BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

EXTENSION OF THE TMT TOOL FOR REPORTING THROUGH THE REPORTPORTAL API

ROZŠÍŘENÍ NÁSTROJE TMT UMOŽŇUJÍCÍ REPORTOVÁNÍ POMOCÍ API NÁSTROJE REPORTPORTAL

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **AUTOR PRÁCE**	**NATÁLIA BUBÁKOVÁ**
SUPERVISOR **VEDOUCÍ PRÁCE**	**Ing. ALEŠ SMRČKA, Ph.D.**

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

156937

Institut:	Department of Intelligent Systems (DITS)
Student:	**Bubáková Natália**
Programme:	Information Technology
Title:	**Extension of the tmt Tool for Reporting through the ReportPortal API**
Category:	Software analysis and testing
Academic year:	2023/24

Assignment:

1. Study the field of the test management. Investigate *tmt* tool for test management. Focus on the way tmt reports results of the executed tests. Investigate *ReportPortal*, a tool for reporting and managing test results. Focus on the tool's API which allows other tools to report their results.
2. Analyse the requirements of automated reporting of test results. Propose and design an extension of the *tmt* which will enable reporting to the *ReportPortal*.
3. Implement the proposed solution as a new *tmt* plugin.
4. Evaluate the implemented plugin on different test suites managed by *tmt*.

Literature:

- Homepage of tmt tool: https://github.com/teemtee/tmt
- Documentation of tmt tool: https://tmt.readthedocs.io/en/stable/
- O. Dubaj. *Systém pro správu výsledků testů doplňující nástroj tmt*. Brno, 2021. Master thesis. FIT BUT.

Requirements for the semestral defence:
The first two points.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Smrčka Aleš, Ing., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	6.11.2023

Abstract

This Bachelor's thesis provides insight into testing processes practiced across teams at Red Hat and describes a new testing infrastructure proposed to improve testing workflow within the company. With this infrastructure serving as the main motivation for the assignment, the work targets several use cases of report functionality resulting from the integration of the Test Management Tool commonly known as tmt and the reporting platform ReportPortal. It examines both of these tools, analyses alternative approaches, and proposes an implementation of a tmt plugin that integrates seamlessly with ReportPortal, via its REST API. The focus of the thesis encapsulates all steps behind a community-driven project, closely examining detailed aspects of design, implementation, and testing of all requested features of the plugin that actively resides within the tmt open-source project.

Abstrakt

Táto bakalárska práca nahliada na testovacie procesy využívané Red Hat tímami v praxi, a zároveň opisuje novú testovaciu infraštruktúru navrhnutú pre účel zlepšenia testovacích praktík v spoločnosti. Infraštruktúra predstavuje hlavnú motiváciou pre zadanie tejto práce, ktorá sa sústredí na niekoľko scenárov tvorby reportov s výsledkami testov, ktoré sú realizované práve prepojením nástroju na správu testov známeho ako tmt a ReportPortalu, teda rozhrania pre zobrazenie výsledkov. Práca skúma oba tieto nástroje a prezentuje implementáciu v podobe tmt rozšírenia plynule prepojeného s ReportPortalom cez jeho REST API rozhranie, čím vylučuje alternatívne prístupy. Práca sa komplexne zaoberá všetkými etapamy projektu, ktorý je realizovaný v spolupráci s komunitou a detailne skúma aspekty návrhu, implementácie a testovania všetkých požadovaných funkcií rozšírenia, ktoré aktívne komplimentuje open-source tmt projekt.

Keywords

Test Management Tool, tmt, fmf, Report Portal, test report, software testing, test result management, test plan, test run, test case

Kľúčové slová

Test Management Tool, tmt, fmf, Report Portal, reportovanie testov, testovanie programu, správa výsledkov testov, plán testovania, spustenie testu, prípad testovania

Reference

BUBÁKOVÁ, Natália. *Extension of the tmt tool for reporting through the ReportPortal API*. Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Aleš Smrčka, Ph.D.

Extension of the tmt tool for reporting through the ReportPortal API

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Aleš Smrčka, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Natália Bubáková
May 15, 2024

Acknowledgements

I would like to express my sincere gratitude to everyone who contributed to the completion of this thesis. Specifically, I extend my thanks to my supervisor Ing. Aleš Smrčka, Ph.D. for his willingness and patience throughout the process. I am grateful to my technical supervisor, Mgr. Petr Šplíchal, and all colleagues who contributed to defining requirements, reviewing and evaluating implementations, and providing assistance in understanding and addressing all related issues. Additionally, I acknowledge those who contributed to the clarity of the documentation of the tmt tool and RHEL processes, and also everyone who supported the main initiative which laid the foundation for this work.
Last but not least, I am deeply grateful to my partner for his enduring support throughout the writing process.

Contents

Chapter 1

Introduction

In the life cycle of a software product, testing emerges as the bedrock of quality assurance. It plays a crucial role not only in the pre-deployment phase but also throughout maintenance, where it monitors behaviour, facilitates easy bug identification, and ensures the stability and functionality of the product with each software update or upgrade iteration. Hence, establishing and maintaining effective test management and automation through a stable and comprehensive system is imperative, particularly within large corporations.

Red Hat, a wide-broad open source company, with its flagship Red Hat Enterprise Linux (RHEL), is no exception to the challenge, testing across a multitude of software components, each with its unique testing requirements. In the field of security and quality assurance of the operating system, software testing is supported by test result management to report, store, and analyze historical development or the current state of the product to detect bugs and develop the quality of the product. Alongside manual execution, this is primarily achieved through test automation, conducted at different intervals, with diverse environmental conditions, and on multiple priority levels, resulting in a variety of result states to be considered.

Across multiple teams and components, these varied needs often lead to inconsistent usage of tools, ranging from outdated systems to adopting several simple tools or even developing custom software solutions to cover specific needs. This can be understood as a widely demanding and inefficient approach.

In response, a cross-functional initiative further introduced as *the Big Picture* has been launched, aiming to develop a large-scale infrastructure of cooperating testing tools with a uniform approach. This initiative serves as the primary motivation for the thesis, as it encompasses the approach to execute tests and report their results for them to be additionally analyzed, filtered, and evaluated. Central to this effort is the necessity for robust and flexible tools to process, display, and store data, accommodating various use cases and ensuring accessibility and active support, thus replacing current obsolete tools.

Aligned with this idea, the thesis targets the integration of the tmt tool and ReportPortal and the reasoning behind it offered as a solution. It explores testing terminology and provides an overview of current tools used within the company and those that will replace them, elucidating test result management processes and all the needs or use cases listed for the implementation. The main focus is however on a key test management tool tmt, its structures, metadata notation, and the way it is integrated with ReportPortal API via the report plugin. Finally, it discusses the possibilities, complications, and resolutions, followed by testing the functionality and evaluation based on the user feedback.

Chapter 2

Software testing and testing tools

Testing is an extensive topic for the thesis, but what testing is and why it is done can hardly be understood without any testing experience beforehand. An understanding of testing purposes is outlined in the standard IEEE 29119-1[2] as follows:

> Testing usually serves more than one purpose. Typical purposes include, but are not restricted to:
>
> (a) detecting defects - this allows for their subsequent removal thus increasing software quality;
>
> (b) gathering information on the test item - testing generates information; this information can serve different purposes, such as:
>
> — developers can use the information to remove defects, increase the code quality, or learn to create better code in the future;
>
> — testers can use the information to create better test cases;
>
> — managers can use the information to decide when to stop testing;
>
> — users eventually benefit from a higher product quality.
>
> (c) creating confidence and taking decisions - by providing evidence that the test item performs correctly under specific circumstances, the stakeholders' confidence that the test item will perform correctly operationally increases; with sufficient confidence, stakeholders can decide to release the test item. Testing may be performed for some or all of the above purposes, and additional purposes not listed may also exist; these purposes should be identified and agreed upon as a starting point for any testing activity.

With all that being said, none of the testing purposes above could be done without the test execution and test report. As all, defect detection, information gathering, and evidence providing refer to a need for a quality overview of test results and other test artifacts, that can provide an easy approach for human reading, further analysis, and additional access. Thus test report and its integration into the testing infrastructure plays an essential role in the idea of testing itself.

Furthermore, the rest of the chapter is dedicated to an understanding of software testing and the practices behind it, followed by a focus on the test report approach and priorities.

2.1 Introduction to software testing

Whilst the thesis will follow the integration of two tools to support the needs for a test report, the project is planted in a real-life operation behind the development and maintenance of a software product. On a corporate scale, such terminology tends to be very specific and testing processes comprehensive enough to keep the operation in order on several levels. This leads to a need to understand the common terms from the work environment, thus define them and use them in the context of test processes that are used.

In the section, all of the main terms and testing processes introduced for purposes of the thesis, are inquired by a broad group of engineers working on a Quality Assurance of the operation system and participating in the Userspace Subsystems for Red Hat Enterprise Linux, which is commonly referred to as RHEL. These are not necessarily Quality Engineers, but rather RHEL Security Engineering teams whose established testing strategies are related therefore can share tools and methods within a comprehensive infrastructure.

2.1.1 Test terminology

Terms used in the thesis are adapted from the RHEL testing practice derived from established standards and testing tools terminology. The terms are defined based on the RHEL community terminology [9], and for better understanding merged with general terms of software testing, defined in the standard ISO/IEC/IEEE 29119-1 [2] and supplemented by the standard ISO/IEC/IEEE 24765 [1].

(a) **Testing** is a set of activities conducted to facilitate the discovery and evaluation of properties of test items.
An act of testing, in the standard referred to as a test, is an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.

  (aa) **Testing activities** include planning, preparation, execution, reporting, and management activities, insofar as they are directed towards testing.

  (ab) **Manual testing** is performed by humans by entering information into a test item and verifying the results.

  (ac) **Scripted testing** is performed based on a documented test script.

  (ad) **Automated testing** uses tools, robots, and other test execution engines to perform tests.

  (ae) **Continuous testing** refers to when a test execution is started via an automated process that can occur on-demand, triggered by a specific event or routine. Continuous testing typically occurs in the context of continuous integration (CI) and continuous delivery (CD).

(b) **Test** is a test unit, in the standard referred to as a test procedure; detailed instructions for the setup, execution, and evaluation of results for a given test case.

  (ba) **Test case** can be an alternative term for a test unit specifically used in terms of test management tools as the lowest level item displayed in the test structure.

  (bb) **Test script** is a test procedure specification document specifying one or more test procedures.

(bc) **Test command** is understood as the smallest unit in the test defined by the testing framework used. It drives the test execution through a command with preconditions, input, and expected results set, and corresponds to one line in the test results log.

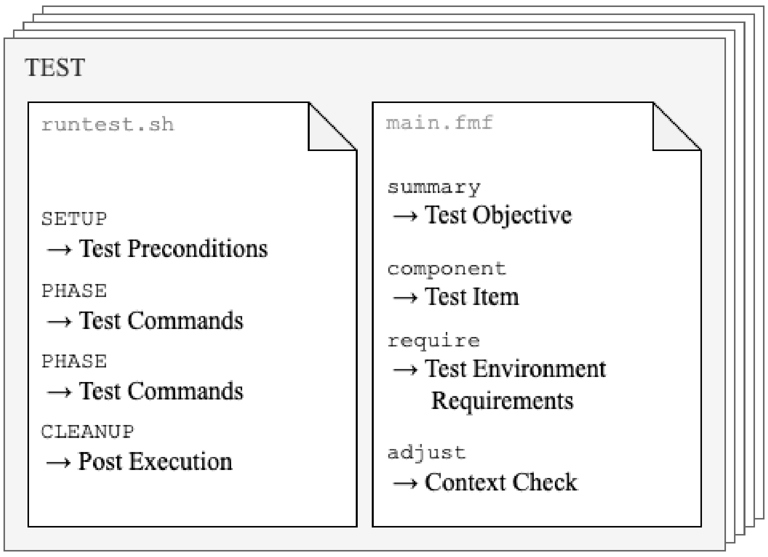


Figure 2.1: Test terminology demonstrated on details of test script and test metadata per test case

(c) **Test item** is a test object, alternatively test subject; a work product to be tested. Example of test items includes software component, system, and user guide procedure.

(ca) **Test objective** is a reason for performing testing

(d) **Test environment** describes an environment containing facilities, hardware, software, firmware, and procedures, needed to conduct a test.

(da) **Test environment requirements** describe prerequisites or necessary properties of the test environment.

(db) **Test preconditions** are conditions that are required to be true for test execution, they include the required state of the test environment, data used by the test item, and the test item itself.

(dc) **Test context** represents an immediate environment in which a procedure or set of procedures operates.

(e) **Test execution** is a process of running a test on the test item, producing actual results

(f) **Test result** is an indication of whether or not a specific test case or test suite has passed or failed, i.e. if the actual results correspond to the expected results or if deviations were observed.

(fa) **Test status** is an alternative term for test result per procedure or per group of procedures, specifically used in the terms of report tools where it can be manually switched.

(fb) **Test artifact** is any additional output of the test suite such as the stdout/stderr output, log files, and screenshots. Test artifacts are for consumption by humans, archival, or big data analysis.

(fc) **Test log** is a chronological record of relevant details about the execution of one or more test procedures.

(fd) **Test report** is defined in the standard as a document that describes the conduct and results of the testing carried out for a system or component. In the context of automated testing and testing tools, it refers to an overview of test statuses and other test artifacts that are easily readable and accessible for human analysis.

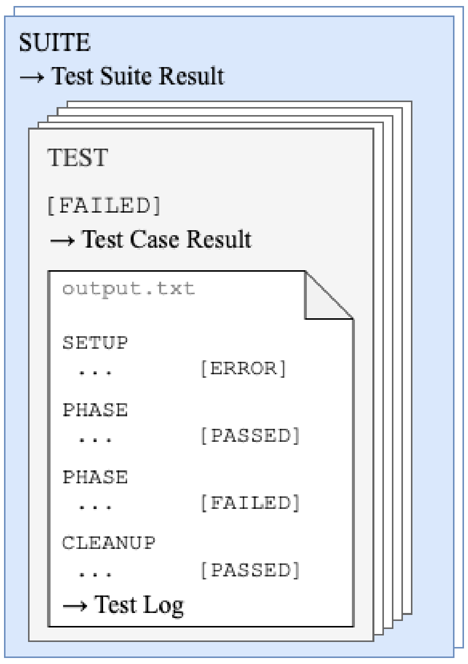


Figure 2.2: Test report terminology demonstrated on analogy of 2.1

(g) **Test run** refers to a single instance of performing a set of testing activities on one or more test cases or plans. It typically identifies the group of tests for test execution, after which it results in the generation of a group of test results and other test artifacts.

(ga) **Test plan** is defined in the standard as a detailed description of test objectives to be achieved, and the means and schedule for achieving them, organized to co-ordinate testing activities for some test item or set of test items. In the context of testing tools, it identifies a group of test cases and its resources for execution with a particular objective against one or more test items.

(gb) **Test suite** is set of test cases or test procedures.

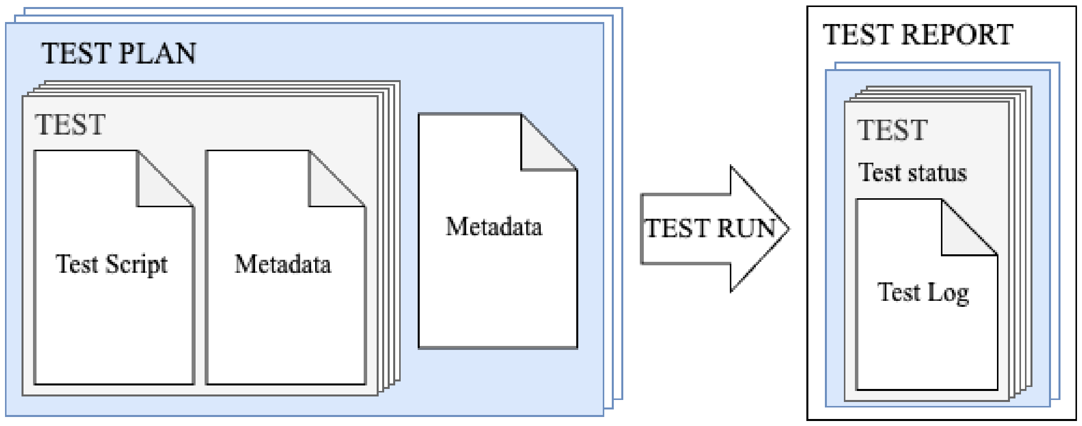


Figure 2.3: Test run terminology demonstrated on relation of test 2.1 and test report 2.2

(h) **Test management** generally involves planning, scheduling, estimating, monitoring, reporting, control, and completion of test activities.

(ha) **Testing tool** is a specific or generic tool that is used for test execution and test management such as test results recording, test results display and interpretation, generation of test scripts, etc.

(hb) **Testing framework** is a library or component that the test suite and tests use to accomplish their job.

(hc) **Testing system** is a CI or other testing system that would like to discover, stage, and invoke tests for a test subject.

(hd) **Testing infrastructure** refers to an ecosystem related to testing, a set of tools and services providing stable and consistent support for testing the test item or group of test items within a product.

2.1.2 Testing tools

As some of the terms above suggest, terminology can widely vary based on the tools used in the work practice. For starters, there are a few essential software tools to obtain an overview of RHEL Quality Assurance and all the processes related, further elaborated in the subsection 2.1.3. Tools information listed within both old and new test infrastructure is mostly defined according to the internal RHEL documentation [9].

(a) **git repositories**
Repositories for source code of tests used to store source code of automated tests. The most used repositories are *github.com* or *gitlab.com* for open-sourced tests together with *internal gitlab* or *dist-git instances* for private projects. Both GitLab and GitHub support continuous integration and continuous delivery (CI/CD) that allows to automate the build, test, and deployment pipeline.

(b) **BeakerLib**
A shell-level integration testing library that serves as a main testing framework for verification.

(c) **Nitrate**
Web-based Test Case Management System, also known as TCMS, is designed for creating, organizing, and analyzing test plans, test cases, and test runs. Nitrate offers a wide range of functionality including robust test life-cycle management, extensible issue tracker, detailed analysis of test results, and fast search capabilities. It is an open source project, written in Python and Django framework, that is currently obsoleted. [5]

(d) **tmt**
The tmt tool, shortened for Test Management Tool, is an open source Python module and command-line tool that provides a user-friendly way to identify tests, prepare the testing environment, execute tests, and report their results. It implements the Metadata Specification which allows storing all needed test execution data directly within a git repository, which can be then remotely referenced. The specification serves as a successor of the Ansible-based Standard Test Interface. [12]

Its test identification is based on the format of an open source Python module and command line tool **fmf**, shortened for Flexible Metadata Format, derived from declarative YAML format. It is an efficient format used to store all test execution metadata in both human and machine-readable ways in one place and offers an alternative to test metadata stored in Makefile. [7, 11]

(e) **Beaker**
An open source software for managing and automating labs of test computers. It enables users and administrators to oversee systems across multiple labs, maintain hardware inventory, provision task environments, schedule tasks across systems, and view stored task results.

(f) **Testing Farm**
An open source Testing System is a Service designed to provide a reliable and scalable service for executing automated tests from various users, such as Fedora CI, RHEL CI, Packit, and others. It serves as a test execution back-end across diverse infrastructures, from private to public clouds. Using the tmt tool format, it abstracts test infrastructure, enabling specific hardware requirements and transparent provisioning. Testing Farm aims to optimize test execution across different environments and projects within the Red Hat ecosystem and open source community.

(g) **ReportPortal**
Service that provides increased capabilities to speed up results analysis and reporting through the use of built-in analytic features. There are multiple instances of ReportPortal running internally.

(h) **Polarion**
A complete web-based Application Lifecycle Management Solution (ALMS) within Linux QE is mostly used for storing test results and related testing-related documents such as test plans, test specifications, or release readiness reports to allow traceability and auditing.

(i) **Bugzilla**
The Red Hat Bugzilla is a Red Hat bug-tracking system for submitting and reviewing defects found in Red Hat distributions.

(j) **Jira**
An issue-tracking and project-management system with advanced visualization capabilities, intuitive hierarchy management, and extensive REST API. It functions as a central tracking, planning, and collaboration tool in Red Hat.

(k) **Errata**
The Errata Tool is a system for managing the Red Hat Errata process. Errata, also known as advisories, is the vehicle by which fixes and enhancements are released to customers for RHEL and other Red Hat products.

(l) **Packit**
An open source project aiming to ease the integration of your project with Fedora Linux, CentOS Stream, and other distributions.

(m) **Brew**
Red Hat's build system. It is designed to build packages from sources in a reproducible and auditable manner and to keep track of those packages for the lifetime of their related products and longer.

(n) **Jenkins**
An open source automation server that enables developers around the world to reliably build, test, and deploy their software. Multiple Jenkins instances are running internally.

2.1.3 Test processes

When software is developed and maintained, it is accompanied by testing activities described by test processes that are generally defined at 3 levels, Organizational test processes, Test management processes, and particularly Dynamic test processes [2]. While the first two serve an important role as wrappers covering mostly organizational and strategic purposes, there lie the Dynamic test processes as a core of the operation. The infrastructure covering the dynamic processes consists of several parts, that dynamically cooperate either fully or semi-automatized.

Firstly there is a source of the test scripts. These depend on the testing framework, and test environment requirements. A tested component or any test item of a requested version must be built and provided either manually or passed into the infrastructure beforehand. Secondly, the test scripts and requirements including the test item are identified and passed to a preset testing environment. Afterward, test execution occurs for test results and additional information to be logged. Lastly, the outcome of the testing is reported into the interface where the logs are stored and can be additionally analyzed and evaluated.

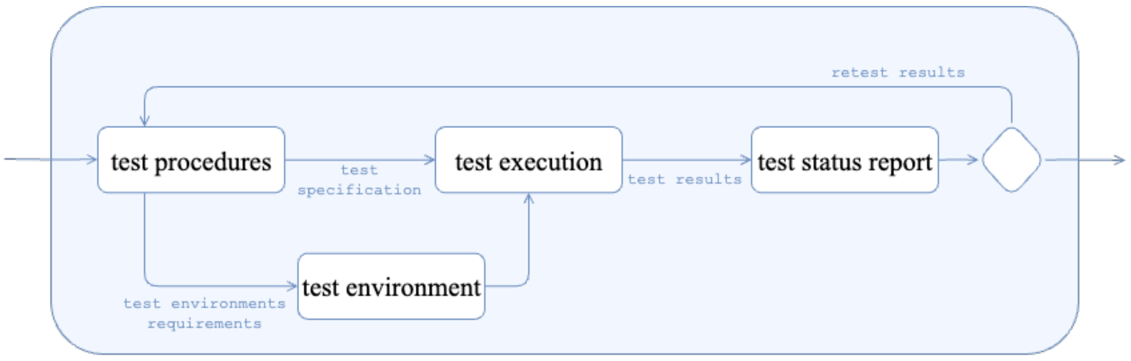


Figure 2.4: Adapted diagram of dynamic test processes based on the standard ISO/IEC/IEEE 29119-1 [2]

With insight into tools listed in 2.1.2, specific testing processes used in the RHEL subsystems can be introduced.

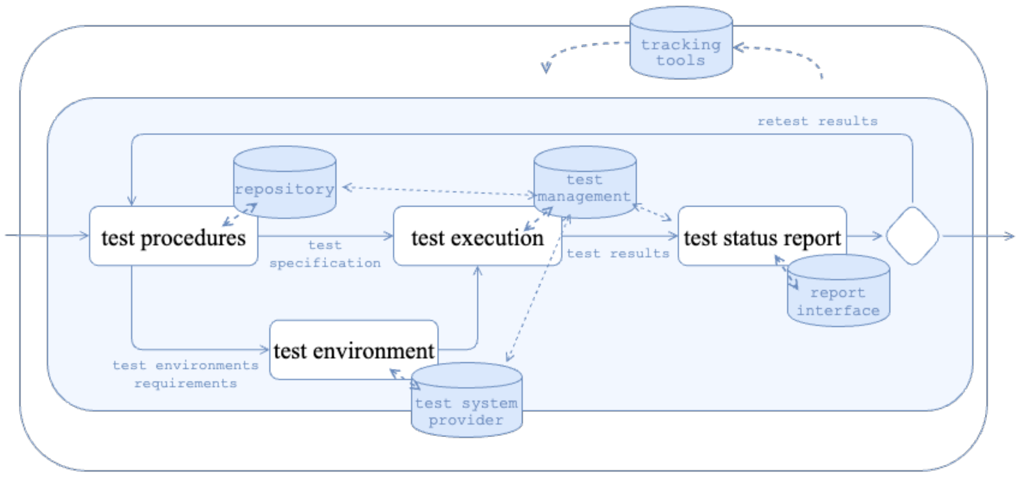


Figure 2.5: Extended diagram of dynamic test processes with tools specification 2.4

Naturally, test suites must be stored in a repository, for this purpose **git repositories** are used. They are either internal instances (*dist-git*) or public code hosting platforms to support an idea of open source (*GitLab*, *GitHub*). Git repository serves as a main test code storage that allows the maintenance of common test code in one place, prevents test code duplication, and also enables integration testing.

Tests stored consist of raw source code and metadata, as shown before in figure 2.1. Code is built with shell-level **testing framework** *BeakerLib*, that provides simple commands to generate test log on execution. It uses shell command to cover the objective, its description, and the expected result to generate the description with test results, each per line, allowing the structure of phases.

```

rlJournalStart
    rlPhaseStartSetup
        rlAssertRpm $PACKAGE
        rlRun 'TmpDir=$(mktemp -d)' 0 'Setup directory'
        rlRun "pushd $TmpDir"
    rlPhaseEnd

    rlPhaseStartTest
        rlRun "touch test" 0 "Create test file"
        rlAssertExists "test"
    rlPhaseEnd

    rlPhaseStartCleanup
        rlRun "popd"
        rlRun "rm -r $TmpDir" 0 "Clean up directory"
    rlPhaseEnd
rlJournalEnd

```

Figure 2.6: Example of test written with BeakerLib framework

Metadata per each test holds information such as test objective, contact name, environment requirements, and much more depending on the type of infrastructure used. Metadata typically uses an additional file and is noted in a markup language, based on the test case management tool used. It can be either passed via Makefile, together with an identifier to link the test case in a web-based test management organization (Nitrate, Beaker), or it is all stored in the metadata file based on YAML format that is read when the run is performed (tmt, fmf). This is further examined in the section 3.1.

In order to execute tests, the test management tool is used to identify the group of tests. This process is usually based on a test plan. Then it creates a test run passing the data to the environment to perform the execution.

When testing RHEL components, the test items are ready in the environment beforehand as they are shipped with the release of the RHEL version. If testing a new package before it is shipped in the release, it must be built in the Brew first and manually specified or via the Errata tool by automated processes passed in the compose. The services responsible for automation are Jenkins, Packit, etc.

By the time the test execution starts, the installation of environment requirements and preconditions is triggered within the **test system providers** *Beaker* or *Testing Farm*. These prerequisites are installed for each test or plan, or within the setup phase of the test case. This includes all tested components, additional components, and libraries used in the test required to be present, otherwise the test concludes with an error or warning.

Only afterwards the test execution can take place, driven by the framework mentioned above, generating well-structured test logs and test results. BeakerLib generates a log with details of execution, a log with a summary of test results, includes protocol details with information about the compose, and allows description used for definition of the test purpose and test contact.

```

...
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Setup
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: [ 12:00:03 ] :: [   PASS   ] :: Setup directory (Expected 0, got 0)
:: [ 12:00:03 ] :: [   PASS   ] :: Command 'pushd /tmp/tmp.oUo' (Expected 0, got 0)
:: [ 12:00:03 ] :: [   PASS   ] :: Command 'set -o pipefail' (Expected 0, got 0)
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Duration: 0s
::   Assertions: 3 good, 0 bad
::   RESULT: PASS (Setup)

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Test
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: [ 12:00:03 ] :: [   PASS   ] :: Create test file (Expected 0, got 0)
:: [ 12:00:03 ] :: [   PASS   ] :: File test should exist
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Duration: 0s
::   Assertions: 2 good, 0 bad
::   RESULT: PASS (Test)

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Cleanup
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: [ 12:00:04 ] :: [   PASS   ] :: Command 'popd' (Expected 0, got 0)
:: [ 12:00:04 ] :: [   PASS   ] :: Clean up directory (Expected 0, got 0)
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Duration: 0s
::   Assertions: 2 good, 0 bad
::   RESULT: PASS (Cleanup)

...
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Duration: 1s
::   Phases: 3 good, 0 bad
::   OVERALL RESULT: PASS ()

```

Figure 2.7: Example of summarized test log generated by BeakerLib framework, analogous to the source code in the figure 2.6

These logs are in the execution tool (Beaker, Testing Farm, tmt) wrapped in another journal supplemented with details of accompanying processes. Within the run reported in the test management tool, they are displayed or passed to another interface for better visualization. Thus test results are reported in **test management tool** like *Nitrate* or *tmt*, or in a dedicated report service such as ReportPortal. They can also pass artifacts linked to the logs (HTML file, Beaker log, Testing Farm log, etc.). Details of data management based on management tools and report services are elaborated in the section 2.2 below.

Generally, a **report interface** such *ReportPortal* or *Nitrate* (supplemented by Beaker logs), allows an analysis of test results with comments, and switchable statuses and ensure its preservation.

Based on the manual verification another decision can be made. Typically it needs a rerun in case the environment setup fails, otherwise it is identified as a bug that must be filed to fix it. Tracking tools used for the issue record are Bugzilla or Jira, which manage the planning behind another release and trigger the next cycle of testing.[9]

2.2 Test report solutions

As one of the main principles, Red Hat embraces open source, and most of the tools developed and used adhere to this philosophy. While not all tools used are entirely Red Hat-developed, most of them flourish in the open source due to contributions from Red Hatters, which help to keep the software tools under control and cover all the internal needs.

With this in mind, a new *Testing Tools* & *Infrastructure* community has emerged with the goal of enhancing tools, infrastructure, and testing processes to improve efficiency, compatibility, and alignment with Red Hat's strategies and open source beliefs. Members of this community have contributed to existing tools, identified gaps, and embarked on building a complex infrastructure capable of meeting future testing needs and facilitating potential improvements.

In line with open source philosophy and the evolving needs of RHEL subsystems, there is a perceived need for a more robust system aligned with current developments, driven by the community's initiative. Consequently, teams lean towards transitioning from the Nitrate test case management tool to the *tmt tool*, a tool specifically made for the needs of testing the operating system in both downstream and upstream conditions. Within its scope, it is superior to alternative test management systems that engage in other areas of Red Hat workflow such as *Polarion*, *Xray* for *Jira*, *codeBeamer* or *PractiTest*.

However, unlike Nitrate, tmt lacks a built-in test report interface. To address this requirement, an integration of tmt with ReportPortal through its plugin is taken into consideration. Before diving into the details of the integration with ReportPortal, it is important to conduct market research to evaluate alternative tools available for comparison.

2.2.1 Report tools

This section prompts us to investigate the reasoning behind the choice of ReportPortal and evaluates alternative options within the spectrum of reporting tools. Aligned with Red Hat's values, the focus is directed towards open source tools that provide straightforward test reporting and analysis capabilities, along with potentially compelling features. Here follows a list of a few popular open source tools meeting these criteria based on an independent source [3] supported with personal research:

(a) **Zebrunner**

- + AI/ML used for auto-classification of failures.
- + Real-time progress reports.
- + Support of test artifacts including logs, screenshots, and video recordings.
- + Customization options for test results.
- + Integration with popular frameworks and tools (Jenkins, Jira, Slack), REST API
- − May require expertise for setup and configuration.
- − Limited community support compared to more widely adopted tools.

(b) **Allure Report**

- + Generates attractive and interactive HTML reports.
- + Historical trend analysis
- + Extensive customization options and visualization of test results.
- + Support of multiple frameworks, REST API.
- − May require expertise for setup and configuration.

(c) **ReportPortal**

- + ML algorithms used for failure prediction and detection.
- + Real-time analytics and visualization of auto-test results.
- + Excellent in acquiring, aggregating, and analyzing test results.
- + Versatile customization of failure types.
- + Integration with major frameworks and tools (Jenkins, Jira), robust REST API.
- + High scalability and performance
- + Large open source community.
- − May require expertise for setup and configuration.

In summary, each of the software solutions listed offers distinct advantages in the realm of test reporting, very similar to each other. However, when considering the specific needs of RHEL Security teams, ReportPortal emerges as the optimal choice. Its adherence to open source principles, extensive reporting and analytics functionalities, scalability, flexibility, and robust community support renders it well-suited for the comprehensive analysis of RHEL components. By offering teams a reliable framework for data-driven decision-making and efficient testing processes, ReportPortal supports the efforts of teams already using the tool and sets the stage for the potential expansion across additional teams and subsystems through the implementation of the report portal plugin within the tmt tool.

Chapter 3

Analysis or requirements specification

Transitioning from general concepts, this chapter first delves into the RHEL security engineering infrastructure, exploring its current state, identifying deficiencies, and the motivation driving the need for improvements, as notable gaps in the transition towards automated testing and the integration of comprehensive test management tools exist. It also provides insights into the details of the planned infrastructure which leads to the core of the problem covered in the subsequent implementation - the integration of tmt and ReportPortal.

To gain a comprehensive understanding of the integration process and its requirements, it is essential to first establish the context and the key factors in the sections 3.2 and 3.3, as they may influence the implementation of requirements listed in the section 3.4.

3.1 Motivation and problem setting

This section aims to further elaborate on the current state of obsolete tools, vaguely introduced in the subsection 2.1.3. These include tools such as Nitrate, Beaker, and Bugzilla, which are currently used but do not fully cover the needs.

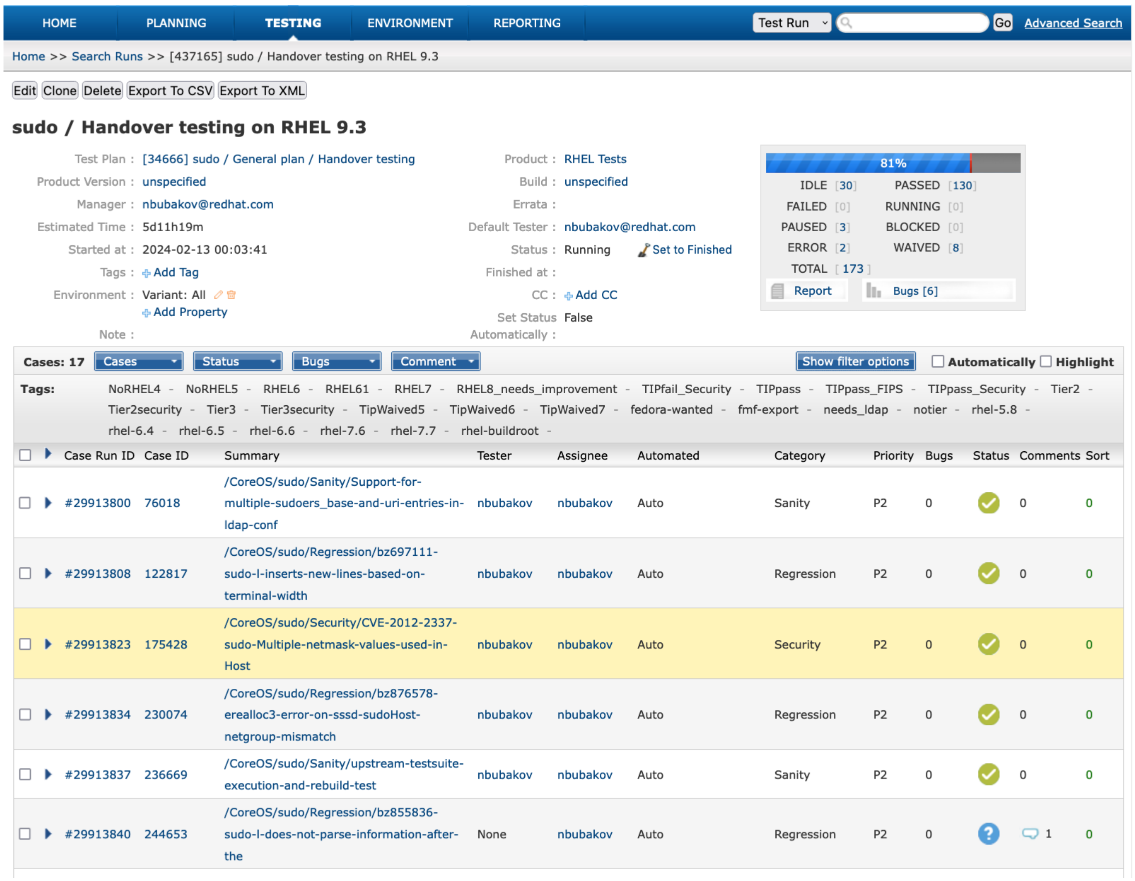
The subsystems that embrace the migration have a variety of different objectives. The components tested within the RHEL operating system are far from trivial; they may require multi-host systems, rebooting, continuity of voluminous procedures, etc. Moreover, RHEL components often align with those in systems such as Fedora or CentOS Stream. Thus, these and a few other supported projects require testing and maintenance as well, ideally in an upstream manner.

Despite the varied system objectives, it is essential for core test automation workflows to be shared by definition and provide clear interfaces for teams to seamlessly connect to these workflows. To address varied needs, a shared set of workflow building blocks is offered as a solution. This allows teams to choose the modules they need while still benefiting from an improving and unifying test experience.

Thus there's the initiative to build a whole new infrastructure that is further elaborated in this section, leading to the objective of the thesis.

3.1.1 Deficiency of the current status

The current solution covers the test management in the Platform Security Subsystem as a representative of RHEL Userspace Subsystems in the core, RHEL subsystems are web-based test case management systems called TCMS or **Nitrate**. It is not only a test management tool but also a report interface. Its report feature allows one to investigate test results, determine the status type, add comments, and link with tracking systems (Bugzilla, Jira). Additionally, it allows efficient filtering, attribute management, and metadata tracking for all its objects. [4]



Home >> Search Runs >> [437165] sudo / Handover testing on RHEL 9.3

sudo / Handover testing on RHEL 9.3

Test Plan :	[34666] sudo / General plan / Handover testing	Product :	RHEL Tests
Product Version :	unspecified	Build :	unspecified
Manager :	nbubakov@redhat.com	Errata :	
Estimated Time :	5d11h19m	Default Tester :	nbubakov@redhat.com
Started at :	2024-02-13 00:03:41	Status :	Running Set to Finished
Tags :	Add Tag	Finished at :	
Environment :	Variant: All Add Property	CC :	Add CC
Note :		Set Status Automatically :	False

81%

IDLE	[30]	PASSED	[130]
FAILED	[0]	RUNNING	[0]
PAUSED	[3]	BLOCKED	[0]
ERROR	[2]	WAIVED	[8]
TOTAL	[173]		

Cases: 17

Tags: NoRHEL4 - NoRHEL5 - RHEL6 - RHEL61 - RHEL7 - RHEL8_needs_improvement - TIPfail_Security - TIPpass - TIPpass_FIPS - TIPpass_Security - Tier2 - Tier2security - Tier3 - Tier3security - TipWaived5 - TipWaived6 - TipWaived7 - fedora-wanted - fmf-export - needs_ldap - notier - rhel-5.8 - rhel-6.4 - rhel-6.5 - rhel-6.6 - rhel-7.6 - rhel-7.7 - rhel-buildroot -

Case Run ID	Case ID	Summary	Tester	Assignee	Automated	Category	Priority	Bugs	Status	Comments	Sort
#29913800	76018	/CoreOS/sudo/Sanity/Support-for-multiple-sudoers_base-and-uri-entries-in-ldap-conf	nbubakov	nbubakov	Auto	Sanity	P2	0	✓	0	0
#29913808	122817	/CoreOS/sudo/Regression/bz697111-sudo-l-inserts-new-lines-based-on-terminal-width	nbubakov	nbubakov	Auto	Regression	P2	0	✓	0	0
#29913823	175428	/CoreOS/sudo/Security/CVE-2012-2337-sudo-Multiple-netmask-values-used-in-Host	nbubakov	nbubakov	Auto	Security	P2	0	✓	0	0
#29913834	230074	/CoreOS/sudo/Regression/bz876578-erealloc3-error-on-sssd-sudoHost-netgroup-mismatch	nbubakov	nbubakov	Auto	Regression	P2	0	✓	0	0
#29913837	236669	/CoreOS/sudo/Sanity/upstream-testsuite-execution-and-rebuild-test	nbubakov	nbubakov	Auto	Sanity	P2	0	✓	0	0
#29913840	244653	/CoreOS/sudo/Regression/bz855836-sudo-l-does-not-parse-information-after-the	None	nbubakov	Auto	Regression	P2	0	?	1	0

Figure 3.1: Nitrate's screen view of the test plan report enabling further analysis of test cases

It provides the following object hierarchy:

(a) **test case**
Representing a specific test with a unique identifier, attributes, and relevance for distributions, it can also be reused across the other plans.

(b) **test plan**
Defining a group of tests, allows feature management and arrangement within a tree hierarchy of plans.

(c) **test run**
Executing a test plan for a specific compose and assigning status per each test case.

Though Nitrate's interface is quite comprehensive, access to the test management is also required by test automation and terminal commands. This is possible via the Python-Nitrate library, a high-level Python API built on the XMLRPC API that Nitrate offers.

Additionally, the approach is also supplemented by internal scripts implementing the primary test case management functions and integration with Beaker, offering possibilities such as adding new test cases, merging duplicates, populating it with Beaker job results, etc.[6]

However such approach Nitrate and its additions offer ends up with test case metadata partially uploaded in the web interface and partially stored in the Makefile for Beaker execution. Some data are passed with automation triggers (Jenkins), some are done manually from the terminal, and some actions are allowed only via the interface. This leads to a system that is too bulky, has data fragmented into several places, and is difficult to build on additional improvements and automation, in addition, the project stopped its development and is considered obsolete.

Based on the internal documents and discussions [10], there are also other core infrastructure parts shared among RHEL teams, including Bugzilla, Beaker, Resource Hub, RHEL compose gating, and others not relevant to the thesis's scope. Unfortunately, the tools and services responsible for integrating these parts for team-specific use cases are fragmented and typically developed and maintained by individuals as secondary responsibilities. This poses risks to pipeline security and contributes to instability and lack of resilience, resulting in increased workload and stress. Given the demands of continuous integration, gating, and the expanding scope of test automation, any infrastructure outages significantly impact team productivity. A shared infrastructure operated as a service by a dedicated team of expert Site Reliability Engineers offers the potential to optimize team capacity over the long term.

3.1.2 Motivation example

To unveil the improvement plan, the initiative outlined by *the Big Picture* aims to address long-term problems in testing infrastructure by implementing the proposed architecture of **Shared OS Testing Infrastructure**.

Based on internal documents and discussions [10], its primary goal is to establish a consistent workflow, familiar to teams across RHEL CI, Fedora CI, CentOS Stream CI, and upstream projects, centered around tests, plans, and runs. It also intends to establish a metadata specification system that liberates metadata from the old test case management system, eliminating inconsistency and data fragmentation. Finally, the initiative seeks to replace current fragmented approaches used within *RHEL Security Subsystems*, enhance and unify the test experience across all supported environments, and easily support any future improvements.

The architecture is composed of building blocks that support the entire testing lifecycle. It begins with the initial pull request, continues through packaging and adding essential context details, organizes tests into plans, and ultimately facilitates their execution. Whether the tests run automatically or manually, the process concludes by reporting and storing the results along with all necessary test artifacts. This comprehensive approach ensures a seamless and efficient testing workflow from start to finish, ideally within a single command or simple automated setup.

The diagram provided in the figure 3.2 interprets how these building blocks integrate to enable consistent workflows and comprehensive test management, aligning with the envisioned goals of *the Big Picture* initiative.

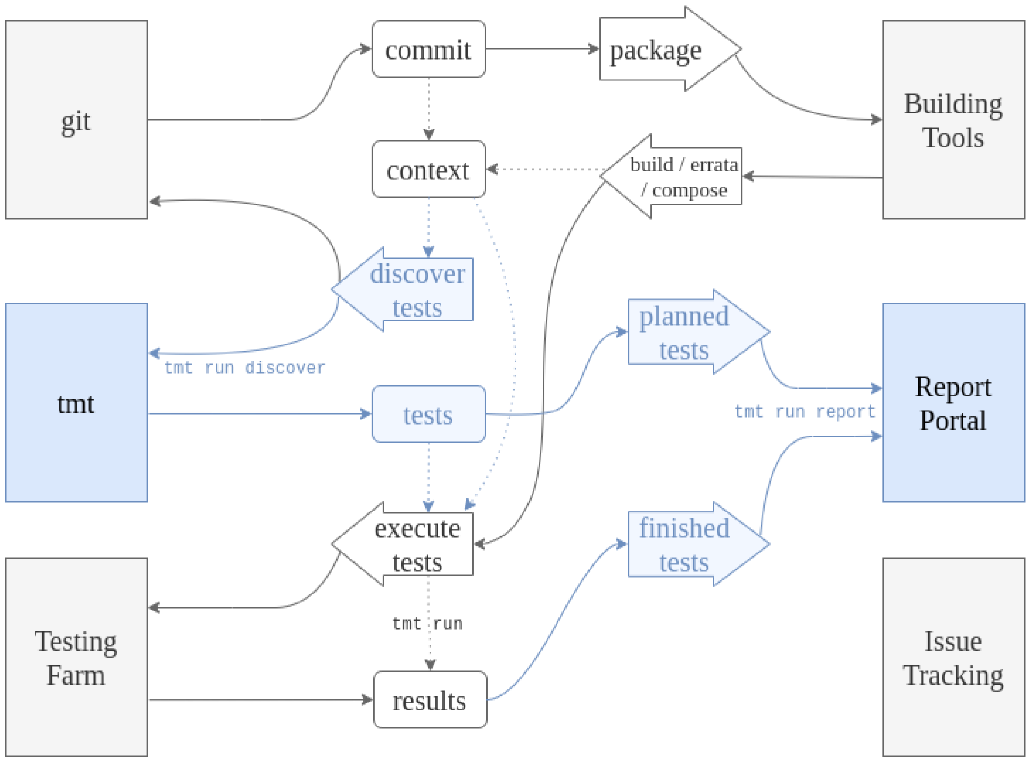


Figure 3.2: An adopted diagram of Shared OS Testing Infrastructure, emphasizing the role of the integration tmt and ReportPortal that is further examined

The first important building block is represented by **git**. The git repository serves as a consistent and stable storage place for the majority of the data, typically application source code, tests, and test metadata. Utilizing the fmf metadata format ensures that all relevant data for execution are kept within the repository without any extra dependencies. In case, the data are split into more repositories (e.g., upstream, downstream), git also provides remote referencing. Consequently, this enables neat contribution, prevents code duplication, and provides an easy approach to enable integration testing.

Alongside git, another key piece is **tmt**, building its functionality on the fmf format. This comprehensive tool provides teams with consistent and concise configuration to execute tests easily. Contents of this crucial building block include the test metadata itself and the plans, which group tests and enable testing. All configurations are stored as plain text versioned under git using a human-readable YAML-based format with inheritance and hierarchy. the tmt specification also has an additional feature of tracking requirements in future stories, such as implementation, test, or documentation coverage, all within the git repository.

In addition to its data specification capabilities, tmt can facilitate the entire testing process through several operational steps, as elaborated in the subsection 3.2.1. It supports the execution based on selected plans, starting from the identification of all test sources and their requirements, through environment preparation and setup based on preferences and requirements, up to the test execution itself. It then reports the test results and artifacts and performs cleanup tasks afterwards.

For purposes such as CI, tmt's function is closely related to the **Testing Farm**, a testing system as a service that provides a variety of test environments supporting rich hardware requirements. It serves as a unified test execution back-end with seamless access to machines, boxes, and virtual machines from systems such as AWS, Beaker, OpenStack, Resource Hub, and more under the Artemis system, all obtained by a simple request. the transition to Testing Farm will enable teams to execute test jobs in the cloud and easily expand testing capabilities in RHEL, Fedora, and CentOS Stream.

Furthermore, the **building tools** such as *Packit* in upstream picking up GitHub pull requests, and other services like *Koji*, *Brew* and *Module Build Service* play a critical role in the automation process. Pipelines orchestrate tasks, listen to events, and trigger activities such as packaging or running tests.

Another essential component is **ReportPortal**, which serves as a platform for displaying uploaded results, logs, and other test artifacts obtained after the execution from tmt data within the environment provided by Testing Farm. This platform not only addresses functionalities utilized by the old system Nitrate, such as deep search capabilities in well-structured test result history and persistent audit logs, but also offers additional features including dashboards, saved filters, custom issue types, and machine learning-based auto-detection for advanced analysis of tests. ReportPortal enhances these capabilities in a more visually appealing and comprehensive manner, providing a robust solution for managing and analyzing test results effectively

Finally, the processes are overseen by issue-tracking systems **Jira** and **Bugzilla**. While both systems are not new in RHEL teams, there is a Bugzilla-Jira transition, so Jira is no longer used only for tracking team activities but covers bug tracking and eventually becomes a central tracking, planning, and collaboration tool. Jira can handle larger product requirements where Bugzilla's feature set is insufficient to meaningfully capture the work effectively. Despite the emphasis on Jira, Bugzilla remains a bug-tracking system used to collaborate with partners and for bugs that need public errata.

To sum up the workflow, it involves triggering tests based on events such as pull requests or commits, initiating builds, and gathering context from artifacts. Test discovery is a critical step that identifies all required tests based on the context, including manual tests. The discovered tests can be passed to a ReportPortal and reported as a planned test to show progress for comprehensive testing. Finally, test execution is performed using the testing farm API, with results and detailed logs updated and stored for audit purposes and investigation. This comprehensive workflow streamlines the testing process and ensures effective management and analysis of test results and artifacts.

3.1.3 Aim of the thesis

As the title of this thesis reveals, the aim is an integration of the tmt tool and ReportPortal tool. Ultimately, the proposed architecture of Shared OS Testing Infrastructure strongly affects the topic of this thesis and vice versa, the architecture also depends on the thesis's implementation much as it creates an essential connection between the key building blocks in the infrastructure.

As indicated in the previous subsection and visualized in figure 3.2, the emphasized flow of data starts with data source in a git repository, where the metadata provides an identification of tests when tmt intends to discover them within its run. Furthermore, tmt passes provided data including the names of plans and tests to ReportPortal in order to create the report structure with idle status prepared for additional upload. Then when tests are executed and results are generated they are passed to the ReportPortal once again, and if the ReportPortal has prepared a report for them, it just updates it with obtained data of test results and artifacts.

To present the targeted connection, it is involved in the part of infrastructure where the specified tests are identified and passed as idle reports to the ReportPortal, further where the results from the test execution are reported to the ReportPortal.

Although not immediately apparent, the tmt specification plays the main role behind all these processes, thus it is an essential factor for reports to be displayed in the interface or ReportPortal.

For purposes of the infrastructure's development, there was a proof of concept conducted independently and outside the scope of this thesis. It involved using a tmt plugin to upload a simple XML file to ReportPortal, as explained in the design chapter in subsection 4.1.1. The objective was to offer a temporary solution for infrastructure development purposes and served as a foundational starting point for the tasks undertaken in this thesis. But this temporary solution does not cover the needs of the integration of tmt and ReportPortal as it omits the use of several features the ReportPortal provides, and even completely skips the step of an initial report and the progress update. This implementation helped with assembling the idea of the plugin realization and supported the analysis for requirements to be set and listed in the section 3.4 and the design elaborated in the section 4.3.

Also tmt is an comprehensive tool with far more use cases than visualized in the plan for Shared Infrastructure, see subsection 3.4.1. And though the shared infrastructure is the main motivation for the development of tmt specification and its integration to ReportPortal, it is not aimed only for this purpose. It can be simulated via isolated functionality of tmt, and afterwards to be adopted in order to run via Testing Farm request. As the Testing Farm is out of the scope of this thesis, the test coverage and evaluation of the implementation targets isolated approach within the functionality of tmt.

3.2 Tool analysis

In the context of this thesis, which focuses on the integration of tmt and ReportPortal, it is essential to conduct an in-depth introduction to these tools. This introduction lays the groundwork for understanding the terminology, functionalities, and capabilities necessary for their effective utilization.

The analysis of tmt and ReportPortal involves a detailed examination of the use cases they address, along with their respective features and limitations. This understanding is crucial for identifying the requirements and design considerations needed to seamlessly integrate these tools within the scope of the thesis project.

3.2.1 Tool tmt

The tmt is a powerful tool that as well uses **test case**, **test plan** and **test run** as the primary objects the test management system is built on, with a few differences from the Nitrate. Above all, it implements metadata specification which allows storing all needed test execution data in one place, directly in a git repository that can even be remotely referenced. The specification of metadata units is covered on several levels:

(a) **core** | L0
Attributes used across all other metadata levels such as summary, description, test contact, id, tag, order, adjust, etc.

(b) **tests** | L1
Attributes closely related to individual test cases such as test script, framework, directory path, maximum test duration or environment requirements, etc.

(c) **plans** | L2
Attributes related to plans, allowing the definition of all details for each step of the test run listed below 3.2.2, in addition, there is context, environment variables, etc.

(d) **stories** | L3
Attributes related to stories in order to track expected or required features, these are title, priority, story and example.

Such metadata is written in YAML based **fmf format**, both human and machine readable, and stored in the repository with source code. For a complex image, the configuration data per plan are composed of summary, description, details of the run steps including a list of tests to be discovered and included in the plan, context attributes and environment variables. The data defined per tests are summary, description, contact, executable file, environment requirements, environment variables, tags and any other additive test data written in the same format. See the example in the figure 3.3.

directory tree

```
.
├── plan
│   ├── plan_01.fmf
│   └── plan_02.fmf
├── test_01
│   ├── main.fmf
│   └── test.sh
└── test_02
    ├── main.fmf
    └── test.sh
```

plan/plan_01.fmf

```
summary: Plan metadata
description: Testing fmf

discover:
    how: fmf
    test:
    - /test_01
    - /test_02
execute:
    how: tmt
report:
    how: reportportal
    project: test_tmt

context:
    component: tmt
    distro: rhel-8
    arch: x86_64
    purpose: upgrade
    milestone: rc

environment:
    RELEASE: rhel8
```

test_01/main.fmf

```
summary: Test metadata
description: Testing fmf

contact: nbubakov@redhat.com

test: ./test.sh
framework: beakerlib

require:
  - name: /smoke_library/basic
    url: https://git.com/smoke/
    ref: master
    type: library
  - library(another/smoke_lib)

tag:
  - flaky_test
tier: '1'

environment:
    TEST_VAR: test_string

duration: 5m
```

Figure 3.3: Example of fmf metadata for test plans and test cases

Moreover, tmt is also an extensive command line tool that allows to create new tests, safely and easily run tests across different environments, review test results, debug test code and enable tests in the CI using a consistent and concise configuration.

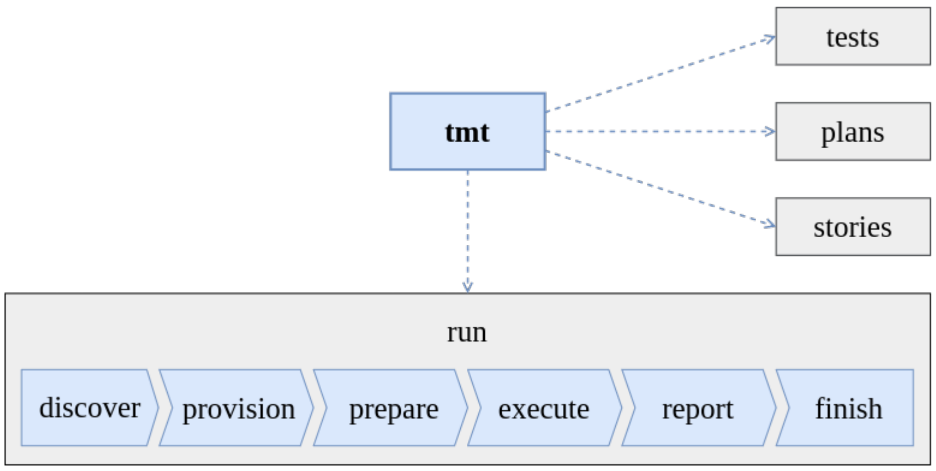


Figure 3.4: Structure of the tmt tool with the focus on run functionality for the purpose of the thesis

Its *run* feature involves 6 main steps, that are each chronologically performed per each plan:

(a) **discover**
Identify test cases and gather information about them.

(b) **provision**
Provision an environment for testing or use localhost.

(c) **prepare**
Prepare the environment for testing.

(d) **execute**
Run tests using the specified executor.

(e) **report**
Provide test results overview and send reports.

(f) **finish**
Perform the finishing tasks and clean up provisioned guests.

Steps drive the run of a plan or group of plans, can be either run all by default or specified to omit others. Each of them includes several plugins to support additional features that are needed, that can be in line easily specified.[12]

The discover plugins enable to specify the identification of tests included in the run. The choice of the environment is done by provision plugins, which allow to run the tests locally, in container via podman or connect to any machine via ssh, it is also integrated with systems such Beaker, Artemis, 1minutetip or TestCloud virtual machines. The prepare plugins define the way of environment setup, via shell script by default, but it can be switched into other ways, for example with the packages defined in the errata.

There are several plugins for each step, and though all the steps are essential for the run, it is the ‘report’ step that is the most relevant for the purpose of the thesis. The ‘report’ step offers three elementary report plugins such **display** for log output in terminal, **html** in order to get a better but simple overview in HTML format, or **junit** for needs of an upload to test report tools. Additionally, there is **polarion plugin** which covers a direct integration with a Polarion Software, and **reportportal plugin** with primitive upload of data via xunit format. The reportportal plugin is where the assingment of the thesis is targeted, so it can be rewritten into a complex plugin via ReportPortal's REST API to allow additional features ReportPortal enables.

For an overview understanding of tmt for purposes of the thesis, the command structure can be summarized with a diagram in the figure 3.5.

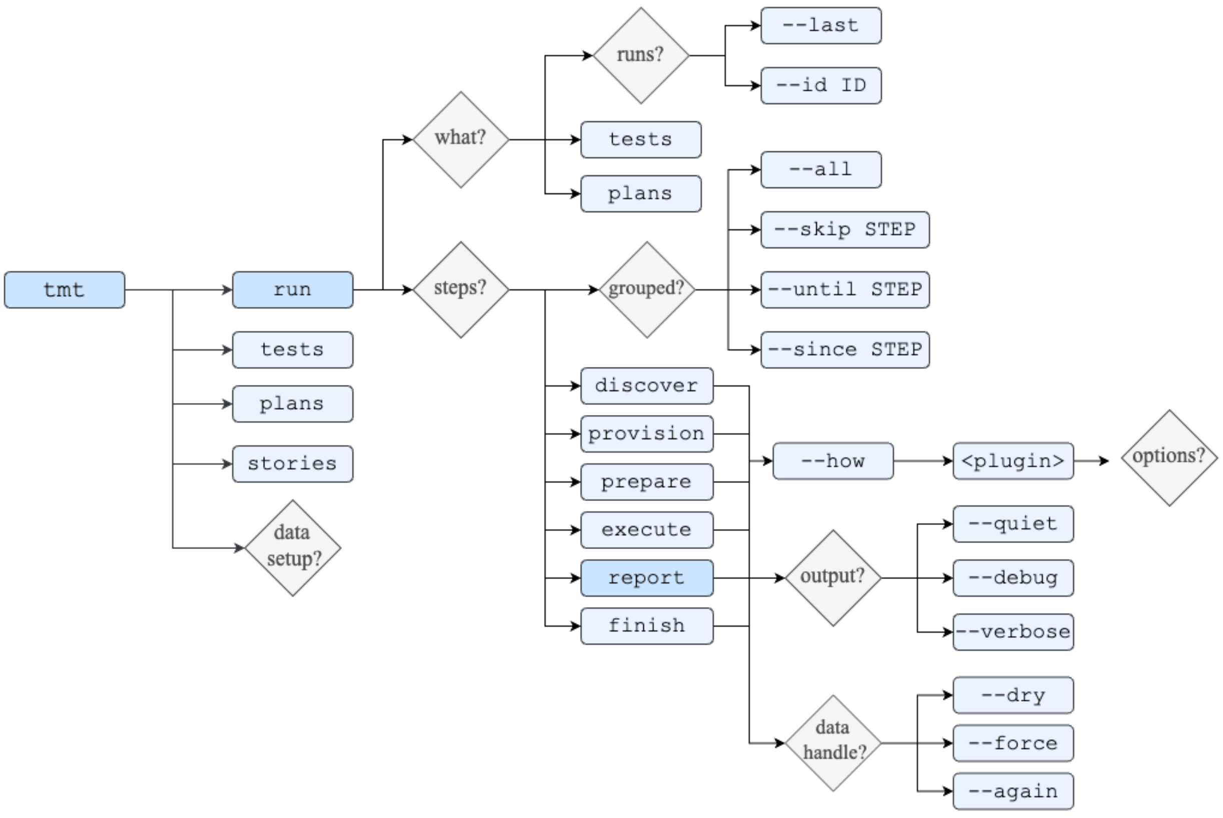


Figure 3.5: Structure of the tmt command tool and its options for the purpose of the thesis

The run steps are performed per each plan chronologically and those to be performed can be selected. To continue with the previous run it is done via `--last` or `--id ID` for its id specification. Each step is performed once per run unless it is specified by –force (deleting the previous data) or –again (preserving the previous data) to repeat the step. There is an universal option `--help` for listing options and details related to any part of the command that is listed before it.

To name a few example commands:

`$ tmt init` to create initialize tmt specification in the repository;

`$ tmt plans ls` to list plans configured in the repository;

`$ tmt run discover tests --name . plans --default` to perform a 'discover' step on the run with all the tests under the current working directory, within the plan that was not pre-configured;

`$ tmt run --until report` to perform run except finishing the run;

`$ tmt run --last --all execute --force` to perform the test execution by force on the latest run with deleting data of previous 'execute' step.

3.2.2 Tool ReportPortal

ReportPortal is a complex service with a clear and intuitive interface. It can accommodate numbers of different projects, which allows convenient and organized parallel work on several projects, or offers a space for private purposes. Each project has a separate organization, customization. Within a project, the access and permissions per user are given by the role such Project Manager, Member, Operator and Customer.

The tab panel on the side offers quick access to dashboards, launches, filters and debugs.

The first space view lists all project **dashboards**, offering the visualization of test analytics within several types of tables or graphs based on filters.

Then there is the main part, the list of launches. The launch term is derived from the elements structure that ReportPortal offers:

(a) **launch**
The main container that encapsulates the hierarchy of all the other elements, with suites or tests accessed directly.

(b) **suite**
An optional container that typically encapsulates other suites or tests.

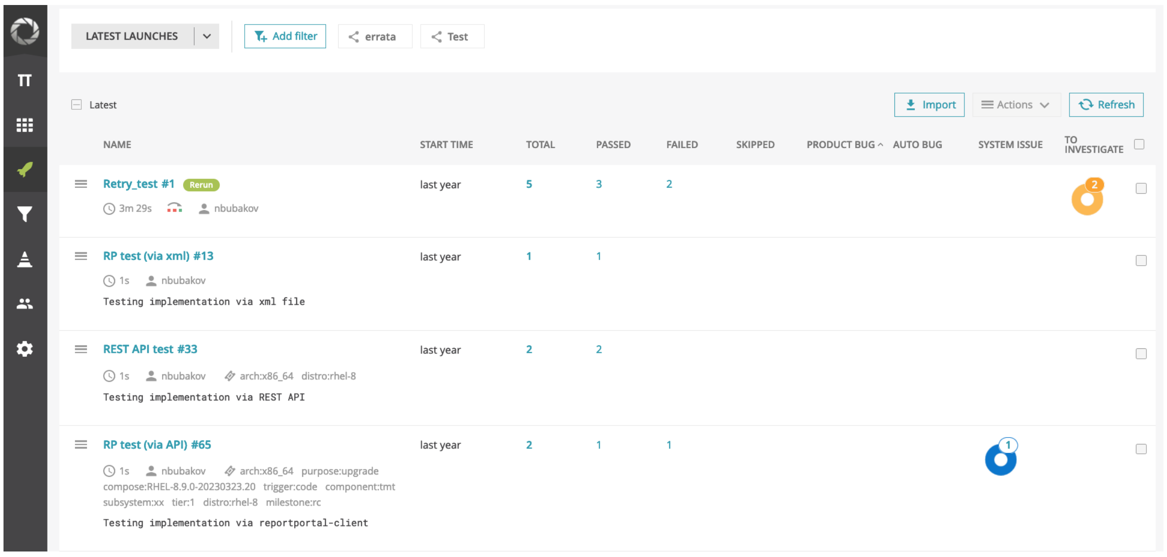
(c) **test**
An item per each test case, encapsulating information about the test case.

(d) **step**
A collapsible element within the log area that wraps other steps or logs.

(e) **log**
An output written in the log area directly or within the step.

Where the items such as launch, suite and test have name, description, attributes, log area and unique identifier in the URL.

To describe the main features in the view of **launches**, there are filter options, buttons for Import, Actions an Refresh. Actions allow to edit, merge, compare, move to debug, force to finish or delete all selected launches. the launches in the main view can be displayed either by run or grouped by the launch name. In the overview, each of them displays its start time and a number of tests in total, those PASSED, FAILED, SKIPPED, WITH PRODUCT BUG, AUTOMATION BUG, SYSTEM ISSUE or labelled as TO INVESTIGATE. Furthermore, there is a similar view on the list of suites after opening one of the launches, alternatively one of the nested suites. Only the overview of test items offers a listing of METHOD TYPE, STATUS, START TIME and specific DEFECT TYPE per each test case. Beside the list view, an overview of suites and tests also offers to display unique errors, log view and history table.



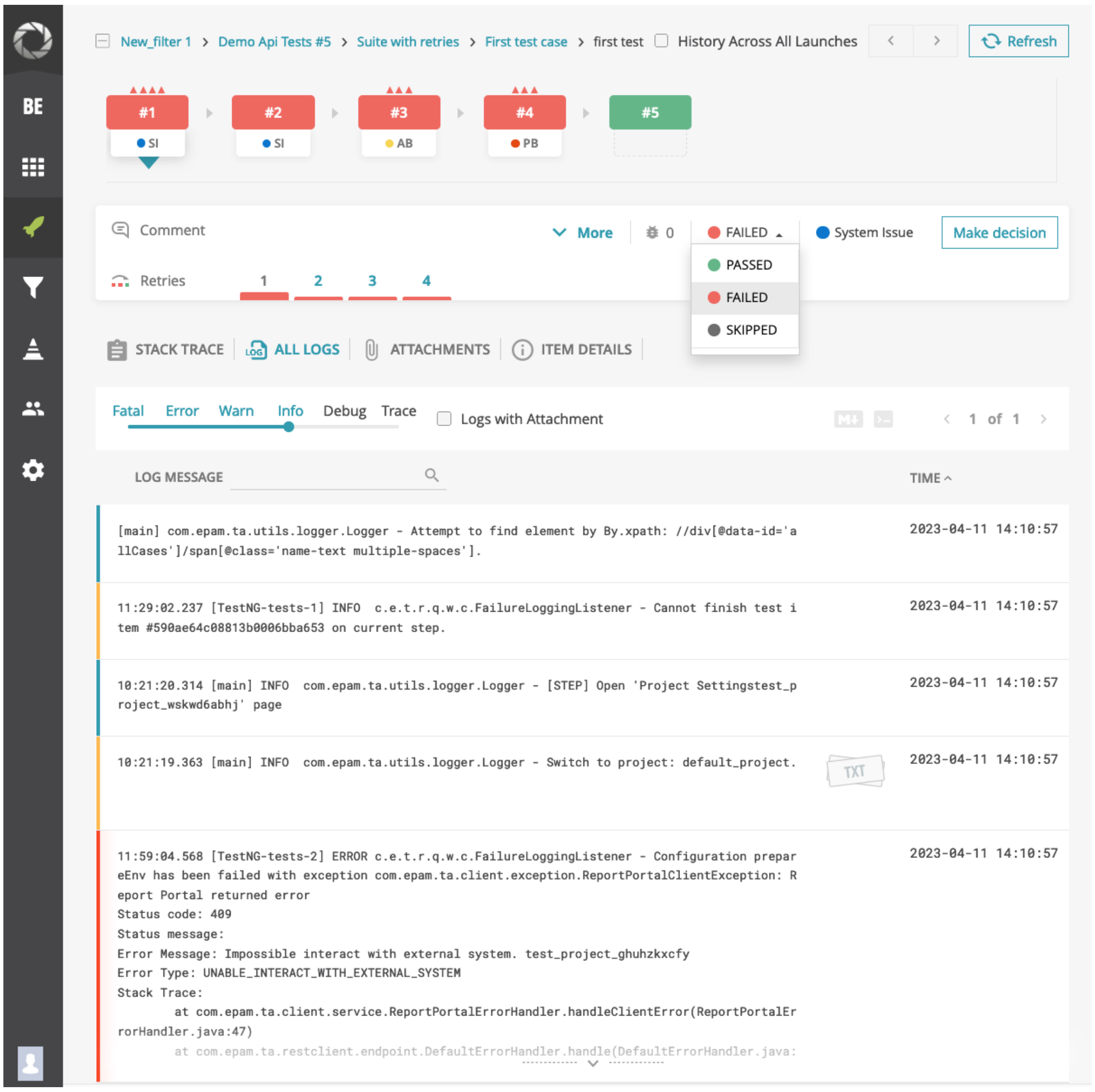
NAME	START TIME	TOTAL	PASSED	FAILED	SKIPPED	PRODUCT BUG	AUTO BUG	SYSTEM ISSUE	TO INVESTIGATE
Retry_test #1 Rerun 3m 29s nbubakov	last year	5	3	2					2
RP test (via xml) #13 1s nbubakov `Testing implementation via xml file`	last year	1	1						
REST API test #33 1s nbubakov arch:x86_64 distro:rhel-8 `Testing implementation via REST API`	last year	2	2						
RP test (via API) #65 1s nbubakov arch:x86_64 purpose:upgrade compose:RHEL-8.9.0-20230323.20 trigger:code component:tmt subsystem:xx tier:1 distro:rhel-8 milestone:rc `Testing implementation via reportportal-client`	last year	2	1	1				1	

Figure 3.6: ReportPortal's screen view of launch list that demonstrates a simplistic visualization of test results per each launch

Finally, opening the **test item** offers the most information including all the details per test case, additionally history line and retry items, both explained later. There is a tab for report analysis that can be expanded and used to comment on it, switch its status, mark the issue or integrate with the issue-tracker. On the central tab there are listed STACK TRACE, ATTACHMENTS, ITEM DETAILS, HISTORY OF ACTIONS and the most important ALL LOGS which stands for the log area. **Log area** contains logs or steps with logs, they all can be collapsed for a better overview and can also be searched for the presence of a wanted expression. The log area also offers several phases like *Fatal*, *Error*, *Warn*, *Info*, *Debug* and *Trace* to reduce the log entries based on the verbosity, where the Fatal phase is the most brief and the Trace phase shows the most. Within the log are also the attachments allowed.

Additionally, the information shown in the item details are test name, description, status, attributes and length of the test run. There are also properties such as code reference, test case ID and parameters intended for environment variables, that are important for **history aggregation** shown in the history line or history table. History of test cases intends to aggregate tests based on the real history of the test case, thus it is built on the uniqueness of the test case identifier provided from the test management system. If the test case ID is not provided, it is generated based upon the code reference and parameters, otherwise based on its path names (test case name and all parents names, except the launch name) and parameters.

The names are generally used as an important identifier in ReportPortal. They are also used to identify launch and items within for a feature generating an additional element of **retry item** within the test item. This feature allows to rerun the tests and report its results within the latest launch with the same name by default, or within the launch specified with an UUID. However, the mapping of items is based on individual path names.



```
[main] com.epam.ta.utils.logger.Logger - Attempt to find element by By.xpath: //div[@data-id='a     2023-04-11 14:10:57
llCases']/span[@class='name-text multiple-spaces'].

11:29:02.237 [TestNG-tests-1] INFO  c.e.t.r.q.w.c.FailureLoggingListener - Cannot finish test i     2023-04-11 14:10:57
tem #590ae64c08813b0006bba653 on current step.

10:21:20.314 [main] INFO  com.epam.ta.utils.logger.Logger - [STEP] Open 'Project Settingstest_p     2023-04-11 14:10:57
roject_wskwd6abhj' page

10:21:19.363 [main] INFO  com.epam.ta.utils.logger.Logger - Switch to project: default_project.     2023-04-11 14:10:57

11:59:04.568 [TestNG-tests-2] ERROR c.e.t.r.q.w.c.FailureLoggingListener - Configuration prepar     2023-04-11 14:10:57
eEnv has been failed with exception com.epam.ta.client.exception.ReportPortalClientException: R
eport Portal returned error
Status code: 409
Status message:
Error Message: Impossible interact with external system. test_project_ghuhzkxcfy
Error Type: UNABLE_INTERACT_WITH_EXTERNAL_SYSTEM
Stack Trace:
        at com.epam.ta.client.service.ReportPortalErrorHandler.handleClientError(ReportPortalEr
rorHandler.java:47)
        at com.epam.ta.restclient.endpoint.DefaultErrorHandler.handle(DefaultErrorHandler.java:
```

Figure 3.7: ReportPortal's screen view of the test case that demonstrates history aggregation, retry items, analysis properties and all logs up to the Info phase, including an attachment.

Moving forward, the high level **filters** can be accessed from the filters tab as well as from the view of launches, they offer a view of launches specified by a variety of condition combinations and can be saved for additional access. This filter can be conditioned by the number of total launches, total tests and tests based on its status or issue type; by the start time or by the presence of expression in the launch name, description, owner or attributes. Though similar but only transient filtering is allowed within the launch or suite, currently there is no filter access to test names, descriptions or attributes within the high level view.

Lastly, the „Debug“ tab is intended for private debugging with no access to the role Customer having almost the same features as „Launches“, only filters cannot be saved.

To conclude ReportPortal, project setting allows a range of modifications such as retention periods, integrations with other systems (Jira, Email Server, etc.), properties of auto-analysis or pattern-analysis and creation of new Defect Types within all issue groups (PRODUCT BUGS GROUP, AUTOMATION BUGS GROUP, SYSTEM ISSUES GROUP, NO DEFECT GROUP, TO INVESTIGATE GROUP). The interface also enables an access to the profile with generated user's token and also provides well-structured API with dynamical requests useful for study purposes of the integration.

3.3 Mapping the terminology

To report results of obtained by tmt tool and send them to ReportPortal, there must be strictly defined the way of mapping the terms used as they may widely differ throughout both systems.

For **tmt**, there are three main elements in terms of the run data, similar to what is known from the Nitrate system:

- **test**
  A specific test, identified by its name, corresponding to the tmt specification data L1 in 3.2.1.

- **plan**
  A group of tests within the hierarchy data unit above the tmt steps (see figure 4.5), corresponding to the tmt specification data L2 in 3.2.1.

- **run**
  Representing a single high-level data unit made by a new run of plans that is separately identified and can be additionally reused.

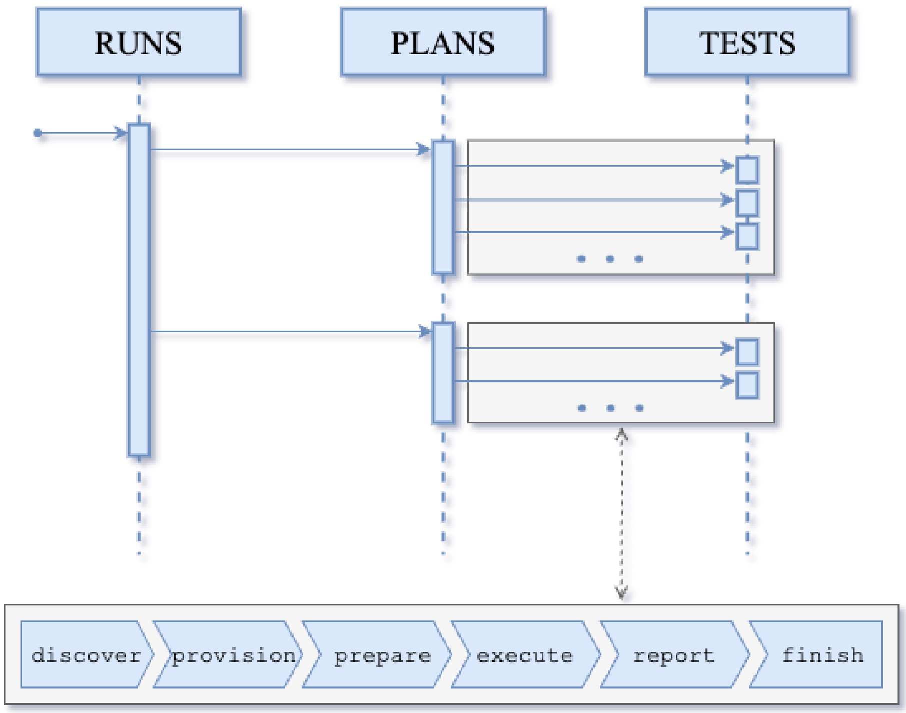


Figure 3.8: Sequence of the process between the primary elements run, plan and test; displayed also with the run steps positioned within the tmt data relation.

This hierarchy is important for understanding the execution data obtained in order to report test results and logs, that is further examined in the design subsection 4.2.1.

While the **ReportPortal** hierarchy is mainly based on the following three elements:

- **test**
  An element for an individual test case, wrapping all its data (name, description, attributes, etc.), logs and analysis details.
- **suite**
  A wrapper of elements involving other suites or test elements, with its own separate data.
- **launch**,
  A high-level wrapper of elements involving suites or test elements, with its own separate data.

Compared to the three-level elements of tmt, the ReportPortal's three elements are not limited to three levels. Due to the optional suite elements with a nesting ability, ReportPortal allows whatever level structure including two and more (*launch* > *test*, *launch* > *suite* > *tests*, *launch* > *suite* > *suite* > *test*, etc.).

For practical purposes, there can be 2 different ways of mapping the tmt and ReportPortal elements. Both have its positive and negative factors and can be prioritised based on the individual use cases or preferences.

The first one implicitly clear is to map them analogously like showed in the figure 3.9 under the term SUITE-PER-PLAN mapping, which is with ReportPortal element to the tmt one mapped as *launch - run*, *suite - plan*, *test - test*. This way of mapping adheres to the familiar well-structured display of reports, often preferred e.g. for purposes of errata testing. However, it makes tmt plans in ReportPortal suites much less accessible, as the subsection 3.2.2 states, the filter from the high-level overview of launches cannot approach the data per suite or test. Though plan data within a suite can contain specific information essential for the analysis or reports such reference to a tested component, compose or architecture, etc. within the field of suite name, description or attributes. These data currently cannot be accessed unless they are pulled out for the launch data.

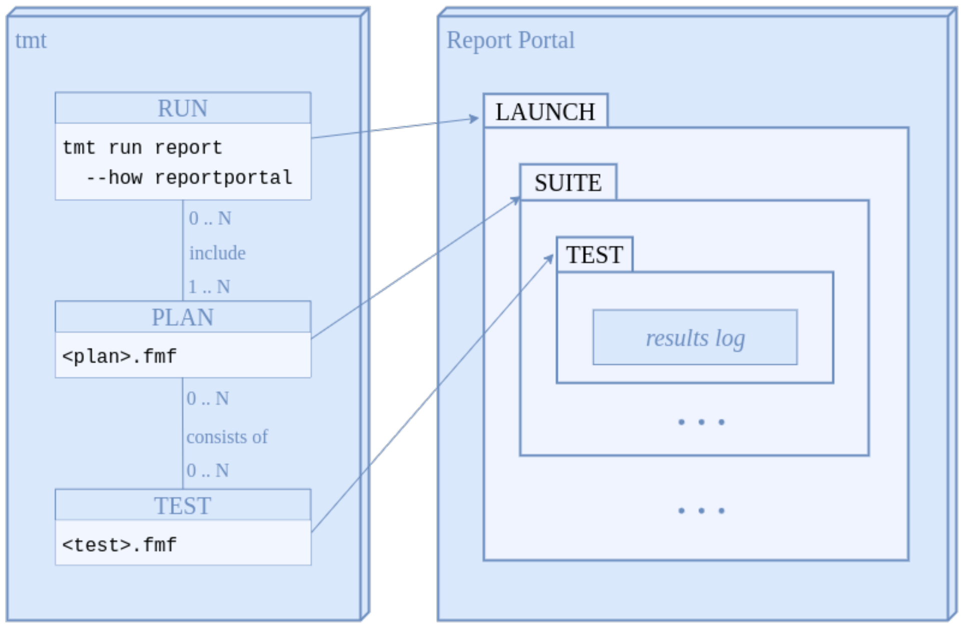


Figure 3.9: Relations of the tmt and ReportPortal elements with SUITE-PER-PLAN mapping

Alternatively, there can be a simple two-level structure established that would map tmt plans to high-level launch elements in the interface of ReportPortal. This would create a group of new launches per run, a new launch per each executed plan, as visualised in the figure 3.10. This approach not only allows a direct search access to the data extracted from the plan, but it is more natural and much easier to implement within the tmt structure, as each plan is processed separately as referred to above (figure 3.8). This type of mapping is further in the thesis referred to as LAUNCH-PER-PLAN mapping.

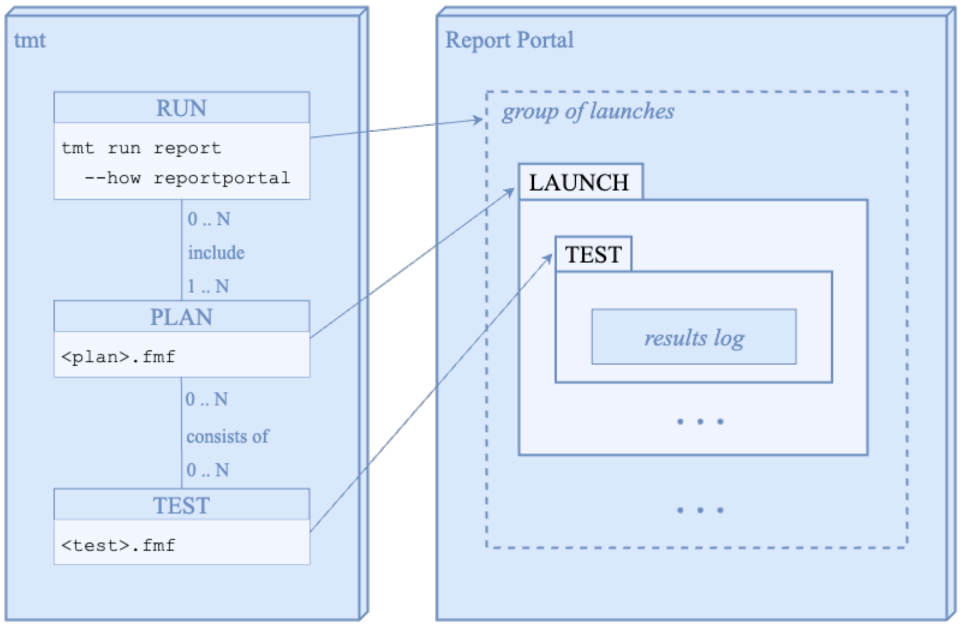


Figure 3.10: Relations of the tmt and ReportPortal elements with LAUNCH-PER-PLAN mapping

3.4 Requirements

The integration of tmt and ReportPortal within the scope of this thesis is driven by a necessity to address the essential needs inherited from the outdated Nitrate system and align with the evolving demands of advanced RHEL testing process, particularly within the domain of Testing Tools community and the Shared OS Testing Infrastructure initiative.

As the mission of this thesis is built on real team needs and real-life scenarios, there were many factors to consider in order to assemble the list of use cases and consequential requirements. This process involved collective input from representatives of the Testing Tools community. Though real requirements can be volatile in the course of time based on continuous tool evaluation, for purposes of the thesis, these are summarized in universal manner to cover all the needs that were communicated at first.

This section examines the use-case demands and the reasoning behind them as a base for setting detailed functional and non-functional requirements. Afterwards, these can be used for the purpose of project design and implementation, providing clarity and guidance for the integration of tmt and ReportPortal into the testing infrastructure.

3.4.1 High-level use cases and analysis

The primary motivation behind plugin implementation via the ReportPortal API is to leverage the full potential of ReportPortal's capabilities. Given that the rapidly growing tool tmt is utilized across many subsystems and extends even beyond the RHEL, Fedora or CentOS Stream reaches, it is important that tmt plugin covers not only essential use cases for neighboring teams but also to offer a variety of options that can be potentially demanded.

A collaborative effort involving key representatives responsible for Testing Tools improvement identified several key use cases that serve as examples for distinct groups of features the plugin can offer. Each of these use cases must be grounded in real-world justifications within testing processes, ranging from basic to advanced demands in terms of implementation or usage practices. They ensure the plugin meets diverse and evolving testing requirements across various contexts and scenarios.

Before naming the use cases, they cover scenarios such as fundamental data upload to ReportPortal on a run, additional upload on rerun of the tmt run or additional upload from different tmt run. Though data that are uploaded can differ between use cases, a data upload generally refers to creating a ReportPortal item and providing it with data such as name, description, attributes, logs, status, etc.

(a) **User can upload test results per each plan.**
This is a simple use case that covers an essential scenario where the user desires to display test results and logs in the ReportPortal after a common test run. The data upload utilizes the straightforward and practical approach of LAUNCH-PER-PLAN mapping, demonstrated in figure 3.10. With this mapping, a new ReportPortal launch is generated after each tmt plan is executed, containing the respective tests within. Each launch possesses a name, description, and attributes corresponding to each plan's data, akin to ReportPortal and tmt tests. This mapping allows an easy approach to all plans data from the high-level list of launches, where their name, description and attributes can be directly searched, especially when plans differ in the component or distribution tested. This use case serves as a foundation for the primary inte-

gration of tmt and ReportPortal before advanced features can be applied, for more implementation details see the section 5.1.

(b) **User can upload test results grouped per run.**
Another use case for the same scenario as above, but with a structured launch with suites and tests within, mirroring the tmt run, plan, and tests with SUITE-PER-PLAN mapping, illustrated in the figure 3.9. This approach addresses purposeful testing like errata testing. Though it utilizes the hierarchy supported by ReportPortal, it requires a more comprehensive design enabling the suite upload to the same launch after execution of each tmt plan. It involves an extra data layer, recommending the user to define the launch name and launch description, and the launch attributes display the intersection of all plans data. This is implemented in the section 5.2.

(c) **User can rerun the tests and update it within the same launch**
Another scenario arises when the tests are already reported to ReportPortal, but a minor change was made in the environment setup, source code of the test, or the test item, making it unnecessary to duplicate the run in another launch. Using the feature that ReportPortal offers, the plugin can enable the retry items within the old test items in the same launch whenever demanded by the user. ReportPortal maps the elements based on their name within the last launch of the same launch name. See section 5.3 for more information.

(d) **Automated process can upload IDLE tests and update them after execution**
This use case addresses the needs of testing automation processes indicated in the OS Shared Infrastructure introduced in the section 3.1. It aims to display activity in the ReportPortal interface, even though the tests processed take a long time to finish execution. The run begins by discovering tests, which are then displayed in the ReportPortal with no logs and IDLE state, prepared for an additional update after the run is executed. To achieve precise mapping, the identifiers of ReportPortal items are stored within tmt, offering additional rerun possibilities such repeated aggregating of logs.Implementation is covered in the section 5.4.

(e) **User can additionally upload new tests to already existing launch**
The final scenario occurs when there is already a reported launch but it is incomplete because some tests were forgotten, or the tests cannot be executed in one run, e.g., multihost testing. In such cases, there should be an option to manually upload additional tests or suites into the launch identified by the launch ID displayed in the URL of the ReportPortal launch. Analogically, in the structure of SUITE-PER-PLAN mapping, there is an option to update one or more tests to a specific suite.
See 5.5.

3.4.2 Functional Requirements

In this subsection, the list of functional requirements assembled based on the use cases named above is presented. the requirements are related to the tmt tool and its integration with ReportPortal within the tmt plugin, supporting both the use of tmt as both command-line tool and metadata specification. They are also further examined and explained in the design section 4.3, most of them particularly in its subsection 4.3.1 defining the options that derived from them.

ID	Requirement	Description
FR1	Reporting to ReportPortal on request	The tmt tool must support integration with ReportPortal, defined either by using ‘`--how reportportal`’ option in the tmt command within the ‘report’ step, or equivalently through metadata specification.
FR2	Connection to ReportPortal instance	The plugin must be able to access the ReportPortal using the mandatory data obtained from user such as authorization token, URL to the ReportPortal instance and name of the targeted project.
FR3	Reporting with LAUNCH-PER-PLAN mapping	New request to upload report should create a new launch item per each plan of tmt run, when prompted by option for LAUNCH-PER-PLAN mapping. The data are uploaded to ReportPortal in the structure of ‘`launches > tests`’.
FR4	Reporting with SUITE-PER-PLAN mapping	New request to upload report should create a new launch item per tmt run with suites within per each plan, when prompted by option for SUITE-PER-PLAN mapping. The data are uploaded to ReportPortal in the structure of ‘`launch > suites > tests`’.
FR5	Printing out a link to the reported ReportPortal launch	The plugin must display a link in the terminal in order to redirect the user to the ReportPortal launch the report is uploaded to.
FR6	Uploading detailed report data	The plugin must upload all test results and logs per each test case obtained from the test execution step to ReportPortal.
FR7	Definition of relevant name and description for the ReportPortal launch	The plugin should support the capability of user to define the launch name or launch description in ReportPortal otherwise use default alternatives based on metadata.
FR8	Definition of relevant attributes per elements in ReportPortal	The plugin should upload relevant context and contact information recognized from the metadata and display them as attributes per each launch, suite and test in ReportPortal
FR9	Definition of test parameters in ReportPortal	The plugin should upload all relevant environment variables recognized from the metadata and display them as parameters per test in ReportPortal
FR10	Definition of test case id in ReportPortal	The plugin should upload the tmt id recognized from the metadata and display it as test case id per test in ReportPortal
FR11	Consistency in ReportPortal’s test history aggregation	The plugin must not disturb the consistency of history aggregation that ReportPortal enables.
FR12	Launch rerun via build-in option for ReportPortal retry items	The plugin should support an additional data uploads to a previous ReportPortal launch via name-based mapping, wrapping it in separated retry item within each test item.

. . . continued on next page

FR13	Launch preparation with IDLE test items for tests only discovered	The plugin should prepare a launch and display tests with IDLE status if tests were discovered but yet not executed.
FR14	Launch rerun via tmt run with stored identifiers	The tmt should support an additional data uploads to an existing ReportPortal launch when rerunning the 'report' step within the same tmt run. This update must use stored ReportPortal item identifiers in tmt for precise mapping.
FR15	Data upload to the launch based on its URL identifier	The plugin should allow additional uploads of new tests or suites to an existing launch based on the URL identifier obtained from the user.
FR16	Data upload to the suite based on its URL identifier	The plugin should allow additional uploads of new tests to an existing suite based on the URL identifier obtained from the user.
FR17	Test coverage of the plugin	The tmt must have the test coverage for all newly implemented features in the report plugin for integration with ReportPortal.
FR18	Usage documentation within the tmt tool	The tmt must provide specification of all added features in its manual requested on `--help`.

Table 3.1: Functional Requirements

3.4.3 Non-functinal Requirements

As it was already implied, the integration of tmt and ReportPortal should be implemented within the tmt tool as a report plugin using ReportPortal's API. Here it is summarized in the table, so it can be further simply referenced.

ID	**Requirement**
NR1	The solution must be implemented within the tmt tool
NR2	The integration of tmt and ReportPortal should use ReportPortal API
NR3	The plugin implementation must be written in Python 3
NR4	The test coverage should be written with BeakerLib framework in bash
NR5	All user data passed to the plugin must be obtained from the command arguments or through fmf metadata
NR6	Language of all operative and descriptive elements should be English

Table 3.2: Non-Functional Requirements

Chapter 4

Design

This chapter examines the possible approaches to achieve the objective of the thesis leading to the integration of tmt and ReportPortal as tmt plugin using the ReportPortal API as the best solution. This decision is thoroughly justified and demonstrated on alternative attempts. Before the design details of the objective, there is elaborated structure of tmt and capabilities of ReportPortal API building up their fundamental functionality. Leading to the design essentials that describe the theory and all steps it takes to fully cover the requirements in the implementation.

4.1 Ways to integrate tmt and ReportPortal

There are two points of view that the integration of the tool can be done either from the side of ReportPortal or from the side of tmt.

There was an attempt to implement a direct access to the ReportPortal via extension of ReportPortal within a diploma thesis Test Results Management System Complementing the tmt Tool [4]. But this solution was later considered non-effective as the extension intervenes to the external tool, requiring an extra responsibility for its maintenance.

That's why it makes sense to examine the ways to implement the objective from the side of tmt, internally developed tool, which is the aim of this thesis. Beside direct code intervention, the ReportPortal offers several approaches for external tools to communicate with its interface. There is a library for Python clients and REST API offered as the only compatible mediators with tmt. Further in the section, there are described three approaches for integration tmt with ReportPortal, simple import of xml file, elaborated communication via the API library for Python client and finally the elaborate solution via REST API. Providing the specification of the attempts that were demonstrated as no sufficient solution for the requirements, and suggestions sufficient for this thesis assignment.

4.1.1 A tmt plugin - Via JUnit XML import

First approach a simple REST API command to import JUnit XML report into ReportPortal. This is done via report plugin establishing the contact with ReportPortal after obtaining all required data as user token, target project, instance URL, launch name and execution data. The data such as test results and logs are then transformed into JUnit XML format via Python module to create structured data that ReportPortal can work with and inserted into API command, as further demonstrated. the results can remain in an XML file or be compressed into ZIP file for more effective upload. There is also the possibility of setting launch description and launch attributes. [13]

```

 import requests

 url = "https://demo.reportportal.io/api/v1/<PROJECT>/launch/import" \
       "?description=<LAUNCH_DESCRIPTION>&launchName=<LAUNCH_NAME>"

 headers = {'Content-Type': 'multipart/form-data',
            'Accept': '*/*',
            'Authorization': 'Bearer <USER_TOKEN>'}

# CREATE LAUNCH AND IMPORT XML/ZIP FILE
response = requests.request("POST", url, headers=headers,
    files={'file': (<FILE_NAME>, <BYTESTREAM>, "application/zip")})

```

Figure 4.1: Demonstration of Python request for importing JUnit XML report via ReportPortal API

Though this approach was implemented in order to provide the proof of concept, out of the scope of the thesis. Its capabilities were too limited in the sake of ReportPortal features, uploading hierarchy of launch > suites > test with test names and logs only. The functionality such as test attributes, parameters, id, code reference or any additional information per item was not supported, failing the requirements FR8 - FR10. Results could be uploaded only after the run finished when all data were available not showing the progress and also reruns or any additional uploads to existing launch, were not supported as the requirements FR12 - FR16 suggest. It was only suitable as a temporary solution and foundation for the task of this thesis.

4.1.2 A tmt plugin - Via API library

Another option is using a common client library for Python-based agents that provides commands to upload the data to ReportPortal, with more detailed approach. The commands provided by the library can create a launch loading it with name, description and attributes, as well as can create other items within it. The items can be defined with a type of the item which is suite, test or step. In case of a test item, its data can be enriched also by parameters, test case ID and code reference (FR7-FR10). What is more, detailed logs are supported, defining the log level or uploading attachments (FR6). Finally, the items can be closed with result status and issue type, and afterwards launch can be finished connection is terminated. It allows uploading data in real-time and also supports the rerun tag for launch in order to update existing launch via retry item (FR12).

```

from reportportal_client import ReportPortalService

client = RPClient(endpoint, project=project, api_key)
client.start()

launch = client.start_launch(name, timestamp(), description)

item_id = client.start_test_item(name="Test Case",
                                 start_time=timestamp(),
                                 item_type="STEP",
                                 description="First Test Case",
                                 attributes={"key1": "val1",
                                             "key2": "val2"},
                                 parameters={"var1": "val1",
                                             "var2": "val2"})

client.log(time=timestamp(), message="Hello World!", level="INFO")
client.log(timestamp(), "Screenshot of issue.", "WARN", attachment)

client.finish_test_item(item_id, timestamp(), status, issue)
client.finish_launch(end_time=timestamp())
client.terminate()

```

Figure 4.2: Demonstration of Python API request for importing JUnit XML report to Report Portal

The library `reportportal_client` mostly showed a potential to cover all the requirements as it was implemented within the scope of this thesis and attached within appendix (A.2.2). However, it showed weaknesses that were discovered only during the implementation as it was actively in development and still not documented properly.

After all it was refused as insufficient solution for the integration of tmt and Report Portal with following reasoning:

(a) Limited in functionality and currently unstable behaviour.

  (aa) It does not allow the update of an existing launch with additional results F13-FR16.

  (ab) It does not show any progress update before the launch is finished and connection terminated.

  (ac) It fails to differentiate the item types such suite, test, step.

  (ad) Rerun feature has unstable behaviour, different in outcome for launch > suite > test structure (new logs within retry items, appending attributes) and for launch > test structure (appending logs, rewriting attributes).

(b) Using library is less stable, efficient and managable than REST API.

  (ba) It is not packaged as RPM, thus it would not work unless installed from pypi or there was invested time to get it packaged in Fedora.

  (bb) It is in an active development with updates that are not backward compatible and break the plugin.

(bc) When new features are added to the REST API it takes time to update the python client, which could slow down the development in the future.

4.1.3 A tmt plugin - Via REST API

After all as the most reliable solution available for the communication between tmt and Report Portal is chosen the Report Portal REST API. This is fully developed, documented and offers the most functionality. the structure of data upload is similar to the one with API library. There is only no need to establish connection with Report Portal, but afterwards the requests are used to create launch and items within it, upload logs and finally close the items with corresponding test results, as further demonstrated in 4.2.2. the REST API allows to define all properties displayed in ReportPortal which covers requirements FR6 - FR10. It provides requests for managing issue types, obtaining any item-related data and also offers the access to the previous launches based URL ID or UUID of created elements. Hence it as well covers the requirements FR13 - FR16 leaving the rest up to the implementation purely within tmt plugin that is fully capable of it.

4.2 Program structures

Following the introduction of tools tmt and ReportPortal in the section 3.2, this section aims to delve into tmt functionality and its code structure in order to understand its building blocks before detailed design of all required features is demonstrated. Furthermore, the REST API of ReportPortal is presented to learn all the possibilities of this mediator between the command-line tool tmt and the interface of ReportPortal.

4.2.1 Class structure of tmt

The program structure of tmt tool source code is very complicated as it integrates many features and plugins. As presented in tmt documentation [12], it is based mainly on two groups of classes: **functional classes** and **data containers**.

As figure 4.3 below with an overview of the first group present, the '`Common`' class is the parent of most of the available classes, providing common methods for logging, running commands and handling of working directory. To name a few, it implements `read()`, `write()` for comfortable file access and `run()` method for an easy command execution.

The '`Core`' class together with its child classes '`Test`', '`Plan`' and '`Story`' cover the Metadata Specification. The other child classes are managed in order to keep structure of tmt steps, particularly '`Steps`' and those under '`BasePlugin`' covering each individual step of tmt run. Therefore, the class for a functionality of test report to a ReportPortal instance must be '`ReportReportPortal`' derived from the common parent of report plugins '`ReportPlugin`'. This class inherits an essential function `go()` from the `GuestlessPlugin` class so the functionality can be performed and logged in 'report' step within the step sequence.

The tmt structure offers in addition to standard python modules its own modules which have many predefined functions and methods that are either inherited or imported throughout the tmt and used for various purposes. To name a few, there is a logging module, module with result definitions and especially the module with utilities. The logging module `tmt.log` provides important functions for communication with user in terminal such `info()`, `verbose()` and `debug()`, with logging priority in this order. It logs in attributes (key: value) and allows specifying the text color and shift integer for indentation. In `tmt.utils`, there is a function `field()` important for defining the option data and other aspects related to option. It is declared with `@overload` which provide a range of data types and numerous parameters offered to define the option. The most relevant are for naming the option, setting the holder for value, setting the default value, providing the description for '`--help`', setting it as flag and even allowing multiple uses for lists or dictionaries.

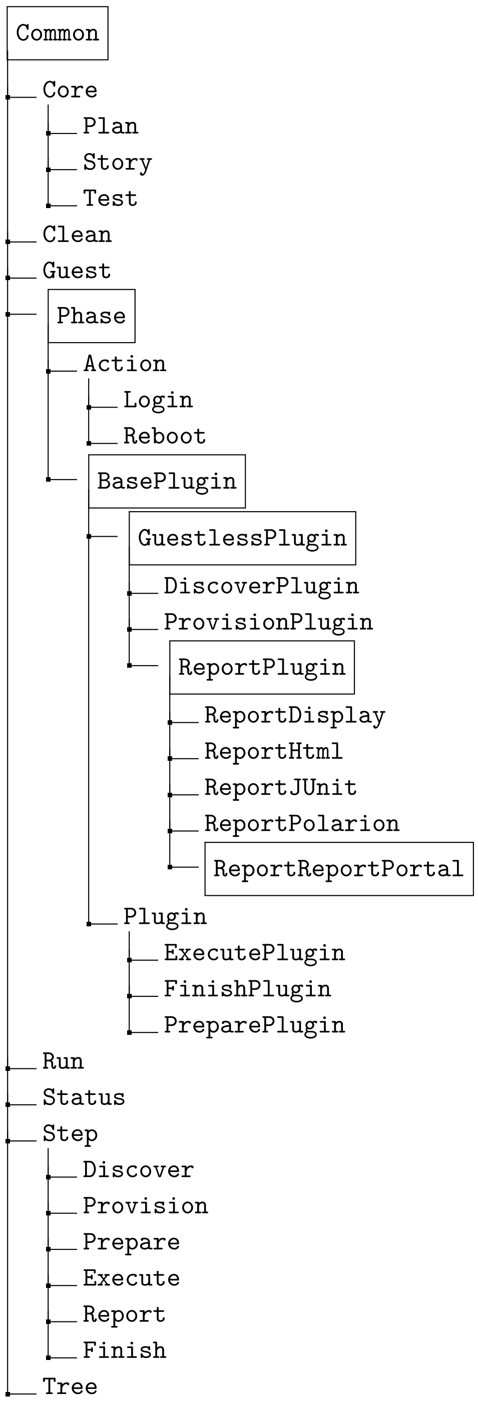


Figure 4.3: Inheritance of tmt method classes

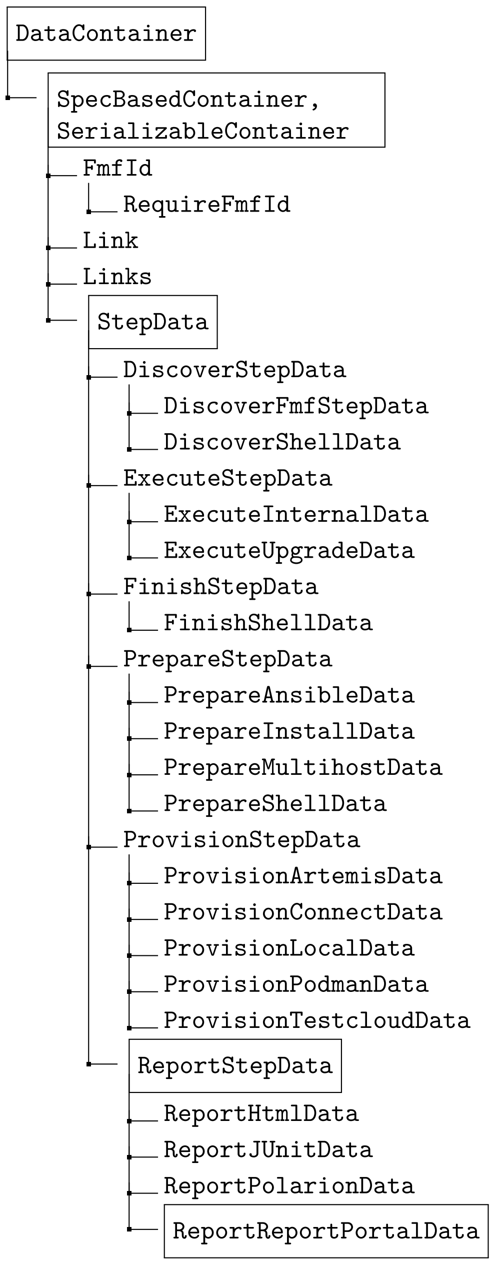


Figure 4.4: Inheritance of tmt data classes

In order to obtain test results and logs from the execution data and report them, it is important to understand the hierarchy of data stored. As the figure 4.5 suggests, the class containers manage the file system created for each tmt run. This file system is labeled by run identifier storing run data and data per each plan. Run data compose of run log and YAML file listing all plans, steps and central data for the run. Each plan is composed of data per step, which reflects that via requested plugins *the steps are performed chronologically and separately per each plan.* List of tests included in the plan and all configuration data are stored under 'discover' step in YAML file. Their results are under **'execute' step**, where YAML file points to the source per each test, storing files such `output.txt`, `journal.txt`, and other related to test metadata and execution results. At last, there is a directory for 'report' step with YAML file prepared for data definition from the report class.

```
/var/tmp/tmt/run-032

.
└── plan
    ├── plan_01
    │   ├── data
    │   ├── discover
    │   ├── provision
    │   ├── prepare
    │   ├── execute
    │   │   ├── data
    │   │   │   ├── test_01-1
    │   │   │   ├── test_01-2
    │   │   │   └── test_02-3
    │   ├── report
    │   └── finish
    └── plan_02
        ├── data
        ...
```

Figure 4.5: Overview of directories within the data structure stored per run

To elaborate the data accessible from the plugin via `ReportReportPortal` class for purpose of the thesis, there are essential plan data providing name (`self.step.plan.name`), brief description (`self.step.plan.summary`), list of context attributes (`self.step.plan._fmf_context.items()`) and other data obtained from fmf plan configuration. Then there are data within the step related to its processing, especially list of tests within the 'discover' step (`self.step.plan.discover.tests()`) and tests results within the 'execute' step (`self.step.plan.execute.results()`). These methods allow a direct access to all the test's attributes and execution data needed for uploading all required data about test cases.

4.2.2 ReportPortal REST API

The ReportPortal REST API provides a comprehensive and well documented set of operations that enable users to interact with the ReportPortal server via HTTP requests. The requests are targeted for integration of ReportPortal functionality into custom workflows, building automation scripts, or developing custom tools.

To introduce the capability and composition of the REST API requests, there are mandatory data elements that are required in each request to be successfully sent to the targeted Report Portal instance. It is a user token used for authorization within a request header. The token serves for an identification of Report Portal user, it is displayed in the user profile of the Report Portal instance. Besides, there are an instance URL and a project name needed to determine the target of request.

```

url = "<REPORTPORTAL_URL>/api/v1/<PROJECT_NAME>/"

headers = {'Content-Type': 'application/json',
           'Accept': '*/*',
           'Authorization': 'Bearer <USER_TOKEN>'}

```

Figure 4.6: Demonstration of mandatory data for Python requests of Report Portal REST API, related to the requirement FR2.

Here are groups of API-supported operations and demonstrations that are most relevant for the purpose of this thesis:

(a) **Launch Controller**

(aa) Create a new empty launch with properties.
(ab) Create a launch with JUnit XML import.
(ac) Merge set of specified launches in a common one.
(ad) Update launch properties.
(ae) Stop or finish launch with status.
(af) Get launch properties, identifiers, result status, etc.
(ag) Search launches based on launch properties.
(ah) Delete launch.

There are up to 30 requests related to launch with many additional functionalities, but mostly varying the options and approach choice, which is mostly via project name, launch ID or launch UUID, in some cases also launch name or other launch properties.

```

# CREATE LAUNCH
response = requests.request("POST",
    url + "launch", headers=headers,
    data=json.dumps({
        "name": <LAUNCH_NAME>,
        "description":<LAUNCH_DESCRIPTION>,
        "attributes": <LAUNCH_ATTRIBUTES>,
        "startTime": timestamp()}))

# CLOSE LAUNCH
response = requests.request("PUT",
      url + "launch/<LAUNCH_UUID>/finish",
      headers=headers,
      data={"endTime": timestamp()})

```

Figure 4.7: Demonstration of Python API requests related to launch within report upload to Report Portal

For creating a new report upload, there must be created a launch and at last finished. With launch creation, the launch parameters are defined. The mandatory launch parameters are launch name and start time, then there are launch description, launch attributes, launch mode and pair of fields for activation of rerun with an optional specification of launch UUID. The request response returns UUID and number of created launch.

(b) **Item Controller**

(ba) Start a root or child item with item parameters.
(bb) Update an item parameters.
(bc) Finish an item.
(bd) Attach external issue for items.
(be) Get item parameters, statistics, history, etc.
(bf) Search items based on item parameters or its contents.
(bg) Delete an item.

There are also up to 30 requests related to items (suites, tests, steps, etc.) with functionalities varying only in its requested scope of details and approaches either via launch ID or launch ID or searches based on particular item parameters.

```

# CREATE TEST ITEM
response = requests.request("POST",
    url + "item", headers=headers,
    data=json.dumps({
        "name": "SUITE_NAME",
        "description":<TEST_DESCRIPTION>,
        "attributes": <TEST_ATTRIBUTES>,
        "parameters": <TEST_PARAMETERS>,
        "testCaseId": "TEST_ID",
        "startTime": timestamp(),
        "type": "TEST",
        "launchUuid": <LAUNCH_UUID>}))

# CLOSE TEST ITEM
response = requests.request("PUT",
      url + "item", headers=headers,
      data=json.dumps({
          "launchUuid": <LAUNCH_UUID>,
          "endTime": timestamp(),
          "status": <TEST_RESULT>,
          "issue": { "issueType": <DEFECT_TYPE>}}))

```

Figure 4.8: Demonstration of Python API requests processing test root item within the report upload to Report Portal, which is related to the requirements FR3 and FR7 - FR10

Within launch there is a suite or test with its parameters, it must be started and closed afterwards. There are root items and child items based on the parent of the item (launch or suite item). The parameters of item include name, description, attributes, parameters (environment variables), test case ID (test management system), code reference, unique ID and type of the item. the Report Portal allows diverse types of item, beside SUITE, TEST, STEP, there is SCENARIO, BEFORE_CLASS, AFTER_CLASS, BEFORE_GROUPS, AFTER_GROUPS, BEFORE_METHOD, AFTER_METHOD, STORY, BEFORE_TEST, AFTER_TEST the response returns the UUID of the item, which is used as identifier of item passed to child item or any other internal data.

(c) **Log Controller**

(ca) Create log

(cb) Get or search log

(cc) Delete log

There are 13 log-related requests in order to manage logs and its parameters. Log request allows uploading file as attachment, and it sorts logs based on the log level, which is FATAL, ERROR, WARNING, INFO, DEBUG. There are also identified either by ID or UUID.

```

# UPLOAD LOG
response = requests.request("POST",
    url + "log/entry", headers=headers,
    data=json.dumps({
        "message": <TEST LOG>,
        "itemUuid": <TEST_UUID>,
        "launchUuid": <LAUNCH_UUID>,
        "level": level,
        "time": result.end_time}))

```

Figure 4.9: Demonstration of Python API requests for logging with report upload to Report Portal, related to FR6

Beside the operations for standard report upload named above, Report Portal API allows management of projects, users, dashboards, integration with other systems, etc.

In the figure 4.10 is an example of GET request for a defect type defined in the project for a check related to FR13.

```

    \item Project setting
    \begin{enumerate}[label = (\alph{enumi}\alph{enumii})]
        \item  Create, get or delete project issue sub-types
    \end{enumerate}

```

Figure 4.10: Demonstration of Python API requests for logging with report upload to Report Portal

4.3 Details of tmt plugin design

Based on the primary use-cases that inspired the requirements, the requirement coverage can be divided into several parts by design, there is a core functionality - report upload either with trivial LAUNCH-PER-PLAN mapping or with more complex SUITE-PER-PLAN mapping, covering the requirements FR1 - FR11. And then there are individual extensions of core report for each of the requirements FR12- FR16.

This section will elaborate the essential factors based either on user choice or tools possibilities and limitations. To cover the user choice, there are options which are necessary for most of the requirements to specify the intended action, which must be further examined for all supported combinations. And then, there can be designed the principles of targeted use-cases, that rely on the tools. Especially understanding of tmt functionality explained in sections 3.2.1 and 4.2.1 is essential to build on. It is especially the sequence of the steps 'discover', 'provision', 'prepare', 'execute', 'report' and 'finish'. From those, only steps'**discover**', '**execute**' and '**report**' are relevant to the plugin design. Together with the fact that the order of steps cannot be changed, the run goes through them per each plan and the run can be reused. This influences the progress report, need to store data so 'report' step can communicate with previous or following 'report' steps within a run and the approach each use case can be achieved.

4.3.1 Definition of plugin options

As FR1 requires, the tmt plugin class enables the selection of options for reporting to ReportPortal by specifying parameters immediately after `--how reportportal`' in the command section of the 'report' step (`tmt run report`'). These options can also be specified within the fmf plan metadata or as environment variables using the template format `TMT_PLUGIN_REPORT_REPORTPORTAL_<option>`. The priority of relevance for these definitions follows this order, with environment variables read only as default values.

`--token USER_TOKEN`
The token from the user profile used to authenticate the user for upload to the ReportPortal instance.

`--url RP_URL`
The URL of the ReportPortal instance where the data should be sent to.

`--project PROJECT_NAME`
Name of the project into which the results should be uploaded.

`--launch-per-plan`
LAUNCH-PER-PLAN mapping, creating one or more launches with no suite structure (launch - test).

`--suite-per-plan`
SUITE-PER-PLAN mapping, creating one launch and continuously uploading suites into it (launch - sute - test).

`--launch LAUNCH_NAME`
Set the launch name, otherwise the tmt plan name is used by default.

`--launch-description DESCRIPTION`
Pass the description for ReportPortal launch (with ‘`suite-per-plan`’) or append the description from the plan summary with additional info (with ‘`launch-per-plan`’).

`--defect-type STATUS_NAME`
Pass the defect type to be used for failed tests. It is defined in the project (e.g. ’Idle’), with ’To Investigate’ used by default.

`--exclude-variables PATTERN`
A regular expression for excluding environment variables from reporting to ReportPortal, using the pattern ’`^TMT_.*`’ by default. Parameters in ReportPortal can display all environment variables, but with ‘`exclude-variables`’ they get filtered out by the pattern to prevent overloading and to preserve the history aggregation for ReportPortal item.

`--launch-rerun`
Rerun the last launch based on its name and unique test paths to create a retry item with a new version per each test. Supported in ’suite-per-plan’ structure only.

`--upload-to-launch LAUNCH_ID`
Pass the launch ID for an additional test/suite upload to an existing launch. ID can be found in the launch URL. To upload specific info into the description see also launch-description.

`--upload-to-suite SUITE_ID`
Pass the suite ID for an additional test upload to a suite within an existing launch. ID can be found in the suite URL.

Where all ‘`token`’, ‘`url`’ and ‘`project`’ are mandatory options in order to establish connection to the instance of ReportPortal and enable reporting there, see FR2. Especially for the needs of these options, the alternative definition via environment variable is very essential.

Then there are flag options ‘`launch-per-plan`’ and ‘`suite-per-plan`’ that define the launch structure and the mapping (FR3, FR4). They are mutually exclusive, and ‘`launch-per-plan`’ is active by default if none is specified.

the ‘`suite-per-plan`’ option is recommended to use with voluntary options ‘`launch`’ and ‘`launch-description`’ to cover the FR7, as there is no relevant data level above plan in tmt structure. Otherwise, by default, the launch is named after the first plan of the run and the description remains empty.

More about the values including the default ones that are to be visualised in the ReportPortal interface, read in the section 4.3.4. And limitation or potential of possible option combinations are elaborated in the next section 4.3.2.

4.3.2 Supported use cases and limitation of option combinations

With the amount of offered option there is a need to limit forbidden combinations and ensure the full potential of supported ones for variety of use cases.

The table 4.1 demonstrates all supported use cases formed by combinations of the plugin options. Most of them are open to options that would affect launch name, launch description, defect type and exclude-variables, which may be either recommended, situational or redundant per given use case.

A The use case 3.4.1(a) representing a simple upload with LAUNCH-PER-PLAN mapping (FR3), open to other options, where the '`launch`' is mostly redundant.

B The use case 3.4.1(b) representing a composite upload with SUITE-PER-PLAN mapping (FR4), open to other options, where the '`launch`' and '`launch-description`' are recommended.

C, D The use case 3.4.1(e) applying the requirement FR15, uploading all tests either directly (C) or in suites per plan (D) to given launch. No new launch is created in the use case thus the launch parameters are not affected by the options.

E The use case 3.4.1(e) applying the requirement FR16, uploading all tests directly to given suite.

F The use case 3.4.1(c) creating a new retry item within existing test items with name-based mapping in the last launch with given name. Supported only with SUITE-PER-PLAN mapping.

G, H The first part of the use case 3.4.1(d), if no 'execute' step is performed before-hand. It applies the requirement FR13 and can be complemented with 'I' below. See the template example in 4.3.3 (a).

I The second part of the use case 3.4.1(d), if the 'discover', 'report' and 'execute' steps were completed beforehand. It covers the requirement FR14 and complements 'G' or 'H' above. It ignores all options and reuses the mapping and the ReportPortal elements based on the UUID identifiers stored by the 'report' step before. It only uploads new logs to given items if the item identifiers are already known. See the template example in 4.3.3 (b).

	`URL, token, project`	`launch-per-plan`	`suite-per-plan`	`launch`	`launch-description`	`defect-type`	`exclude-variables`	`upload-to-launch`	`upload-to-suite`	`launch-rerun`
A	×	×		?	?	?	?			
B	×		×	?	?	?	?			
C	×	×		-	-	?	?	×		-
D	×		×	-	-	?	?	×		-
E	×	-	-	-	-	?	?	-	×	-
F	×		×	?	?	?	?			×
G	×	×		?	?	×	?			
H	×		×	?	?	×	?			
I	×	-	-	-	-	-	-	-	-	-

Table 4.1: the table demonstrates combinations of plugin options per row, where the '×' means the option is applied, the '?' means the option may be applied, and the '-' means that the option is ignored (even when applied) in given use case

Here follows the list of derived limitations.

(a) When one of the options '`URL`', '`token`' and '`project`' is not defined, the report is unsuccessful, therefore *error should be raised.*

(b) When there are both '`launch-per-plan`' and '`suite-per-plan`' defined, an unexpected behaviour may appear, therefore the default value is used and *warning should be logged.*

(c) When '`upload-to-launch`' and '`upload-to-suite`' are defined, an unexpected behaviour may appear, therefore '`upload-to-suite`' is prioritised and *warning should be logged.*

(d) When '`launch-rerun`' together with '`upload-to-launch`' or '`upload-to-suite`' is defined, an unexpected behaviour may appear, therefore '`launch-rerun`' is ignored and *warning should be logged.*

(e) When '`launch-rerun`' and '`launch-per-plan`' is defined, an unexpected behaviour may appear as current version of ReportPortal does not support this functionality, therefore *warning should be logged.*

4.3.3 Design decisions in tmt plugin

To dig into details of the core report upload, as the report plugin is within the run performed once per each plan it makes the mapping LAUNCH-PER-PLAN trivial, creating a new launch with launch - test hierarchy per plan.

On the other side, the LAUNCH-PER-PLAN mapping intends to add several plans from several 'report' steps to one launch. That's why this requires to create a new launch once for the first plan and pass the launch UUID to the following plans so they can be added and the last plan can close the launch. The report upload in this case is structured with launch - suite - test hierarchy

In both cases the launches and items within are created, then filled with corresponding logs and they are closed afterwards to show real time progress grouped by tmt plans. The current implementation of tmt does not allow a mutual concurrence of individual steps for progress update more frequent than per each plan, as the 'report' step is always performed after completed 'execute' step per plan.

The problem of step succession must be considered also for the scenario of uploading an empty report with IDLE state (FR13) and then updating it after execution (FR14).

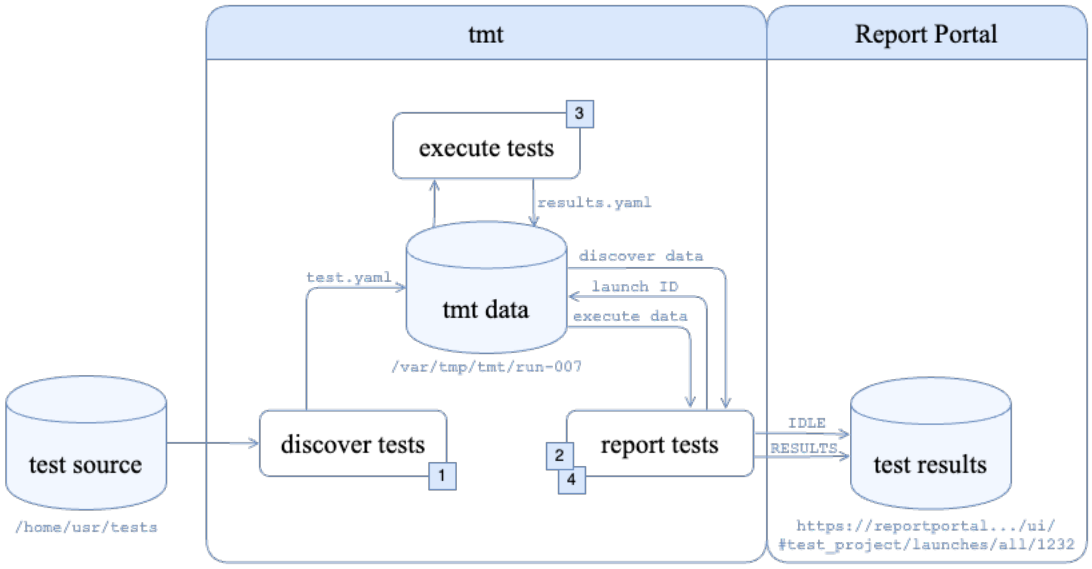


Figure 4.11: An activity diagram that demonstrates an intended communication between tmt steps '**discover**', '**execute**' and '**report**' to cover use case 3.4.1(d) and the requirements FR13 and FR14

The figure 4.11 drafts the idea of data flow for the use case 3.4.1(d), hence as the plugin is called from the 'report' step, this cannot be done in one run unless the tmt step succession is modified. On the other hand, tmt does allow storing data into file structure per run that can be additionally reused. So the main idea of the use case for the purposes of transparent automated testing is to perform it in two commands.

(a) Prepare the report with IDLE state (FR13)
Template example:

```
$ tmt run discover report --how reportportal --defect-type IDLE
```

Here the 'discover' step discovers and stores all the test data within the run. Skipping the 'execute' step, the 'report' step obtains only descriptive data per each test with no results. These data are used to create a launch and prepare the test items (or suites and test items within) with name, attributes and other parameters. The items are labeled as IDLE if opted as a defect type. To allow updating with precise mapping, all item UUIDs must be stored in tmt structure.

(b) Update the report with results (FR14)
Template example:

```
$ tmt run --last --all report --how reportportal --again
```

The other step requires to enable repeating the 'report' step with ability to reuse data achieved with the option `--again`. This time the 'report' step follows the 'execute' step, thus it obtains test results that can upload to ReportPortal. In case, it has the identifiers to ReportPortal items stored, it does not create new but reuse the prepared ones, where it updates the data parameters including test results and appends the logs.

Mapping of additional data to already created items is much more precise with stored identifiers than with default mapping the Report Portal offers for reruns (FR12). Though the implementation is built-in in the API command, it has its limitations and can be used on plans with unique test names only.

Additionally, to upload additional data in new items within an existing launch (FR15, FR16), there is needed the UUID of the target. Though the user knows only the ID found in the URL of launch or suite, this is passed by option and can be used to obtain its UUID via API command.

To sum up the design decisions, there must be check for known identifiers or ReportPortal launch or items that can be reused, either UUIDs stored in the previous 'report' step or IDs passed by user. If there is no existing launch available, new one is created and the UUID is stored, as well for the rest of the items based on the hierarchy opted. All provided data are loaded in the data parameters either from the fmf metadata or directly from the user via plugin options. After logs are uploaded, independent items are closed. Only after all data are uploaded the launch is closed, so the visualisation of launch loading in ReportPortal represents an actual progress in Report Portal.

Additional tmt design decision is to print plugin update in output, on verbose in terminal. This involves printing names of launches, suites or tests to display the progress of report to ReportPortal. And finally the URL is printed per each launch at the end (FR5).

4.3.4 Design decisions in ReportPortal interface

The goal view in ReportPortal is specified by the design decisions that involve mapping data to the parameters of ReportPortal elements that are visualised in the interface.

For **names**, there are used tmt names derived from the paths of plans and tests. For launch, there is prioritized name specified by the user in the option, otherwise plan name is used by default. If the launch name is not defined in combination with `--suite-per-plan` option, there is a first plan name used.

Description of elements is filled with a summary obtained from fmf metadata per plan and per test. Only the description of launch created for `--suite-per-plan` option would remain empty if it is not specified by the option.

For FR8, the **attributes** on the plan level (launch or suite items), are obtained from the plan context, where tmt allows to define the architecture, compose, and many others, including non-defined personalised attributes. The attributes of test items contain all inherited *context attributes* that tmt holds per each test, additionally with *contact* per test. The launch attributes when `--suite-per-plan` is opted, are collected by the intersection of all plans that the launch contains.

The test parameters are filled with environment variables as FR9 states. However, tmt and other tools involved (Testing Farm etc.) may generate the environment variables that are unique by their character (`TMT_TREE=/var/tmp/tmt/run-012`, etc.) and therefore break the history aggregation (FR11) if use case ID is not provided. In case, there is an use case for displaying these variables, they cannot be fully omitted, see the option `--exclude-variables`.

Finally, the use case ID serves for purposes of test identification which corresponds with the tmt ID generated within fmf metadata (FR10).

Chapter 5

Implementation details

The implementation strategy can be divided into two parts, core base and extension features built on it. Simple upload of test results to ReportPortal serve as a proof of concept. Before additional requirements can be delivered, the core base must be verified and then used as a building block for the extension.

This chapter elaborates major details of final implementation of tmt plugin integrating with ReportPortal via its REST API.

Using all the potential of report plugin within tmt and requests of ReportPortal API, it aims to address all the functions that are either required by named use cases and requirements in the section 3.4 or beneficial in testing processes that tmt is or may be used for.

Throughout the process of implementation the plugin, all new features are also manually tested so any partial issues can be addressed immediately and name the resolution.

Finally, the overall implementation is polished with all the functionalities together resulting in the final code implementation attached in the appendix (A.1). Beside the core plugin code, the tmt tool integrates the plugin throughout its implementation in additional files including definition of report schema and documentation specifications. The tmt directly involves also test coverage but it will be examined in the next chapter 6.

5.1 Upload plan to ReportPortal

This core part of the implementation serves for purposes of applying trivial functionality on plugin structure and laying foundation for additional features. Using the classes and functions in the tmt structure along with requests to the ReportPortal REST API, build upon the knowledge from the section provided in appendix (A.2.3).

As elaborated in the section 4.2.1, the report plugin stands on two essential classes, data class and common class. Where the data class defines all supported options and the common class `ReportReportPortal` derived from the class `tmt.steps.report.ReportPlugin` envelopes the main functonality with `go()` function inherited from the class `GuestlessPlugin`.

The core upload of test results to the interface of ReportPortal tool covers the initial connection establishment, and upload of logs in the bare structure of launch-test items, particularly based on direct LAUNCH-PER-PLAN mapping and additional processes and features indicated by the use case 3.4.1 (a), covering the requirements FR1-FR3 and FR5-FR11.

The implementation in this section targets to cover the following command:

```
$ tmt run --all report --how reportportal   --url <REPORTPORTAL_URL>
  --token <USER_TOKEN> --project <PROJECT_NAME> [--launch <LAUNCH_NAME>]
  [--launch-description <DESCRIPTION>] [--exclude-variables <PATTERN>]
```

Where at least the URL and token must be supported to pass via the environment variable, all options can be passed via fmf metadata instead and there is no mapping option needed as a LAUNCH-PER-PLAN mapping only is used in a core implementation.

5.1.1 Establish the connection to ReportPortal instance

Unlike the reportportal library for python plugin, the REST API does not need an initial connection establishments, but it still needs the mandatory data as URL of ReportPortal instance, user token and name of the targeted project that are used for each REST API request. These all are obtained from user via option, which need to be defined in dedicated data class `ReportReportPortalData`.

The option is defined via the function `field(option, metavar, default, help)`, setting the option name, metavar to represent the value, the default value, and help description. The default value is either the environment variable `„TMT_PLUGIN_REPORT_REPORTPORTAL_" + option.upper()` or `None` if the variable is not defined. The 'None' value influences the data typ and therefore requires using a predefined type `Optional[str]`.

The functional part of the plugin is encapsulated within the class `ReportReportPortal` which must be labeled with decorator `@tmt.steps.provides_method(„reportportal")` to be identified as 'reportportal' method. This will guarantee a proper integration with tmt report step so the plugin can be called on '`report --how reportportal`' and cover FR1. This class must involve the definition for class `ReportReportPortalData` and function `go()` which calls an equally named superfunction and includes entire report functionality.

The plugin body starts with option handling. Even if the options `server_url`, `token`, and `project` are allowed to be `None`, they are mandatory, therefore each of them must be handled by raising an error if not provided. For this is purpose, tmt utilities provide `tmt.utils.ReportError(message)` to raise an exception with error message.

Afterwards, token is used to prepare headers and the server url with project name are used to form a link that is throughout the plugin used to establish connection with ReportPortal within every API request and successfully cover FR2.

```

    url = f"{server_url}/api/v1/{project}"
    headers = {"Content-Type": "application/json",
               "Accept": "*/*",
               "Authorization": "Bearer " + token}

```

Figure 5.1: Variables for URL and headers are defined to be further used for API requests to establish connection with ReportPortal, based on the template 4.6 in design chapter.

Finally, a function from tmt utilities is used to start communication with ReportPortal via API requests. Defined variable `session` then provides requests such as GET or POST.

```
with tmt.utils.retry_session() as session:
```

Figure 5.2: Define a session to commence the communication with Report Portal.

5.1.2 Create launch and test items

Before sending the first API request, the necessary data must be collected to fill all launch or item parameters. the options that affect these parameters are `launch`, `launch-description` and `exclude-variables`. They are defined the same way as previous options introduced above. Beyond these, data are obtained from tmt structure based on fmf plan and test metadata.

Only an option `exclude-variables` has a pattern of a regular expression '`^TMT_.*`' as a default value to avoid reporting tmt variables that may break the history aggregation which FR11 requests. In case, an user uses the option but sets it with an empty string instead of pattern, it is assumed that they do not want to use any filter thus a pattern '`$^`' is applied. And it is further used to filter all environment variables by external function. The environments variables are obtained from plan metadata labelled by 'environment'. These must be prepared as a list of dictionaries with key and value of each environment variable to pass it to parameters of ReportPortal test item. Similarly, the attributes are obtained from plan context within fmf metadata.

```
    envar_pattern = self.get("exclude-variables") or "$^"
    env_vars = [{'key': key, 'value': value}
                for key, value in test.environment.items()
                if not re.search(envar_pattern, key)]

    attributes = [{'key': key, 'value': value[0]}
                  for key, value in self.step.plan._fmf_context.items()]
```

Figure 5.3: Constructing list of dictionaries to prepare environment variables and context attributes for report to ReportPortal.

With all data constructed, launch can be created via the API request with prepared url, headers and json data with all paramters made by options and tmt metadata such plan name, plan summary, initial execution time of plan and already explained attributes. From the response, after it is handled for possible errors, a launch UUID is obtained to use it further for report of additional data.

```

response = session.post(
    url=f"{url}/launch",
    headers=headers,
    json={"name": self.data.launch or self.step.plan.name,
          "description": self.step.plan.summary + self.data.launch_description,
          "startTime": self.step.plan.execute.results()[0].start_time,
          "attributes": attributes})
self.handle_response(response)
launch_uuid = yaml_to_dict(response.text).get("id")
assert launch_uuid is not None

```

Figure 5.4: Demonstration of API request creating a launch in ReportPortal, based on 4.7.

To create a test item within the launch, tests cases are processed in a loop to get result data from 'execute' step ('`result in self.step.plan.execute.results()`') a and test data from the 'discover' step ('`test in self.step.plan.discover.tests()`'). This allows an access to the tmt data from fmf test metadata, such as name tmt name for test case, test summary, test ID and additional data that may be possibly useful when displayed in ReportPortal. List of test attributes is based on launch attributes with a contact added per each test, and list of environment variables obtained from fmf metadata per each test as explained in the figure 5.3 into details. Basically these parameters cover requirements FR7 - FR10. After all, type of item is set to test, launch UUID from previous response was inserted and then another UUID for test item is received from the new response to enable logging into the element.

```

    response = session.post(
        url=f"{url}/item",
        headers=headers,
        json={
            "name": test.name,
            "description": test.summary,
            "attributes": item_attributes,
            "parameters": env_vars,
            "testCaseId": test.id or None,
            "codeRef": test.web_link() or None,
            "startTime": self.time(),
            "launchUuid": launch_uuid,
            "type": "step"})
    self.handle_response(response)
    item_uuid = yaml_to_dict(response.text).get("id")
    assert item_uuid is not None

```

Figure 5.5: Demonstration of API request creating a test item in ReportPortal, based on 4.8.

5.1.3 Upload details of test results

With UUID of created test items the test logs obtained from the execution step can be processed and uploaded via request to ReportPortal.

```

for index, log_path in enumerate(result.log):
  log = self.step.plan.execute.read(log_path)

  response = session.post(
      url=f"{url}/log/entry",
      headers=headers,
      json={
          "message": log,
          "itemUuid": item_uuid,
          "launchUuid": launch_uuid,
          "level": level,
          "time": result.end_time})
  self.handle_response(response)

```

Figure 5.6: Demonstration of API request uploading standard log in ReportPortal, based on 4.8.

As framework including BeakerLib may generate multiple logs (output, journal, etc.) and tmt offers filtering based on errors (‘`result.failures(log)`’), the level parameters for ReportPortal log can be used with multiple logging. Failures are filtered from the standard log and uploaded with ERROR level, a full standard log is used for INFO level and rest of the logs for TRACE level. This feature is not necessary but can be beneficial for test result analysis.

5.1.4 Close launch and test items

After all logs with test result details are uploaded, the test results can be assigned to each test item. Each result status is based on the results provided by tmt. In tmt, there are differentiated 5 types of result status, which is ‘PASS’, ‘FAIL’, ‘ERROR’, ‘WARN’ and ‘INFO’. While in ReportPortal there are ‘PASSED’, ‘FAILED’, ‘SKIPPED’ which are relevant to evaluation of test results. Therefore the mapping is done with dictionary as ‘PASS’: ‘PASSED’, ‘INFO’: ‘SKIPPED’ and rest of tmt’s states are assigned to ReportPortal’s ‘FAILED’. the test results are uploaded together at closing the test item. It requires item UUID obtained at its creation for the url.

```

                response = session.put(
                    url=f"{url}/item/{item_uuid}",
                    headers=headers,
                    json={"launchUuid": launch_uuid,
                          "endTime": self.time(),
                          "status": status})
                self.handle_response(response)
                launch_time = result.end_time

```

Figure 5.7: Demonstration of API request closing item in ReportPortal, based on 4.8.

When all test item are closed, the launch gets closed as well, with the stored UUID used as well. As demonstrated at the bottom of 5.8, the launch item at closing returns the URL link to the launch item that is useful to display it in the terminal via tmt standard logging function and cover the requirement FR5.

```

            response = session.put(
                url=f"{url}/launch/{launch_uuid}/finish",
                headers=headers,
                json={"endTime": launch_time})
            self.handle_response(response)

            link = yaml_to_dict(response.text).get("link")
            self.info("url", link, "magenta")

```

Figure 5.8: Demonstration of API request closing launch in ReportPortal, based on 4.7.

In this point the basic upload to ReportPortal is fully functional. Ater closing the launch report steps ends, and the upload is repeated for another plan when the report step is called.

5.2 Grouping several plans to ReportPortal

This section aims to introduce boolean flag options for mapping, `launch-per-plan` option for a core implementation from the previous section and **suite-per-plan** which will extend the core implementation with suite hierarchy.

The main problem is that a launch needs to contain numerous plans therefore numerous 'report' step executions need to approach this launch. It leads to a need of a launch UUID to be stored after launch is created in the first plan. Value for purposes of a report step can be saved within the `ReportReportPortalData` class together with the options as follows.

```
launch_uuid:  Optional[str] = None
```

In the body section, mapping options must be handled so exactly one type is applied. If `suite-per-plan` is active, a new launch is created only if there is no '`launch_uuid`' in report step data within a first plan, and within it suite - test structure is created.

When launch is created it, the only difference in the implementation of its parameters from the `launch-per-plan` mapping is the value for attributes. Launch attributes are composed of intersection of all plan attributes. It is constructed on comparison of each plan attribute list with a lastly composed temporary list, which as at last made into key-value form for ReportPortal. This way, the final list involves only attributes that are truly relevant for each plan. The code algorithm for this process follows in the figure 5.9.

```
merged_plans = [{key: value[0] for key, value in plan._fmf_context.items()}
                for plan in self.step.plan.my_run.plans]
result_dict = merged_plans[0]
for current_plan in merged_plans[1:]:
    tmp_dict = {}
    for key, value in current_plan.items():
        if key in result_dict and result_dict[key] == value:
            tmp_dict[key] = value
    result_dict = tmp_dict
launch_attributes = [{'key': key, 'value': value}
                     for key, value in result_dict.items()]
```

Figure 5.9: Demonstration of the algorithm obtaining intersection attributes of all plans as explained in the text above

Otherwise the launch request is mostly reused from the `launch-per-plan` implementation. Within the launch, there is a suite created as a root item as well as test item above. This suite is filled with tmt plan data for parameters such name, description and attributes.

```
self.info("suite", suite_name, color="cyan")
response = session.post(
    url=f"{self.get_url()}/item",
    headers=self.get_headers(),
    json={"name": self.step.plan.name,
          "description": self.step.plan.summary,
          "attributes": attributes,
          "startTime": self.time(),
          "launchUuid": launch_uuid,
          "type": "suite"})
self.handle_response(response)
suite_uuid = yaml_to_dict(response.text).get("id")
assert suite_uuid is not None
```

Figure 5.10: Demonstration of API request closing launch in ReportPortal, based on 4.8.

The suite UUID obtained from the response is inserted into the link used to create its child test items. For this, the core implementation of test items can be slightly modified to switch between the root item and child item based on the mapping used. It takes only one-line modification within a link as presented below:

```
url=f„self.get_url()/itemf'/suite_uuid' if suite_uuid else ""
```

All the rest remains same until the conclusion, where the launch is closed for `suite-per-plan` option only if it is the last plan being processed. This way, the run process is simulated in the visual interface while all the plans are being executed and continuosly uploaded. After all, the URL of one launch for all plans is reported.

5.3 Support of reruns

ReportPortal build-in function that allows rerun by creating retry items within a test item (FR12) is easily approached within API command for launch creation. There is only added one more parameter in the json data on line 5 of figure 5.4 for activating rerun as below, where `launch_rerun` is assigned with a value from the `launch-rerun` option.

```
„rerun": launch_rerun
```

With name-based mapping this rerun approach is not suitable for plans with repeating names of tests, therefore there is an alternative way to rerun those tests and upload the data, though the logs can be only appended to previous ones or to empty tests instead of adding new retry items. It is done via storing all UUIDs per each launch, suite or test item and using them for a precise mapping when a rerun is requested (FR14).

```

     launch_url: Optional[str] = None
    launch_uuid: Optional[str] = None
    suite_uuid: Optional[str] = None
    test_uuids: dict[int, str] =
        field(default_factory=dict)

```

Figure 5.11: Definition of values within the `ReportReportPortalData` class storing UUIDs of ReportPortal elements.

Unlike the name-based build-in rerun, mapping via stored identifiers can be applied only within the same tmt run to reuse the report data. However, the option **`--again`** must be used to allow repeating the report step and avoid deleting stored data. In that case, when there is in subsequent report step with a launch, suite and test UUIDs already defined, no elements are created again but reused the existing ones based on their UUIDs.

5.4 Report with an idle status

The use case 3.4.1 (d) is based on 'report' step being run twice, which can be currently done only by running the tmt run twice as elaborated in the design section 4.3.3.

To support the first part of use case, the report step must allow reporting even if execution data are empty and upload only data from discover step without logs and results. Though the plugin is targeted further than the neighbouring teams reach, the idle status is no defined value in ReportPortal projects unless the project administrators choose so. That's why this use case should be supported an option that support a wider range of demands thus allows setting any requested value for tests which would be reported as unsuccessful with no result data.

It is defined under '`defect-type`' within `ReportReportPortalData` class, while the class of a functional code section includes a whole function dedicated to this feature. The function demonstrated in the figure 5.12 is responsible for obtaining the locator of requested defect type value, as it is needed to use it for issue report in case of failures.

If none specific defect type is requested, the function returns the locator of a default value for failures, which is '**To investigate**' with a static locator `ti001`) under the defect type group ‘To investigate’. Otherwise it gets the locator via API request and finds it defined under one of the defect type groups ‘To investigate’,‘No defect’,‘System issues’, ‘Automation bugs’ or ‘Product bugs’. If given value is not defined in the project, the error must be raised.

```

    def get_defect_type_locator(self, session: requests.Session,
                                defect_type: Optional[str]) -> str:
        if not defect_type:
            return "ti001"

        response = self.get_rp_api(session, "settings")
        defect_types = yaml_to_dict(response.text).get("subTypes")
        if not defect_types:
            return "ti001"

        groups_to_search = ['TO_INVESTIGATE', 'NO_DEFECT',
                            'SYSTEM_ISSUE', 'AUTOMATION_BUG', 'PRODUCT_BUG']
        for group_name in groups_to_search:
            defect_types_list = defect_types[group_name]
            dt_tmp = [dt['locator'] for dt in defect_types_list
                    if dt['longName'].lower() == defect_type.lower()]
            dt_locator = dt_tmp[0] if dt_tmp else None
            if dt_locator:
                break
        if not dt_locator:
            raise tmt.utils.ReportError(f"Defect type '{defect_type}' "
                        "is not be defined in the~project {self.data.project}")
        self.verbose("defect_type", defect_type, color="cyan", shift=1)
        return str(dt_locator)

```

Figure 5.12: Demonstration of a function that returns a defect type locator to report the issue in case of failed or empty report in ReportPortal, further explained in the text above.

The function is called within a single parameter added in the json data on line 5 of figure 5.8 to report the issue if a failed or empty test is reported.

```
„issue": „issueType": self.get_defect_type_locator(session, defect_type)
```

Though this functionality offers variety of possibilities, it enables reporting an idle status as well, in case a project administrator allowed this feature with a setup of a defect type ‘IDLE’.

Eventually, it allows uploading an empty report with defined status, which can be ‘IDLE’ therefore covers a requirement FR13. Together with an implementation mentioned in the previous part, it allows to update these empty reports with a rerun of the same tmt run via precise mapping covering requirement FR14 and also a full use case they form together.

5.5 Additional upload to the launch

In the long run, the last supported function should upload new additional tests or suites into an existing launch. This can be approached with an implementation of FR14 described in the section 5.3, which supports an upload of tests to a launch with given UUID. If suites or tests are not bound with its UUID in the 'report' step data, a new suite or test is created. Though if a new tmt run is used to upload a report to an existing launch, it has no UUID stored and is expected to be obtained from the user via dedicated option.

In contrast with a launch ID, UUID is no freely available identifier for a common user to know. Therefore the plugin expects an user to pass the launch ID which can be simply found in the launch URL. For this an option `upload-to-launch` is defined within the `ReportReportPortalData` class. Afterwards, the UUID is obtained via API request based on the provided ID, as can be seen in 5.13

```

    if launch_id:
        response = session.get(url=f"{self.get_url()}/launch/{launch_id}",
                               headers=self.get_headers())
        self.handle_response(response)
        launch_uuid = yaml_to_dict(response.text).get("uuid")

```

Figure 5.13: Demonstration of an API request in order to obtain a launch UUID with a launch ID.

With `upload-to-launch` option, launch UUID is obtained and used for all uploads, therefore based on mapping option either suites will be created within the launch, or all tests will be directly inserted into the launch, which does cover the requirement FR15. It is not recommended to run more than one plan with mapping based on `launch-per-plan` option, as this approach supports only uploads to one launch per run.

Analogically, this is done for FR16 with the option `upload-to-suite`. If given the option, no launch or suite is created, but suite UUID is obtained with its ID and reused to upload the launch. As a common practice of plugin based on the requirements builds either launch - test or launch - suite - test structure, and no launch - suite - suite - test structure, then it cannot allow creating more suites within a suite. Therefore mapping options `launch-per-plan` ad `suite-per-plan` are ignored in this case and only a direct upload of tests is supported. Again, it is not recommended to run more than one plan with this option, as this approach supports only uploads to one suite per run.

5.6 Finalization and documentation of tmt plugin

After the main part of plugin within the folder structure dedicated to plugin implementation is done, there are several additional modifications needed to fully integrate the plugin to the tmt structure.

Firstly, for purposes of FR18, there must be a support of `help` argument to print a description of plugin functionality and instructions to the options supported. This is done within the file with plugin implementation. The functionality description is generated from the Python **plugin docstring** within the comment section under the ReportReportPortal class and the **description to individual options** is supported in the function `field()` under the '`help`' parameter.

Additionally, there are other files where the documentation related to plugin can be updated. The most relevant involves the **specification for plugin**, which is displayed in web interface as part of tmt documentation, see [12]. It falls under the report specification in the file further specified in the appendix (A.2.3). The plugin specification is described in YAML with a summary, story, description, link to the source file, and several examples that present a variety of supported use cases in the form of both command and metadata specifications.

Among the other relevant files to the plugin, there is a dedicated file for schema definition for metadata specifying the reportportal plugin options. It is marked up in JSON language and used to verify options with tmt command or metadata.

```

    properties:

      how:
        type: string
        enum:
          - reportportal

      name:
        type: string

      project:
        type: string

      launch-per-plan:
        type: boolean

      suite-per-plan:
        type: boolean

```

Figure 5.14: Demonstration of a part of reportportal plugin schema specifying the option properties.

Chapter 6

Evaluation of the plugin

Beside the implementation, tmt requires a test coverage which verifies the functionality implemented is truly working. the test coverage aims to focus on the functionality of tmt in relation to the plugin as well as towards the targeted functionality, particularly the integration with ReportPortal tool. This can be tested on regular basis, supporting automated testing as well.

Moreover, the requirements definition in the real setting of the work environment is a process based on analysis and gained experience. Though it began with an idea that is elaborated in the chapter 3 under the sections with use cases and requirements. The plans for an initial implementation were sceptical against the possibilities of tools, therefore the implementation was separated into two parts the core one and the extensions as indicated in the previous chapter. After an implementation of each of them, an user feedback was very essential factor, which is further explained and summarizes the evaluation in the last section.

6.1 Test coverage

Based on the segmentation of implementation goals derived from the supported use cases, the test coverage of the plugin is structured into two main parts.

The first part primarily focuses on core functionality related to detailed aspects of a common report upload. In summary, it ensures that all ReportPortal properties including logs and result statuses within a launch-test structure are successfully uploaded and no problem appeared in the communication between tmt and ReportPortal.

The second part has a broader scope, verifying the primary functionality of all features that extend beyond the core report requirements. It involves multiple tmt runs, each addressing different aspects derived from all use cases, besides the core one. This part includes only brief verification of details already covered in the first part with goal to validate all supported use cases.

Encapsulating features related to both tmt and ReportPortal, both of them read standard output based on the server response and verify ReportPortal interface values via the REST API. Also both parts are written in bash with BeakerLib framework, organized within a test that is divided into phases, with each phase mostly corresponding to one test objective. Beside the targeted test phases, there is setup and cleanup phase for an approach to the temporary data that the framework implements.

In addition to the executable bash file, the test includes a data directory containing sample tests with metadata demonstrating both different statuses, with a range of parameters, see 6.1. This directory has tmt initialized (`.fmf` directory) and contains fmf plan metadata to support the plans within tmt run. The plan definition includes several parameters interesting for plugin functionality like summary, context attributes, environment variables and report specifications.

```

/bad:
    summary: Failing test
    contact: tester@redhat.com
    test: echo "Something bad happened!"; false

/good:
    summary: Passing test
    contact: tester_2@redhat.com
    test: echo "Everything's fine!"
    id: 63f26fb7-69c4-4781-a06e-098e2b58129f

/weird:
    summary: an~error encountered
    test: this-is-a-weird-command

```

Figure 6.1: Three versions of simplified tests with metadata, targeting three types of statuses.

Furthermore, the test itself can be executed by tmt, involving a brief fmf file at the same level to support the test automation. It aims to cover requirements FR1 - FR16. This way the test suite goes through all supported scenarios described in the thesis and fully verifies its functionality in each execution.

6.1.1 Test coverage for core functionality

With an intention of a thorough verification, this part is divided into two phases. Starting with a tmt run representing an essential functionality of the plugin, the tmt run executes and reports three tests targeting good, bad and unexpected result, all specified in metadata. The run is performed with default values (LAUNCH-PER-PLAN mapping, metadata values) and shared in first two phases for detailed approach.

The first phase verifies whether all tests were correctly reported from the side of tmt. Reading the the standard output, it uses `grep` to assert launch link FR5 and names of all reported tests to ensure the connection and full communication was successful FR2. It also reads ID and UUID identifiers of launch and test elements to assert they are not empty. Parts of this phase are reused also throughout the test suite for testing extended features as well, as it has an access to names and identifiers which are essential for approaching the launch and all items within via REST API.

The second phase tests this run from the ReportPortal point of view via the **REST API requests**. It obtains a response from the API request and compares it with a static expected data, it is data read from the metadata and data as identifiers printed out in standard output acquired from the previous phase. To illustrate the idea of testing, figure 6.2 provides an example with a **beakerlib assert** function that is used for the comparison.

The function takes comment for log, followed by tested data and expected data to generate a status for log. For searching terms within json response and fmf metadata are used Python tools **jq** and **yq**.

```

    response=$(curl -X GET "$URL/api/v1/$PROJECT/launch/uuid/$launch_uuid" \
                    -H  "accept: */*" -H  "Authorization: bearer $TOKEN")

    rlAssertEquals "Assert the~URL ID of launch is correct" \
                   "$(echo $response | jq -r '.id')"  "$launch_id"
    rlAssertEquals "Assert the~name of launch is correct"   \
                   "$(echo $response | jq -r '.name')"  "$launch_name"
    rlAssertEquals "Assert the~status of launch is correct" \
                   "$(echo $response | jq -r '.status')"  "$launch_status"

```

Figure 6.2: Demonstration of a few asserts testing launch properties URL ID, name and status based on the response of the API request.

It starts with a launch request to validate the launch ID, launch name, launch status and launch description. Additionally, it goes through all metadata and compares it with launch attributes.

Another step is based on a test request, where the test goes in a loop through all three tests and verifies details such UUID, name, or result status. It also checks all environment variables in parameters individually, including a negative assert for filtered environment variables that should be omitted by the default value of `exclude-variables` option.

Finalizing with a log request, where each log is processed and asserted on its contents and assigned level in ReportPortal.

In summary, with these first two phases passing without problem, it validates the support of requirements FR1 - FR10 with an exception of the first requirement which is mostly implicit based on the valid use of the plugin, and the requirements defining mapping on request (no default mapping) which are covered in the second part of test coverage described below.

6.1.2 Test coverage for advanced use cases

Targeting verification of all supported features in addition to the core report upload, there are several phases to cover each of them. In this part, each phase involves at least one separate tmt run to simulate an use case. And each of them includes brief verification to cover key properties of requirements FR3 - FR4 and FR11 - FR16. Having a similar approach to testing the core functionality, they sum up it all in one phase per each obejctive while they do not focus on details already covered in the first part of previous subsection.

To name them in the order of requirements, an essential phase to begin with is one that supplements the coverage of core functionality with LAUNCH-PER-PLAN mapping on demand. It tests the functionality of plugin with a mapping requested on demand by `launch-per-plan` option and verifies if no suite structure is created. In addition to the core test coverage, this ensures that all expectations were met for the requirement FR3.

Another phase is analogical to the previous one with pure SUITE-PER-PLAN mapping on demand. It verifies the presence of launch-suite-test structure, but also replicates detailed testing examined in the previous subsection to ensure no problem appeared with a change of launch structure and mapping. Therefore, it meets the requirement FR4.

There is also a need to validate the integrity of historical data aggregation, as requested in FR11. This validation process involves a sequence of two tmt runs. During the first run, a launch is created containing items that serve as direct predecessors to the items in the launch created in the subsequent run. The validation is conducted through an API request that retrieves a history of depth 2. This API response is then utilized to assert the identifiers of the items, ensuring that the historical aggregation remains intact.

This phase is extended by another test run using the `exclude-variables` option, targeting FR9, to demonstrate valid functionality in relation to test parameters, test case ID and history aggregation in ReportPortal. It adopts a similiar approach, but involves tmt environment variables that tend to be unique per run and therefore may disrupt the history aggregation. It verifies the preserved history aggregation of test items with a test case ID present and identifies any interruptions otherwise.

Next phase targets the ReportPortal built-in feature of launch rerun, which is objective of FR12 covered by two runs and launch mapped on name basis. There is an initial run that creates a launch and provides its identifiers for verification of rerun feature in the second run. It asserts the report is mapped to the previous launch with same name, and its test items contain retry items. The access is mostly enabled via API request that lists all test items and its data filtered by any property, particularly a launch ID.

Furthermore, the rerun functionality is also tested using a different approach in another test phase, aligning with the requirements specified in FR14. This involves a single tmt run, initially executed to perform a standard upload. Subsequently, the run is rerun with the option **`--last`** and option **`--again`** specifically on the 'report' step to append logs based on the UUID identifiers of ReportPortal elements. During this process, assertions are used to ensure that identifiers of reported items are correctly mapped and that the logs are accurately appended.

In another test phase, the UUID-based rerun functionality on a top of an implementation for a requirement FR13 is used to simulate the scenario of intagrating the plugin in Shared OS Testing Infrastructure introduced as motivation example in section 3.1. Within tmt run 'discover' step and 'report' step are performed opting `defect-type` to label the empty report in ReportPortal as 'idle'. Afterwards the same tmt run is rerun and updates the report. While the API request helps to verify the validity of the process through defect types assigned to each test item and logs uploaded to the corresponding items

Finally, there are three test phases implemented using the options `upload-to-launch` for both launch-suite-test' and launch-test' structures, and `upload-to-suite` specifically for the 'launch-suite-test' structure, in alignment with the requirements outlined in FR15 and FR16. These phases involve creating a launch with an initial run, followed by an additional run to supplement new ReportPortal items. Validation of the upload process is performed using the same API request to retrieve all items with details per launch. This validation includes comparisons of identifiers and parsing of content to ensure accurate upload.

In conclusion, this section comprehensively addressed and tested all specified requirements through the implementation of various test phases. Each test phase, including either core or key properties of extended functionality, was meticulously explained and executed to validate the functionality of the system according to FR1 - FR16 requirements. Whereas the requirement FR17 is implicit given this test suite and FR18 is covered by tmt functionality that is tested as well, once the plugin description is defined in corresponding functions as stated in 5.6.

Ultimately, all proposed requirements were met, with functional requirements fully satisfied as detailed above and non-functional requirements adequately covered. This achievement is implicitly demonstrated through the implementation discussed within the implementation, chapter 5.

Importantly, all test phases resulted in successful outcomes, confirming the system's adherence to requirements and demonstrating the achievement of all intended functionalities of the plugin as well as the purposes of the thesis. This robust evaluation of the thesis' implementation underscores the system's adherence to requirements and successful realization of its intended functionalities.

6.2 User feedback

In real-world scenarios involving the developer, client, and end-user, initial requirements and goal setting are often distorted, insufficient and unclear, leaving room to adjustments during the implementation process. Especially, if all three authorities of the scenario converge as the assignment evolves in a 'community-driven' approach to improve the tools used via community discussions. Therefore, just as the design and implementation, which are summarized earlier in this thesis for clarity, the evaluation of the implementation also occurred in two steps, with user feedback being a crucial factor after each of them.

This way, the evaluation of the implementation was conducted in two steps as well, with user feedback playing a pivotal role. Especially, after the core implementation based on initial requirements that were barely ambitious. With a few concepts as plans or 'nice-to-have-features' to be further evaluated based on a profound experience of core report implementation.

The evaluation process involved meetings with key users transitioning to these tools, as well as representatives of current and potential users. The survey was based on a sample of colleagues who participated in meetings and those who voluntarily contributed feedback through a dedicated chat channel for tmt, GitHub issues related to tmt, or any direct communication with one of the representatives.

This approach was possible due to a direct contribution to tmt tool living in GitHub repository, which is newly released on monthly basis.

Here are the main points summarized from the first session after all members had a opportunity to test the plugin with core report functionality, uploading test results, logs and other properties to ReportPortal within launch - test structure, covered in section 5.1.

(aa) **launch - suite - test hierarchy**
Transitioning exclusively to a direct hierarchy of the launch - suite structure was deemed impractical and unrealistic based on feedback from users accustomed to a hierarchical structure with three levels in past approaches. This feedback was particularly targeted at addressing the needs of errata testing, which involves grouping results from multiple plans based on different approaches, architectures, or components to validate specific components or features. To accommodate a broader scope of testing requirements, the addition of options for switching the hierarchy and mapping was identified as a viable solution.

(ab) **update of idle tests via rerun**
The initial plan to generate an empty report by triggering the report' step from the discover' step in one run was considered redundant due to the significant alterations required to tmt functionality and the need to allow for repeating the report

step. Instead, the problem was simplified by utilizing options `--last` and id RUN-ID to enable reruns, prompting exploration of methods to repeat the step. This challenge was addressed by the timely implementation of the `--again` option, inspired by a similar option `--force`, with the key distinction of preserving data before repeating the step.

(ac) **defect types**
The plan to report empty tests with an idle status became partially irrelevant due to evolving processes and the expanding scope of tmt. It became challenging to anticipate the general impact and necessity of this flag, especially as new flags emerged. Therefore, it was reconsidered to allow users to define any supported value within the project, keeping the possibilities open even if the concept of reporting idle tests is no longer viable.

(ad) **UUID-mapped rerun**
There appeared an interest for rerun functionality. While the name-based rerun feature provided by ReportPortal generates nicely structured retry items, it proved insufficient for covering plans where test cases are repeated with changing conditions. Therefore, only UUID-based mapping would be accepted, even if test results are rewritten and logs appended. This approach could be leveraged from the point referenced in (ab). Although retry functionality was considered a „nice to have“ feature and was not prioritized for broader support, it was still planned for implementation due to its simplicity, with the understanding that it would not be extensively supported.

(ae) **appending the description**
The original plan to add files was deemed redundant and ineffective for storage, as the testing farm already provided URLs to the test artifacts. Consequently, attachments were replaced with the ability to append a string such as URL directly within the launch description.

To evaluate the initial phase of analysis, design, implementation, and feedback, it was essential to explore the capabilities of the ReportPortal REST API and establish realistic expectations for integrating tmt with ReportPortal.

In the gathered feedback, key users highlighted specific challenges and requirements that influenced subsequent refinements and adjustments to optimize the system's usability and alignment with user expectations. Building on former plans, there were provided valuable insights laying the groundwork for subsequent development iterations. Eventually, the feedback resulted in finalizing all requirements planned for the integration or tmt and ReportPortal and finally implementing this comprehensive solution.

The community's feedback was again solicited to evaluate and assess the effectiveness of these enhancements.

(ba) **all options available via environment variables**
As the functionality of tmt is in Testing Farm run via metadata specifications and environment variables, it lacks the functionality of plugin options unless they can be approached via environment variables.

(bb) **option for REST API version**
Given the availability of two different versions of the ReportPortal API offering syn-

chronous (v1) and limited asynchronous (v2) approaches, there was a desire to leverage the asynchronous functionality. As a solution, an option was proposed to allow switching between these two versions as needed.

(bc) **an option targeted to link to artifacts**
A link to artifacts within a launch description was found insufficient for scenarios involving merged launches or additional uploads into launches. Therefore, a new option is intended to be dedicated specifically for this purpose, allowing users to insert the link into the description at each level within the ReportPortal launch.

(bd) **need of an instance upgrade**
Execution of complex test suites often results in timeouts, preventing the completion of uploads. This issue was identified and resolved within ReportPortal, and the solution is available in a new version of the software.

Despite receiving feedback that highlighted certain deficiencies, these were promptly addressed. The issues mentioned in (ba) were included in the thesis's implementation, while others were deemed out of scope for this project. Ultimately, the plugin's implementation was deemed a success, effectively meeting all requested testing needs and replacing previous systems. This achievement instilled a deep sense of gratitude for the functionality delivered and fostered ambitions for ongoing improvements. Through iterative evaluation steps and active engagement with users, the implemented solution was refined to enhance user experience and align with practical needs and expectations in real-world usage scenarios. The incorporation of user feedback proved instrumental in driving meaningful improvements and shaping the final successful implementation of the system.

Chapter 7

Conclusion

This thesis embarked on an extensive exploration of testing terminology, processes, and tool integration within Red Hat's testing ecosystem. The primary objective was to facilitate the seamless integration of the tmt tool with ReportPortal, driven by the overarching goal of establishing a Shared OS Testing Infrastructure that addressed the complexities of testing diverse software components within Red Hat Enterprise Linux (RHEL).

Throughout the study, alternative solutions were considered, yet none proved sufficient to surpass the final design of the tmt plugin integrated via the REST API of ReportPortal. The thesis analyzed the functionalities of existing and forthcoming tools, focusing particularly on tmt's capabilities and the possibilities the ReportPortal REST API offers.

The implementation phase was structured into two essential components. Firstly, the core report functionality addressed the need for reporting to ReportPortal. However, this implementation fell short of fulfilling all requirements for automated testing within the tmt and ReportPortal. Following thorough testing and successful execution of the core implementation, extensive feedback was solicited and incorporated into plugin's design and implementation. Secondly, advanced features were introduced, carefully designed, and implemented to cover a broad range of scenarios, leveraging the full potential of integrated tmt and ReportPortal API. Each new feature was incrementally built upon previously implemented components, manually tested, and validated with full test coverage to ensure that all requirements were met without disrupting existing functionalities.

Feedback from stakeholders and users was instrumental in shaping the direction of this project, driving meaningful improvements, and fostering ambitions for ongoing enhancements within Red Hat's testing infrastructure. The positive response and constructive suggestions received underscored the value of collaborative efforts in advancing software testing practices and achieving higher standards of quality assurance.

Looking ahead, the integration of tmt with ReportPortal represented a significant step towards the modernization and optimization of testing processes within Red Hat. As Red Hat transitioned from legacy systems to innovative solutions, the commitment remained strong to refining and expanding testing capabilities, ultimately delivering improved outcomes for RHEL teams and contributing to the evolution of software testing practices in the open-source community.

In conclusion, this thesis underscored the importance of effective test management, automation, and collaboration in ensuring the quality, stability, and functionality of software products. By integrating tmt with ReportPortal and leveraging user feedback, a solid foundation was laid for future advancements in testing methodologies and infrastructure within Red Hat.

Bibliography

[1] ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary. *ISO/IEC/IEEE 24765:2017(E)*. 2017, p. 1–541, [cit. 2024-04-10]. DOI: 10.1109/IEEESTD.2017.8016712.

[2] ISO/IEC/IEEE International Standard - Software and systems engineering –Software testing –Part 1:General concepts. *ISO/IEC/IEEE 29119-1:2022(E)*. 2022, p. 1–60, [cit. 2024-04-10]. DOI: 10.1109/IEEESTD.2022.9698145.

[3] Bravin, A. *Top 11 test reporting tools to supercharge your QA process* [online]. [cit. 2024-04-14]. Available at: https://zebrunner.com/blog-posts/top-11-test-reporting-tools-to-supercharge-your-qa-process.

[4] Dubaj, O. *Systém pro správu výsledků testů doplňující nástroj tmt*. Brno, CZ, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/23921/.

[5] Open Source Contributors. *Nitrate* [online]. [cit. 2024-04-18]. Available at: https://nitrate.readthedocs.io.

[6] Red Hat Contributors. *BaseOS QE Project Page* [online]. [cit. 2024-04-16]. Internal document accessible upon request.

[7] Red Hat Contributors. *fmf* [online]. [cit. 2024-04-21]. Available at: https://fmf.readthedocs.io.

[8] Red Hat Contributors. *GitHub repository: tmt* [online]. Available at: https://github.com/teemtee/tmt.

[9] Red Hat Contributors. *RHEL Development Guide* [online]. [cit. 2024-04-18]. Internal document accessible upon request.

[10] Red Hat Contributors. *Shared OS Testing Infrastructures* [online]. [cit. 2024-04-17]. Internal document accessible upon request.

[11] Red Hat Contributors. *Testing Tools* [online]. [cit. 2024-04-16]. Internal document accessible upon request.

[12] Red Hat Contributors. *tmt* [online]. [cit. 2024-04-30]. Available at: https://tmt.readthedocs.io.

[13] ReportPortal Contributors. *What is ReportPortal?* [online]. [cit. 2024-04-25]. Available at: https://reportportal.io/docs/.

Appendix A

Contents of included storage media

(A.1) `plugin_implementation`
A source code of tmt tool with implemented plugin, documentation and test coverage.[8]

(A.1.1) `README.md`

(A.1.2) `tmt`

```
├── spec
│   └── plans
│       └── report.fmf
├── tests
│   └── report
│       └── reportportal
│           ├── data
│           │   ├── plan.fmf
│           │   └── test.fmf
│           ├── main.fmf
│           └── test.sh   <-
└── tmt
    └── schemas
    │   └── report
    │       └── reportportal.yaml
    └── steps
        └── report
            └── reportportal.py   <-
```

(A.2) `alternative_plugin_implementations/`
Alternative or partial implementations of the plugin for reference.

(A.2.1) `via_junit_xml_import/` [NOT AUTHORED BY ME]

(A.2.2) `via_api_library/`

(A.2.3) `via_rest_api_core_only/`

(A.3) `output.txt`
A log from an executed test coverage with detailed test results.

(A.4) `docs`
Documentation including PDF file and LaTeX source code.