivé položky tabuľky, čo je predstavené na obrázku 5.6.



Obr. 5.6: História výsledkov

História akcií Okrem histórie výsledkov, systém zaznamenáva aj históriu zmien, ktoré boli vykonané na danom objekte. Príkladom môže byť zmena typu chyby z *To Investigate* na *Automation bug* spolu so zmenou statusu z *Manual* na *Failed* (obrázok 5.7) alebo zmena komentáru a popisu chyby. Zaznamenáva sa daná činnosť, kto ju vykonal a s ktorým objektom súvisí.

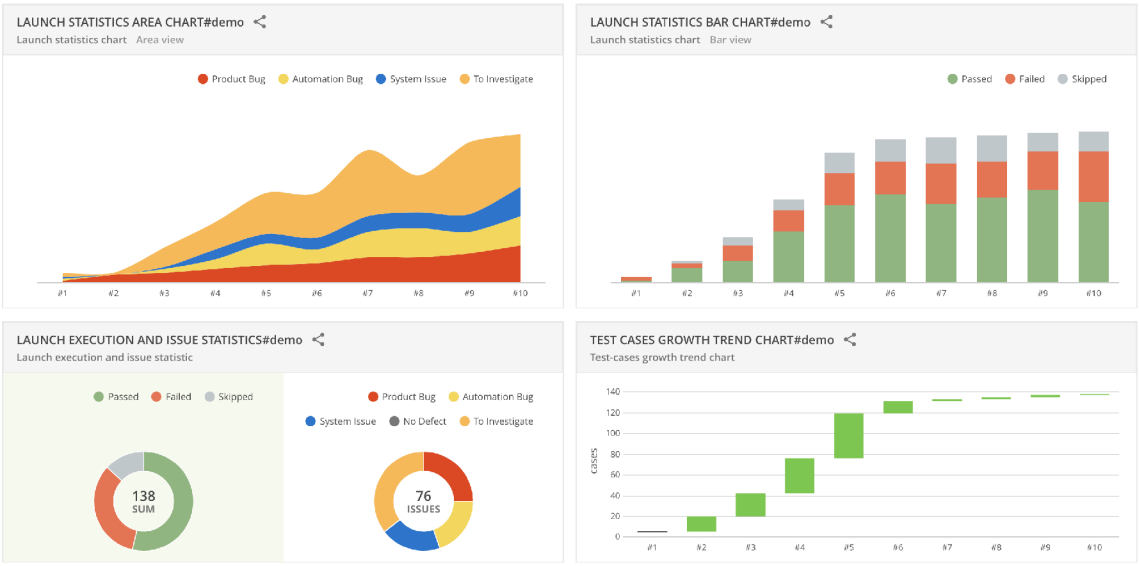


Obr. 5.7: História činností

Zlučovanie testovacích behov V prípade rozdelenia testovania do viacerých exekúcií, je možné ich následne pre prehľadnosť zlúčiť do jedného objektu. Následne budú pod jednou položkou dostupné výsledky všetkých spustených testov.

Export výsledkov Aplikácia umožňuje export výsledkov vybraných testovacích behov do formátov PDF, XLS a HTML.

Widgety a dashboardy ReportPortal umožňuje zobrazovať užívateľsky prívetivé grafy a widgety pre určité informácie, ktoré sú zväčša naviazané na konkrétne užívateľom definované filtre. Umožňuje to efektívnejšie analyzovať trendy vývoja testov, ich úspešnosť, množstvo, čas a rôzne štatistiky. Systém taktiež dokáže rozoznávať aj nestabilné testy spomedzi vybraných testovacích behov. Týmto testom a funkcionalite, ktorú testujú je možné následne venovať väčšiu pozornosť. Príklady widgetov je možné vidieť na obrázku 5.8.



Obr. 5.8: Widgety a dashboardy

Správa projektov a užívateľov Ako každá štandardná aplikácia, aj táto ponúka možnosti správy projektov a im prideleným užívateľom. Rozoznávame niekoľko rolí:

- **Administrátor** – plné privilégia pre všetky projekty;
- **Projektový manažér** – plné privilégia pre konkrétny projekt;
- **Člen** – prístup ku všetkým informáciám, ktorých je vlastníkom v rámci projektu bez možnosti administrácie;
- **Operátor** – prístup ku všetkým informáciám v rámci projektu bez detailných logov a príloh;
- **Zákazník** – prístup k dashboardom a výsledkom bez možnosti ich editovať.

Integrácia rôznych testovacích frameworkov ReportPortal podporuje základné typy testovacích frameworkov pre niektoré štandardné programovacie jazyky (*Java*, *JavaScript*, *Python*, *PHP*, *.Net*). Import výsledkov je možný iba s využitím aplikačného rozhrania avšak v prípade potreby je možné implementovať vlastné rozšírenie pre iný framework. Import pomocou *XML* je podporovaný iba pre formát Junit generovaný frameworkom *JUnit5*.

Podporované zásuvné moduly Do systému je možné integrovať rôzne zásuvné moduly pre podporu externých aplikácii, ako e-mailových notifikácií, trackovacích systémov (*Jira*, *Rally*) a autorizačných serverov (*LDAP*, *Active directory*, *SAML*, *OAuth*). V prípade potreby je rovnako ako v prípade testovacích frameworkov možné implementovať vlastný zásuvný modul pre vybranú externú aplikáciu.

5.2 Chýbajúca funkcionalita a návrh zmien

Vzhľadom na vyššie popísanú funkcionalitu a jej prienik s užívateľskými požiadavkami bol systém ReportPortal vybraný ako vhodný kandidát pre doplnenie funkcionality nástroja **TMT** a nahradenie nástroja **Nitrate**. Pre splnenie ďalších požiadaviek je potrebné doplniť istú funkcionalitu a zároveň ponúknuť isté časti chýbajúcej funkcionality vývojárom **ReportPortal-u**. Mimo implementácie novej funkcionality bude potrebné implementovať modul **listener**, ktorý bude komunikovať s *UMB*, **sadu skriptov** pre spracovanie prichádzajúcich správ a modul **parser**, ktorý bude preklápať dáta do požadovaného formátu a následne výsledky importovať do **ReportPortal-u**. V nasledujúcich podsekciách si predstavíme jednotlivé oblasti, v ktorých bude doplnená funkcionalita nástroja.

Manuálne testy Systém **ReportPortal** je určený primárne pre prácu s automatizovanými testami a neponúka žiadnu možnosť uložiť, prípadne zobraziť výsledky manuálnych a poloautomatizovaných testov. Na základe diskusie s užívateľmi je táto funkcionalita vysoko prioritná a preto je potrebné ju implementovať. Aplikácia ponúka možnosť manuálnej zmeny stavu výsledku testu, kde momentálne existujú tri základné stavy **Passed**, **Failed** a **Skipped**.

Manuálne a poloautomatizované testy sú súčasťou existujúcich plánov a budú vo výsledkoch testov reportované ako nespustené testy, ktoré vyžadujú interakciu s testerom, kde logy týchto testov budú obsahovať návod na vykonanie testovania. K existujúcim stavom je možné pridať ďalší stav **Untested**, ktorý bude signalizovať, že daný test nebol vykonaný a je manuálny/poloautomatizovaný. Takýto test bude označený ako „**To Investigate**“, aby ho nebolo možné prehliadnuť a aby boli vykonané potrebné kroky. Po manuálnom vykonaní testu užívateľ zmení stav výsledku testu do aplikácie podľa získaných hodnôt spolu s komentárom popisujúcim priebeh testovania. Týmto bude manuálny test vykonaný a výsledok uložený do aplikácie. Zároveň je to predpríprava pre implementáciu budúcej funkcionality pre vystavanie kostry testovania, kde v danej kostre budú nespustené testy označené stavom **Untested**. Túto funkcionalitu, žiaľ v rámci tejto práce nebude možné plne implementovať, nakoľko v spoločnosti *Red Hat* ešte nie je pripravená infraštruktúra, ktorá by túto kostru vedela pred spustením testovania vygenerovať.

O túto špecifickú funkcionalitu prejavili záujem aj vývojári nástroja **ReportPortal**, kde návrh týchto zmien bol s nimi konzultovaný.

História exekúcií a filtrovanie prvkov V histórii exekúcií testov sú výsledky prvkov zobrazené neprehľadným spôsobom, kde položka v tabuľke sa viaže ku konkrétnemu testu, avšak nie ku špecifickému testovaciemu behu. Tabuľka zobrazuje niekoľko posledných exekúcií daného testu, čo dopĺňa určitú formu obrazu o stabilnosti testu, avšak nie je možné určiť v rámci akého testovania test prebehol. Požiadavka je, aby história výsledkov testov tvorila ucelenú tabuľku, kde na ose Y budú k dispozícii testované objekty (podľa úrovne zanorenia) a na ose X mená testovacích behov, ktorým daný výsledok prislúcha. V danej tabuľke sa budú môcť vyskytovať aj prázdne polia, nakoľko niektoré testy pre rôzne testovacie behy nemusia byť spustené.

ReportPortal umožňuje efektívne filtrovať objekty na ose Y, čo ale nie je podporované pre osu X, nakoľko tam v súčasnosti žiadne pokročilé usporiadanie prvkov neexistuje. Bude potrebné implementovať filter atribútov testovacích behov, aby bolo možné túto požiadavku pokryť.

Import vlastného formátu XML Vzhľadom na skutočnosť, že nástroj ReportPortal umožňuje import výsledkov testov pomocou formátu *XML* v ktorom sú naše výsledky testov k dispozícii na UMB, je pre nás výhodné upraviť formát výsledkov, aby bolo možné interpretovať výsledky testovacích fáz alebo testov spustených na viacerých architektúrach a zároveň upraviť funkciu importu aby dokázal tento formát akceptovať, preniesť a zobraziť do navrhnutej štruktúry. V rámci **ReportPortal-u** bude potrebné implementovať akceptáciu nových značiek *XML*, ich sémantiku a prenos do existujúcich objektov databáze, z ktorej budú výsledky následne zobrazené. V nasledujúcej časti bude predstavený **štandardizovaný formát XML**, ktorý bude základným vstupným rozhraním nástroja **ReportPortal**, kde z neho budú vychádzať zmeny a novo-implementované značky.

Správa „false negative“ výsledkov Správa „false negative“ výsledkov je k dispozícii aj v súčasnej podobe aplikácie vo forme typu chyby **No Defect**. Užívatelia však vyjadrili požiadavku, aby bola táto funkcionalita mierne pozmenená a boli upravené isté prvky štatistík. Užívatelia v rámci stretnutí vyjadrili žiadosť zmeniť tento stav na **Minor Defect**, nakoľko lepšie vystihuje požadovaný stav. Ďalej sa tento typ chyby nezobrazuje v štatistikách a chartoch, kam by ho radi pridali a tiež je neustále interpretovaný ako chybový stav, čo nemusí korešpondovať s realitou, nakoľko by sa takto označený neúspešný test nemal interpretovať v štatistikách zlyhania. V tomto bode sa jedná skôr o úpravy a nastavenie systému pre naše potreby, avšak bude potrebné implementovať isté zložky dopĺňajúce tento stav do štatistík.

Implementácia stavu aktuálne bežiaceho testu Pre potreby vystavania kostry testov pred ich spustením je potrebné v systéme navrhnúť a implementovať taktiež stav, ktorý by označoval aktuálne bežiaci test v rámci vystavanej kostry. Tento stav bude predstavovať jeden z dvoch stavov do ktorých sa automatický test po prechode zo stavu **Untested** môže dostať. Tento nový stav bude označený ako **Running** a bude dopĺňať stav **In Progress**, ktorý je v nástroji **ReportPortal** implementovaný. Dôvovom nevyužitia stavu **In Progress** pre túto potrebu je nemožnosť návratu objektu z koncového stavu s valídnym výsledkom naspäť do stavu **Running**. Je to z dôvodu, že ReportPortal neponúka možnosť znovu spustenia určitého testu, čo je v rámci bežnej práce v spoločnosti *Red Hat* bežný prípad použitia. Taktiež v prípade, že by sme chceli využiť stav **In Progress**, zmeny by boli vzhľadom na jeho implementáciu tak rozsiahle, že je pre nás jednoduchšie zachovať pôvodnu funkcionalitu tohto stavu a implementovať nový stav **Running** presne pre naše

využitie. V rámci spoločnosti sa snažíme čo najviac zachovať princíp zachovania funkcionality pôvodného softvéru z dôvodu udržovateľnosti kódu pri nasadení nových verzií nástroja a je predpoklad, že stav **In Progress** môže v budúcnosti nájsť využitie v rámci rôznych testovacích scenároch.

5.3 Nastavenie systému

Po implementácii chýbajúcej funkcionality je potrebné systém **ReportPortal** nastaviť tak, aby vyhovoval potrebám užívateľov. Zároveň je potrebné namapovať starý workflow a funkcionalitu na nový, avšak nie nutne pozmenený systém. Oblasti, v ktorých je potrebné dodatočné nastavenie a úpravy sú predstavené v nasledujúcich podsekciách.

Štruktúra testov a ich zobrazenie Pre uloženie a zobrazenie výsledkov testov využijeme flexibilitu existujúcej štruktúry tak, aby bolo možné efektívne prezerať výsledky testov v rôznej granularite. Pre vytvorenie štruktúry bude využitá možnosť opakovaného vnorenia objektu **Suite** popísaného v predchádzajúcej časti.

Základom je vytvorenie projektu pre určitú skupinu užívateľov, v ktorom budú zobrazené výsledky komponent s ktorými daná skupina pracuje. Na vrchole hierarchie v rámci projektu bude meno testovacieho behu, ktoré bude označovať k akému testovaniu sa daná exekúcia testov viaže, prípadne k akej komponente a jej konkrétnej verzii. Nasledovať budú **testovacie úlohy** (určitá izolovaná podmnožina testov konkrétneho plánu) viažúce sa k určitej verzii vybranej distribúcie operačného systému. V rámci každéj úlohy bude existovať predom definované množstvo automatizovaných, poloautomatizovaných a mauálnych testov. Následne môže byť každý test spustený na rôznych architektúrach, kde na každej architektúre sú spustené všetky fázy daného testu. Tieto fázy sú interpretované ako samostatné testy, kde výsledky každej fázy sú zobrazené v systéme. Každá úroveň tejto hierarchie môže obsahovať logy z testovania, avšak pre naše účely sú logy momentálne potrebné iba pre testy spustené na špecifických architektúrach a fázy. Z vyššie popísaného vyplýva, že výsledky budú tvoriť usporiadaný strom, kde na každej úrovni bude možné vidieť súhrn výsledkov danej vetvy s prislúchajúcimi štatistikami a taktiež históriu.

$$\textit{Test-run(Launch)} > \textit{Job(Suite)} > \textit{Test(Suite)} > \textit{Test-arch(Suite)} > \textit{Phase(Test)}$$

Namapovanie stavov testov z Nitrate na nové stavy Namapovanie stavov testov z Nitrate na nové stavy v nástroji **ReportPortal** je dôležité pre zachovanie zaužívaných workflow procesov v rámci spoločnosti. Užívatelia vo všeobecnosti neradi menia svoje zvyky a ocenia zachovanie existujúcich procesov s možnosťou dynamicky rozšíriť niektoré prvky o potrebné elementy. Presne túto možnosť nám ponúka **ReportPortal** v podobe vytvárania podkategórii zlyhaní. Navrhnutým kategóriam je možné dynamicky pre každý projekt zadefinovať lokálne podkategórie, ktoré lepšie zadefinujú množinu zlyhaní testov pre každý tím osobitne. V praxi je bežné, že tím vyvíjajúci a testujúci komponenty jadra OS bude mať inú sadu kategórii zlyhaní testov ako tím, ktorý sa venuje užívateľskému rozhraniu. Týmto spôsobom nie je potrebné hľadať kompomisy medzi týmami, ale umožňiť spomínanu voľnosť a flexibilitu.

Predtým, ako budeme definovať kategórie chýb, ktoré sú rozšírením statusov, je potrebné si zadefinovať nové statusy, ktoré vychádzajú zo statusov systému **Nitrate**, ktorých prechodový diagram je predstavený na obrázku 2.1. Využijú sa už spomenuté novo-implementované

Tabuľka 5.1: Tabuľka mapovania pôvodných statusov na nové

Nitrate status	ReportPortal status	Defect type
Blocked	Failed	to_investigate
Paused	Failed	to_investigate
Idle	Untested	-
Running	Running	-
Passed	Passed	-
Waived	Failed	known_bug, minor_defect
Error	Failed	system_issue, test_bug
Failed	Failed	new_bug

statusy **Running** a **Untested**, aby mohol byť prechod na iný systém čo najplynulejší. Mapovanie statusov spolu s novými kategóriami zlyhaní je možné vidieť v tabuľke 5.1. Pre účel namapovania stavov bolo zorganizované niekoľko-hodinové stretnutie s užívateľmi, kde boli vyjasnené všetky požiadavky a možnosti. Využila sa už spomínaná flexibilita nástroja **ReportPortal**, kde boli **Nitrate** stavy **Waived** a **Error** explicitne rozšírené o kategórie zlyhaní, aby lepšie popisovali daný stav. **Waived** zahŕňal očakávané zlyhania a taktiež nepodstatné, čo bude v **ReportPortal-e** rozlíšené a **Error** zahŕňal chyby infraštruktúry aj testov, čo bude taktiež rozlíšené.

Z vyššie uvedenej tabuľky vyplýva, že je potrebné kategórie zlyhania (defect type) odstrániť a niektoré pridať. Jedná sa o odstránenie kategórii **Automation Bug**, **No Defect** a pridanie už spomínaného **Minor Defect**, **Test Bug** a rozšírenie kategórie **Product Bug** o subkategórie **Known Bug** a **New Bug**.

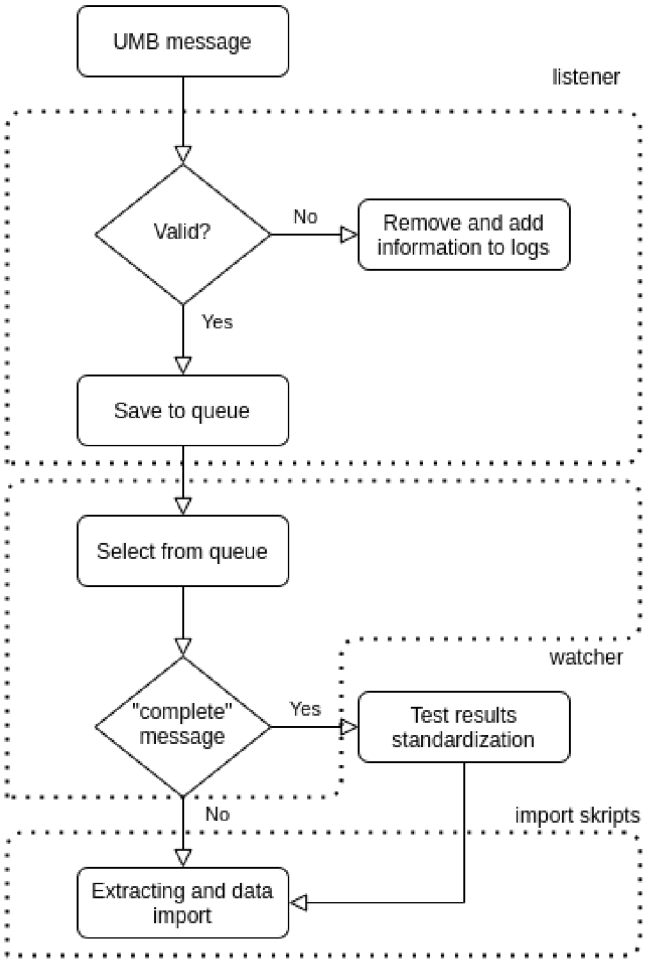
Uloženie metadát Pre uloženie metadát a informácií o testovaní bude použitá prehľadná štruktúra atribútov, kde na každej úrovni zanorenia je možné špecifikovať iné hodnoty. Systém umožňuje efektívne filtrovanie v rámci týchto atribútov, čo zapadá do nášho konceptu.

Nastavenie potrebných zásuvných modulov Vzhľadom k dostupným zásuvným modulom, bude potrebné nastaviť možnosť prihlasovania užívateľov cez OAuth2 alebo SAML protokoly.

Do úvahy pripadá aj možnosť aktivácie automatickej analýzy chýb, avšak táto možnosť je primárne určená pre výstupy z podporovaných testovacích frameworkov a nie je v súčasnej dobe jasné, či bude daný zásuvný modul spoľahlivo fungovať aj pre nami poskytnuté výstupy a framework *BeakerLib*. V rámci tohto frameworku je časté používanie náhodne vygenerovaných mien súborov, prípadne vstupných dát, čo z nich robí ťažko analyzovateľné objekty na základe zhody textových reťazcov. Je pozitívne a do budúcna veľmi prísľubné, že daná funkcionalita je k dispozícii, avšak vzhľadom na strednú prioritu tejto požiadavky, nutnosť pokročilej analýzy, testovania v prevádzke a prípadných zmien funkcionality pre potreby spoločnosti *Red Hat*, nie je aktivácia tohto zásuvného modulu prioritná pre túto prácu a bude pravdepodobne predmetom záujmu možných rozšírení po úspešnom nasadení nástroja **ReportPortal** do používania.

5.4 Návrh infraštruktúry pre komunikáciu s UMB

Infraštruktúra bude zapezpečovať tok dát do nástroja **ReportPortal** v požadovanej forme. Bude to sada skriptov zachytávajúca a validujúca pre nás dôežité správy z **UMB**, ktoré budú ukladané do perzistentnej fronty na disku. Správy z tejto fronty budú vyberané v poradí v akom dorazili, budú z nich extrahované dôležité informácie. Následne sa prevedú do štandardizovanej formy a importnú do nástroja **Reportportal**. Grafické zobrazenie vyššie popísanej postupnosti je možné vidieť na obrázku 5.9.



Obr. 5.9: Fungovanie jednotlivých častí infraštruktúry

Modul listener pre zachytávanie správ z UMB Pre zachytávanie správ z UMB bude slúžiť modul **listener**, ktorý bude načúvať na kanáloch obsahujúcich informácie ohľadom testovania RPM balíkov (*brew-build*) a modulov (*redhat-module*) pre *Red Hat Enterprise Linux*. Na týchto kanáloch sú k dispozícii štyri typy správ predstavujúce aktuálny stav testovania konkrétnej úlohy, ktoré sú uložené vo formáte *JSON*. Prvou z nich je správa **queued** predstavujúca zadanie testovania do fronty pre zadávateľa, avšak v tomto prípade testovanie ešte nezačalo. Správa **running** signalizuje začiatok behu testov danej úlohy a následne je očakávaná koncová správa, ktorou môže byť správa **error** signalizujúca, že nastal

problém s testovacou infraštruktúrou alebo **complete** obsahujúca výsledky testov danej úlohy.

V súčasnosti modul listener nebude zachytávať správu **queued**, nakoľko je to zbytočné vzhľadom na to, že táto správa momentálne neobsahuje žiadne údaje o plánovaných testoch. V prípade, že budú v budúcnosti tieto správy rozšírené o informácie o konkrétnych plánovaných/bežiacich testoch, bude aj listener rozšírený o túto funkcionalitu. Tieto informácie sú potrebné pre vystavanie potrebnej kostry testov, ktorá bola vysvetlená v súvislosti so statusmi **Untested** a **Running**, kde tieto stavy by mali kopírovať konkrétne správy.

Cieľ pre túto prácu bude zachytávať správy **running**, **error** a **complete**, kde každej správe bude overená validita a v prípade úspechu uložená do perzistentnej fronty.

Perzistentná fronta správ Perzistentná fronta bude implementovaná formou obyčajných súborov uložených na disku v konkrétnom adresári. Toto riešenie bolo zvolené na základe potreby uchovať dáta aj v prípade výpadku určitej podčasti systému. Pre tento adresár bude implementovaný modul **watcher**, ktorý bude sledovať zmeny súborov tohto adresára a bude na to reagovať. V prípade uloženia nových správ do fronty zo strany modulu **listener**, bude **watcher** tieto správy vyberať v poradí v akom dorazili na **UMB** a predávať ich skriptom zabezpečujúcim štandardizáciu formátu dát a následný import do **ReportPortal-u**.

Modul parser pre štandardizáciu dát získaných zo správ V prípade prijatia správy **complete** a uloženia do perzistentnej fronty budú výsledky testovania, ktoré táto správa obsahuje, spracované modulom parser a prevedené do štandardizovaného formátu. Tieto výsledky sú uložené vo formáte *XML* podobajúce sa štandardu *JUnit XML*. Štandardizácia vstupných dát je potrebná z dôvodu, že na **UMB** sú zachytávané správy z rôznych testovacích prostredí a neexistuje presný formát, podľa ktorého by mali tieto výsledky testov vyzerať. Je potrebné preto zjednotiť formáty výsledkov prevodom do definovaného štandardu, ktorý je akceptovaný nástrojom **ReportPortal** a je k dispozícii v prílohe B. Daný štandard sa snaží čo najbližšie držať štandardu *Junit XML*, avšak je pre naše účely rozšírený o nové prvky. Vzhľadom k tomu, že dáta o konkrétnych testoch obsahuje momentálne iba správa **complete**, nie je pre dáta v správach **running** a **error** potrebná štandardizácia, nakoľko tieto správy obsahujú iba metadáta o konkrétnej úlohe.

Sada skriptov pre import dát do ReportPortalu Sada skriptov pre import dát bude mať funkciu extrahovania potrebných metadát zo správ a následný import týchto informácii do ReportPortalu, potenciálne aj so štandardizovanými výsledkami testovania v prípade správy **complete**. Pre každú správu bude definovaný skript, aby bola zachovaná prehľadnosť a možnosť jednoduchého rozšírenia pre budúcnosť.

Pri správe **running** bude vytvorená bežiaca úloha, ktorá neobsahuje informácie o bežiacich testoch a následne pri príchode správy **complete**, budú informácie o výsledkoch testovania doplnené. V prípade, že vznikne problém s infraštruktúrou a **listener** obdrží správu **error**, bude úloha označená ako zlyhaná s chybou infraštruktúry.

Kapitola 6

Implementácia infraštruktúry a chýbajúcej funkcionality

Implementácia chýbajúcej funckcionality spolu s infraštruktúrou pre import výsledkov prebiehala iteratívnym štýlom. Po úvodnom zozbieraní užívateľských požiadaviek bol vytvorený prototyp vychádzajúch z **ReportPortal-u** verzie 5.2. Cieľom tohto prototypu bolo predviesť požadovanú funkcionalitu užívateľom na nasledujúcich stretnutiach a potvrdiť, že nástoroj **ReportPortal** je použiteľný a zhoduje sa s predstavami užívateľov.

V úvode bolo podstatné zoznámiť užívateľov s týmto nástrojom a diskutovať o možných zmenách a nastaveniach pre potreby spoločnosti *Red Hat*. V momente, kedy požiadavky začali naberať konkrétne rysy, začala postupná implementácia jednotlivých častí. Ako prvé bolo potrebné nastavenie štruktúry testov a overenie správnosti zobrazenia výsledkov. Po potvrdení nasledovala implementácia správneho zobrazenia histórie podľa požiadaviek užívateľov a k tomu prislúchajúcich filtrov, čo bola značne problematická časť. Po následnom overení nasledovala implementácia stavu **Untested** spolu s návrhom a implementáciou štandardizovaného formátu XML potrebného pre hromadný import výsledkov do nástroja. Stav **Untested** mal odzrkadlovať použitie manuálnych testov a ktorého funkcionalita bola diskutovaná aj s vývojármi **ReportPortal-u** cez konferenčný hovor. Táto funkcionalita bola následne prijatá upstreamom a bude súčasťou oficiálnych verzií nástroja.

Po implementácii vyššie popísaných prvkov bolo podstatné systém nasadiť do testovacieho používania a dodať do neho reálne dáta. To vyžadovalo implementáciu infraštruktúry, ktorá získava dáta o testovaní z **UMB**, spracuje a importuje ich do **ReportPortal-u**. Implementácia tejto časti prebiehala štandardným spôsobom, nakoľko požiadavky aj štandardizovaný formát bol už definovaný a nebolo potrebné z užívateľmi konzultovať podrobnosti.

Po úspešnej implementácii infraštruktúry bol prototyp nasadený do testovacieho používania, ktoré odhalilo problémy spojené s importom reálnych dát, otestovalo správnu funkčnosť systému a potvrdilo správne pokrytie určitých požiadaviek. Prototyp bol nasadený za pomoci technológie **Kubernetes**, čo sa však ukázalo ako nespoľahlivé riešenie a preto nebolo použité pri finálnom nasadení. V celkovom obraze panovala spokojnosť užívateľov s nástrojom, avšak boli isté nezrovnalosti, ktoré bolo potrebné doplniť. Až pri prototype sa ukázala potreba presne namapovať stavy nástroja **Nitrate** no nové stavy, čo viedlo k potrebe implementácie stavu **Running** a taktiež úpravy a implementácie nových kategórií zlyhaní. Tieto zmeny, spolu s drobnými úpravami chovania niektorých prvkov a užívateľského rozhrania, boli následne z časti premietnuté do prototypu a úplne do finálnej verzie, ktorá vychádzala z **ReportPortal-u** verzie 5.3. Výhoda vyššie popísaného procesu a využi-

tia prototypu znamenala zachytenie problematických prvkov hneď v úvode vývoja a taktiež bezproblémového behu a nasadenia finálnej verzie. Pri finálnom nasadení bola použitá už otestovaná infraštruktúra, čo taktiež minimalizovalo množstvo problémov.

V nasledujúcich sekciách je detailne predstavená implementácia alebo úprava jednotlivých častí systému, ktoré popisujú jej finálny stav po nasadení do používania.

6.1 Nástroj ReportPortal

Pre implementáciu nástroja ReportPortal sú použité štandardné technológie pre vývoj webových aplikácií. Aplikačná logika je implementovaná v jazyku *Java* využívajúci *Spring framework* poskytujúci aplikačné rozhranie, užívateľské rozhranie formou hrubého klienta za pomoci *ReactJS* a dátová vrstva využíva *PostgreSQL*. Vzhľadom na to, že v tejto práci sa venujeme primárne rozširovaniu a upravovaniu funkcionality nástroja **ReportPortal**, budeme tieto technológie využívať aj my.

Funkcionalita nástroja nie je implementovaná v rámci jedného repozitára, ale je logicky rozdelená do viacerých, podľa poskytujúcej funkcionality alebo časti systému. Najmä aplikačná logika systému je rozdelená do množstva repozitárov, kde každý implementuje určitú časť funkcionality, ako napríklad hlavný repozitár aplikačného rozhrania obsahujúci *Controllery*, repozitár implementujúci *Moduly*, repozitár pre implementáciu *Handlerov* pre Moduly a mnoho iných. Následne sú tieto repozitáre pomocou nástroja *Gradle* skompilované do jedného modulu. Rovnaký princíp rozdelovania funkcionality do logicky oddelených častí je použitý pre väčšinu častí nástroja. Implementácia konkrétnych zmien bude predstavená v nasledujúcich podsekciách.

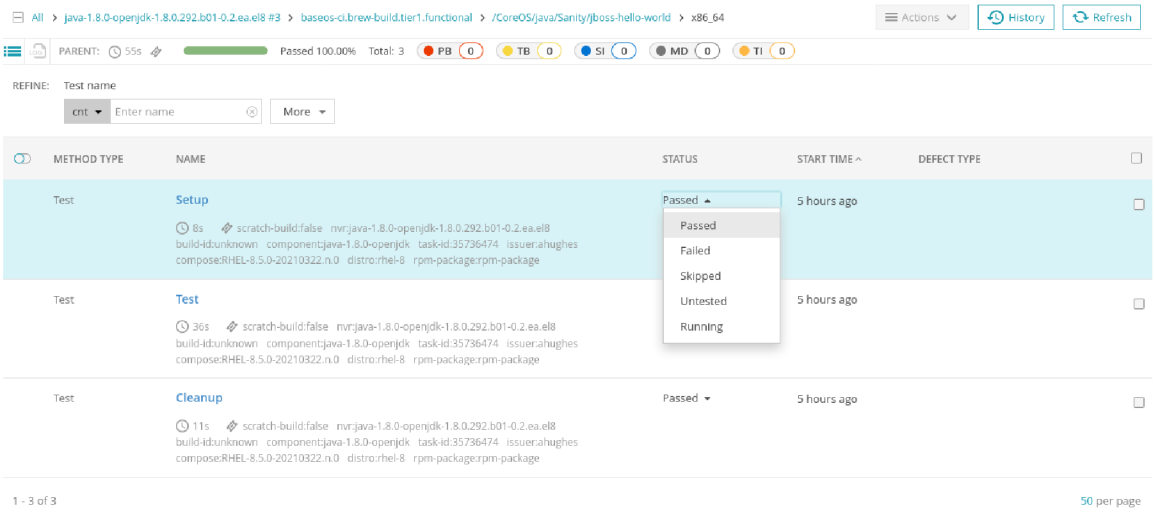
6.1.1 Implementácia stavov Untested a Running

Potreba implementácie stavov **Untested** a **Running** bola vysvetlená v predchádzajúcej časti tejto práce. Išlo o zmenu, ktorá sa dotkla všetkých častí nástroja ReportPortal. Počínajúc databázovou vrstvou, bolo nutné rozšíriť aktuálnu množinu stavov testov o nové, ktoré sú v rámci inicializačných skriptov uvedené ako dátový typ *enum*, rozšírenie tabuľky štatistík o nové stavy a úprava automatických procedúr alebo triggerov pre automatické prehodnotenie stavu rodičov daného objektu pri zmene jeho stavu.

Úpravy v rámci užívateľského rozhrania zahrňovali pridanie stavov do objektov zobrazujúcich štatistiky, filtre, nastavenie CSS a pridanie nových objektov zobrazujúcich a popisujúcich dané stavy v rámci aplikácie. Jednalo sa o doplnenie nových položiek do existujúcej funkcionality, čo z implementačného hľadiska nebolo náročné, avšak bolo potrebné poznať danú aplikáciu a doplniť potrebné kusy kódu na správne miesta.

V rámci aplikačnej logiky sa zmeny sústreďovali na implementáciu možnosti zmeny statusu testu. Táto zmena zahrňovala potrebu implementácie nových tried **ToRunningStatusChangingStrategy**, resp. **ToUntestedStatusChangingStrategy**, kde obe tieto trie-dy rozširovali abstraktnú triedu **AbstractStatusChangingStrategy** zabezpečujúcu zme-nu statusu testu a jemu definovaných rodičovských prvkov. Okrem implementácie nových tried prebehlo aj doplnenie statusov do rôznych **Handlerov** spracúvajúcich štatistiky alebo filtre. Dôležitou zmenou bolo doplnenie a úprava jednotkových testov a s nimi súviasiacimi testovacími dátami, ktoré sú vyhodnocované pri každej kompilácii kódu a zabezpečujú kontrolu nezanesenia regresných chýb pri zmenách kódu. Výslednú podobu implementácie nových statusov testov je možné vidieť na obrázku 6.1 a zobrazenie štatistík týchto statusov a ostatných typov chýb na obrázku 6.3.

Ako bolo zmienené aj v predchádzajúcich kapitolách, o implementáciu stavu **Untested** prejavili záujem aj vývojári **ReportPortal-u**, kde bolo toto nami implentované rozšírenie ponúknuté formou pull-requestov a z ich strany prijaté a mergnuté do oficiálnej verzie nástroja. V budúcnosti nie je z toho dôvodu potrebné udržiavať tieto zmeny a stav **Untested** bude súčasťou oficiálnej verzie k dispozícii pre akéhokoľvek užívateľa, ktorý sa rozhodne **ReportPortal** využívať.



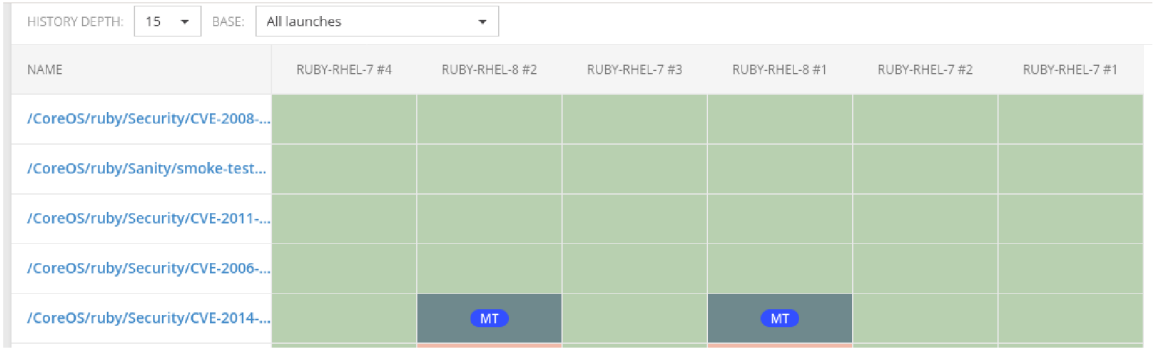
Obr. 6.1: Nové statusy testov

6.1.2 Úprava zobrazenia histórie a implementácia aditívnych filtrov

Zobrazenie histórie exekúcií v rámci oficiálnej verzie ReportPortalu nebolo pre naše potreby príliš použiteľné. Ako bolo predstavené v návrhu, je potrebné zoradiť prvky histórie do ucelenej tabuľky, kde na ose X sú definované mená testovacích behov a na ose Y testované objekty podľa úrovne zanorenia. Vzhľadom k tomu, že endpoint aplikačného rozhrania **ReportPortal-u** dodáva kompletné informácie o požadovaných testoch, je možné tieto dáta usporiadať podľa časovej osy a vytvoriť na ose X hlavičku tabuľky definujúcu, ktoré testovacie behy budú zobrazené v histórii. Na základe tejto hlavičky je možné získané dáta o výsledkoch testovania usporiadať a vytvoriť prehľadnú tabuľku, ako je to možné vidieť na obrázku 6.2 namiesto iba sekvenčného vypísania výsledkov daného testu za posledných M definovaných spustení, ako to je vidieť na obrázku 5.6. Tieto zmeny sa týkali modulu definujúceho užívateľské rozhranie, nakoľko neboli potrebné zmeny v rámci aplikačnej logiky, ktorá poskytovala všetky potrebné dáta.

Po úprave zobrazenia histórie sa ponúkla možnosť filtrovania v rámci zobraziteľných testovacích behov, nakoľko táto funkcionalita nebola v oficiálnej verzii **ReportPortal-u** prítomná. Cieľom bolo, aby bolo možné fitrovať prvky zobrazené na ose Y, tak ako aj na ose X a vytvárať ucelený užívateľom definovaný obraz. V tejto fáze nastal problém, nakoľko filtre sú v rámci tohto nástroja implementované spôsobom, ktorý ich nedovoľuje jednoducho rozširovať o nové prvky. Filter je definovaný v určitom kontexte z ktorého si dynamicky generuje použiteľné fitre. Pre jednoduchosť si uvedieme príklad v kontexte testovacích behov a testov. V prípade, že by sme chceli v rámci kontextu testov pridať filtre použité v kontexte testovacích behov, nastáva problém, nakoľko tieto veci si navzájom odporujú. Našim cieľom bolo existujúci filter v kontexte testov rozšíriť o filter atribútov testovacích

behov, ktorý je logicky prítomný v kontexte testovacích behov. Vzhľadom k využitému polymorfizmu vo vrstvách užívateľského rozhrania aj aplikačnej logiky nebolo v tom prípade možné využiť existujúcu logiku užívateľského rozhrania a kontext pre tieto filtre, ale bolo potrebné definovať úplne nový prvok s pevne definovaným kontextom. Tento nový prvok pre fitrovanie atribútov testovacích behov bol naviazaný na upravený endpoint aplikačného rozhrania, ktorý zabezpečoval dáta pre zobrazenie histórie. Tento filter musel byť definovaný ako samostatný parameter HTTP požiadavky, nakoľko pridanie do filtra daného kontextu nebolo možné z dôvodu rozdielneho kontextu. Následne boli upravené dotazy na databázovú vrstvu aby filtrovanie potrebných atribútov prebiehalo správne. Táto požiadavka mala vysokú prioritu, bola konzultovaná aj s vývojármi **ReportPortal-u**, ktorí potvrdili nemožnosť pridania filtrov z iného kontextu. Z toho dôvodu bolo potrebné uchýliť sa k API nekompatibilným zmenám, aby sme dosiahli požadovaný výsledok a vyhoveli užívateľom.



Obr. 6.2: História výsledkov po úprave

6.1.3 Implementácia možnosti importu nového formátu XML

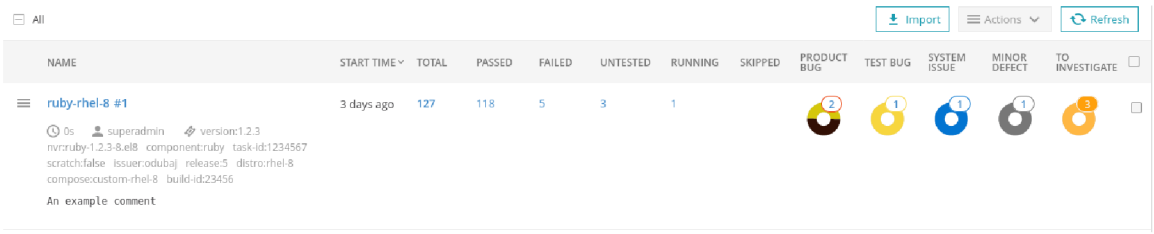
Funkcia importu výsledkov pomocou formátu XML bola v rámci nástroja **ReportPortal** implementovaná, avšak iba v miere, kde akceptovala veľmi oklieštený formát *JUnit XML*. Pre použitie v rámci spoločnosti *Red Hat* bolo potrebné navrhnúť a implementovať nový **štandard formátu XML**, ktorý by bol akceptovaný **ReportPortal-om** a pokrýval by potrebné užívateľské požiadavky.

Import výsledkov zabezpečuje trieda **ZipImportStrategy**, rozširujúca abstraktnú triedu **AbstractImportStrategy**, zabezpečujúca spracovanie XML súboru skomprimovaného v archíve, kde trieda **ParseResults** pomocou handleru **XunitImportHandler** spracuje obsah vstupného súboru a uloží dáta. Vzhľadom k tomu, že v oficiálnej verzii nástroja chýbala podpora uloženia metadát vo forme atribútov, bola trieda **ParseResults** rozšírená o túto funkcionalitu. Ďalej bolo potrebné v triede **XunitReportTag**, ktorý reprezentuje výčet akceptovaných elementov, rozšíriť množstvo XML tagov o novo-navrhnuté elementy a atribúty podľa definovaného štandardu. Ako posledný krok bolo nutné upraviť funkcionalitu handleru **XunitImportHandler** pre konkrétne spracovanie jednotlivých elementov, aby boli sémanticky správne interpretované, ukladané a následne zobrazené v samotnej aplikácii. Handler je implementovaný ako množina metód reagujúca na začiatok alebo koniec elementu, jeho názov, obsah alebo vybrané atribúty a ich hodnoty.

6.1.4 Implementácia nových kategórii zlyhaní

Potreba implementácie nových užívateľsky prívetivých kategórii a podkategórii zlyhaní bola definovaná užívateľmi v rámci štúdia funkcionality prototypu a mapovania stavov testov z **Nitrate** do nástroja **ReportPortal**. Vychádzajúc z návrhu a tabuľky X, našim cieľom bolo odstránenie kategórii **Automation Bug** a **No Defect**, pridanie kategórii **Minor Defect** a **Test Bug** a rozšírenie kategórie **Product Bug** o subkategórie **Known Bug** a **New Bug**. Rovnako ako pri implementácii nových statusov testov, nešlo o pridanie novej funkcionality, ale o úpravu, resp. o rozšírenie tej existujúcej. Implementované kategórie je možné vidieť na obrázku 6.3, kde zmeny je možné porovnať s obrázkom 5.4.

Tieto úpravy sa dotkli dátovej vstvy, kde bola potrebná úprava inicializačných skriptov databázy a taktiež rozšírenie tabuľky pre zaznamenávanie štatistík o nové položky. V rámci aplikačnej logiky išlo o doplnenie týchto kategórii do Handlerov spracúvajúcim štatistiky, čo súvisí s úpravou dátovej vrstvy. Následne boli rozšírené jednotkové testy spolu s testovacími dátami pre zabezpečenie kontroly nezanesenia regresii do systému. V rámci užívateľského rozhrania bola práca sústredená na doplnenie týchto kategórii do filtrov, boli vytvorené nové objekty pre správu a zobrazenie štatistik a taktiež potreba modifikácie správania kategórie **Minor Defect**, ktorú si užívatelia definovali ako možnosť pre správu „False Negative“ výsledkov. Požadovaný princíp bol, aby boli testy so stavom **Failed** a kategóriou **Minor Defect** interpretované ako úspešné. V tomto konkrétnom prípade bolo potrebné upraviť niektoré zadefinované výpočty štatistík v rámci užívateľského rozhrania, nakoľko implementácia nižších vrstiev nedovolovala modifikovať túto funkcionalitu a interpretácia výsledku testu v štatistikách je závislá iba od stavu testu. Vzhľadom k tejto zmene bolo potrebné pri spracovaní túto kategóriu testovať a následne upraviť prvky v rámci zobrazenia histórie, filtrov a vykreslovania grafov štatistík.



Obr. 6.3: Kategórie zlyhaní s prehľadnými štatistikami a uložením metadát

6.1.5 Nastavenie štruktúry testov, uloženie metadát a úpravy užívateľského rozhrania

Nastavenie štruktúry a zapúzdrenia testov v rámci testovacích plánov a architektúr je priamo závislé od formátu vstupného štandardizovaného XML súboru, ktorý svojou štruktúrou definuje a následne do nástroja **ReportPortal** importuje tieto informácie tak, aby boli pre užívateľov čitateľné. Znamená to, že vytvorenie požadovanej štruktúry sa deje pri vytváraní a štandardizovaní výsledných dát z testovania, kde **ReportPortal** slúži v tomto smere pre spracovanie a efektívne zobrazenie informácii. Tento koncept nám ponúka možnosť veľmi flexibilne v budúcnosti pridávať aj iné typy testovaní a meniť štruktúru testov podľa potrebných kritérii bez zásahu do zdrojvého kódu aplikácie.

Rovnakým spôsobom je riešené uloženie metadát do systému (obrázok 6.3), kde tieto dáta sú taktiež súčasťou štandardizovaného vstupného súboru. Okrem atribútov a paramet-

rov testovania je v nich prítomný aj odkaz na zdrojový kód testu, aby užívateľ v prípade potreby dokázal efektívne porovnávať kód s jednotlivými logmi výsledkov testov. Za zmienku stojí aj uloženie identifikátora testu na základe ktorého sa vyhodnocuje história exekúcií. Tento identifikátor je vyhodnotený spolu s parametrami pre spustenie testu a následne zašifrovaný do unikátneho hashu. V súčasnosti je pre identifikátor využité meno testu, avšak v budúcnosti bude pre tento účel využitý FMF identifikátor predstavený v kapitole 3.

Vzhľadom na nie úplne vyhovujúci systém správy užívateľov bolo potrebné upraviť oprávnenia istých rolí. V pôvodnej verzii užívatelia nemali možnosť prezerať a modifikovať testovacie behy, ktorých neboli vlastníkmi. Nakoľko vlastníkom všetkých testovacích behov bude automatický systém importujúci dáta do nástroja **ReportPortal**, bolo potrebné upraviť oprávnenia rolí **Member** a **Project_Manager**, aby mali prístup k akémukoľvek testovaciemu behu v rámci daného projektu.

V rámci úprav užívateľského rozhrania prebehli drobné zmeny týkajúce sa zobrazenia jednotlivých prvkov, úprava CSS, orezanie a prípadné modifikované zobrazenie určitých mien objektov, všetko na základe požiadaviek užívateľov, ktoré boli definované v rámci využitia prototypu.

6.2 Testovacia infraštruktúra

Pre implementáciu testovacej infraštruktúry pre zachytávanie dát z **UMB**, ich štandardizáciu a následný import do nástroja **ReportPortal** boli vybrané jazyky *Python* a *Bash*. *Python* bol vybraný ako jazyk ponúkajúci množstvo modulov a nástrojov pre efektívne zachytenie správy, overenie správneho formátu, štandardizáciu vstupného súboru a jeho následné predanie na spracovanie ďalšiemu skriptu.

V jazyku *Bash* sú implementované skripty pre sieťovú komunikáciu s nástrojom **ReportPortal**, kde je využívaný nástroj **curl**, pomocou ktorého sú vystavané HTTP požiadavky pre komunikáciu. *Bash* bol zvolený z dôvodu veľmi priamočiarej a prehľadnej implementácie týchto požiadaviek, hoci *Python* tiež ponúka túto možnosť pomocou modulu **pycurl**, napokon prevážila prehľadnosť a jednoduchosť *Bash*-u.

6.2.1 Implementácia modulu listener pre zachytávanie správ z UMB

Pre implementáciu modulu listener pre zachytávanie správ z UMB bol využitý jazyk *Python*. Základ skriptu tvorí trieda **UMBReceiver**, ktorá implementuje triedu **MessagingHandler** z modulu **proton**, poskytujúca funkcie pre komunikáciu s *AMQ Broker* pomocou *AMQP* protokolu.

Samotný modul listener počúva na vopred definovaných kanáloch predstavených v návrhu, z ktorých zachytáva správy, overuje ich validnosť pre účely ďalšieho spracovania a následne ukladá vo forme JSON súborov do perzistentnej fronty. Trieda **UMBReceiver** obsahuje niekoľko metód pre správne fungovanie modulu **listener**, zachytávanie a napokon spracovanie správ. Pri spustení modulu listener sa inicializuje trieda za pomoci metódy **___init___()**, následne sa listener pomocou metódy **on_start()** a certifikátu pripojí k danej komunikačnej zbernici a začne zachytávať správy z vopred definovaných kanáloch. Pri zachytení takejto správy sa aktivuje metóda **on_message()**, ktorá danú správu dekóduje, overí či je pre naše účely validna a ak áno, uloží do perzistentnej fronty. V prípade, že správa validna nie je, bude zahodená, avšak informácia o zachytení takejto správy bude uložená v logoch, pre ktorých implementáciu bol využitý štandardný modul **logging**. V prípade, že s modulom **listener** nastane nejaký problém s pripojením, spracujú ho metódy

on_link_error() alebo **on_transport_error()**, ktoré zaručia jeho plynulý chod. V prípade, že by sa vyskytol problém, kde by **listener** neočakávane zlyhal, bude po 10 sekundách nanovo inicializovaný a spustený.

Pre spustenie a trvalý beh modulu **listener** aj po odhlásení užívateľa z daného stroja bol využitý nástroj **nohup** [15], ktorý ignoruje *SIGHUP* signál vyvolaný pri zavretí terminálu, teda pri odhlásení uživateľa. Jeho výhodou je, že je možné nechať bežať skrity neobmedzene dlho na akomkoľvek stroji. V našom prípade bol skript modulu **listener** spustený aj na pozadí (výpis 6.1), aby svojou činnosťou neobmedzoval užívateľa v ďalšej práci.

```
1    nohup python3 -u listener.py &
```

Výpis 6.1: Ukážka spustenia modulu listener

6.2.2 Implementácia modulu watcher a perzistentnej fronty

Rovnako ako u modulu listener, aj pre implementáciu modulu watcher bol použitý jazyk *Python*. Základom skriptu je využitie modulu **watchdog** v dvoch triedach **Watcher** pre sledovanie perzistentnej fronty, ktorá je implementovaná vo forme priečinku, kde položky sú vo forme súborov a triede **Handler** implementujúcej triedu **FileSystemEventHandler** pre obsluhu fronty.

Trieda **Watcher** na začiatku inicializuje a vytvorí inštanciu triedy **Observer** z modulu **watchdog.observers**, ktorá zabezpečuje sledovanie daného priečinku pre zmeny. Tento objekt pri zaznamenaní zmeny alebo vytvorenia súboru má definovaný handler, ktorý je objektom triedy **Handler** a spracováva dané položky vo fronte. Trieda **Handler** ma definovanú metódu **on_any_event()**, ktorá pri udalosti reaguje a obslúži danú požiadavku. V prvom kroku začne spracovávať danú položku, ktorou je správa UMB uložená vo formáte JSON podľa toho, či sa jedná o správu pre *module-build* alebo štandardný *brew-build*. Rozdielne spracovanie je z dôvodu, že dané správy majú mierne odlišný formát a sú z nich extrahované informácie z rôznych položiek pre ďalšie spracovanie. Ďalej na základe toho o aký typ správy sa jedná (**running**, **complete**, **error**), je zavolaný príslušný skript pre import dát, ktorý získané informácie spracuje do jednej alebo niekoľkých HTTP požiadaviek a importuje ich do **ReportPortal-u** pomocou aplikačného rozhrania. V prípade, že sa jedná o správu **complete**, sú v rámci triedy **Handler** extrahované výsledky testov prítomné v položke xunit, uložené do súboru a predané k spracovaniu skriptu pre import dát zo správy complete.

V prípade, že by prišlo k situácii, že sa do fronty uloží väčše množtvo položiek naraz, **watcher** ich bude spracovávať v poradí v akom boli zachytené na **UMB**, až pokým nebude fronta prázdna. Taktiež sa všetky činnosti modulu watcher ukladajú do logovacieho súboru na čo je rovnako ako v prípade modulu **listener** využitý modul **logging**. Pre spustenie a trvalý beh aj po odhlásení užívateľa z daného stroja je využitý nástroj **nohup** [15] predstavený v predchádzajúcej podsekcii fungujúci na rovnakom princípe ako u modulu **listener**.

6.2.3 Implementácia modulu parser a skriptov pre import dát

Ako už bolo uvedené v prechádzajúcej časti, na UMB nás v súčasnosti budú zaujímať tri typy správ na vopred definovaných kanáloch. Medzi ne patrí správa **running**, definujúca začiatok behu testov určitej úlohy, správa **complete** signalizujúca koniec behu testov obsahujúca podrobné informácie o výsledkoch testovania a správa **error** vyjadrujúca chybu

testovacej infraštruktúry. Všetky tieto správy sa viažu k určitému konkrétnej verzii RPM balíku na ktorom je vykonávané testovanie. Tento RPM balík je jednoznačne identifikovaný položkou **task-id**, ktorý má podobu celočíselného dátového typu, čo znamená, že pod konkrétnym testovacím behom sa budú zjednocovať testovacie úlohy práve s týmto **task-id**. V praxi to vyzerá, že meno launchu je zároveň menom balíka, jeho verziou a položkou release. V reálnych dátach sa vyskytujú taktiež aj výsledky testov viažuce sa ku testovacím kompiláciam RPM balíkov, ktoré slúžia k otestovaniu v rámci vývoja a k jednoduchému spusteniu testov pre túto verziu balíka. Tieto testovacie RPM balíky sú v metadátach označené ako „*scratch*“, aby ich bolo možné jednoducho odfiltrovať.

Po spracovaní správy modulom **watcher**, je určený jej typ a na základe neho watcher spúšťa príslušný skript pre import dát. V prípade správy **running** skript bude vytvárať bežiacu úlohu v rámci štruktúry testov predstavenej v návrhu. V prípade, že neexistuje ešte testovací beh s daným **task-id**, vytvorí sa. Spolu s vytvorením objektov sa do **ReportPortal-u** uložia aj metadáta testovania ako atribúty. Správa **running** spolu so správou **queued**, ktorú v momentálnej fáze nezachytávame, nie sú z hľadiska infraštruktúry UMB implementované úplne, nakoľko v nich chýbajú informácie o aktuálne bežiacich alebo vybraných testoch, kde by sa na základe týchto informácii vystavala kostra testov.

V prípade spracovania správy **error**, skript pomocou aplikačného rozhrania ReportPortalu vyhľadá aktuálne bežiacu úlohu podľa mena a definovaného **task-id**, zastaví ho a doplní do neho informácie o zlyhaní spolu s odkazmi na detailné logy. Rovnako ako v predchádzajúcom prípade, ak neexistuje daná úloha alebo testovací beh, väčšinou z dôvodu stratenej správy, sú dané objekty vytvorené, aby bola zachovaná konzistencia dát. V tomto prípade ale daná úloha nebude v stave **running**, ale priamo označená ako zlyhaná.

Je zrejmé, že pri správe **complete** bude scenár spracovania podobný s predchádzajúcimi správami. Avšak správa **complete** nesie informáciu o podrobných výsledkoch testovania, kde tieto informácie musia byť extrahované, štandardizované do navrhnutého formátu predstaveného v prílohe B a až následne importnuté do **ReportPortal-u**. Pre štandardizáciu bol použitý skript **standardize_xunit.py**, ktorý pomocou modulu *lxml* načíta dáta zo vstupného súboru, spracuje ich a vytvorí nový súbor vo formáte, ktoré ReportPortal akceptuje. Daný parser je implementovaný princípom rekurzívneho zostupu, kde počas načítavania informácii priamo generuje nový formát.

Tento vstupný súbor je modulom **watcher** uložený v priečinku označenom **task-id** spolu s prijatými správami na **UMB** a následne taktiež so štandardizovanými výsledkami. Je to z dôvodu možnosti preskúmania prijatých správ alebo generovaných súborov v prípade chýb. Po štandardizácii výsledkov, skript vyhľadá bežiacu úlohu s daným **task-id**, označí ju ako dokončenú, importuje výsledky spolu s metadátami do **ReportPortal-u** a následne tieto dve úlohy zjednotí do jednej. Je to z dôvodu nemožnosti doplňať výsledky hromadne do už existujúcej úlohy. Je pre nás výhodnejšie využiť aplikačné rozhranie **ReportPortal-u**, vytvoriť novú úlohu a následne ich zjednotiť. Malú ukážku kódu pre import výsledkov do nástroja **ReportPortal** vidieť vo výpise 6.2.

```
1 # import XML file to ReportPortal
2 function import_xunit() {
3     local project=$1
4     local api_token=$2
5     local file=$3
6
7     echo $(curl --header "Content-Type: multipart/form-data" \
```

```
8           --header "Authorization: Bearer $api_token" \
9           --request POST \
10          --form "file=@./$file" \
11          ${RP_URL}/api/v1/${project}/launch/import)
12  }
```

Výpis 6.2: Ukážka implementácie funkcie pre import výsledkov do nástroja ReportPortal

Kapitola 7

Nasadenie systému, testovanie, výsledky a možnosti rozšírenia

Vzhľadom na to, že testovanie je neoddeliteľnou súčasťou vývoja softvéru pre účel overenia správnej funkcionality a návrhu systému budeme sa v tejto kapitole venovať popisu nasadenia a testovania systému či už formou automatizovaných jednotkových testov alebo manuálnemu testovaniu v rámci experimentálneho nasadenia systému **ReportPortal** do používania. Taktiež bude získaná spätná väzba od užívateľov **ReportPortal-u**, kde budú zhrnuté výsledky tejto práce a na záver si predstavíme možnosti rozšírenia a postup ďalšieho vývoja tohto systému.

7.1 Nasadenie systému a testovanie

Prvotné testovanie zmien nástroja **ReportPortal** prebiehalo formou automatických jednotkových testov, ktoré sú súčasťou daného projektu. Tieto testy boli rozšírené a upravené na základe nových zmien, ktoré boli do **ReportPortal-u** implementované. Jednalo sa predovšetkým o automatické testy pre kontrolu správnej funkcie end-pointov v rámci aplikačného rozhrania, overovanie správnej funkcionality štatistík, ktoré naväzovali na zmeny v dátovej vrstve a inicializačných skriptoch databázy a taktiež testovanie správnej práce s novými statusmi testov.

Nakoľko sa systém pre správu výsledkov testov neskladá iba z nástroja **ReportPortal**, ale aj z infraštruktúry dodávajúcej dáta, bolo potrebné úvodné požiadavky užívateľov na tento systém implementovať a experimentálne nasadiť, aby sa overila ako správna funkcionalita infraštruktúry, tak aj správnosť užívateľských požiadaviek. Je všeobecne známe, že užívatelia dokážu oveľa efektívnejšie zadefinovať svoje potreby pri práci s existujúcim softvérom, ako iba formou brainstormingu a domýšľania rôznych situácií, ktoré môžu nastať.

7.1.1 Nasadenie a testovanie prototypu

Po úvodnom zozbieraní požiadaviek, kde sa formou iteratívnych stretnutí užívatelia schádzali a diskutovali ohľadom nového systému, bol vybraný nástroj **ReportPortal** ako vhodný kandidát. Tento nástroj bol užívateľmi odsúhlasený a boli do neho implementované základné požiadavky, kde podoba tejto implementácie bola na každom stretnutí s užívateľmi diskutovaná. Týmto spôsobom postupne vznikal prototyp systému pre experimentálne nasadenie. V čase, kedy boli užívateľské požiadavky jasnejšie zadefinované a systém použiteľný pre reálne nasadenie, prebehla implementácia infraštruktúry, ktorej cieľom bolo získavať dáta

z **UMB**, spracovávať ich a následne importovať do nástroja. Implementácia infraštruktúry bola základne manuálne otestovaná vývojármi a následne s prototypom **ReportPortal-u**, otestovaným formou jednotkových automatických testov, nasadená to experimentálneho používania. Experimentálne bol nástroj **ReportPortal** nasadený pomocou technológie *Kubernetes*, kde každý kontajner s bežiacou službou bol nasadený na vlastnej jednotke zvanej *pod* a testovacia infraštruktúra na *Fedore 32* bežiacej za pomoci technológie *Openstack*, čo je cloudová platforma vyvíjaná spoločnosťou *Red Hat*.

Výhodou experimentálneho nasadenia bolo odhadenie chýb a overenie správnej funkcionality infraštruktúry spolu s umožnením užívateľom vyskúšať si nástroj **ReportPortal** počas práce s reálnymi dátami. Toto nasadenie bolo sprevádzané so značným odhalovaním funckionálnych chýb najmä pri práci s **UMB** a neštandardizovaným formátom výsledkov testov, avšak po odladení týchto problémov bolo užívateľom umožnené prihlásenie a práca s týmto systémom. Prihlasovanie bolo možné pomocou existujúcich účtov na platforme *GitHub*, ktoré má väčšina užívateľov štandardne založených.

V rámci používania systému a následných iteratívnych stetnutí dokázali užívatelia jasnejšie definovať svoje požiadavky, kde tieto boli do systému súčasne zapracovávané a konzultované počas nasledujúcich stretnutí. Zároveň prebiehal zber dát pre finálne nasadenie, kde sa získavali skúsenosti ohľadom pamäťovej a výkonnostnej náročnosti systému.

Z vyššie uvedeného vyplýva, že experimentálne nasadenie prototypu bolo uskutočnené z dôvodu ako otestovania správnej funkčnosti testovacej infraštruktúry v spolupráci s nástrojom **ReportPortal**, tak z dôvodu doplnenia plného obrazu požiadaviek užívateľov na výsledný systém. Toto nasadenie splnilo účel a výrazne uľahčilo finálne nasadenie dokončeného systému.

7.1.2 Finálne nasadenie

Finálnemu nasadeniu systému pre správu výsledkov testov predchádzalo experimentálne nasadenie, počas ktorého sa uzavrel zber požiadaviek na nástroj **ReportPortal** a zároveň overila správna funkčnosť infraštruktúry dodávajúcej do **ReportPortal-u** dáta. Vzhľadom k týmto skutočnostiam nebolo potrebné v tejto fáze riešiť úpravy infraštruktúry, nakoľko táto už bola riadne otestovaná približne 8 týždnovým používaním s reálnymi dátami získavanými z **UMB**. Počas tejto fázy bolo zaznamenané, že priemerne systém počas pracovného dňa spracuje približne 250 UMB správ zo sledovaných kanáloch. Tieto správy sú vyfiltrované z približného množstva 10 až 15 tisíc správ, čo svedčí o spôsobilosti systému spracovať efektívne veľké množstvo dát.

Zmeny nastali v nástroji **ReportPortal**, kde pred finálnym nasadením boli zmeny systému migrované z verzie 5.2 na verziu 5.3. Migrácia prebehla úspešne a bez väčších problémov, kde správna funkcionalita aplikačného rozhrania bola overená pomocou jednotkových testov. Následne bola výsledná verzia systému nasadená za pomoci technológie *Docker-compose* na *Fedoru 32* bežiacu za pomoci technológie *Openstack*. Pre zmenu technológie nasadenia **ReportPortal-u** sme sa rozhodli z dôvodu častej poruchovosti jednotlivých častí (*pod*) v rámci *Kubernetes cluster*, čo nám spôsobovalo problémy s výpadkami celého systému. Túto zmenu môžeme hodnotiť pozitívne, nakoľko od riadneho nasadenia systému sme nezaznamenali žiadny problém so stabilitou. **ReportPortal** spolu s infraštruktúrou nevyžaduje žiadne prehnané požiadavky čo sa týka pamäťovej alebo výkonovej náročnosti. Daný Openstack ponúka 500GB disk spolu s 16GB pamäte RAM, čo je pre momentálne potreby dostačujúce a isté výkonové optimalizácie sú nastavené iba v rámci *PostgreSQL* databázy pre vyvažovanie spracovania požiadaviek. Historické dáta a logy sú v

rámci **ReportPortal-u** aj infraštruktúry premazávané automatickými skriptami, kde logy infraštruktúry sú udržované 30 dní a detailné dáta testovania 6 mesiacov. Kôli premazávaniu dát sa pamäťová náročnosť celého systému pohybuje v jednotkách GB mesačne, čo v súčasnej dobe je veľmi prijateľné číslo.

Po nasadení celého systému a overení správnej funkčnosti a spolupráce infraštruktúry s **ReportPortal-om**, bolo potrebné nastaviť bezpečnú autentifikáciu užívateľov v rámci internej siete. Vzhľadom na veľké množstvo interných aplikácii v spoločnosti *Red Hat* sa pre autentifikáciu využíva autentifikačná schéma *SSO*, ktorú v spolupráci s protokolom *SAML* implementuje aj **ReportPortal**. Z tohto dôvodu to bola pre nás najbezpečnejšia a priamočiara varianta. Po nastavení autentifikácie prebehlo verejné virtuálne stretnutie, na ktoré boli pozvaný zamestnanci spoločnosti *Red Hat*, ktorí o to prejavili záujem. Stretnutie prebiehalo formou demo ukážok funkcionality **ReportPortal-u** a odpovedaním na otázky užívateľov, akým spôsobom je pracovanie s nástrojom čo najjednoduchšie.

Vzhľadom na to, že sa daného stretnutia mohlo zúčastniť iba obmedzené množstvo užívateľov, bolo vytvorené aj krátke demo video popisujúce základnú funkcionalitu a používanie nástroja. Toto video bolo následne spolu s krátkym úvodom umiestnené na verejný mailing-list, kde ho užívatelia mohli ďalej zdieľať a vyvolalo pozitívnu odozvu na celý koncept vývoja **ReportPortal-u** ako systému pre správu výsledkov testov. Dané video je dostupné a možné zhliadnuť na priloženom DVD. V súčasnosti prevláda trend využívania nástroja, avšak ako pri všetkých nových technológiách je očakávané, že bude plne akceptovaný až po odstránení aktuálne využívaných nástrojov, v tomto prípade *Nitrate*. Rovnaký trend je zaznamenaný aj pri využívaní nástroja TMT.

7.2 Výsledky a možnosti rozšírenia

Výsledný systém pre správu výsledkov testov bol úspešne nasadený a je využívaný zamestnancami spoločnosti *Red Hat* ako jeden z nástrojov pre efektívne prezeranie výsledkov testov. Daný systém v súčasnom stave ešte nenadobudol svoj plný potenciál, aj vzhľadom na chýbajúce informácie na **UMB** a nie plné nasadenie nástroja **TMT** s ktorým by mal primárne kooperovať. Momentálne je ale použiteľný pre výsledky testov, ktoré sú k dispozícii na **UMB** a je možné s nimi plnohodnotne pracovať.

Spätná väzba bola zhromažďovaná v rámci pravidelných stretnutí užívateľov podieľajúcich sa na vývoji a správe testovacej infraštruktúry v spoločnosti *Red Hat* vo forme neformálnych poznámok, kde väčšina vecných námietok bola do systému zapracovaná. Na týchto stretnutiach bol získaný názor od 34 užívateľov a v rámci postupného behu aplikácie sa vyvíjal. Zo začiatku v rámci testovacieho nasadenia a behu prototypu boli poznámky vecné a týkali sa chýbajúcej funkcionality a prípadných zmien nástroja **ReportPortal**. V záverečných týždňoch diskusia prebieha výlučne ohľadom možných rozšírení systému, ktoré sú detailnejšie predstavené v nasledujúcej časti. Z toho usudzujeme, že s aktuálnou verziou systému prevláda všeobecná spokojnosť a prioritou je momentálne rozširovanie množstva užívateľov pre získavanie nových nápadov a propagáciu používania systému v rámci spoločnosti *Red Hat*. Za pozitívnu spätnú väzbu je možné považovať aj prejavený záujem o aktívne podieľanie sa na ďalšom vývoji a smerovaní celého nástroja od 5 užívateľov. Z celkového hľadiska prevláda názor, že nový systém pre správu výsledkov testov **ReportPortal** bude v budúcnosti po naplnení svojho potenciálu plnohodnotnou náhradou zastaralého systému *Nitrate*. V súčasnosti evidujeme približne 50 zaregistrovaných užívateľov, kde 20 z nich využíva **ReportPortal** na pravidelnej báze. Z tohto dôvodu vidíme potenciál nárastu užívate-

ľov po pridaní ďalšej funkcionality nakoľko toto číslo je pre tak mladý systém aj v porovnaní s **TMT**, ktoré existuje už pár rokov, veľmi slušné.

Diskusie ohľadom potenciálnych rozšírení nástroja **ReportPortal** sa týkajú najmä možností rozšíriť **UMB** správy **queued** a **running** o informácie o plánovaných a aktuálne bežiacich testoch. V tomto prípade, je možné jednoduchým spôsobom rozšíriť infraštruktúru a aj samotný nástroj **ReportPortal** o akceptáciu týchto informácii, kde sú v súčasnosti pripravené stavy a scenáre pre adoptovanie tejto novej funkcionality. V tomto smere by bolo pre užívateľov možné prehliadať celý testovací plán spolu s aktuálnymi stavmi testov priamo v **ReportPortal-i**, čím by sa mohli z používania vylúčiť iné aplikácie pre prezeranie aktuálneho stavu testovania ako napríklad *Nitrate*.

Ďalším rozšírením je posielať informácie o **Tier** a **Errata** testovaní na UMB, kde pokiaľ budú mať tieto dáta rovnaký formát ako výsledky CI testovania, je možné ich prakticky bez zmeny akceptovať a zaradiť do používania. Ako bolo vysvetlené v predchádzajúcej časti tejto práce, tento fakt závisí najmä od exekútorov týchto testov, nakoľko súčasní exekútori nedisponujú funkcionalitou pre komunikáciu s UMB. Je to ale určite jeden z krokov smerom vpred, ktorý je pre daný systém plánovaný, aby sa bolo v budúcnosti možné absolútne oslobodiť od používania nástroja *Nitrate*, ktorý je primárnym systémom pre správu výsledkov týchto testov.

Posledným a jediným plánovaným veľkým rozšírením dotýkajúcim sa iba nástroja **ReportPortal** je zavedenie umelej inteligencie do automatického vyhodnocovania zlyhaných testov na základe predchádzajúcich manuálnych vyhodnotení. Túto funkcionalitu má nástroj **ReportPortal** implementovanú na základe porovnávania reťazcov logov. Pre naše potreby nie je táto funkcionalita v tejto forme využiteľná, nakoľko logy medzi jednotlivými behmi testov sa môžu výraznejšie líšiť a vyhodnocovanie by mohlo byt problematické. V tomto smere sa ponúka možnosť vytvoriť ku každému testu očakávanú predlohu logov výsledkov testov, kde by sa na základe nej dokázali vyhodnocovať typy chýb podľa toho, v ktorej časti logov nastali nezhody oproti danej predlohe. Táto možnosť by rátala s využitím funkcionality pre porovnávanie reťazcov, ktorá je implementovaná v rámci nástroja ReportPortal. Táto funkcionalita je momentálne preberaná v rámci diskusií a hľadajú sa možnosti, akým spôsobom by bolo možné ju nasadiť aspoň pre experimentálne použitie.

Z vyššie uvedených bodov vyplýva, že nástroj **ReportPortal** má potenciál stať sa plnohodnotou súčasťou práce vývojárov a testerov v rámci spoločnosti *Red Hat*, kde zmeny uvedené v tejto práci sú len prvým krokom k plnej funkcionalite a vývoj neustále pretrváva.

Kapitola 8

Záver

Táto práca sa venovala vývoju a úspešnému nasadeniu systému pre správu výsledkov testov v rámci spoločnosti *Red Hat*. Vzhľadom na obšírnosť tejto témy, boli v úvode preštudované a predstavené aktuálne používané technológie pre správu a vyhodnocovanie výsledkov testov v rámci danej spoločnosti. Postupne sme sa venovali predstaveniu nástroja **Nitrate**, ako systému pre správu testov a výsledkov testov, interne vyvinutému testovaciemu frameworku **BeakerLib**, systému pre zaznamenávanie chýb softvéru **Bugzilla**, komunikačnej zbernici **Unified Message Bus** a taktiež nástroju **Jenkins** využívanému pre CI a CD, kde predstavenie daných tém slúžilo ako teoretický úvod do práce za účelom oboznámenia čitateľa s aktuálnymi technológiami.

Následne sme si predstavili aktuálne vyvíjaný systém pre správu testov **Test Management Tool**, ktorý bude v budúcnosti priamo kooperovať s novovyvinutým nástrojom pre správu výsledkov testov. V rámci práce boli popísané jeho vlastnosti a predstavená jeho funkcionalita.

V spolupráci s užívateľmi prebehlo zhodnotenie súčasného stavu technológii v spoločnosti a taktiež zozbieranie funkčných ale aj nefunkčných požiadaviek pre vývoj nového systému pre správu výsledkov testov, ktorý spolu s **TMT** v budúcnosti nahradí a vyradí z používania nástroj **Nitrate**. Táto spolupráca prebiehala formou organizovaných stretnutí, kde v úvode bol diskutovaný celkový obraz kooperácie týchto systémov. V rámci získaných vedomostí prebiehal paralelný výber vhodného nástroja pre dané použitie, kde pre tento účel bol vybraný open-source projekt **ReportPortal**.

Vzhľadom na rozsiahlosť požiadaviek, bol nástroj **ReportPortal** použitý pre stavbu prototypu a iteratívne obohacovaný o novú funkcionalitu prediskutovávanú v rámci stretnutí s užívateľmi. V istom bode, keď bol **ReportPortal** použiteľný pre reálny chod, prebehlo experimentálne nasadenie, ktoré overilo ako správnosť požiadaviek na systém, tak aj správny chod infraštruktúry pre import dát do nástroja.

Na základe spätnej väzby na experimentálny chod systému s reálnymi dátami, boli doplnené chýbajúce požiadavky a zakomponované zmeny do finálnej verzie daného systému. Finálna verzia systému bola otestovaná jednotkovými testami, ktoré sú súčasťou implementácie, ale aj experimentálnym chodom samotného prototypu, čo nám prinieslo veľa dôležitých dát pre finálne nasadenie a spoluprácu systémov.

Po implementácii finálnej verzie systému a jej nasadení bolo usporiadané demo pre širšiu skupinu užívateľov za účelom oboznámenia sa s nátrojom **ReportPortal** a spôsobom jeho využitia.

Vo výsledku hodnotíme danú prácu ako úspešnú vzhľadom na pozitívnu spätnú väzbu od užívateľov, kde daný systém je reálne využiteľný v problematike testovania softvéru v rámci

spoločnosti *Red Hat*. Jej výsledky sú taktiež prospešné aj pre open-source komunitu, kde niektoré časti tejto práce boli akceptované vývojármi nástroja ReportPortal a sú dostupné všetkým užívateľom. Prihliadajúc na to, že u daného nástroja existujú veľké možnosti rozšírenia na ktorých sa už v súčasnosti pracuje, je potenciál, že daná téma bude napredovať v rámci riešení.

Z vyššie uvedeného konštatujeme, že daná práca splnila cieľ, vzhľadom na to, že bol vyvinutý a nasadený nový systém pre správu výsledkov testov doplňujúci nástroj **TMT**, ktorý v budúcnosti nahradí zastaralý nástroj **Nitrate**.

Literatúra

[1] *Bugzilla – Bug Fields* [online]. [cit. 2020-12-29]. Dostupné z: https://bugzilla.suse.com/page.cgi?id=fields.html#resolution.

[2] *Introducing Red Hat AMQ 7* [online]. [cit. 2021-02-21]. Dostupné z: https://access.redhat.com/documentation/en-us/red_hat_amq/7.5/html/introducing_red_hat_amq_7/index.

[3] *Introduction - Errata* [online]. [cit. 2021-03-26]. Dostupné z: https://wiki.test.redhat.com/ErrataWorkflow/Introduction.

[4] *Nitrate Status* [online]. [cit. 2021-03-26]. Dostupné z: http://wiki.test.redhat.com/BaseOs/BestPractices/NitrateStatus.

[5] *RHEL Workflows: CDW MCW* [online]. [cit. 2021-03-26]. Dostupné z: https://one.redhat.com/rhel-developer-guide/#_rhel_workflows_cdw_mcw.

[6] *UMB 1.2 Infrastructure* [online]. [cit. 2021-03-26]. Dostupné z: https://source.redhat.com/groups/public/enterprise-services-platform/it_platform_wiki/umb_12_infrastructure.

[7] *Jenkins* [online]. 2016 [cit. 2020-12-29]. Dostupné z: https://www.jenkins.io/doc/book/.

[8] *ReportPortal - AI-powered Test Automation Dashboard* [online]. 2016 [cit. 2020-12-29]. Dostupné z: https://reportportal.io/.

[9] *Fedora CI* [online]. 2021 [cit. 2021-03-24]. Dostupné z: https://docs.fedoraproject.org/hu/ci/.

[10] *Polarion QA* [online]. 2021 [cit. 2020-12-29]. Dostupné z: https://polarion.plm.automation.siemens.com/products/polarion-qa.

[11] *ReportPortal - Test automation CI/CD tool* [online]. 2021 [cit. 2020-03-24]. Dostupné z: https://solutionshub.epam.com/solution/report-portal.

[12] CloudBees. *What is Jenkins?* [online]. 2010-2021 [cit. 2020-12-29]. Dostupné z: https://www.cloudbees.com/jenkins/what-is-jenkins.

[13] Heger, J. *Performance Optimization of Testing Automation Framework Based on Beakerlib*. Brno, CZ, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/20237/.en.

[14] Meliora. *Official support for Jenkins Pipelines* [online]. 2019 [cit. 2021-03-26]. Dostupné z: https://www.melioratestlab.com/2019/02/01/official-support-for-jenkins-pipelines-2/.

[15] Meyering, J. *Nohup* [online]. [cit. 2021-03-24]. Dostupné z: https://linux.die.net/man/1/nohup.

[16] Miller, D. et al. *Bugzilla* [online]. 1998-2021 [cit. 2020-12-29]. Dostupné z: https://www.bugzilla.org/.

[17] Müller, P. et al. *BeakerLib - a shell-level integration testing library* [online]. 2017 [cit. 2020-12-29]. Dostupné z: https://github.com/beakerlib/beakerlib/wiki/man.

[18] Ross, A. et al. *Nitrate - Test Case Management System* [online]. 2013-2021 [cit. 2020-12-29]. Dostupné z: https://nitrate.readthedocs.io.

[19] Vadkerti, M. et al. *CI Messages* [online]. 2019-2021 [cit. 2021-03-24]. Dostupné z: https://pagure.io/fedora-ci/messages.

[20] Šplíchal, P. et al. *FMF - Flexible Metadata Format* [online]. 2015 [cit. 2020-12-29]. Dostupné z: https://fmf.readthedocs.io.

[21] Šplíchal, P. et al. *TMT - Test Management Tool* [online]. 2019 [cit. 2020-12-29]. Dostupné z: https://tmt.readthedocs.io.

Príloha A

Obsah priloženého pamäťového média

- **docker-compose.yml** – konfiguračný súbor pre beh aplikácie ReportPortal v prostredí Docker
- **download.sh** – skript pre jednoduché stiahnutie zdrojových súborov ReportPortalu
- **README.txt** – súbor obsahujúci inštalačný postup
- **xdubaj00.pdf** – technická správa diplomovej práce
- **demo_RP.mp4** – demo video pre používanie ReportPortalu
- **/latex** – zdrojové súbory technickej správy
- **/Infrastructure** – zdrojové súbory testovacej infraštruktúry
- **/ReportPortal** – zdrojové súbory aplikácie ReportPortal

Príloha B

Štandardizovaný formát ReportPortal Xunit

V tejto prílohe bude predstavený štandardizovaný formát ReportPortal Xunit pre zjednotenie formátu správ s výsledkami testov v rámci spoločnosti Red Hat. Pre popis štandardu bol použitý jazyk *XML Schema*.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="failure">
        <xs:complexType mixed="true">
            <xs:attribute name="type" type="xs:string" use="optional"/>
            <xs:attribute name="message" type="xs:string" use="optional
                "/>
        </xs:complexType>
    </xs:element>

    <xs:element name="manual-test">
        <xs:complexType mixed="true">
            <xs:attribute name="type" type="xs:string" use="optional"/>
            <xs:attribute name="message" type="xs:string" use="optional
                "/>
        </xs:complexType>
    </xs:element>

    <xs:element name="error">
        <xs:complexType mixed="true">
            <xs:attribute name="type" type="xs:string" use="optional"/>
            <xs:attribute name="message" type="xs:string" use="optional
                "/>
        </xs:complexType>
    </xs:element>

    <xs:element name="skipped">
        <xs:complexType mixed="true">
            <xs:attribute name="type" type="xs:string" use="optional"/>
```

```
27            <xs:attribute name="message" type="xs:string" use="optional
                  "/>
28        </xs:complexType>
29    </xs:element>
30
31    <xs:element name="untested">
32        <xs:complexType mixed="true">
33            <xs:attribute name="type" type="xs:string" use="optional"/>
34            <xs:attribute name="message" type="xs:string" use="optional
                  "/>
35        </xs:complexType>
36    </xs:element>
37
38    <xs:element name="running">
39        <xs:complexType mixed="true">
40            <xs:attribute name="type" type="xs:string" use="optional"/>
41            <xs:attribute name="message" type="xs:string" use="optional
                  "/>
42        </xs:complexType>
43    </xs:element>
44
45    <xs:element name="properties">
46        <xs:complexType>
47            <xs:sequence>
48                <xs:element ref="property" minOccurs="0" maxOccurs="
                      unbounded"/>
49            </xs:sequence>
50        </xs:complexType>
51    </xs:element>
52
53    <xs:element name="global_properties">
54        <xs:complexType>
55            <xs:sequence>
56                <xs:element ref="global_property" minOccurs="0"
                      maxOccurs="unbounded"/>
57            </xs:sequence>
58        </xs:complexType>
59    </xs:element>
60
61    <xs:element name="arch-properties">
62        <xs:complexType>
63            <xs:sequence>
64                <xs:element ref="arch-property" minOccurs="0" maxOccurs
                      ="unbounded"/>
65            </xs:sequence>
66        </xs:complexType>
67    </xs:element>
68
```

```
69      <xs:element name="property">
70          <xs:complexType>
71              <xs:attribute name="name" type="xs:string" use="required"/>
72              <xs:attribute name="value" type="xs:string" use="required"/>
73          </xs:complexType>
74      </xs:element>
75
76      <xs:element name="global_property">
77          <xs:complexType>
78              <xs:attribute name="name" type="xs:string" use="required"/>
79              <xs:attribute name="value" type="xs:string" use="required"/>
80          </xs:complexType>
81      </xs:element>
82
83      <xs:element name="arch-property">
84          <xs:complexType>
85              <xs:attribute name="name" type="xs:string" use="required"/>
86              <xs:attribute name="value" type="xs:string" use="required"/>
87          </xs:complexType>
88      </xs:element>
89
90      <xs:element name="parameters">
91          <xs:complexType>
92              <xs:sequence>
93                  <xs:element ref="parameter" minOccurs="0" maxOccurs="
                        unbounded"/>
94              </xs:sequence>
95          </xs:complexType>
96      </xs:element>
97
98      <xs:element name="parameter">
99          <xs:complexType>
100             <xs:attribute name="name" type="xs:string" use="required"/>
101             <xs:attribute name="value" type="xs:string" use="required"/>
102         </xs:complexType>
103     </xs:element>
104
105     <xs:element name="system-err" type="xs:string"/>
106     <xs:element name="system-out" type="xs:string"/>
107
108     <xs:element name="testcase">
109         <xs:complexType>
110             <xs:sequence>
111                 <xs:choice minOccurs="0" maxOccurs="unbounded">
112                     <xs:element ref="skipped"/>
113                     <xs:element ref="error"/>
114                     <xs:element ref="failure"/>
115                     <xs:element ref="system-out"/>
```

```
                <xs:element ref="system-err"/>
                <xs:element ref="manual-test"/>
            </xs:choice>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="time" type="xs:string" use="required"/>
        <xs:attribute name="id" type="xs:string" use="required"/>
        <xs:attribute name="arch" type="xs:string" use="required"/>
    </xs:complexType>
</xs:element>

<xs:element name="testsuite">
    <xs:complexType>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="testsuite"/>
            <xs:element ref="properties"/>
            <xs:element ref="parameters"/>
            <xs:element ref="testsuite-arch"/>
            <xs:element ref="testcase"/>
            <xs:element ref="system-out"/>
            <xs:element ref="system-err"/>
        </xs:choice>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="id" type="xs:string" use="required"/>
        <xs:attribute name="href" type="xs:string" use="required"/>
        <xs:attribute name="tests" type="xs:string" use="optional"/>
        <xs:attribute name="failure" type="xs:string" use="optional
            "/>
        <xs:attribute name="skipped" type="xs:string" use="optional
            "/>
        <xs:attribute name="manual" type="xs:string" use="optional
            "/>
        <xs:attribute name="running" type="xs:string" use="optional
            "/>
        <xs:attribute name="untested" type="xs:string" use="optional
            "/>
    </xs:complexType>
</xs:element>

<xs:element name="testsuite-arch">
    <xs:complexType>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="arch-properties"/>
            <xs:element ref="testcase"/>
            <xs:element ref="system-out"/>
            <xs:element ref="system-err"/>
        </xs:choice>
        <xs:attribute name="name" type="xs:string" use="required"/>
```

```
159             <xs:attribute name="id" type="xs:string" use="required"/>
160             <xs:attribute name="href" type="xs:string" use="required"/>
161             <xs:attribute name="tests" type="xs:string" use="optional"/>
162             <xs:attribute name="failure" type="xs:string" use="optional
                    "/>
163             <xs:attribute name="skipped" type="xs:string" use="optional
                    "/>
164             <xs:attribute name="manual" type="xs:string" use="optional
                    "/>
165             <xs:attribute name="running" type="xs:string" use="optional
                    "/>
166             <xs:attribute name="untested" type="xs:string" use="optional
                    "/>
167         </xs:complexType>
168     </xs:element>
169
170     <xs:element name="testsuites">
171         <xs:complexType>
172             <xs:choice minOccurs="0" maxOccurs="unbounded">
173                 <xs:element ref="properties"/>
174                 <xs:element ref="global_properties"/>
175                 <xs:element ref="testsuite"/>
176             </xs:choice>
177             <xs:attribute name="name" type="xs:string" use="optional"/>
178             <xs:attribute name="time" type="xs:string" use="optional"/>
179             <xs:attribute name="tests" type="xs:string" use="optional"/>
180             <xs:attribute name="failure" type="xs:string" use="optional
                    "/>
181             <xs:attribute name="skipped" type="xs:string" use="optional
                    "/>
182             <xs:attribute name="manual" type="xs:string" use="optional
                    "/>
183             <xs:attribute name="running" type="xs:string" use="optional
                    "/>
184             <xs:attribute name="untested" type="xs:string" use="optional
                    "/>
185         </xs:complexType>
186     </xs:element>
187 </xs:schema>
```