BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

APPLICATION FOR OPENSHIFT PLAFORM FOR TESTING OF STUDENTS PROJECTS

APLIKACE PLATFORMY OPENSHIFT PRO TESTOVÁNÍ STUDENTSKÝCH PROJEKTŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. MARIÁN ORSZÁGH**
AUTOR PRÁCE

SUPERVISOR — **ALEŠ SMRČKA, Ing., Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2019/2020

Master's Thesis Specification

Student: **Országh Marián, Bc.**
Programme: Information Technology Field of study: Information Technology Security
Title: **Application for OpenShift Plaform for Testing of Students Projects**
Category: Software analysis and testing
Assignment:

1. Study Openshift container platfrom. Study the methods of requirement-based testing. Get familiar with system test automation.
2. Design a framework for testing of student projects. Focus on scalability of the system when running multiple tasks of the same type. The designed framework should allow debugging and reporting of test results.
3. Implement the framework as a web service. Implement the text user interface (e.g. by utilising ncurses library).
4. Desing and implement automated test suite for verification of basic functionality. Demonstrate the framework on testing selected student projects.

Recommended literature:

- L.H. Tahat ; B. Vaysburg ; B. Korel ; A.J. Bader. Requirement-based automated black-box test generation. 2001. In Proceeding COMPSAC '01 Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development. Pages 489-495
- Homepage of OpenShift. https://www.openshift.com/

Requirements for the semestral defence:

- The first two items.

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: November 1, 2019
Submission deadline: June 3, 2020
Approval date: May 19, 2020

Abstract

The aim of this thesis is to design a service for automated requirements-based testing of student programming assignments, and thereafter implement this service using the OpenShift, Python and Git technologies. By creating such a service, a foundation is set for a unified testing process, which includes executing the test suites in separate Linux containers. Such a process is intended for simplification of the grading by teachers and teacher assistants, and at the same time improvement of student's performance in such tasks.

This Master's thesis explains the basics of software testing, while focusing on requirements-based testing specifically. Furthermore, it provides insight into the container technology, as well as how these themes are applied in the project's design, and how they are reflected in the service's requirements. Afterwards, the implementation details of the service are put under examination in order to provide a reference material for any future extensions of the project.

The implemented service allows for basic operations, including testing of multiple student projects in separate containers concurrently, creating a containerized debugging session, or automatically building a testing container image for a given assignment.

Abstrakt

Cieľom tejto práce je navrhnúť službu pre automatizované testovanie študentských programovacích projektov na základe požiadaviek a následne implementovať túto službu za použitia technológií OpenShift, Python a Git. Vytvorenie takejto služby stavia základ pre zjednotený proces testovania študentských projektov, ktorý zahŕňa spúšťanie testovacích sád v oddelených Linuxových kontajneroch. Vylepšený testovací proces má viesť ku zjednodušeniu známkovania vyučujúcimi a taktiež zlepšeniu výsledkov študentov pri týchto úlohách.

Táto diplomová práca vysvetľuje základy testovania softvéru, pričom sa sústredí na testovanie založené na požiadavkách, poskytuje náhľad do technológie kontajnerov a objasňuje, ako boli tieto témy zahrnuté pri návrhu služby a taktiež, ako sa ich použitie odrazilo na požiadavkách na ňu. Okrem toho je implementácia tejto služby podrobená detailnej analýze, ktorá má slúžiť ako referenčný materiál pre jej akékoľvek budúce rozšírenia.

Implementovaná služba je schopná vykonávať základné operácie, zahŕňajúce paralelné testovanie študentských projektov v oddelených kontajneroch, vytvorenie kontajnerizovaného ladiaceho prostredia, alebo automatické zostavenie kontajnerového obrazu pre konkrétne zadanie.

Keywords

testing, requirements-based testing, containers, openshift, kubernetes, fitest, git, python

Kľúčové slová

testovanie, testovanie založené na požiadavkách, kontajnery, openshift, kubernetes, fitest, git, python

Reference

ORSZÁGH, Marián. *Application for OpenShift Plaform for Testing of Students Projects.* Brno, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Aleš Smrčka, Ing., Ph.D.

Rozšírený abstrakt

Zámerom tejto práce je navrhnúť a implementovať službu slúžiacu na automatizované testovanie študentských projektov v prostredí platformy OpenShift. Cieľom takejto služby je modernizácia procesu testovania a známkovania projektov, z pohľadu opravujúcich a v budúcnosti aj zlepšenia prístupnosti testov študentom.

Vo svojom úvode táto práca popisuje princípy *testovania založenom na požiadavkách*, do ktorého spadá aj väčšina testovania študentských projektov. Znamená to, že pre jednotlivé projekty je na začiatku definovaný súbor požiadaviek (zadanie), ktoré musí dané riešenie splniť a následne je odovzdané riešenie, v rámci hodnotenia a známkovania porovnané s pôvodnými požiadavkami. Práca sa zaoberá základmi testovania softvéru a termínmi dôležitými pre pochopenie metodík, ktoré tvoria teoretický základ pre túto službu. Ďalej objasňuje využitie automatizovaného testovania založeného na požiadavkoch v kontexte opravovania študentských projektov a vysvetľuje dôležité termíny ako *testovacie prostredia* a *test reporting*.

Po objasnení základov terminológie testovania softvéru sa práca venuje existujúcim riešeniam na problém automatizovaného testovania študentských projektov. Popisuje platformy ako *The Marmoset Project*, *Mimir Classroom*, alebo *Codeboard* a vysvetľuje ich výhody, ale zároveň aj nedostatky, pre ktoré by nemohli byť použité ako alternatívy pre službu, ktorú táto práca navrhuje. Okrem špecifických nedostatkov bola najväčším problémom týchto platforiem ich limitovanosť v kontexte konfigurácie testovacích prostredí.

Ďalej sa práca zaoberá rôznymi spôsobmi, ako vytvárať testovacie prostredia a vysvetľuje, prečo boli pre túto aplikáciu zvolené *kontajnery*. Nakoniec sa čitateľ dočíta o kontajnerizačných platformách *Kubernetes* a *OpenShift*, technológiách *Git* a *Python*, ako spolu súvisia a prečo boli zvolené na implemetáciu tejto služby. Obsahuje tiež podrobnejší opis dôležitých termínov z dokumentácií platforiem Kuberentes a Openshift, ako *Build* alebo *Job*, ktoré boli aplikované pri návrhu a implementácii testovacej služby. Tento opis má nad rámec dokumentácie slúžiť aj ako referenčný materiál pre možné budúce rozšírenia navrhovanej testovacej služby.

Po vysvetlení teórie testovania a technickej terminógie sa práca sústredí na návrh služby pre automatizované testovanie študentských projektov, pomenovanej FITest. Obsahuje cieľové požiadavky na túto službu, popisuje, aké aktivity je možné pomocou nej vykonávať a tiež, čo služba neautomatizuje a dáva za úlohou jej používateľovi. Ďalej opisuje architektúru služby FITest, rozdelenie úloh medzi klienta, server, OpenShift a Git a približuje, ako sú tieto prvky spojené a vzájomne využívané. Na základe popísanej architektúry ďalej vysvetľuje návrh jednotlivých procesov, ako napríklad spustenie testov alebo vytvorenie prostredia pre daný projekt.

Tento návrh sa premieta do realizácie samotnej služby, na základe čoho sú dokumentované jednotlivé prvky a procesy, implementované jazykom Python. V rámci implementácie serveru služby sú opísané aj súčasti konfigurácie objektov pre platformu OpenShift, ako dané objekty fungujú a ako serverová aplikácia využíva funkcionalitu tejto platformy na spúšťanie automatizovaných testov v izolovaných kontajneroch. Dôležitou súčasťou serveru je tiež jej webové *API*, pomocou ktorého je možné ovládať službu vzdialene, cez sieť. Taktiež stručne opisuje implementáciu klientskej terminálovej aplikácie, ktorá slúži na ovládanie funkcionality pomocou spomenutého API. Spôsob návrhu a implementácie služby dovoľuje, aby bola v budúcnosti jednoducho, modulárne rozšíriteľná.

Vo svojom závere práca sumarizuje a hodnotí, ako bola služba testovaná a do akej miery boli splnené definované požiadavky. Taktiež pripúšťa, že vzhľadom na použité technológie

a postupy môžu nastať komplikácie vo fungovaní tejto služby. Na tieto problémy tiež ponúka riešenia a vysvetľuje ako im predísť.

Služba samotná je vo stave, v ktorom dokáže vykonávať úkony potrebné na automatizované testovanie študentských projektov jednotlivými opravujúcimi. Vzhľadom na rozsah hypotetickej univerzálnej testovacej služby táto práca nerieši všetky možné problémy a neimplementuje rôzne špeciálne prípady jej využitia. Na druhej strane vytvára solídny základ pre širokú variabilitu budúcich rozšírení, ktoré by z nej vytvorili univerzálnu testovaciu službu využiteľnú ako opravujúcimi, tak aj študentmi.

Application for OpenShift Plaform for Testing of Students Projects

Declaration

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

.....
Marián Országh
June 3, 2020

Acknowledgements

I would like to express my gratitude to Aleš Smrčka, for supervising this work, as well as providing valuable insights and offering professional guidance throughout its creation. Furthermore, I want to thank my family and friends for standing beside me and offering their support and motivation throughout my work. Thank you all.

Contents

Chapter 1

Introduction

Testing of students' programming assignments at the Faculty of Information Technology has often been a target of criticism by many students attending various courses, where often the majority of points, obtainable throughout the semester comes from these projects. Plenty of students complain about the fact that the tests are frequently executed against their solutions completely automatically, without regard for the nature of the encountered errors, which in the end leads to a small amount of points acquired for the projects. Unfortunately, the quantity of student submissions is usually quite high and the resources necessary to manually test every failed submission are simply too demanding for the teachers, or assistants, who are tasked with the grading of the assignments.

The aim of this thesis is to create a service, that provides an environment for remote, automated execution of such tests and gives the means to distribute the same system configuration to the students, who can test their implementations this way.

This Master's thesis provides a detailed insight into the development and functionality of the FITest service, a service implemented to automate the testing of students' programming assignments. In the beginning, the basic concepts surrounding requirements-based testing are explained, due to the fact, that the current process of testing of student's projects is based on this methodology. Afterwards, the existing solutions, with similar goals as the proposed service were presented, along with their shortcomings in the context of the problem. Next, the technologies proposed to be used in implementation were described, including important terms related to them. In Chapter 4, the requirements for the service are defined, together with the design of the service, which would fulfill these requirements. In the following chapter, a detailed explanation of how this design was used in the implementation of the service is provided. Lastly, the implementation of the service is evaluated, and compared to the original requirements, including any possible issues that may come up during usage of the service.

Chapter 2

Requirements-based Testing

This chapter shortly summarizes all the necessary concepts behind requirements-based software testing, vital for understanding the subsequent chapters of this documentation. It describes underlying concepts behind requirements-based testing, followed by terms such as *test fixtures*, or *test reporting*. Finally it gives an insight into the application of these approaches in testing of students' programming assignments.

2.1 Software Testing

The role of software testing grows hand-in-hand with the fashion of computerization, as the products we use are expected to be performing perfectly during their entire lifespan. To cite the best-seller of software testing, *Art Of Software Testing* [24], a book which has been first published in 1979 and is kept up-to-date with current trends of development, software testing is *„a process, or a series of processes, designed to make sure computer code does what it was designed to do and, conversely, that it does not do anything unintended“*. This definition conveys the idea, that a software product needs to perform, as it was designed, while avoiding any ambiguous, or unaccounted situation, in which it might fail.

2.1.1 Software Testing Life-Cycle

As opposed to common misconception, software testing is not a single activity, but a sequence of interconnected steps, which are (or should be) executed alongside the development of the actual software they are designed to test. In general, these steps are described as **Software Testing Life-Cycle** (STLC) [13][15][50], for which the exact definition is not completely decided upon, but for the scope of this thesis can be summarized as the following steps:

1. requirements specification,
2. test planning/design,
3. test development,
4. test execution,
5. test result and test coverage analysis,
6. conclusion and closure.

The method applied in this thesis focuses mainly on steps 1, 2 and 5, in the method known as **Requirements-based testing**, which is looked upon more thoroughly in Section 2.2.

2.1.2 Models of Software Testing

The approaches to software testing are very closely tied to the model of software development, be it the software life-cycle model, or the software behaviour model [16]. This section is going to focus primarily on creating tests based on the **Software development life-cycle** (or shortly SDLC) models.

SDLC can be considered a framework, which defines individual steps carried out by a development team, from a product's initial design, through implementation and testing to maintenance of the final product, presented to its customers [16][19][53]. There are multiple methods, varying in ways to execute and evaluate each step, among the most widely-known being

- the waterfall model, being the foundation for the rest of the methodologies,
- the V-model (see Figure 2.1),
- the incremental model,
- the spiral model and lastly,
- the agile models.

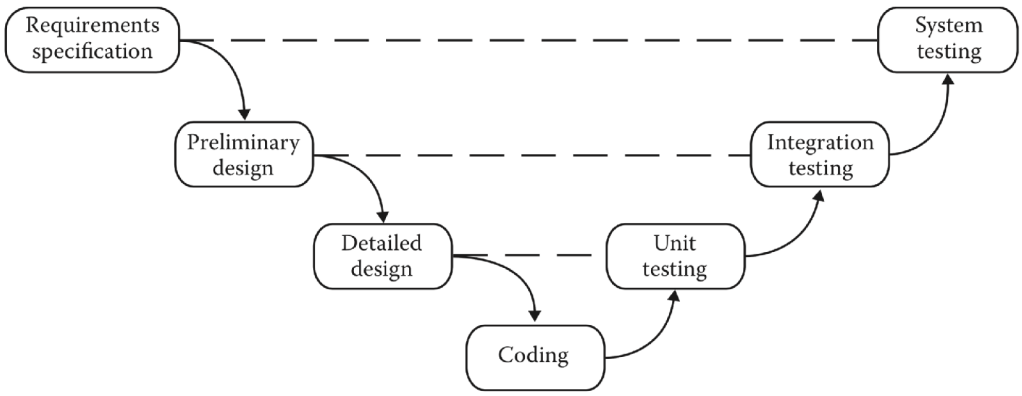


Figure 2.1: Diagram of the V-model.
Source: Software Testing: A Craftsman's Approach [16]

The **V-model** can be used to present the basic notion of the SDLC models to the reader, as it is quite straightforward and simple to understand. As seen in Figure 2.1, the entire process starts with the specification of requirements (most important phase for requirements-based testing). Following a top-down[1] approach the system is designed and implemented. Once a functioning SUT is created, bottom-up[2] testing is carried

[1] gradual decomposition of the whole system into smaller functional components
[2] joining elementary, smaller components to create larger ones

out, verifying that all components are performing correctly (*Unit testing*) and if their incorporation into bigger system segments does not introduce any bugs to the system (*Integration testing*). In the concluding stage, the *System testing* phase, the product's behaviour as a whole is tested and laid against the original specification of requirements from the first step. The dashed lines in the diagram represent relation between the design and testing (Unit testing tests the Detailed design of the components, etc.).

2.2 Requirements-based Testing Methodology

As already mentioned in the STLC's list of steps in Section 2.1, the primary focus of Requirements-based testing (RBT, not to be confused with *Risk-based testing*), or, alternatively, *Black-box testing* (as it is the outside behaviour being tested, instead of evaluating the code modules), comes from having a **clear set of definite specifications** for the tested software product before ever laying a finger on the actual code of the project. This procedure helps to shape the scope of the project early on and to avoid complications and unnecessary alterations in later stages, while focusing on validity of the specifications throughout the entire process [3], [52].

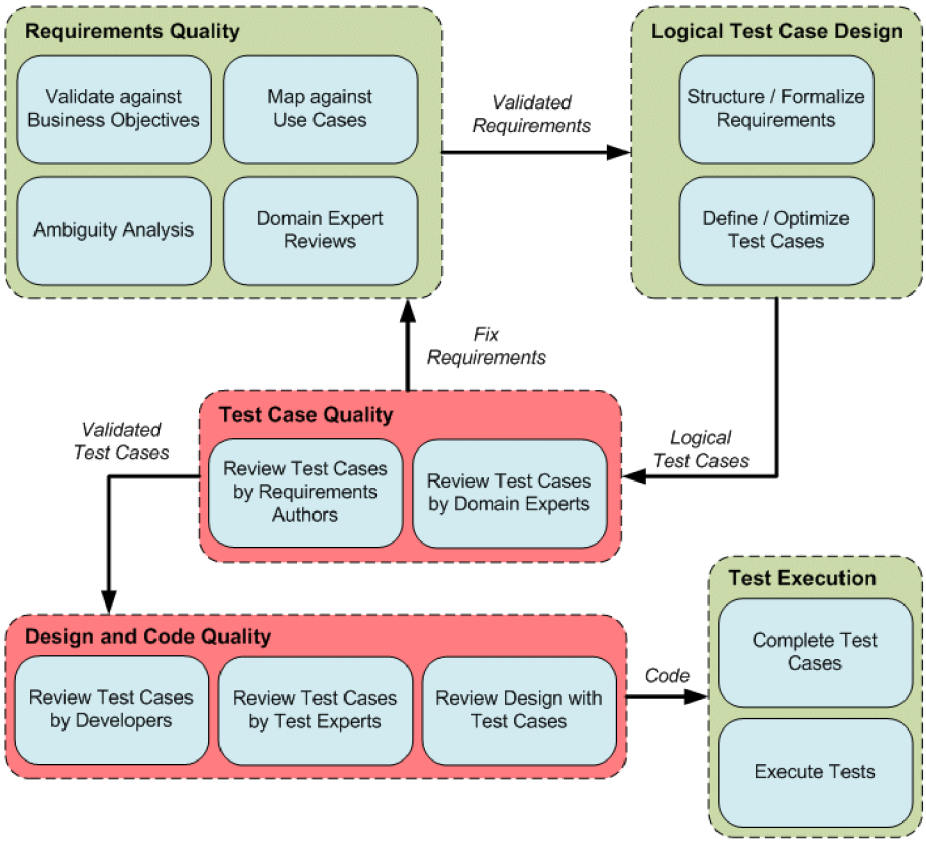


Figure 2.2: Requirements-based testing methodology diagram.
Source: Requirements-based Testing Process in Practice [50]

Multiple guidelines and sources [3], [17] define the Requirement-based testing methodology as a 12-step process (see Figure 2.2), which initiates with defining, examining and tailoring the project's requirements, avoiding any possible ambiguities. Following these steps, the specifications are reviewed by domain experts and users, and a structured, formal definition of requirements is formed and inspected for logical inconsistencies.

Having a consistent formal set of requirements gives way to actual design of the test cases, that are reviewed by the requirements' authors, users and/or domain experts and finally, the developers. The test cases are thereafter used in design review [3], followed by code review [4]. Finally, the implementation of the test cases' code is put into progress along with execution against the code and comparing results to expectations.

In practice, to reduce the required resources, the test cases are usually designed to cover multiple requirements. To put this into an example, let's consider an excerpt from a hypothetical command-line application's requirements:

```
...
- The output value is an integer value, greater than zero
- The output value is enclosed in matching brackets
...
```

An example test case covering both of these requirements could at first check, if the output value is a string with a left bracket as its first and a right bracket as its last character. Afterwards, ignoring the bracket characters it would check if the value is a string convertible to a positive integer (i.e. exclusively containing numerical characters). If all these conditions are met, the test case passes, otherwise it is marked a failure.

2.2.1 The Advantages of Applying Requirements-based Testing

Skoković and Rakić-Skoković showed in their research [50], where they compared the development process of two different web-portals (connected to the same databases, with the same available technology and domain knowledge), each of which applied RBT methodology in different stages of the project.

The team working on the first portal employed parts of RBT during the implementation phase, after it was clear they are likely going to exceed the project's planned deadline and possibly the budget too. Compared to the second portal's development, which finished even before the planned deadline and which applied these methodologies early on, the first team encountered more severe issues and was forced to spend more time on re-specifying the requirements [5].

2.3 Test Reporting

Test reporting is the most important phase of the testing process as it provides the developer, or the tester team with valuable information if the tested product is either working as it is supposed to or alternatively, that a bug has occurred and a test or multiple tests are failing, meaning there's a fragment of code that needs to be fixed.

[3] confirming that the initial requirements are satisfied by the final design

[4] the process of investigating if all parts of the code are working as expected

[5] The presented data showed that erroneous requirements caused circa 30% of all issues (the rest being human factor and technology obstacles) for Portal 1 as opposed to the 2nd one's 10%.

A good report should contain information, that makes the bug removal as easy as possible. This data includes, but is not limited to:

- Success status of each test, conventionally stated as *Pass* (the test has finished its execution successfully), *Fail* (the test's success conditions were not met), or *Error* (an unexpected error, unrelated to the testing scope occurred). If the test has not passed correctly, additional information is often included, such as a description of the error, stack trace[6], the position of the error call in the code, etc.
- Percentage and count of Passed / Failed / Error tests.
- The amount of time spent on the execution of the tests.
- The scope of the executed tests.
- Testing environment (system configuration, package versions, etc.).

2.4 Test Fixtures

The last concept in this chapter regarding software testing theory are test fixtures (sometimes also referred to as test context [65]). According to xUnit Patterns, [66] a test fixture is *„all the things we need to have in place in order to run a test and expect a particular outcome“*. The same source provides a definition of test context as *„everything a system under test needs to have in place in order to exercise it for the purpose of verifying its behavior“* [65].

In other words, they are environments used for performing a test, or a test suite[7] that are usually set up with identical configuration each time to avoid test failures that do not originate from the SUT, but from the varying environment (e.g. two systems with different versions of packages), or any other fault that is not associated with the testing process.

2.4.1 Standard Fixture

A *Standard Fixture* is a pattern most related to Requirements-based testing, as it involves designing an environment, that is reusable by **multiple** (or all, in the best instance) test cases, as opposed to a *Minimal Fixture*, which aims at creating a separate, smallest possible test fixture suited for a single test case. The aim is to form such an environment upfront, before the actual implementation of the tests, instead of attempting to fit one to the tests, after they have been constructed. [21]

Shared Fixture

A *Shared Fixture* (a.k.a. Shared Context, Reused Fixture) is a Standard Fixture, that is **reused** for multiple tests (possibly entire test suite). This approach is applicable in cases, when it can be very time consuming to build a *Fresh Fixture*[8]. [21]

[6] sequence of called functions that resulted in the error, or failure

[7] a set of common, related test cases used to test a single SUT

[8] Fresh Fixture is rebuilt for every test case, for its own use and is torn down after the test case has finished executing

This pattern is especially useful in the design of this thesis, as restarting a container for every single test case (even though they are quite fast, compared to VMs, explained more in Section 3.2) would still result in a hefty amount of time spent just on *Set-Up*[9] and *Tear-Down*[10] phases of the testing process, for each test case.

Prebuilt Fixture

Prebuilt Fixtures are, in their essence, Shared Fixtures with the additional characteristic of being prepared and built before the execution of a test suite. This is appropriate in cases, where assembling a fixture is a very resource (time, finances, etc.) demanding process.

The primary idea is for the Prebuilt Fixture to exclude the Set-Up phase from every instance of the testing process (see Figure 2.3), which results in the conservation of the aforementioned resources. [21]

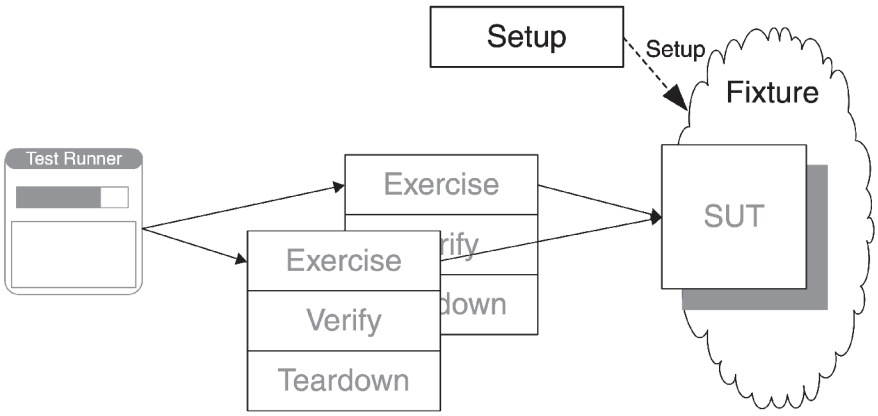


Figure 2.3: Diagram illustrating the approach of a Prebuilt Fixture
Source: xUnit Test Patterns [21]

2.5 Automated Testing of Student Projects

All of the aforementioned principles are commonly used in the of testing programming assignments developed by students and are generally used to verify the student's technical skills acquired from attending different courses, however, the role of test fixtures is often very vague. Following sections explain the application of the principles in the testing process of the students' projects and most importantly, how a more advanced application of test fixtures can improve the practice of automated testing of such assignments.

[9] creation of a testing environment
[10] deletion and clean up of a testing environment

2.5.1 Requirements-based Testing of Course Assignments

The programming assignments this thesis targets usually test a small area of the student's knowledge, whether it is the ability to write a relatively short Python script utilizing a certain module, or a C program that expects the student to be familiar with, let's say, DHCP (Dynamic Host Configuration Protocol). Such assignments have predefined requirements for the final product concerning the output, modules/libraries the students can use and the way the application should be compiled and executed.

The major difference from the traditional requirements-based testing lies in the execution of the tests. In the majority of the time, students are provided with no tests and are left to implement all tests themselves in accordance with their understanding of the requirements (i.e. the assignment). In some cases, this approach leads to poorly graded assignments, where a low score originates not from the student's lack of skill, but instead from minor issues such as wrongly formatted output (a single extra white-space character), wrong file-path for compiler target, etc. The points a solution acquires are frequently calculated by comparing the expected and actual tests output character by character, utilizing an automated script. This approach is commonly used simply because the quantity of student's projects is too large compared to the number of teachers, or assistants who are tasked to grade the assignments.

In the scope of requirement-based testing, this thesis aims to reduce the occurrence of similar cases by giving the teachers a way to easily distribute preliminary, or complete test suites, that are constructed from the requirements, allowing students to shift their focus from testing their projects' output formatting to improving their overall solution's quality (be it the quality of the code, or the implemented algorithm). In the case a test suite cannot be distributed for various reasons, students can be also provided with a test fixture lacking the actual tests, but encompassing the environment, where the tests are going to be executed.

It's very important to mention that executing tests for these student projects is not a common practice in terms of software engineering. The main difference comes from the fact, that the RBT methodology involves running tests against a solution implemented by a development team. In the case of student projects, the main aim is to test **multiple** and **different** implementations of the same project, described by identical requirements, and then testing and evaluating these implementations separately (as illustrated by Figure 2.4).

2.5.2 Reporting of Students' Results

In this thesis, the value of the reported information varies depending on the receiving party. The instructor tasked with the process of grading all the submissions is much more interested in general information, such as how many tests have failed for each student, or the average fail count for the whole set of students (generic Pass / Fail / Error statistics). At the same time, a student is likely going to be interested more in the information regarding their specific implementation, such as what caused a test to fail (usually found in stack trace information), or why their program or script couldn't be launched at all (e.g. a compiler error).

However, this task is dependant on the person, or people responsible for the implementation of the specific test suites. The verbosity of the tests depends solely on their consideration and is not being solved completely by the service proposed by this thesis, aside from a basic level of reporting. The service itself aims at merely delivering the final test

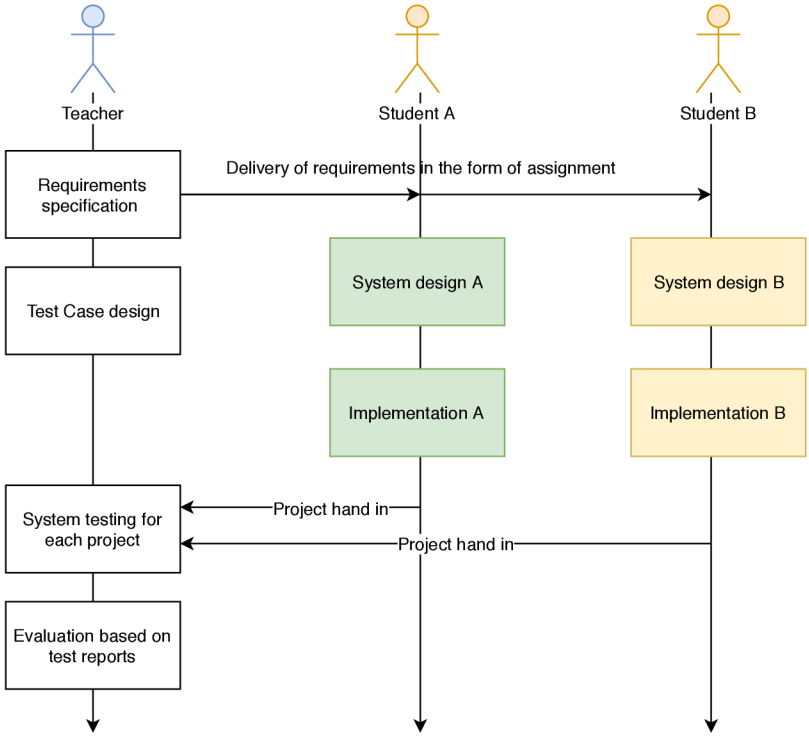


Figure 2.4: Diagram illustrating current process of a programming assignment implementation and evaluation.

results, from the test fixtures to the teacher, or the assistant tasked with the testing, or the evaluation of given assignments.

2.5.3 Test Fixtures in Relation with Students' Solutions

The application of fixtures in this thesis consists of creating an isolated environment with all the necessary and specific dependencies in order to test (and possibly grade) every student's project in the exact same manner automatically while saving resources for both the students and the teachers. Besides, students can use these fixtures locally, on their systems at any time, to develop their projects while avoiding issues with different versions of various packages and having to modify their own systems.

To summarize the intent to use the xUnit-defined patterns (Section 2.4) in designing service for testing student assignments automatically, the fixtures can be applied as follows:

- All tests for a given assignment are executed inside the same *Standard Fixture*.
- Every test suite execution (testing of a single implementation) is wrapped inside a *Shared Fixture*.
- To save resources, a *Prebuilt Fixture* is created in advance and re-set for every test suite.

The specific technical parallels applied in this project, associated with these relations, are explained more thoroughly in Subsection 3.3.2.

Fixtures also pose as a security feature, in the sense of protection from unintentionally (or worse, intentionally) malicious code, by isolating the SUTs to their own environment,

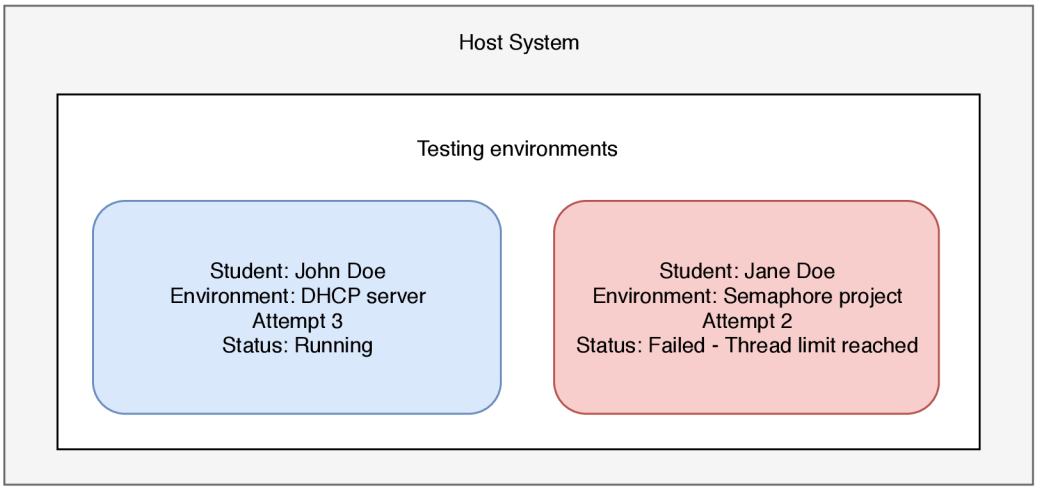


Figure 2.5: Simple illustration of test fixtures application to course assignments.

meaning that if, for example, the SUT doesn't handle its memory allocation well, instead of using up all allocable memory on the host system while executing the test suite, a fixture runs out of its allocated resources and is restarted, or terminated. Figure 2.5 illustrates this design by showing two testing environments, running tests while being isolated from each other and from the host system.

Students often utilize shared servers for the testing of programs, which are frequently unstable in this way, with no regards to other students. This behaviour sometimes leads to unavailability of the servers, making it impossible for other students to use them for other activities. In such regard, the proposed service might help with offloading the servers and at the same time, for the students to not be forced to test their projects on them.

Chapter 3

Existing Solutions and Applied Technologies

In the beginning of this chapter the reader is provided information about existing submission and evaluation systems for student programming assignments, along with their advantages and disadvantages. Afterwards, in a similar fashion, various ways to create and manage testing environments (fixtures) are depicted, along with their contribution to this thesis. The last part of the chapter explains the crucial terms related to OpenShift Container Platform.

3.1 Existing Applications for Automatic Evaluation and Grading of Programming Assignments

This section covers multiple services that focus on automatic evaluation of programming assignments, using requirements-based testing and explains why these services are not applicable to the issue this thesis is addressing.

3.1.1 The Marmoset Project

The Marmoset Project, developed at the University of Maryland (originally as a PhD. thesis in 2006) is „*a system for handling student programming project submission, testing and code review*“[64]. The platform gives the students a chance to test their solutions before submitting it for grading. A student is assigned a fixed number (usually 2, or 3) of „submission tokens“, that refresh in periods of 24 hours. This approach of so-called „release tests“ is supposed to increase the students' activity earlier before the deadline and also improve their software and test development skills while giving the instructors an insight on what the students are struggling with the most [51].

The main issue with the project is the fact that even though it's released under an open-source licence, its official git repository[1] is at the moment in an archived state and the GitHub repository[2] shows no activity since 2015. In addition, the documentation for the code-base is barely existent, mostly addressing issues of compiling and running the already existing codebase (e.g. configuring a build server) with no mentions on how

[1] Archived repository available at `http://code.google.com/p/marmoset`.
[2] Repository available at `https://github.com/paulproteus/marmoset`.

to alter and use the code. Lastly, a lot of references on the project's landing page are unavailable, and/or provide no informational value.

3.1.2 Mimir Classroom

Another service aiming to automate and improve the grading of programming assignments is Mimir Classroom [23]. Mimir Classroom is an online web-based application providing environment for developing (within its own IDE[3]) and evaluating programming assignments while incorporating a wide range of useful instruments, such as online code review, or group project support in a clean and intuitive user interface.

Its usefulness is however limited by the extent of possible customization applicable to a project. When creating a course the instructor is able to choose the target programming language or framework, but no other options to configure the system are accessible, narrowing the possible spectrum of testable projects to simpler instances, that don't require any additional, or custom modules. Another issue with Mimir Classroom is the pricing of the service, 25 USD per semester for each student [4]. To consider an example, the Introduction to Programming Systems is an entry-level course taken by freshmen students in their first semester. In the winter semester of the 2019/2020 academic year, 744 students enrolled in the course (including inactive, and/or dropped off students). This means that with no discounts, the first semester would cost the department over 18,000 USD, which isn't a suitable long-term option.

3.1.3 Codeboard

Initially a research project at ETH Zurich [5], Codeboard's goal is to provide tool for Computer Science teachers that allows them to create and distribute programming assignments in class easily [6]. The application's functionality is very similar to the one of Mimir Classroom (3.1.2), with the additional support for different Learning Management Systems, such as Moodle, or Coursera and, unlike Mimir Classroom, is free to use. In addition, the application is distributed as an open-source project, with all of its source code available on GitHub[6].

Unfortunately its abilities are again narrowed down by the lack of options to configure the system, inside which the test are executed. As stated in the configurations in the GitHub repository, each of the programming languages has a fixed container environment. Another issue is the public repository, which reports the last activity to have occurred in 2016. This might mean that even though the application is still up and running, it's development and maintenance have stopped for the moment.

[3] Integrated Development Environment, an application providing an integrated set of tools used for software development

[4] according to the official pricing, available at https://www.mimirhq.com/classroom/pricing

[5] Eidgenössische Technische Hochschule (ETH) Zürich - Swiss Federal Institute of Technology in Zurich

[6] Repository available at https://github.com/codeboardio.

3.2 Various Technologies Supporting Test Fixtures

Currently, there are multiple approaches a test fixture can be implemented, ranging from creating local copies of the SUT's files to virtualizing entire operating systems during a test suite's run. The following sections explain in more detail different ways of creating test fixtures, which are applicable to this thesis, along with their advantages and disadvantages.

3.2.1 File System Environments

The first category to be mentioned are environments based solely on containing run-time libraries, or modules in a specific location in the file system.

The first notable examples are **Virtual Environments**. A virtual environment is a tool often used with the Python programming language. According to the official documentation [30], a virtual environment is *„a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages“*. This approach is often utilized during development, or testing of Python applications, or scripts. A virtual environment is easily created and using it is quite intuitive, and allows the user to create a special setting for an application that might, for instance, require a specific version of a certain module.

Another example is the **Node Package Manager** [25], available with the popular JavaScript run-time environment *Node.js*. When installing a package for a Node application, the implicit installation path is the working directory containing the application files. This way every application's module library is contained inside its own workspace, avoiding issues with conflicting module versions. The biggest disadvantage of the Node Package Manager comes from the fact, that it does not have an implicit, global package library and relies on installing all packages locally, for every project. This often leads to unnecessary redundancy from the point of the development work, as many Node.js applications utilize the majority of common basic libraries, which can be quite heavy on the side of disk space.

These tools still provide a very versatile and easy-to-use way to create isolated environments for development work. However, the fact that both are very different in their implementation and control and that they are aimed directly at their respective programming languages make them invalid candidates for a system proposed by this thesis, that allows running and isolating projects written in any programming language. Their second disadvantage is that they only isolate the code modules, without restricting the application's access to the host system in any way.

3.2.2 Virtualization

Virtualization (sometimes also referred to as *full virtualization*) is an approach that allows creating full-featured instances of guest operating systems inside a host system, which have access to the underlying hardware with the help of a *hypervisor*. A hypervisor is a process that serves the purpose of controlling the activity of guest systems, their access to the physical hardware (by which it also removes the necessity of modification of the guest system) and isolating them from the host system and from each other (see Figure 3.1) [2].

The biggest advantage of virtualization is its independence on the host system, hence a virtual machine can be migrated between different systems, or servers very easily. Another

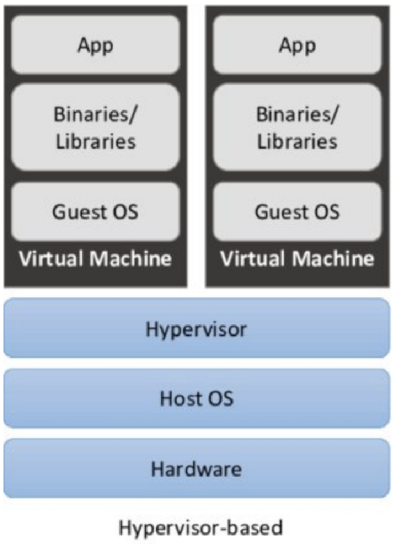


Figure 3.1: Diagram illustrating the Hypervisor-based system virtualization.
Source: A Comparative Taxonomy and Survey of Public Cloud Infrastructure Vendors [49]

benefit is the complete isolation, meaning that a process cannot access any data outside the guest system (e.g. in case of a system error or malicious code).

Unfortunately, this complete isolation comes at a cost - virtualizing a full-featured operating system requires a lot of system resources and setting up a new instance of the system takes a considerable amount of time. In the scope of this thesis this approach is quite impractical, and even though the programming assignments often require a relatively small portion of system resources for their execution, running a virtual machine for their testing would result in a very inconvenient trade-off between the degree of isolation and the necessary performance of the host system. In addition, as a test fixture requires to be identical for all test executions, the resulting amount of time spent on setting up a new virtual machine for every assignment would be too great to be overshadowed by possible advantages. [2][48]

3.2.3 Containers

The basic concept Linux containers builds upon what is known as *Linux Control Groups* (shortly *cgroups*) and *namespaces*. Cgroups and namespaces provide mechanisms for aggregating, or isolating system processes with different behaviour into separate groups. Their primary task in a Linux system is resource tracking (task of cgroups), and the ability to divide processes and their children into the hierarchical trees, creating isolated environments for each of them (taken care of by namespaces).

Each cgroup can be assigned to a specified *subsystem*, which represents a system resource. A process, along with its children can be explicitly assigned to that group (or left in the implicit root cgroup, which stands at the top of the cgroup hierarchy). Cgroups can be furthermore used for prioritization (prioritizing the access of different groups to subsystems) and even accounting, which allows its users to monitor the activity of each group and how much of the assigned resources are being used by them. [20][31]

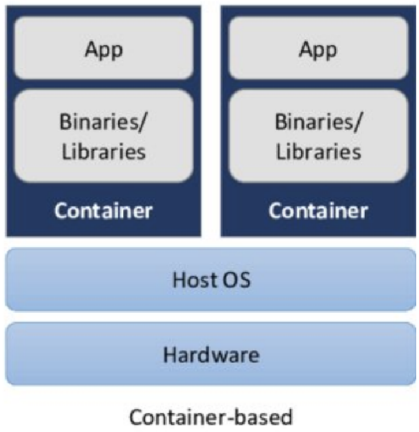


Figure 3.2: Diagram illustrating the Container-based system virtualization.
Source: A Comparative Taxonomy and Survey of Public Cloud Infrastructure Vendors [49]

The ability to isolate process groups in this manner allowed for the creation of **Linux Containers**. A Linux container can be defined as a group of processes running on the host system, isolated from the rest of the system's processes, including its own PID (process ID) namespace, binary files, and libraries, its own file-system, user namespace, network interface, etc. While containers seem like a separate operating system functioning completely on their own (to a process running inside a container there is almost no way of telling it is in reality containerized), in fact, they use the host system's kernel's services and, if necessary, libraries, and/or binaries to function and communicate with the hardware (see Figure 3.2). [44][47][27]

Containers give their users, who can range anywhere from software developers to security engineers, a way to quickly pack their applications and deploy them elsewhere while retaining all the vital dependencies without the need to install more packages on the host system (with the exception for a container management service). Containers fit the aim of this thesis the best, as they are the right compromise between the level of provided isolation and their performance - they are much faster in the means of setting up clean environments than virtual machines while at the same time provide a greater degree of isolation than file system environments.

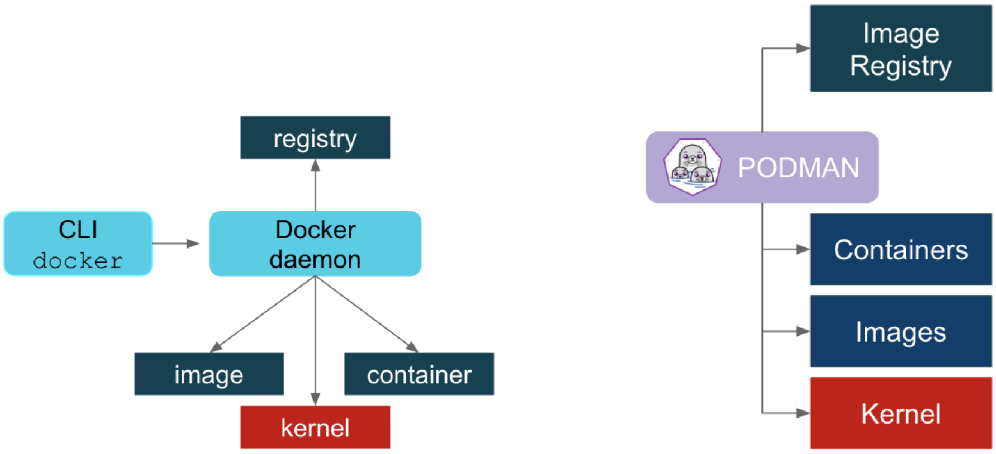
There are many different approaches to building and managing containers and the rest of this section describes the most popular ones, along with their advantages, disadvantages and their possible assets to this thesis.

Docker Community Edition

Docker (or Docker *Community Edition* (CE), since a rebranding in 2017 [11]) is an open-source platform used to develop, build, share and manage containerized applications. Docker CE is a tool leveraging the functionality of the **Docker Engine**, a client-server application that controls the container building and running operations [10]) to create stable Linux containers.

The central component of its architecture is the **Docker Daemon**, which has the task of interconnecting the **Image Registry**[7], the Docker Engine, and the **Docker Client**,

[7] storage service containing container images



(a) Docker (b) Podman

Figure 3.3: Comparison of Docker's and Podman's components' connection.
Source: Podman and Buildah for Docker users [14]

a command-line interface application used to administrate Docker. The Community Edition is aimed at developers and small teams experimenting with the container technology [7].

Docker also provides a service called *Docker Swarm*, which is a platform built on Docker, used for container orchestration[8], and gives a built-in functionality for cluster management. Nevertheless, they present Kubernetes (see the following Section) as their primary orchestration instrument [9], meaning that the focus of this thesis can be shifted towards Kubernetes.

For the sake of the proposed service, using Docker, or Docker Compose (a tool for creating multi-container applications using a single configuration file) would require more convoluted design, as they don't provide functionality for simple scaling of applications.

On the other hand, Docker poses as a good option for a student, who wants to test a single-container application on their machine in order to avoid any system-wise, or package-wise issues that may occur in the final evaluation of the project.

Podman

Podman takes a somewhat different approach to the same issue as Docker. Similarly, it is a tool used for managing containers and container images, which has the same range of capabilities as Docker (except for Swarm mode; Podman doesn't natively possess instruments for orchestration) and even uses the same set of commands for the command-line interface. The difference, however, is their architecture, as Podman doesn't implement any kind of Daemon service and interacts with the components using the command-line interface directly (see Figure 3.3), avoiding various single point of failure problems. [54][14]

In the scope of this thesis, the advantages and the disadvantages of Podman are the same as in the case of Docker and its bare functionality would not be sufficient for implementing

[8] managing containerized applications and service in one, or multiple clusters

the platform as a whole. However, Podman's functionality can be be used for building container images and pushing them to a registry, during the development of the service.

Kubernetes

Kubernetes[9] (*K8s*) is, according to it's documentation, an *„open-source system for automating deployment, scaling and management of containerized applications“* [62], developed by Google. Kubernetes provides utilities for managing containerized applications in a production environment, including, but not limited to tasks such as load balancing, storage management, automated rollouts and rollbacks and secret management. [63]

With these aspects in mind, Kubernetes comes off as a strong candidate for the implementation of the proposed testing service, as it gives ways for managing test fixtures (image, or containers) and more importantly easy scalability of the environments, i.e. easily creating a testing container for a single student's implementation of the given assignment. Furthermore, in the long run, it also provides a way of having the testing service spread across multiple servers as a cluster, to enhance its performance.

OpenShift

OpenShift is a container orchestration platform, which builds upon Kubernetes' functionality to provide mechanisms for Continuous Integration, Delivery and Deployment[10]. Tomasz Cholewa, a cloud-native architect, summarizes the main differences between Kubernetes and OpenShift in his article [5], out of which two give OpenShift an upper hand in terms of assets to the implementation of this thesis:

- The first advantage is a result of the way OpenShift manages images - with something called *Image Stream*, its user can easily change the image used in a project simply by setting its identifying tag in the configuration file. This removes the necessity of changing the configuration of the project every time the requirements for the assignment's environment changes (e.g. a new, more stable version of a certain library is available, or a fix related to a testing script is necessary).
- The second major asset of OpenShift is its web-based user interface, which makes it very easy to manage all the projects and configurations, and even gives its user an easy way to browse through containers' log files and consoles, which is very useful in terms of test execution and reporting.

The following section gives a more in-depth description of what OpenShift is, and more importantly, how it works, what are the operational concepts, and how they relate to the task this thesis aims to achieve.

[9] originating from Greek, meaning governor, helmsman or pilot [62]

[10] conventional practices used in agile software development, automating the steps from the integration of the code to delivering the product to its customers

3.3 OpenShift Container Platform

OpenShift Container Platform is a PaaS[11], developed as the flagship business product of the American multinational open-source software company Red Hat, with its latest release being OpenShift 4.

The popularity of the OpenShift Container Platform emanates from its ability to scale the clients' applications simply and effectively. The installation and deployment size of the platform can range from a small installation on a user's own system (using a deployment called MiniShift, explained in detail in Subsection 3.3.1) to a large public computational cluster provided as a paid service.

The following subsections offer a more detailed insight into how OpenShift works, paraphrasing the most important information from its official documentation [42].

3.3.1 OKD - The Community Way

OKD (The Origin Community Distribution) is a community-driven alternative of OpenShift. The implementation of this project aims at creating a service deployable into OKD for two major reasons:

1. OKD doesn't require a paid subscription, and even though it is stripped of the official technical support and relies on the community's help, the scope of implementation required by the design of this thesis, it doesn't pose an issue, as the documentation is sufficient.

2. OKD can be run in CentOS, unlike OpenShift, which requires to be installed on Red Hat Enterprise Linux (and once again, needs an active paid subscription).

Aside from the lack of official support, OKD is by no means limited in its functionality and offers the same features as its monetized counterpart. [40]

As of writing of this thesis, the current version of OKD (3.11) is behind OpenShift's (4.2), which, however, is not a drawback for the design of this project, and might be even advantageous during the implementation of Pod's resources limitations (see Subsection 3.3.3).

MiniShift

MiniShift is a **containerized version of OKD**, which allows for testing the cluster environment on a single computer, no matter its operating system [41]. Its asset to this project is its usefulness for testing and developing the containers and configurations that are to be deployed in the OKD cluster, used to test the student's projects.

This provides a very easy way to start, restart and even shut down the cluster whenever necessary for any kind of administrative, or debugging processes. Furthermore, it allows for a much larger amount of control over the system, as opposed to running it on a remote system, or having the cluster provisioned by a third party.

[11] Platform as a Service (typically provided by a certain provider) allows its customers to create specific environments for the applications conveniently and easily

3.3.2 Kubernetes' Concepts

Subsection 3.2.3 gives an introduction into what Kubernetes is and what purpose it serves and establishes why it became a standard for container orchestration. Here, its operational concepts are explained more deeply, with the aim of clarifying the underlying architecture and operational concepts of OpenShift.

The following text is a paraphrase of the official Kubernetes documentation [56], referencing each corresponding chapter directly, for the sake of the reader's convenience in case a further explanation to the concepts is necessary.

Nodes

The first term to understand in relation to Kubernetes' architecture is **node**. A node is an instance of Kubernetes running on a IaaS[12] platform, or on a physical, or a virtual server, providing all the necessary services. There are two major types of nodes in the Kubernetes architecture, namely

- **Worker** nodes, running and administrating the containers and
- **Master** nodes, which control and manage the Worker nodes and their workload.

Usually, there are multiple Worker nodes controlled by a single Master node, which also requires much more resources to supervise the function of the Worker nodes. [43]

Pods

Pods are the basic deployment[13] units used in Kubernetes. Their purpose is to encapsulate one, or multiple Linux containers inside their own environment, creating a basic execution unit in a Kubernetes cluster. This unit, which can be thought of as a computer system running container(s) implementing a single application, has its own IP address, storage, resource management, etc. [43][61]

Using Kubernetes, these Pods can be scaled up easily (i.e. increasing the number of instances of the same Pod) with the goal of enhancing the overall performance of the application.

Images

To create containers, their underlying system needs to be prepared in advance. Kubernetes takes advantage of Docker's format of container images. A Docker image is composed of multiple layers, each of them adding content to the previous (installing packages, creating files, or users, etc.), creating a final, desired system, on top of which, the uppermost, Container Layer carries out all the functionality (see Figure 3.4). [8]

As illustrated by Figure 3.4, an image is composed of a *Base Image* (in this specific case it's Ubuntu 15.04), supplementary *Image Layers* created during an image build process (explained in the following section) and finally a *Container Layer*, which is the only one denoted as an R/W-capable layer.

[12] Infrastructure as a Service platforms provide virtual servers/machines with specific system attribute, e.g. Amazon Web Services[1] or Microsoft Azure [22]

[13] *„A Deployment runs multiple replicas of an application and automatically replaces any instances that fail or become unresponsive“* - Kubernetes documentation [12]

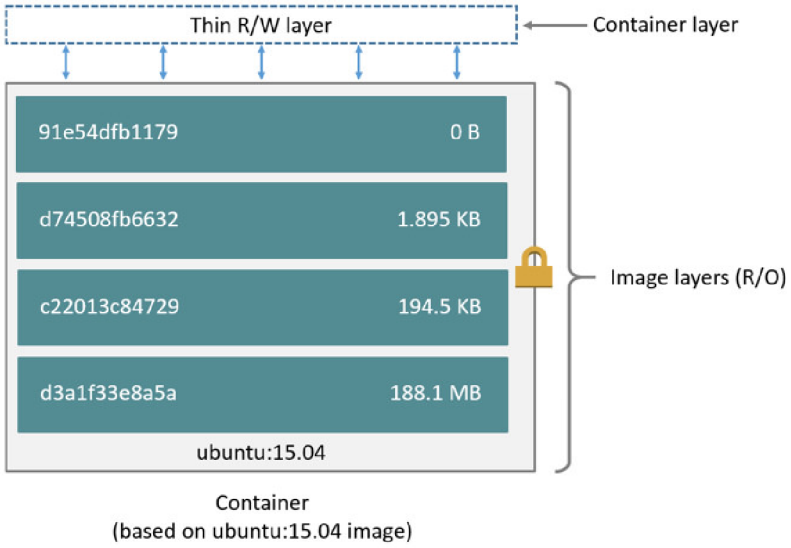


Figure 3.4: Layering of a Docker image.
Source: About images, containers, and storage drivers [8]

Dockerfile

A Dockerfile is a set of instructions for Docker meant for building Docker images. These instructions wrap command-line calls that set up the image, layer by layer (each Dockerfile instruction creates a separate image layer), producing the desired final setting.

```
1  FROM ubuntu
2  RUN useradd testuser
3  USER testuser
4  CMD echo "This is an example."
```

Listing 1: An example of Dockerfile syntax.

A Dockerfile uses its own specific syntax for the commands. For example, by using Dockerfile defined in Listing 1 the following image is created:

1. The `FROM ubuntu` command specifies the base image for the build (the image which is at the bottom of the image layers), which is pulled from an image registry during the build.

2. `RUN useradd testuser` invokes the system's utility `useradd`, to create a user with the name `testuser`.

3. `USER test` instruction changes the image's default user to `testuser`, who was created in the previous step, meaning every command thereafter is executed under the specified user.

4. Finally, `CMD echo „This is an example."` runs the `echo` command, printing „This is an example.“ into the command line.

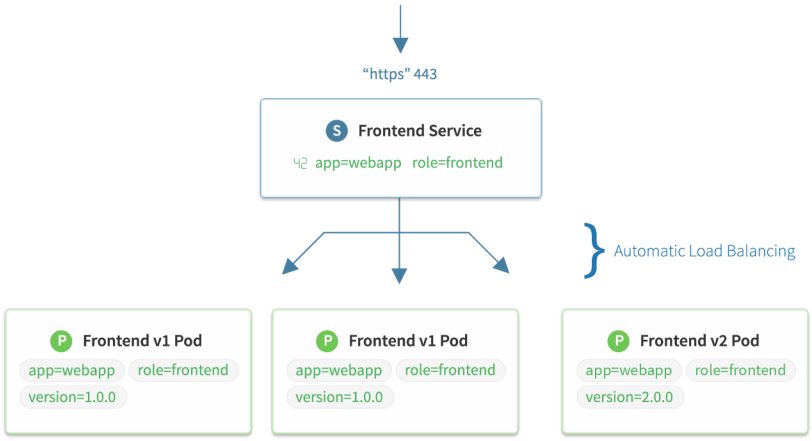


Figure 3.5: Illustration of a service's load balancing features.
Source: Overview of a Service [39]

Services

Services enable their users to expose their Pods as network services, leading to them being able to be accessed inside the scope of an application in the same manner, throughout the application's lifetime. Even though all Pods have assigned IP addresses, these are assigned randomly whenever a Pod is created and released on a Pod's deletion, meaning no static IP address is assigned to the containers by default.

To illustrate on an example, consider a web-based application with two Pods, out of which the first one is a back-end service, and the other is a front-end user interface, both communicating with the other via their assigned IP addresses. In the case of a failure on the back-end's side, the Pod is easily restarted and functioning again in a few moments, but the communication channel is broken as the newly created Pod has a different IP address than the original, which results in non-functionality of the application.

Services serve the purpose of assigning a way to access a defined Pod (or a set of Pods, see Figure 3.5) in the same way in the scope of the application, eliminating the aforementioned problems and providing automatic load balancing between the Pods. [59][39]

Persistent Volumes

Another important functionality of Kubernetes the proposed service takes the advantage of are **Persistent volumes**. First of all, Kubernetes **Volumes** are disk storages (directories) assigned to a Pod. By default, all data in a Container Layer is lost, whenever this container exits. By using Volumes it is possible to share data across a Pod (between containers) and more importantly, preserve the data containers use throughout the entire lifespan of a Pod. [60]

Persistent volumes are Volumes that are not assigned to any specific Pod, and their life-cycle is completely independent, as they are managed as a resource such as a Node and are further set up by the administrator(s) of the system instead of the user. The user's Pods afterwards request a portion of this storage space by creating a *Persistent Volume Claim*, stating how much disk capacity they demand, what kind of storage format they wish to use, what should happen to the storage after the Pod's exit and so on. [58]

Jobs

Jobs' main purpose is to run a set of a Pod's replicas to **completion** (and the Pods' exit). It provides information about the task's progress, and detailed statistics about the replicas (which failed, which succeeded, etc.). Once a certain number of successful tasks has finished, the Job exits and is marked as complete.

To explain on an example, consider a service that is designed to execute multiple instances of a complex computational task, with varying parameters and which runs on a weekly basis. A Job can be used to implement such a service, where the Pods are replicated based on the number of varying parameter sets, each of them executing their part (inside a separate, isolated environment, side by side with each other) and once they have finished their task, the results are collected and the Job is stopped. [55]

For the purpose of this project, Jobs are especially useful for running test suites on multiple implementations, with each of them running in different replica and ending once the test suite has finished its execution. After the Job is completed, the tests' results can be collected and further processed.

3.3.3 OpenShift's Concepts

In addition to terms explained in Section 3.3.2, OpenShift further expands on Kubernetes' functionality to provide a better service for container management. Again, similarly to the previous sections, the following information is taken directly from OpenShift's official documentation [42] and is only a fraction of the original, necessary for understanding this project's design and implementation.

Image Registry

An image registry is a service that allows users to store, manage and distribute Docker Images, along with their metadata[14]. Implicitly, Kubernetes supports using external registries (in addition to the default Docker Hub public image registry) [57] out of the box, but OpenShift expands on this and provides a built-in image registry, that comes with the platform, named **OpenShift Container Registry** (OCR).

This feature allows the users to store their images privately and use them to build their Pod environments. Furthermore, it removes a necessity (but not possibility) for extra authentication that would be required with other external private image registries. OCR also informs OpenShift about new images being pushed, which is especially convenient when updating containerized applications. [38]

Projects

Projects in OpenShift are in their essence namespaces meant to wrap an application, or multiple applications, along with their own configurations, resources and most importantly, Pods and isolate it from other applications running in the cluster. These projects are by default created according to a configurable template, which is set-up by the platform's administrator. [36]

[14] further information about the image, such as its maintainer, date of creation, networking information, etc.

Deployments

Deployments, in relation to Projects, are OpenShift API objects that provide methods for fine-grained management over common user applications. They are composed of the following attributes:

- A *Deployment Configuration* object, defining what **state** a Pod is supposed to be in,
- *Replication Controllers* which define how many replicas (instances) of a Pod are meant to be running and what are the conditions for their execution, and possible restart, or shutdown,
- definition of one, or multiple Pods that the application consists of.

In addition, they also allow the users to deploy a new version of their application, whenever a Pod's image changes, which is very convenient for the scope of this thesis, as it would allow for updating of the assignments' test environments simply by uploading a new image. These objects, along with their technical details are explained more thoroughly in Chapter 5 of this documentation. [35]

Build Configurations and Builds

A **Build Configuration** (i.e. a `BuildConfig` object) is a template for a **Build** operation, which includes the definition of *how* a container image should be built, and also a set of triggers, which determine *when* the image should be built. Another important term to understand is **Build Input**, which is a resource used to create a new container image (for instance a Dockerfile, explained in Subsection 3.3.2). For the scope of this project, the **Git** source was chosen, as it allows for building the image from source code and files contained in a Git repository (see Section 3.3.4). [33][45][34]

Quotas and Limit Ranges

Quotas are objects allowing the administrators to limit the number of objects created inside an OpenShift Project and also limit computing resources, such as CPU, or storage space.

Limit Ranges, similar to Quotas, allow resources to be limited, on the level of Pod, Container, Image, or a Persistent Volume Claim. They are more fine-grained than Quotas and are very useful in the scope of this project as they draw a sharp border around resources a student's project inside a Pod can use and avoid getting stuck, because of a memory allocation bug in the code, a deadlock, etc. [37]

Above-mentioned Concepts in Parallels with xUnit Patterns

The relation of the xUnit Patterns 2.4 to the above-mentioned concepts, in order to implement the proposed testing service is quite simple:

- Standard Fixture - implemented as a Pod (possibly a single-container Pod) with a specific Dockerfile configuration, designed by the tester.
- Shared Fixture - all tests for a given assignment are executed inside a container with identical configuration, i.e. based on the same image.

- Prebuilt Fixture - an image, on which the executive containers are based, is built in advance and stored in the built-in Container Registry, removing the necessity of building a new image for every run of the tests.

These associations give a clearer idea on how the service is going to be designed (expanded more upon in chapter 4), which operational concepts fulfilling what purposes and how the service as a whole can look like.

3.3.4 Version Control: Git

A Version Control System (VCS) allows for managing and tracking changes to information (e.g. source code, documentation, etc.) in order to simplify browsing through past edits and improve efficiency during collaborative work including multiple people working on the same task. [4]

Git is a VCS, which instead of tracking the changes of its files creates miniature snapshots of the repository's files and saves a reference to this snapshot, effectively storing the state of each file via its reference. If a file has not changed, this reference is efficiently passed to the next snapshot, instead of creating a new one. [4]

For the scope of this project, Git was chosen for two main reasons - its overall popularity and specific functionality in OpenShift, allowing for the building of images from a Git repository (see Listing 3).

3.3.5 Python

Python is an interpreted programming language, nowadays utilized for a wide range of purposes, anywhere from system scripting to web development. [46][29]

Its usefulness (and therefore selection) towards this thesis lays in the following points:

- Python is a widely-recognized programming language with a large user base and high-quality community support.
- Object-oriented programming capabilities allow for an efficient design along with a greater deal of abstraction, resulting in more readable and maintainable code.
- An official Python module for the OpenShift client, `openshift`, is available from the Python Package Index (PyPI) [28].
- Python's wide support on various operating systems.

For the actual implementation, version **3** is used, since as of the making of this thesis, Python 2.7 has reached it's end-of-life deadline and is no longer officially supported, and the support for various packages is more prevalent with Python 3.

Chapter 4

Design of the FITest Service's Architecture and Processes

This chapter takes a look at the architecture and process design of the proposed service for testing students' programming assignments, with FITest having been chosen as the working name for it.

4.1 Requirements

The following section contains the requirements for the proposed testing service, along with possible activities which the service should provide and also the activities that the service does not cover and are left upon the service's user.

4.1.1 Features

The final service should support the following features:

1. Testing student submissions automatically, according to the user's requirements.
2. Execution of tests in an identical manner (environment, test suite) for multiple solutions.
3. Evaluating the tests in a standard Pass/Fail/Error manner.
4. Reporting the test results through a command-line interface.
5. Reporting the test results through log files, stored on the server and a corresponding Git repository.
6. Controlling the service via a client command-line interface, from the user's system.
7. Launching tests inside an isolated environment (such as a container running in an OpenShiftPod).
8. Customization of the testing environment by the user, including, but not limited to
   (a) base image,
   (b) system packages,

(c) configuration by own means (commands, scripts).

9. Pushing locally user-made text fixtures to the service.
10. The container's resources (memory, CPU, etc.) can be limited by the user.
11. Running preliminary tests to detect possibly malicious code.
12. Isolating the SUT from the test code and reports.
13. Control via an API[1] (by which the service will be controlled from a command-line interface) supported by the server application.

The satisfaction of the requirements is later referenced in the documentation, when describing a fraction of the design relating to each one of them.

4.1.2 Activities

The service should allow for the following activities, carried out by the user:

1. Running entire test suite on multiple implementations at once.
2. Running entire test suite against a single implementation (repeatedly).
3. Debugging, by connecting to the container and/or starting the container without running the tests.
4. Downloading the testing data.

4.1.3 User's Responsibilities

The responsibilities of a user relate to the activities which are not covered by the service. Namely, they are:

1. Implementation of a test suite.
2. The inspection and validation of the test suite (e.g. by running it against a template implementation).
3. The valuation of the test results.
4. The design of the tests' output format.
5. The design of the container image used for the testing.

[1] Application Programming Interface, defined as a set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or another service.[18]

4.2 Architecture Design

Basic architecture outline, illustrated by Figure 4.1, consists of three main components:

- an OKD cluster, running the service's **server** and the **testing Pods**,
- client application, utilized by the user, to issue commands to the server, manage the assignments and retrieve the test results,
- a Git service (e.g. Gitlab) used for storing the test suites' and the student assignments' code, along with any other resources necessary to build a testing container image.

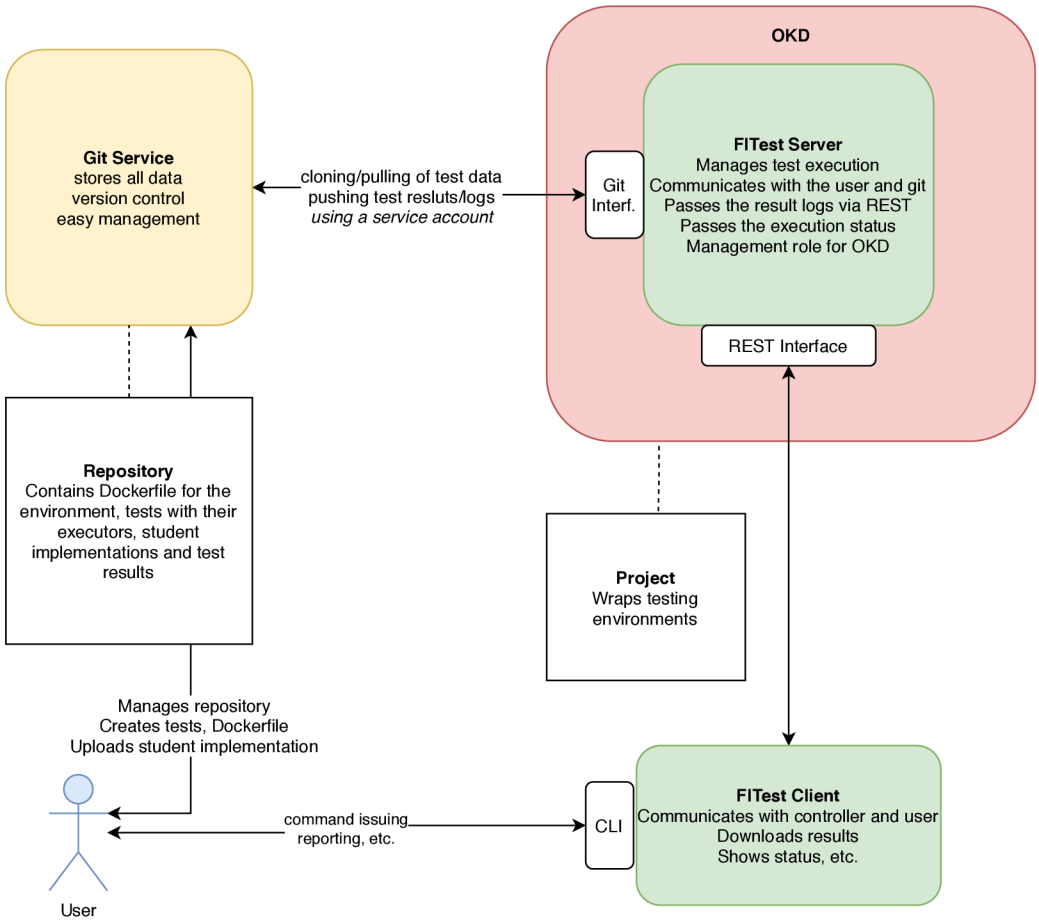


Figure 4.1: The service's architecture.

In the proposed design, a user's task is to store the testing resources in a Git repository. This approach eliminates unnecessary points of failure, and furthermore, Git's functionality allows for many advantages, such as version control and collaboration.

4.2.1 Architecture of the Server Application

The proposed architecture for the server (Figure 4.2) is as follows:

1. **FITest Server** - a separate application that listens on its API endpoint for Client issued commands *(Feature 13)*. It communicates with:
   (a) The OpenShift (OKD) cluster, to control Jobs and Builds, and scaling of the test environments, and the tests' further execution.
   (b) Persistent Volume Claim (2) (CONFIGS) used for backing up the test resources, used by the server and test executors.
   (c) Persistent Volume CLaim (3) (TESTS) used for distribution of students' implementations to the Jobs' pods and temporary storage of test results.
   (d) Git service, to retrieve test resources and manage test reports.
2. **CONFIGS Volume**, storing all the data related to the testing service. This includes the test's results, various possible log files and other test artifacts that can be involved in the testing process.
3. **TESTS Volume**, storing test files, results and logs for the duration of the tests' execution. A separate file system is especially useful for isolating the tested applications from from each other and also, possibly, from accessing the tests' results. *(Feature/12)*
4. **OpenShift Jobs** wrapping the testing environments.

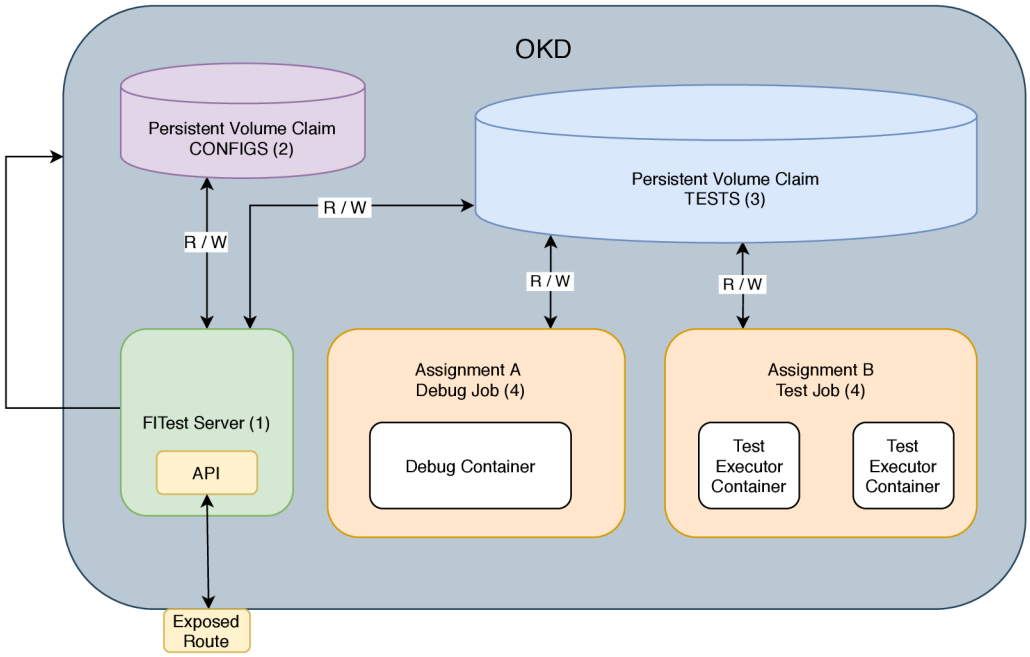


Figure 4.2: Diagram illustrating the server deployed in OpenShift.

4.2.2 Architecture of the Client Application

Proposed architecture for the client-side controller application, implemented in Python (Figure 4.3) is, as follows:

1. **Client Controller** – a central class controlling which endpoint to choose based on the user's commands, also responsible for passing server's responses back to the user (4).
2. **FITest API Endpoint** – a class watching over communication with the server (e.g. launching tests, receiving test results) (2). *(Feature 13)*
3. **Command-Line Interface** – the user endpoint, verifying and passing user input to the Controller and also passing the information from the server back to the user (1). *(Features 4 and 6)*

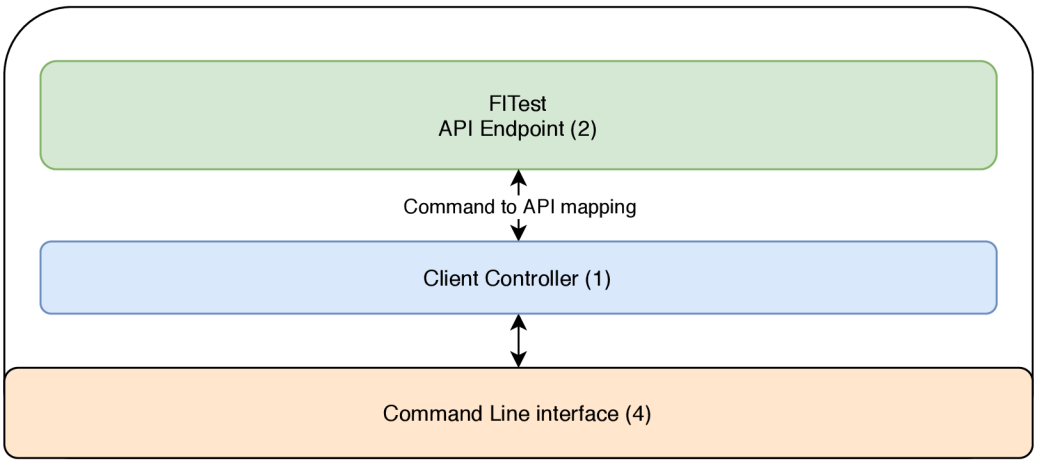


Figure 4.3: Client side application used for interaction with the service.

This fairly simple design should provide for a good level of abstraction and isolation of the aforementioned classes, while also creating a module-friendly environment, which can be expanded with any custom libraries, or scripts to implement additional functionality.

4.2.3 Assignment

An *assignment*, or a *project* object represents a single programming assignment. Its attributes in the scope of this thesis include:

- Project **name**,
- a **Dockerfile**, or a **Container Image** defining the testing environment *(feature 8)*,
- **script, or scripts defining how the test suites are supposed to be executed** *(feature 1, 2)*,
- and **the students' solutions**, which are to be tested.

```
project-name/
├── solutions/
│   └── xlogin00/ ..... Directory with a student's solution.
│       └── program.c
├── tests/
│   ├── run.sh ..... User-defined script that launches tests
│   ├── test_suite.py
│   └── check_user_access.py
├── Dockerfile
└── setup_environment.py ..... Example script used during image build
```

Listing 2: Example of a project directory tree.

Project Environment Folder Structure

The configuration for each assignment testing environment is stored in a Git repository with a specific folder structure. Listing 2 shows an example folder structure for a project-testing environment configuration. The structure is composed as follows:

- The `tests/` folder contains the test suite meant to be executed to test the students' implementations. It contains a script named `run`, which executes the test suite. The `run` script inside the `tests` is a required folder structure element, as the service's test executor expects it to be present during the testing. Any additional scripts, or custom libraries not installed using the system package managers, necessary for the testing (in for example the `test_suite.py` script) should be included inside this folder and the commands used to launch them must be present in the `run` script.
- The `solutions` directory contains the students' solutions, each of them stored in their separate subdirectory.
- `Dockerfile` contains instructions on how to build the testing container image (see Section 3.3.2). In this example, the `setup_environment.py` script is supposed to be included in the build phase of the image. Any artefacts necessary for the image build need to be included in the build context (i.e. the root folder of the repository, or whichever folder specified in the build configuration).

Furthermore, after the first test execution is carried out, a folder named `results` is added to the project, which contains the test reports from the executed tests.

4.3 Process Design

During the design phase, after the architecture of the service was defined, the necessity to specify how each action (e.g. running tests, creating a project, etc.) in detail came up. The following sections explain on a higher level how each of these actions works.

4.3.1 Building an Image

The process of building a Project's container image consists of creating a Dockerfile (see Subsection 3.3.2), and utilizing OpenShift's capabilities to build a Docker image online.

This approach removes the necessity of building images locally and pushing them to the Image Registry, which can be often a very time-consuming, or just an unnecessary task.

```
source:
  git:
    uri: "https://github.com/openshift/ruby-hello-world"
    ref: "master"
  contextDir: "app/dir"
  dockerfile: "FROM openshift/ruby-22-centos7\nUSER example"
```

Listing 3: Example setting of build source to a Git repository. This setting allows for building of container image directly from this repository.

Listing 3 shows an example excerpt from a Build Configuration[2] definition. This configuration allows for the creation of a container image by using a Git repository, specified by its URL (`uri` option), along with a branch (`ref` option). The `contextDir` option specifies the directory, inside which the build resources are located (which, by default, is the root directory of the repository). Additionally, an optional `dockerfile` option is set, containing a string with a Dockerfile-formatted code, which overrides any existing Dockerfile that might be present in the Git repository. The override possibility is especially useful in cases, where an image already exists and has been pushed into the OpenShift Container Registry beforehand, or if there is a necessity to test certain functionality related to the image itself.

4.3.2 Creating an Assignment

Assignment creation consists of two phases, one on the Client's side, the other in the Server (as illustrated by Figure 4.4):

1. The user runs a Client application's command to create a project.
2. The Client sends a request to the server.
3. The Server creates a Git repository, based on the project's name and configuration.
4. From the newly created Git repository, an initial image is built.
5. The User can now clone the empty (only containing the necessary folder structure and pre-generated files) Git repository and update the files.

From the user's viewpoint, this creates an empty template that needs to be filled with the necessary data (SUTs, test suites, Dockerfile, build artefacts, etc.) and does not serve the purpose of creating any tests, or specific configurations deviating from the defaults. This is mainly due to the service's aim to create simple workflows, without the necessity to learn OpenShift Project configuration. However, this design does not prevent the user from designing any advanced configuration, to create a more complex testing environment or functional expansions to the service itself.

[2] an object defining how to build a container image in OpenShift

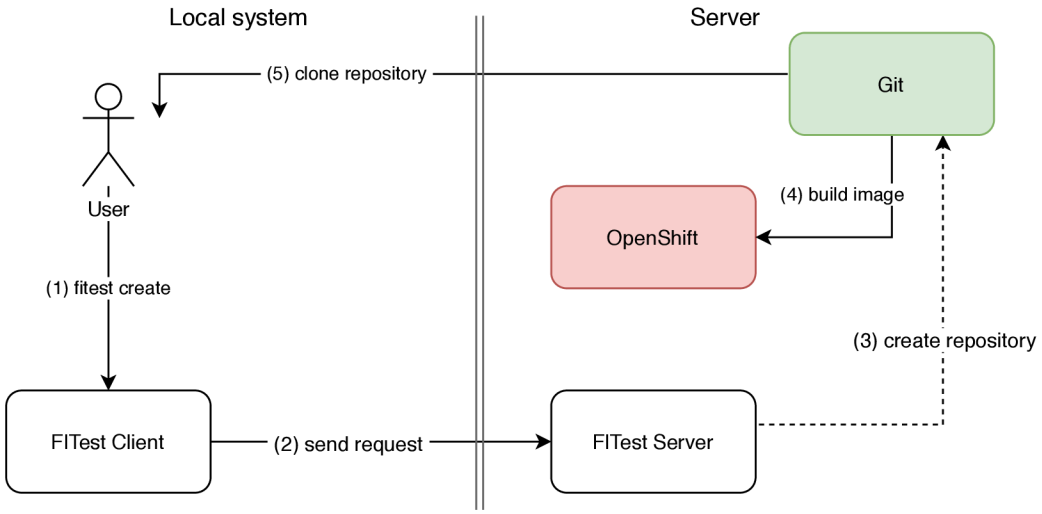


Figure 4.4: Steps in the creation of an assignment, from the point of view of the user.

Pushing the Student Implementations

The students' implementations are meant to be maintained and stored inside the project's Git repository (the same repository containing the testing setup). During the testing phase, the implementations are simply pulled from the repository by the Server and passed to the executing Pods via the TESTS Persistent Volume. This way, backing up and maintaining student's code is made easy and results in a more fail-proof design of the server application.

4.3.3 Configuring a Testing Environment

The primary way to set up the environment, inside which the tests are executed is to provide a Dockerfile (see Subsection 3.3.2). Using this approach the user is able to generate an image configured with its operating system prepared for the tests' execution, along with all system preferences and packages prepared for the task. (*feature 8*)

OpenShift's *Image Creation Guidelines* [32] contain very useful advice on how to properly write a Dockerfile (reusing images as much as possible, clearing temporary build files, configuring environment variables inside Dockerfile, etc.). Furthermore, they present OpenShift-specific information which is necessary to be taken into account by the user when creating an image which is meant to be used with the testing service. The most important of these features is OpenShift's usage of **Arbitrary User IDs**, which, according to the documentation [32] prevents any process from escaping the container and gaining access to administrator privileges on the host machine. This means that the user has to bear in their mind, that the processes ran inside a container will be executed under a random user ID, meaning that certain elevated privilege functionality may not be available.

4.3.4 Updating an Assignment

Changing of the testing environment for a given assignment is handled by OpenShift image build `source` functionality, as it allows for the rebuild of a source image triggered by a change in the Git repository [45]. The user merely needs to update the repository with any necessary changes and push these changes to a configured branch. For example,

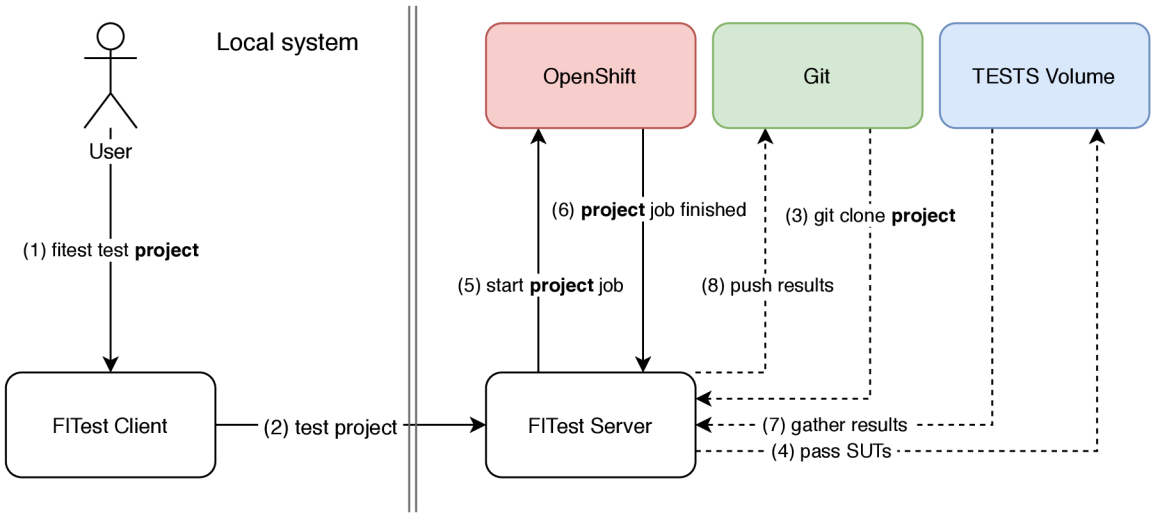


Figure 4.5: System-wide interaction during test execution, including the user's role.

the build policy can be set up in a way, that rebuilds the container image whenever a push to the `master` branch is carried out. This rids the user of the necessity to trigger the image build manually for each small change of the testing environment.

Of course, the option to use a custom image is still available with OpenShift. The user builds the image locally, in their system, using Docker, or Podman, and pushes it into the OpenShift Container Registry *(feature 9)*. In this case, the user has to keep up with the naming convention (explained in Section 5.3.3) when tagging the image.

4.3.5 Running the Tests Against Student Implementations

Test execution (illustrated by Figure 4.5) is performed on the server's side, and invoked by the user, via the API. With a single command, the user starts the Job execution and depending on its settings, two different outcomes are possible:

1. **All student implementations** are pulled from their Git repository. The tests are executed against the SUTs, starting a new Pod for each of them. *(Feature 2, Activity 1)*

2. Only **a subset of students' implementation** is passed to the Job. *(Activity 2)*

In both cases, independent on the quantity of the SUTs, the following steps are carried out for each container, as illustrated by Figure 4.6:

1. The container's *test executor* (explained in Subsection 5.3.1) picks a student implementation. This pick is based on the condition that a log file corresponding with the implementation has not yet been created.

2. To mark the project as occupied, the executor creates an **empty log file**. To satisfy *Feature 12*, the executed script has write permission to the log file but isn't able to read the file. This way the implementation itself is not able to access the logs of the tests.

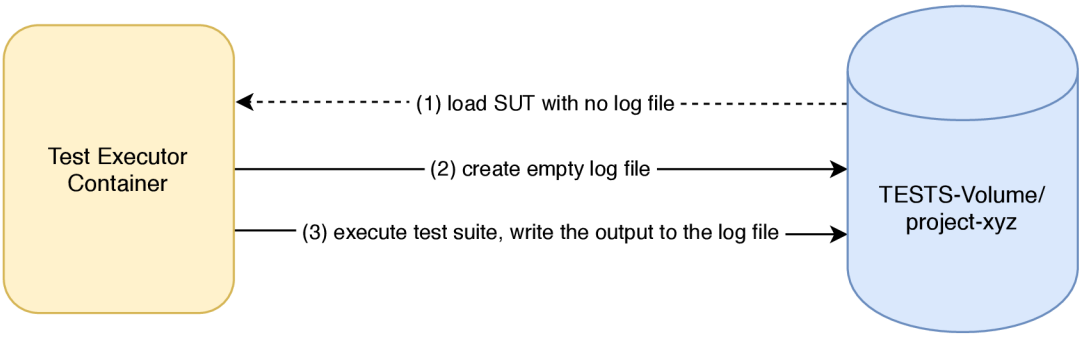


Figure 4.6: Interaction between a Pod and the TESTS Persistent Volume Claim during test execution.

3. The executor then copies the student's solution's directory into the `tests` folder, where it is expected to be present, executes the tests and redirects the output of the test suite to the created log file.

After all Pods have exited, the testing Job is marked as complete, and the results are retrievable. The Server application then collects the testing logs, along with any created test artifacts which are thereafter pushed into their corresponding Git repository, into a subfolder specific for the given test execution.

Debugging Mode

Aside from the aforementioned process of automatic test execution, the service offers a „Debugging mode“ *(Activity 3)*, which allows the user to manually inspect the tests, student's code, and the testing environment. The described functionality is achieved by the following steps:

1. A testing Job is started without the executor script, but instead with an infinitely looping terminal process, preventing it from exiting and/or restarting.

2. The user then connects to the Pod's terminal via OpenShift's built-in console interface, allowing for a live, interactive terminal session.

After they have connected to the Pod's console, the user can investigate on a specific implementation's details (e.g. why the tests are failing, inspecting a part of code, changing a part of code, etc.) without the necessity of building and starting a container on their local system, and injecting the test files manually.

When the user is done with the debugging process, they can simply issue a command which will erase the debugging session, which means that the debugging Job and its Pod are deleted from OpenShift.

Chapter 5

Implementation Details of FITest's Code and Configuration

This chapter goes into detail on how the service was implemented, what specific steps were taken, which aforementioned technologies were used and how the final service as a whole works in order to automate the testing of student programming assignments.

5.1 Development Environment

During the implementation of the service following additional tools/technologies were utilized for the purpose of testing and development:

- Containerized version of GitLab[1], to test the Gitlab-related features, such as source image building,
- MiniShift (see Subsection 3.3.1), and
- Postman, an API development platform[2], used for the testing of FITest's API configuration.

5.2 Initial configuration

During the initial phases of the implementation, several model configurations were necessary to test out the newly developed features.

5.2.1 Creating an example SUT and corresponding test suite

The first step to implement the service was to create an example program posing as a student solution, and a simple test suite that would test this program and produce expected, consistent output.

The example SUT was a simple Python script `find_max.py`, which accepts a set of numbers as command-line arguments and returns the largest of them. The respective test suite was implemented as a straightforward shell script, which compared the output of the `find_max.py` script, given specific input, with the expected output. The output of the test suite is demonstrated in Listing 4.

[1]Source code and documentation available at https://github.com/sameersbn/docker-gitlab.
[2]Official landing page of the project available at https://www.postman.com/.

```
$ ./run.sh
Test 1 - OK
Test 2 - OK
Test 3 - OK
...
```

Listing 4: Example output of the model test suite.

```
1  apiVersion: v1
2  kind: Template
3  metadata:
4    name: example-project
5  objects:
6    - apiVersion: batch/v1
7      kind: Job
8      metadata:
9        name: example-job
10     spec:
11       template:
12         metadata:
13           labels:
14             name: example
15         spec:
16           containers:
17             - name: test-container
18               image: 172.30.1.1:5000/myproject/example-image
19           restartPolicy: Never
20       replicas: 1
```

Listing 5: A simple Template Configuration for a testing Job, used for early development.

This simple output was for checking if the testing Job (see next section) was executed and ended correctly.

5.2.2 Basic Job Template

The next step was to define the above-mentioned testing Job Configuration. The aim of the template was to give the foundation for implementation of more complicated template which would be used later, with minimal necessary changes.

Figure 5 shows the initial configuration. In this example, the highlighted lines specify the following configuration of the Job, based on Kubernetes' documentation [55]:

- Line **7** specifies the `kind: Job` setting, which means that this application will be started and after its execution exited, as opposed to a service deployment, which is meant to run constantly, and possibly restart on an exit due to whatever reason. When using a Job the user must bear in mind that in order to execute a Job to its end,

the container must exit with a 0 return code, otherwise, it's restarted, until it succeeds (or until a certain number of containers fails; by default, OpenShift sets this value to 6 retries).

- Lines **16** through **19** define, what container (or containers) should the executing Pod create. The name of the container is specified by the `name` property, but more importantly, the image, which the given container should be based on is referenced by its URL in the `image` attribute. In this particular case `172.30.1.1:5000/myproject/example-image` was used, as the local MiniShift instance assigns its Container Registry a fixed IP address `172.30.1.1:5000`, via which it is accessible from its Projects.

- Finally the `restartPolicy: Never` setting on line 19 specifies what should happen the moment when a Pod execution fails. The `Never` value causes the Pod to exit and restart in the case where the execution fails, incrementing the failure counter. As counterintuitive as this sounds, it is actually based on the difference between the restart being carried out by the Job scheduler and the Kubelet (Pod) itself. The actual difference this makes for the scope of this project is, that setting the value to `Never` makes sure that the failed containers and their logs will stay present in OpenShift, as opposed to `OnFailure`, which would delete them. Further information about this setting can be found in Kubernetes' documentation [55].

5.3 Implementation of FITest's Server

After the testing configurations were prepared and tested, gradual implementation of the Server's functionality was initiated.

The Server is a containerized application running in an OpenShift cluster, within its own OpenShift Project (namespace). This application can interact with an allocated GitLab namespace (along with a dedicated service account) and furthermore possesses the ability to control Jobs and Build operations inside this namespace. With these functions, it is able to retrieve the data necessary for testing, execute the tests and then if the testing has succeeded, push the test results along with any other essential test artefacts back to the GitLab repository, which the data was initially pulled from.

The details of these processes are described in the following subsections, along with the documentation of important functions, which can be expanded upon in the future.

5.3.1 Test Executors

Test Executor is a containerized Python script responsible solely for test execution, as illustrated by Figure 4.6. The `Executor` class is implemented as a singleton class[3], which operates as follows:

1. First, the setting and validity of environment variables pointing to the tests' and solutions' folders is verified, and if any defect is detected, the executions exits with an exception. The setting of these variables is explained in more detail in a later part of this section and also in Subsection 5.3.3.

[3] A class, which is meant to have only a single instance

2. If all the checks pass, the respective directories are copied, side by side, into a third, **testing environment directory** (in case that there are no solutions left to be tested, the script ends with a 0 exit code). This testing directory, unlike the original two, is located inside the Executor container, which means that the testing is not directly affecting the volume, which is being shared with the Server. At the same time, a log file for the solution is created in the directory, where all the solutions are located. To simplify the setup, an environment variable, `$FITEST_STUDENT_LOGIN` is created, and contains the name of the solution, which is being tested by the specific executor (i.e. the SUT's folder can by accessed, from the context of `run.sh` as `../$FITEST_STUDENT_LOGIN/`[4]).

3. Once the files are all set up, the class checks for the `run.sh` script, and if it's present, the main method of the class launches it as a command. In this part, the setup of the test–solution dependencies (e.g. copying files between the two directories prior to the testing), is managed by the user, and is expected to be done in the `run.sh` script. The output of the testing is redirected to the aforementioned log file. For the sake of concealing the log file's contents during the testing, this file can be written into only by the user running the execution script.

Executors are utilized by **execution Jobs**, which are started by the Server (explained in more detail in Subsection 5.3.3), after receiving a corresponding command from the user (see Subsections 5.3.4 and 5.3.5). The containers which are started by these Jobs are injected with an environment variable containing the path to a pseudo-randomly generated temporary folder path (point 1 in the previous list). After the execution of the Job has finished (or failed, in case the tests did not perform correctly), the log file's access rights are set to read-only, and the Pods and their respective logs are kept in the OpenShift project (namespace), but only until the next batch of tests is started. This means that before starting a testing Job for a specific assignment, the previous Job's data is erased from the project. This is done for two major reasons:

- First of all, to keep the project's resources orderly. Had the Jobs not been cleaned like this, a long history of Pods could be generated and could prove to be quite hard to navigate, or manage, especially with assignments tested for large numbers of students.

- Secondly, the controlling of the Job objects in using OpenShift's API, with this restriction is much more definite and fail-proof.

It is important to note, that due to the fact that the execution logic was implemented using Python 3, any container images designed for testing **need to contain the Python 3 package** in order to run the tests.

5.3.2 Server's Configuration Management

The server's configuration is handled by a class named `ConfigLoader`. The main purposes of this class are

1. loading the server's configuration related to GitLab and the file system environment (paths to configurations files), and

[4]Even though this approach is a bit complicated, it was chosen due to the possible errors which could come from merging the directories.

2. loading and storing the configuration of each project into a designated YAML file, which contains information such as Git repository's URI, status of the build, status of the test execution, etc.

The configuration of the Server is done during the initialization of the main Server class, by reading the terminal's environment variables. The variable list is as follows:

- `FITEST_SERVER_PROJECTS_RECORD`: The path to the configuration file containing the information about the projects.
- `FITEST_TEST_DIR_ENV`: Specifies the path to the mount point of the persistent volume dedicated to passing the test data between the server and the executors.
- `FITEST_REPO_STORE`: The path to a directory, which temporarily stores the data retrieved from GitLab, for the duration of the tests.
- `FITEST_GIT_HOST`: Contains the URL to the GitLab instance, the dedicated FITest namespace is located. This value is used for generating the URIs to the projects' repositories, which are then used to access the test files.
- `FITEST_GIT_TOKEN`: This value is a string containing the GitLab OAuth Access Token, associated with the GitLab service account, used for authorization against the service, during various Git operations. The account, to which the token is bound **needs to have proper access rights** (e.g. owner, or developer), to be able to delete and create repositories and to push and pull data from them.
- `FITEST_OPENSHIFT_NAMESPACE`: The name of the OpenShift project, where the server Pod is deployed. Defaults to `fitest`.

5.3.3 OpenShift Operations

A more extensive class, `OpenshiftControl`, contained in the `fitest.openshift_control` module, is in charge of communicating with the cluster, where the service is running. It utilizes the official `kubernetes` and `openshift` modules, available from the Python Package Manager, to carry out a set of tasks related to image building, and Job execution. During its initialization, an instance of this class

1. checks if the initialization is being executed out inside a Kubernetes (i.e. OpenShift) cluster, (if it's not the case, it raises an Exception, exiting the script and shutting down the Server operation),
2. the in-cluster settings of the executing Pod's service account are loaded, for the purpose of authenticating against the OpenShift API (see Subsection 5.3.9),
3. based on these settings, a Kubernetes client is created, and further passed to an OpenShift `DynamicClient` object, used for the OpenShift API operations.

Hereafter follows the documentation of the methods and resources implemented by the `OpenshiftControl` class, that are used by the Server application.

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: {job_name}
spec:
  successfulJobsHistoryLimit: 1
  failedJobsHistoryLimit: 1
  parallelism: {solution_count}
  completion: {solution_count}
  template:
    spec:
      containers:
        - name: executor-container
          image: {registry}/{namespace}/{project_name}-image
          volumeMounts:
            - mountPath: /tests-volume/
              name: tests
          env:
            - name: FITEST_SOLUTION_PATH
              value: "{solution_path}"
            - name: FITEST_TEST_PATH
              value: "{test_path}"
          command: {command}
          {limits}
      restartPolicy: Never
      volumes:
        - name: tests
          persistentVolumeClaim:
            claimName: tests-claim
      replicas: 1
```

Listing 6: Parametric Job configuration used by the Server to create Executor Jobs.

Job Configuration

In order to execute various testing, or debugging Jobs (see Subsection 3.3.2), the class uses an adjustable YAML string (this string is updated by the `OpenshiftControl` class for every Job execution), which contains the definition of the aforementioned Jobs.

The configuration of the Job shown in Listing 6 shows the definition of the object. Values contained in braces represent parametric values, which are dynamically filled in on a Job creation. The description of the configuration is as follows:

- The configuration starts with the definition of the `apiVersion` attribute, whose value, `batch/v1` in this case, refers to the API namespace containing the methods for Job handling. The `kind: Job` attribute specifies the type of OpenShift object this configuration is describing.
- The `metadata.name` field contains the name of the Job to be created, by which it is referenced when using the OpenShift API. The `{job_name}` value is a parametric field, which is filled in during a specific Job's definition and corresponds with the name of the FITest project, for which it is being executed.
- The `spec` dictionary contains the specification of the **Job's technical parameters**:
  - Both `successfulJobsHistoryLimit` and `failedJobsHistoryLimit` attributes set the limit for history of successful and failed Job Pods, respectively. Based on this value a certain amount of log data is kept around in OpenShift, after the Jobs have ended. Due to the fact that the Jobs re-created on each run, this value is set to `1`, as there is no necessity to store more data, and acts more as a fail-safe mechanism, in case a Job delete operation fails.
  - The `parallelism` field sets how many of the Job pods should attempt to be executed at the same time (attempted, as in using as many resources as assigned and then queuing the rest). The `{solution_count}` value's input is referring to the number of SUTs chosen for testing. This means, that in an ideal situation, all student solutions are tested in parallel, and the duration of the execution depends on the duration of the slowest test Pod. In the case, where a debugging session is to be created, this value is set to **1**.
  - Using the same parametric value, as the previous field, the `completion` property defines how many Job Pods must exit the testing **successfully**, in order for the Job to be marked as „**Finished**“. This way, the testing Job is successful only if the test execution of all the chosen solutions is successful (as described in Section 5.3.1). **While a test may fail, the test suite itself has to be able to recover from possible errors and exit with a success return code**, otherwise the Job will be restarted, and fail, until a Pod failure limit, by default set to 6 attempts, is reached. In such case, the entire test execution would result in a failure.
- The `spec.template.spec` describes **what the Job's Pod should look like**:
  - The `containers` field is used to define one, or more containers, inside the Pod, which are supposed to carry out the Job. Inside, the `executor-container` is set up. This container contains the code which carries out the testing, as described in Subsection 5.3.1. This container is build upon container image pulled during

the Job's creation, from URL contained in the `image` field. The parametric value `{registry}/{namespace}/{project_name}-image` consists from:

  * The URL to a container image `registry`, where the image is stored. In the scope of this implementation, this value is defaulted to the built-in OpenShift container registry.
  * `namespace` refers to the sub-path of the registry, which in the case of the internal registry points to the name of the **OpenShift project**, in which the image is supposed to be used. The value must correspond to the name of the project, where FITest is deployed.
  * Lastly `project_name` is the name of the FITest project object, meaning that every project has a separate image dedicated to it (further explained in Subsection 5.3.4).

  The `volumeMounts` defines which volume object (`name`) should be mounted to the container, and specifies the path (`mountPath`), accessible from the container, where this volume should be mounted. In this case, as defined in the Server's Architecture (see Section 4.2) the value is set to `/tests-volume/`.

  The `env` attribute is used to inject environment variables in the container — as described in Subsection 5.3.1, the paths to solutions and test suites are injected this way, and their values are generated by the Server prior to the testing (see Subsection 5.3.4). Lastly, the `command` field is used to invoke a shell command in the container (furthermore, it overrides any `CMD` setting which could have been defined in the image's `Dockerfile`).

- If any CPU, or memory limits were requested by the user, the `limits` object is added in place of the `{limits}` parameter. If no limits were requested, an empty string is added instead.
- `restartPolicy` is set to `Never`, as described in Section 5.2.2.
- The `volumes` is a list of references to Persistent Volume Claims. Each item defines how the volume should be referred to by the containers (`name`) and which Persistent Volume Claim should be used (`persistentVolumeClaim.claimName`).
- Lastly, `replicas: 1` limits the Job's Pod to a single instance, as it is not necessary to scale an execution Pod this way. The replication is instead assured by the `parallelism` and `completion`.

Build Configuration

Similarly to a Job Configuration, a Build Configuration (see Subsection 3.3.3) is created from a pre-defined YAML string with parametric attributes, some of which are the same as in the above explained Job Configuration:

- Again, the object's definition begins with `apiVersion`, which is, in this case, `build.openshift.io/v1`, and contains API operations for Builds and Build Configurations.
- The object's `kind` is `BuildConfig`, and even though this might come across as a confusing naming (Build and Build Configuration), this object is merely a **configuration for any number of future Builds**, which are supposed to build a given image.

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: {project_name}
spec:
  runPolicy: Serial
  triggers:
      - type: ConfigChange
  source:
    git:
      uri: {project_git_uri}
      ref: master
    sourceSecret:
      name: gitlab-token
  strategy:
    dockerStrategy:
      forcePull: true
      dockerfilePath: Dockerfile
    type: Docker
  output:
    to:
      kind: DockerImage
      name: {registry}/{namespace}/{project_name}-image:latest
```

Listing 7: A parametric Build Configuration used for building of container Images by the Server.

- Again, the object is named after the project (inside a different namespace, e.g. `jobs/` and `buildConfigs/`).
- A BuildConfig's `spec`, however, contains a different set of attributes:
  - `runPolicy: Serial` forces any Builds triggered (i.e. started) during a previous build to wait until all previous Builds have finished. This way, the output of the build is always predictable and consistent. Due to the way the build mechanism is carried out in the specific case of FITest (see **Build operations** in Subsection 5.3.3), there should be always only a single Build instance for every given project, meaning this setting serves as a fail-safe purpose.
  - The `triggers` field contains a list of events, which trigger a Build based on this Build Configuration. In this specific case, a single trigger, with the type `ConfigChange` is set up, which starts a Build, whenever this BuildConfig is created in OpenShift (as confusing, as it seems, the official documentation section regarding Build triggers states, that the functionality triggering the Builds on every configuration update will be added in a future release, and is not supported in the version 3.11). Furthermore, once the service is running inside a cluster accessible from the internet, a trigger based on **GitLab Webhooks** can be added, meaning that a Build can be started, whenever any associated data in a GitLab

repository is updated (at the time of implementation it was not possible to add this feature, as the limited resources of Minishift would not allow for the testing of webhook triggers). [34]

- The way of retrieval of data necessary for an image 's build (Dockerfile, local packages, etc.) is defined by the `source` field. Since the service is designed to use Git to store these files, a `git` object is defined as the source, along with `uri` to the repository and `ref:master` option specifying which branch to use (in this case, the `master` branch). Furthermore, for the purpose of authorizing against the Git service, the `sourceSecret` field is added, to reflect this fact. The `gitlab-token` value is an identification, of a `secret` object stored in OpenShift in advance (see Subsection 5.3.8).
- The `strategy` property defines what mechanisms are supposed to be used when building the image. `type:Docker` setting specifies that the image will be built using Docker (i.e. building according to a Dockerfile) and `dockerStrategy` further expands on the possible options. `dockerfilePath` contains a file path, inside the build context, where the Dockerfile for the image is located. In combination with the `git` source, the `Dockerfile` value in the field translates to building the image from the provided repository, where the Dockerfile for the build is located in the root folder of the Git repository.
- Lastly, the `output.to` field specifies, where the image created by the Build should be stored. As mentioned in the previous section, every project has a dedicated image, which is stored in the internal OpenShift registry. Based on this information, the output of the build is set to `kind: DockerImage`, meaning a Docker-formatted image file will be pushed to the URL `{registry}/{namespace}/{project_name}-image`, as described in the previous Section. This image is also attributed the `latest` tag, meaning that the Job configuration will always point to the latest built image.

Job Operations

The `OpenshiftControl` class provides multiple methods for interaction with the OpenShift Job API, which are used by the `Server` class to execute the tests:

- `OpenshiftControl.start_job()`: The objective of this method is to start a testing Job using the aforementioned Job Configuration. At first, the method deletes any existing testing jobs associated with the project. Afterwards it creates a Job using the Job API and passes all necessary parameters from its input to the Job Configuration string's parameters. The command the Job uses is `[„python3", „executor_path"]`, which means that the executor script is launched on the container's start. If an error occurs during this process, an error message is returned by the method.

- `OpenshiftControl.get_job_status()`: This method returns the current status of a Job and returns this status as a dictionary. If a Job with the specified name is not found, an error message is returned instead.

- `OpenshiftControl.wait_for_job_finish()`: During the tests' execution the `Server` class needs to wait for the testing Job to finish, so it can finalize the testing process. `wait_for_job_finish()` polls OpenShift's API for the Job's status and returns a status message, once the Job has succeeded, or failed.

- `OpenshiftControl.delete_job()` is used to delete a Job, and, if any Pods belonging to the Job are left, it deletes them as well.
- `OpenshiftControl.create_debug_session()` works very similarly to `start_job()`, with 2 differences: There is always only a **single Pod active**, no matter the number of the solutions, and the command the Pod executes is `[„sh", „-c", „tail -f /dev/null"]`, which is meant to **keep the Pod alive**, until the user decides to delete the session.
- `OpenshiftControl.end_debug_session()` simply deletes the debugging Job, if one exists.
- `OpenshiftControl.get_job_pod_name()` returns the name of the **first** Pod belonging to a specific Job. This method is used for controlling the debug session.

Further usage of these methods is explained in Subsection 5.3.4.

Build Operations

Along with Job operations, the class also provides functionality for basic Build operations:

- `OpenshiftControl.update_build_config()` has two main tasks: First, it removes any previously existing Build Configurations associated with the specified FITest project, after which it creates a new Build Configuration, based on the aforementioned Build YAML configuration string — this, as explained previously, triggers a new image Build.
- `OpenshiftControl.get_build_status()` returns the state in which the latest build for the FITest project is, according to the OpenShift Build API.

5.3.4 Parent Server Class

The `Server` acts as a mediator between the API endpoints, and the executive features. Its purpose is to wrap the `OpenshiftControl` class' functionality with file management, and reporting features, related to Git and API clients. The class' public methods include the following:

- `Server.add_project()` creates a new project record, along with a new, fresh GitLab repository in the designated namespace, and pushes the basic structure of the repository, along with a dummy Dockerfile, as the initial commit. If a project with the provided name already exists, or the repository with the same path exists, returns an error message. If all GitLab operations carry out correctly, a build, using the dummy Dockerfile is started. This feature guarantees that there will always be an image present for the project, available as a fallback option in case a future build fails.
- `Server.delete_project()` deletes a projects record in the Server's configuration and removes the associated Git repository, by accessing the GitLab API.
- `Server.build_project_image()` wraps the `update_build_config()` method, adding the check for an existing build **in progress**. If such a build exists, a message, informing the user of an active build is returned. If no Build is active, it starts one.

- `Server.get_build_status()` simply wraps `get_build_status()`, without any additional features and passes its output to the API.
- `Server.get_project_info()` returns all stored information about a project. If a project's name does not exist in the records, an empty dictionary is returned.
- `Server.run_project_tests()` contains the functionality for the test execution:

  1. In the beginning, it sets up the testing environment by cloning the corresponding repository, creating a temporary folder inside the volume, which is being shared with executor pods, with a randomly generated file path and moving the tests and solutions from the repository to this directory. If the repository cloning fails, the repository's directory structure differs from the defined structure (see Figure 2), or no student solution's names matched the optional list of solutions to be tested, a respective error message is set. In such a case, the method skips the execution and goes straight to finalization (3).
  2. After the environment is set up, the method starts the Job, with the data provided by the setup method. Thereafter, the method waits for the execution to be finished.
  3. When the execution has finished, in case there are any test results available, they are uploaded to a respective Git repository, and the temporary local files are erased from the storage.

  This process is illustrated in more detail by Figure 5.1.
- `Server.stop_project_tests()` is used to stop a project's test execution prematurely, without pushing any results. This functionality is achieved by removing the testing Job, which is registered by the wait function, meaning the testing process is finalized (file cleanup), but no results are pushed into the corresponding Git repository.
- `Server.start_project_debug()` sets up the environment in a similar way to `Server.run_project_tests()`, but creates a persistent single-Pod Job, using OpenShift control's `create_debug_session()`. After this Job has been set up, the method retrieves the name of the Job's single Pod, and stores it in the project's information, along with the path to the temporary testing environment.
- `Server.end_project_debug()` is used to remove the debugging Job, along with its Pod, utilizing `OpenshiftControl.end_debug_session()`, and clearing any debug related data from the project's stored information. The method returns any error message that `OpenshiftControl` passed to it during Job removal.

5.3.5 API Endpoints and Flask

The Server's API functionality was implemented using a Python framework called **Flask** [26]. The framework was used to map specific HTTP requests[5], made to pre-defined URLs, to methods of the `Server` class. This way, a service's user is able to invoke the Server's functionality by using these HTTP requests (explained in more detail in the Client section).

[5]FITest uses only a subset of existing HTTP requests' types. More information about different types of HTTP requests available at https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html

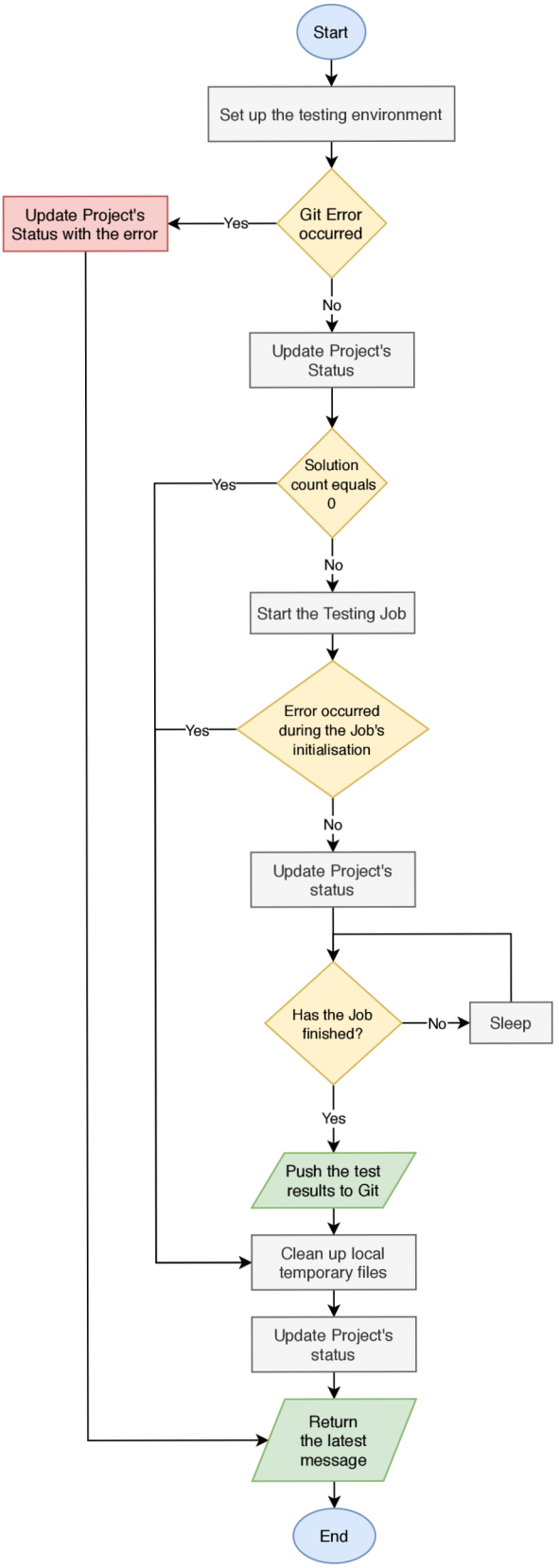


Figure 5.1: Flow chart describing how the test execution works on the Server's side.

The base of the URL is set to `[prefix]/api/`, where the `prefix` part of the URL is the address, where the service is exposed and available on the network. The following routes, parametrized by the `project-name` (if a project with given name is not present on the Server, a 404 response is returned), with **specified HTTP methods**, have been defined for the Server's functions:

- `GET /api/project/project-name` runs the `Server.get_project_info()` method and wraps the project information in the response with status 200.
- `POST /api/project/` is used to create a **new project**. This method expects a `'project-name'` field in the request's body. The value of this field is then compared with existing projects, and if such a project does not exist, the `Server.add_project()` method is called, to create a new project. Its output, whether a success, or an error message, is returned in the response. A project's name can consist only of alphanumeric characters, along with the „-“ and „_“ symbols.
- `DELETE /api/project/project-name` executes `Server.delete_project()` to remove the project from the Server and Git.
- `GET /api/project/project-name/build` invokes the `Server.get_build_status()`, and returns the **latest build's status** in the response.
- `POST /api/project/project-name/build` starts the project's image's build with `Server.build_project_image()` and returns the initial build status, or error, within the response.
- `POST /api/project/project-name/execute` initiates the test execution for the given project by calling `Server.run_project_tests()` and either returns an error message in the response, or waits for the execution to finish and returns a status message. The timeout mechanics of the response are explained in Section 5.4. Furthermore, in the body of the request, this endpoint accepts information about how the container's resources should be limited.
- `DELETE /api/project/project-name/execute` stops a test execution prematurely by calling `Server.stop_project_testing()`.
- `POST /api/project/project-name/debug` creates a debugging session by running `Server.start_project_debug()` and returns the links to the created debugging resources, or an error message in the response. In a case, when a debugging session is already active, the method returns the references to the existing session in the response.
- `DELETE /api/project/project-name/debug` uses the `Server.end_project_debug()` to delete an active debugging Job for the project and returns a message in the response. If no active session is present, a message, informing the user of this fact, is returned.

5.3.6 Server's OpenShift Configuration

An important part of the development was to create an OpenShift Template[6] for the server. This Template contains everything necessary for the server application to be deployed in an OpenShift environment. The following listings contain the Template objects (due to the length of the single file, it has been split into multiple smaller listings).

Server's Deployment Configuration

Listing 8 shows the `DeploymentConfig` object, which defines how the server itself should be ran. The `spec.template` defines the **Pod Template**, i.e. what Pods are supposed to be run in the Server's application. Since the Server itself requires only a single Pod for its functionality, the `container` setting looks as following, starting on line 12:

- The server's container is named `server-container` and uses image located at `172.30.1.1:5000/fitest/fitest-server`. The IP address, as explained in Subsection 5.2.2, is assigned to the internal Container Registry by default, by OpenShift. This value refers to the container image named `fitest/fitest-server`, which is stored in the internal Container Registry. Furthermore, the Deployment configuration is set up in such way, that it is deployed in **OpenShift project named** `fitest`, hence the image's path's prefix `fitest`.
- The container's environment variables are defined in the `env` attribute and contain values explained in Subsection 5.3.2. All variables contain hard-coded strings, except for `FITEST_GIT_TOKEN`, which is using a reference to the project's **Secret Storage**. This secret referencing is explained more thoroughly in Subsection 5.3.8.
- The `volumeMounts` array works as explained in the Job part of Subsection 5.3.3, but also contains a second volume claim `configs`, which is used to store the project information (the usage of this volume is explained in Section 4.2.1).
- Lastly, the container's port **8080** has to be exposed for the container to be able to receive TCP communication. This is done by specifying the port inside the `port` array on line **33**.

The `replicas: 1` setting specifies how many copies of the Pod defined in the Template should be active when deployed. Replication is used for scaling of the application in case, where more resources are necessary (e.g. a webserver receiving too many requests). As in this prototype form of the project the server is not expected to receive heavy loads of traffic, one replica of the Pod is sufficient. If a need to scale the Server's application comes up in the future, only a small change, consisting of adding an automatic scaler, would be necessary.

Lastly the `strategy.type` is set to `Rolling`, which means that whenever a new version of the Deployment should be used (updates of the configuration, image change, etc.), the transition between the old and the new instance of the service will be without any actual outage. This is achieved by first spinning up the new version, redirecting all traffic from the old one to the new one, and finally terminating the old instance.

[6] template, which describes multiple various OpenShift objects, such as Builds, or Deployments

```yaml
apiVersion: v1
kind: DeploymentConfig
metadata:
  name: fitest
spec:
  template:
    metadata:
      labels:
        name: fitest
    spec:
      containers:
        - name: server-container
          image: 172.30.1.1:5000/fitest/fitest-server
          env:
            - name: FITEST_SERVER_PROJECTS_RECORD
              value: "/config/projects.yaml"
            - name: FITEST_TEST_DIR_ENV
              value: "/tests-volume/"
            - name: FITEST_REPO_STORE
              value: "/tmp/fitest-repo-store/"
            - name: "FITEST_GIT_TOKEN"
              valueFrom:
                secretKeyRef:
                  name: gitlab-token
                  key: password
            - name: "FITEST_GIT_HOST"
              value: "https://pajda.fit.vutbr.cz/"
          volumeMounts:
            - mountPath: /tests-volume/
              name: tests
            - mountPath: /config/
              name: config
          ports:
            - containerPort: 8080
              protocol: TCP
          command: [python3, /fitest/app.py]
      restartPolicy: Always
      volumes:
        - name: tests
          persistentVolumeClaim:
            claimName: tests-claim
        - name: config
          persistentVolumeClaim:
            claimName: config-claim
    replicas: 1
    strategy:
      type: Rolling
```

Listing 8: The Server's OpenShift Deployment Configuration.

```
1  - kind: PersistentVolumeClaim
2    apiVersion: v1
3    metadata:
4      name: tests-claim
5    spec:
6      accessModes:
7        - ReadWriteOnce
8      resources:
9        requests:
10         storage: 500Mi
11     volumeName: pv0001
12
13 - kind: PersistentVolumeClaim
14   apiVersion: v1
15   metadata:
16     name: config-claim
17   spec:
18     accessModes:
19       - ReadWriteOnce
20     resources:
21       requests:
22         storage: 100Mi
23     volumeName: pv0002
```

Listing 9: The Server's Persistent Volume Claim Configuration.

Access to Persistent Volumes

Next, Listing 9 shows, how the Persistent Volume Claims referred to in the Deployment Configuration (Listing 8, line 28). Before a Persistent Volume can be used by a Pod, a Claim to the Volume must be made by the application. As shown in the `spec` attribute of the object, in Listing 9, a Persistent Volume Claim's settings are:

- The `accessModes` array, in this case, contains only a single value, `ReadWriteOnce`, which means that only a single Node is able to access the volume in a Read/Write fashion at any given time[7]. This mode was chosen as it provides best support for different file systems, but may change once the service has been deployed in a multi-node cluster.

- `resources.requests` specifies what resources are being requested for the given Claim. In this specific case, `storage: 500Mi` value is set, which means that this Claim is requiring 500MB of storage space to be allocated on the Persistent Volume for it.

- Finally `volumeName` is the name of the Persistent Volume object (see Section 3.3.2), for which the Claim is making a request. For the two aforementioned Claims, defined for the FITest service, `pv0001` and `pv0002` were chosen, mainly for development purposes, since these objects are default Persistent Volumes, provided by MiniShift.

[7]The rest of the modes are listed at https://kubernetes.io/docs/concepts/storage/persistent-volumes/#access-modes

```
- apiVersion: v1
  kind: Service
  metadata:
    name: fitest
  spec:
    selector:
      name: fitest
    ports:
      - name: http
        port: 8080
        protocol: TCP
        targetPort: 8080

- apiVersion: v1
  kind: Route
  metadata:
    name: fitest
  spec:
    to:
      kind: Service
      name: fitest
    port:
      targetPort: http
```

Listing 10: Service and Route objects' configuration.

Networking Configuration

Lastly, Listing 10 shows the configuration of the Server's `Service` and `Route` objects. These objects are used for enabling access to the Server via network, both from inside and outside of the cluster:

- The `Service` object is used to assign a in-cluster static IP address to a Pod (as explained in Subsection 3.3.2). The `selector` property is used to find all containers with name matching `fitest` (i.e. the Server Pod(s)), and associate the Service to them. The main part of the object is the `ports` attribute, which is used to specify which port(s) of the associated Pods should be exposed using this service. In this case, the `8080` port gets exposed (the `targetPort` field specifies through which port the communication will arrive from outside of the Pod) to `TCP` communication. The selected port is also given a name, so it can be used in the following object.

- A `Route` object is used for exposing a Service outside of the OpenShift cluster, which means it will be available vie a given hostname (e.g. `https://fitest.com`). The `to` attribute specifies that the Route should lead to a `Service` object with the name `fitest`. The `port` field specifies to which port of the Service (in this case the port named `http`) the internet traffic should be directed to.

5.3.7 Creating the OpenShift Namespace

To run an application in OpenShift, an OpenShift Project (see Subsection 3.3.3) needs to be created in order to wrap the application. For this task, the **OpenShift Command Line Interface**[8], a command-line client tool for interaction with the OpenShift API, was used. To create the Project, the following commands were issued:

```
$ oc login -u developer -p developer
$ oc new-project fitest \
    --description="FITest Testing Service" \
    --display-name="FITest"
```

The `oc login` command was used to log in to the API, as the default MiniShift non-admin user called `developer` and the `oc new-project` command was executed afterwards, to create the Project. The `fitest` argument specifies **the name of the Project object**, `--description` specifies the summary information about the Project, and lastly, the `--display-name` argument sets what name should be displayed in the Web Console application. Unlike the object's name, the display name can contain capital letters and spaces.

5.3.8 Creating a Secret Resource for the GitLab Access Token

The next step in the creation of the application was storing the above mentioned GitLab Access Token, which would be used for authentication and authorization against GitLab (see Sections 5.3.2 and 5.3.6). Again, the CLI application was used, executing the following command:

```
$ oc create secret generic gitlab-token \
    --from-literal=password=[TOKEN_SECRET] \
    --type=kubernetes.io/basic-auth
```

The `oc create secret generic secret-name` command is used to create a secret from a local file, directory or literal value[9] and store it as an object with the name `secret-name`. The `--from-literal=password=token-secret` flag specifies that the secret should use the value from the provided literal, and save it as a password (other options, for instance `--from-literal=user=username`, could be added, if, for instance, a user-password basic authentication was desired[10]). The `type` flag specifies the type of the secret object. Since the secret is created from a non-specified token value, even though the token is a GitLab OAuth token[11], the best option to store and access it is as a `basic` type of secret.

This value is then retrieved by the Server's Deployment Configuration on lines 21 through 25 in Listing 8, by using the `secretKeyRef` setting, inside which the `name` of the secret is looked up and the value stored in `password` attribute (hence the `=password` option specified in the literal flag) is injected into the specified environment variable.

[8] More information about the tools usage available at https://docs.openshift.com/container-platform/3.11/cli_reference/get_started_cli.html.

[9] Further information about types of secrets available at https://docs.openshift.com/container-platform/3.5/dev_guide/secrets.html

[10] More information about the basic authentication model available at https://docs.openshift.com/container-platform/3.11/dev_guide/builds/build_inputs.html#source-secrets-basic-authentication

[11] Information on how the OAuth authentication works is documented at https://docs.gitlab.com/ee/api/oauth2.html#access-gitlab-api-with-access-token

5.3.9 Elevating the Project's Service Account's Privileges

For the Project's Service Account[12] to be able to start Jobs, a special privilege had to be added to it. Using the following command, a pre-defined role, called `job-controller`[13] was added to the project's default service account:

```
$ oc policy add-role-to-user system:job-controller \
    system:serviceaccount:fitest:default
```

By adding the privileges, the `OpenshiftControl` class (Subsection 5.3.3) is able to execute Jobs, since the Kubernetes configuration loaded during the Class's instance's initialization uses this specific service account. Furthermore, no other roles were necessary to add to the Service Account, as all remaining functions were already contained in the `default` account's privileges.

5.3.10 Deploying the Testing Service

The final deployment of the Server application consists of

1. building and pushing the Server's container image into the internal container registry, and
2. applying the application's Template to the project.

The actual setup of the Template, described in Section 5.3.6, is simply executed by the following command:

```
$ oc process -f server.yaml | oc apply -f -
```

The first part of the command pipeline processes a Template configuration, stored in the `server.yaml` file and its output, which consists of the definition of the above mentioned objects, is then redirected to the `oc apply` command, which creates (or updates) these objects within the OpenShift Project.

5.3.11 Automated Deployment to MiniShift

For the purpose of making it easy to deploy the application into MiniShift for development and testing purposes, a Bash shell script was created. This script automates all previously mentioned steps.

As seen in Listing 11, the variable containing the GitLab OAuth token is supposed to be filled into the `$TOKEN_SECRET` variable by the script's user, after which all the commands are carried out automatically.

It's also worth mentioning, that the part of the script executing Docker commands utilizes MiniShift's macros (`minishift docker-env` and `minishift openshift registry`)

[12] A Service Account is an OpenShift object, used for authorization for various operations, such as Job, or Build execution. Unlike a regular User Account, it is not necessarily bound to a single user (i.e. person), but instead to an OpenShift Project's namespace. More information about different accounts in OpenShift available at https://docs.openshift.com/container-platform/3.6/dev_guide/service_accounts.html.

[13] More information about OpenShift's Role-based Access Control available in the official documentation, at https://docs.openshift.com/container-platform/3.11/admin_guide/manage_rbac.html.

```
TOKEN_SECRET='INSERT_TOKEN_HERE'
# Login and create the project in openshift
oc login -u developer -p developer
oc new-project fitest \
    --description="FITest Testing Service" \
    --display-name="FITest"
# Create the secret for gitlab operations
oc create secret generic gitlab-token \
    --from-literal=password=$TOKEN_SECRET \
    --type=kubernetes.io/basic-auth
# Gives the service account of the server access to job operations
oc policy add-role-to-user system:job-controller \
    system:serviceaccount:fitest:default
# Push the server's image
eval $(minishift docker-env)
docker login -u developer -p $(oc whoami -t) $(minishift openshift registry)
docker build -t $(minishift openshift registry)/fitest/fitest-server .
docker push $(minishift openshift registry)/fitest/fitest-server
# Apply the template to the project, deploying the server application
oc process -f server.yaml | oc apply -f -
```

Listing 11: Script for automated deployment of FITest to MiniShift.

to automatically direct the Docker daemon to its internal container registry. The `$(oc whoami -t)` command is used to return the currently logged in user's password for accessing the internal registry, which changes on every login.

5.4 Client Application

For easier interaction with the FITest API, a command-line application was implemented, using Python 3. The application itself is a script, which uses the `requests` module for issuing HTTP requests to the API, and the `argparse.Argparser` class for parsing the user's input. Each input command maps to a different API endpoint (see Section 5.3.5) and prints the parsed response from the Server. The mapping of the commands is as following:

- `$ client.py info project-name` uses `GET /api/project/project-name` to retrieve a project's information and print it in the console.
- `$ client.py create project-name` sends the `POST /api/project/` request, including the `project-name` value in the body of the request to create a project.
- `$ client.py build project-name` starts the build of a project's image with the `POST /api/project/project-name/build` request. If the `--status` flag is added after the `build` argument, the application instead retrieves a build's status by sending the `GET /api/project/project-name/build` request.
- `$ client.py test project-name` executes a project's build by invoking the `POST /api/project/execute` command. By default, no solutions are specified, meaning that all of the provided ones are going to be tested. If the `--solutions` flag is set,

and a comma separated list of values is provided (e.g. `--solutions=xample00,...`), the specified solutions are passed in the request's body to the Server as well.

If the `--stop` switch is added, the testing is stopped by calling `DELETE` on the `/api/project/project-name/execute` endpoint.

Furthermore, the `--cpu-limit` and `--mem-limit` switches can be used to limit the testing containers' maximum CPU time (in milicores) and memory (in megabytes), respectively.

- `$ client.py debug project-name` sends the `POST /api/project/debug` request to create a debugging session and prints the link to the debugging Pod's terminal's web interface, in the console. If the `--stop` flag is supplied, a debugging session is deleted instead (if one was active).
- `$ client.py results project-name` uses the `GET /api/project/project-name`, in a similar fashion as `info`, but only prints the URL to the latest test logs in the console.
- Lastly, the `$ client.py delete project-name` command can be used to delete a project, along with its Git repository by invoking the `DELETE` command on the `/api/project/project-name` endpoint.

The way the Client is implemented allows for very easy modifications to the code, meaning if a change of the API occurs, or new features are implemented in the feature, the application can be updated accordingly with minimal effort.

Chapter 6

Evaluation of FITest

During, and after the implementation of FITest, the service was being tested on multiple levels and after it has been finished, the final product was compared against the requirements defined in Chapter 4. Furthermore, this chapter mentions various error, that could manifest while using the service, caused by either minor design choices, or the OpenShift platform.

6.1 Compliance with the Requirements

The aim of the implementation was to fulfill the largest possible subset of the defined requirements defined in Section 4.1.

6.1.1 Features

The following features of the project were met, or not achieved as follows (in order of their definition):

1. User-defined automatic testing is the foundation of the service and was achieved the earliest. The user's requirements are defined by the `run.sh` script and allow them to use any functioning test suite, which runs in a Linux environment.

2. Identical test execution is attained by using the Executor Pods, replicated to the number of different solutions, which are being tested.

3. Based on point **1**, the test evaluation depends solely on the user's definition and allows for any kind of text-formatted output. However, FITest will still report a Failure status, if the test suite in execution had failed.

4. Results reporting from the command-line application was dropped due to complications it would bring into the scope of the project, as it would require further functionality regarding dynamic displaying of output in the terminal. As a substitute, the Client provides the URL to the results directly, in a project's Git repository, where the results can be browsed using GitLab's native file explorer, in an orderly fashion.

5. As mentioned in the previous point, the results for a project are accessible via its Git repository. Storage of the projects on the service's server was dropped in the course

of implementation, as no serious need for such functionality was found (and was instead fully replaced by using the Git storage).

6. A client application (see Section 5.4) for interaction with the service's API was implemented, which allows for the usage of all available API endpoints.

7. Isolation of tests was achieved by using separate Executor Pods for each test suite's run (see point **2**). These Pods are isolated from any other Executor Pods and are only accessing the shared Persistent Volume to retrieve the necessary testing data.

8. Customization of the test fixture is attained by using Docker images, built by the user, or by the OpenShift *Source to Image* capabilities, allowing for the building of container images directly from a project's Git repository, containing a Dockerfile, and, if necessary, any required build artifacts.

9. As illustrated by Figure 4.1, a user's responsibility is to maintain a repository containing a project's testing data, meaning they can supply the service with any test fixtures they deem valid, by pushing the test files to a project's Git repository.

10. Container resource limitation was implemented, by using the `limits` container attribute, as explained in Subsection 5.3.3.

11. Explicit separation of preliminary tests was dropped, due to the way the test execution is defined by the user. If a user desires to run multiple different test suites, they can simply define these steps in the `run.sh` script. This approach furthermore gives the user more control, in case they need these steps to be connected in any way.

12. The isolation of the test reports was partially achieved by changing the access right to the log file before and after the tests. However, the isolation of the test suites proved more difficult, due to variability of the test suites, and was not implemented for the sake of customizability of the test execution. Changing owner permissions to a non-specific set of files could result in the test suites not working as expected, or not functioning at all.

13. A Python Flask application was defined on top of the `Server` class (see Subsection 5.3.5), which implements a standard API protocol.

6.1.2 Permitted Activities

The activities defined in Subsection 4.1.2 were all made available thanks to the way the service was designed. Both activities **1** and **2** were achieved by having the option to choose a subset (even a single one) of solutions to be tested, or omitting this option, to test all solutions present in the repository. Activity **3** is made available by the debugging session functionality, described in Section 5.3, which creates a single debugging Pod and returns a URL for accessing the container to the user. Lastly, addressing activity **4**, the user can browse, and download any test results created by the service, via a given project's Git repository.

6.2 Testing and Validation

The service's testing was done using three main approaches:

- **Unit testing**: The source code's unit tests were implemented using Python's `unittest` module, alongside the development of the code, and are meant to be executed using the `pytest` module[1]. Each of the aforementioned classes (see Subsections 5.3.1, 5.3.2, 5.3.3 and 5.3.4) was tested using a dedicated test suite. These unit tests were executed both during the development of the modules and also, with the help of the Continuous Integration features of GitLab, where the source code is stored, automatically, whenever a new change was pushed into the repository. These tests often revealed errors introduced to the code by minor code changes and communication changes (e.g. how the classes report their errors and statuses between each other and to the user).

- **Integration testing**: Integration testing was done manually, in the progress of the development of the modules. The first to be verified was the functionality of the `Executor` class in the OpenShift environment. In this phase, the aforementioned example tests and configurations were used (See Subsections 5.2.1 and 5.2.2), as the dynamic generation of the settings was yet to be implemented. Next, when the `ConfigLoader` class, and the prototype for the `Server` were implemented, the correct behaviour of the code was ensured, along with the proper functionality of the Git operations. After the correct behaviour of the combination of these classes was assured, the `OpenshiftControl` class was added, and its integration was tested in two phases: first, verifying only the non-testing functions (i.e. build, info, project create, etc.), and secondly, including the test execution functionality, where the `Executor` class was used in Jobs. After the different functional elements of the Server were combined, the Flask application was implemented to incorporate the Server's functionality into an API. The application was tested via the network, using the aforementioned Postman platform (see Section 5.1). After the Client application had been created, its functionality was verified by comparing the output of Postman's calls to its output, and also checking the debugging logs of the Server. This phase of testing was able to expose errors caused by varying time spans taken by the Server's and OpenShift's operations, which often led to incorrect reporting, or command issuing from the Server's side.

- **Acceptance testing**: The acceptance phase of testing was carried out, using an anonymised set of student data and automated tests for the **Introduction to Programming Systems** course[2]. The provided SUTs contained simple programs made in the C programming language. A container image, based on Ubuntu Linux was created, in accordance with the test suite's requirements. Each tests test suite had an execution time of around 2 minutes and the entire testing, including Job starting and Git operations took only a little more than 2 minutes (multiple executions varied on the scale of seconds due to variables such as CPU load and network connection at the moment of execution).

- **Self-testing**: As a final proof-of-concept, a project for FITest's source code was created and the unit tests mentioned above were used as the tests for the execution. After a short configuration of the `run.sh` the service was able to test its own code, with expected results. During this part of testing, the main part of focus was the preparation

[1] Further information about the module available at https://docs.pytest.org/en/latest/.
[2] More information about the course available at https://www.fit.vut.cz/study/course/13376/.en

of the testing data. Multiple changes needed to be issued due to the way Python's `pytest` testing framework handles module importing, although the overall setup was still quite straightforward.

6.3 Known Issues and Possible Bugs

During the implementation, testing, and further research of the OpenShift documentation a few issues were revealed, that may result in incorrect, or inconsistent behaviour of the service.

Firstly, issues related to containers' creation and removal may lead to various reporting and functional issues. On occasions, multiple identical commands such as `build` may result in the service to attempt to create a new container for a given action, while the old one is still being removed. Thanks to integration testing, multiple checks for events like this were added to improve crash resistance of the service, however, there is still a possibility that similar errors may manifest in the future (e.g. if new functionality is added). As the service is meant to run in OpenShift, it provides an easy way to debug, and fix such errors by accessing OpenShift's Web Console interface, which allows for real-time container status and log inspection.

Secondly, errors resulting from the OpenShift configuration may reveal themselves when deploying the service to an OpenShift cluster (as opposed to MiniShift). These issues would be likely related to access rights or other limits issued by the administrator. For the most part, these faults can be solved by applying minor changes to the configurations, and/or changing the project's, or cluster's settings from the point of an administrator.

Lastly, in the state that the service is in as of writing the documentation, the only Git service, which FITest supports, is GitLab. The reason behind this is that token-based authentication, and repository management varies between different Git services, and the target Git server, which is used with the service in the future is also GitLab. If it was necessary to support multiple Git services later, code changes related to authentication and project management, which reflect this, would have to be added.

Chapter 7

Conclusion

During the course of this project, the FITest testing service has been designed, implemented, and tested. The service's desired functionality is working according to the requirements and is ready to be deployed into an OpenShift (OKD) cluster. It is designed to be used by a teacher, or a teacher assistant in order to automatically test a set of student's solutions to programming assignments. For the ease of use of FITest's API a command-line interface application was implemented, which allows the user to use all available API endpoints. Furthermore, a technical evaluation of the service was done, including a comparison of the service's features to the original requirements, as well as a description of its testing process. Testing of the service incorporated acceptance testing done using real, anonymised data from the Introduction to Programming Systems course.

It is important to note that this project does not provide a full solution to the problem, but instead sets a solid, functioning foundation for various future improvements and expansions. The state, in which the service is presented in this thesis, provides the means necessary to test student projects automatically, within dynamically generated test fixtures, and report the results of these tests. Examples for future enhancements of the service could include:

- A client web interface, used for interaction with the service.
- User-based authentication for the API.
- Separate error reporting interface, with file explorer capabilities.
- Division of the service into multiple micro-services, for an even larger degree of modularity and scalability.
- Increasing the Executor Pods' degree of isolation by injecting the tests and SUTs without using a shared Persistent Volume.

Furthermore, the reporting side of the service could be improved by using the *Markdown*[1] language, whose rendering is directly supported by various Git services, directly in their web interfaces. This means, that the tests' output could be formatted during its creation and shown directly in GitLab.

Another extension, which would allow the reports to be viewed by the students could work by creating a repository for each student, where the test output could be stored. This

[1] A simple markup language, developed by John Gruber. more information about the project available at https://daringfireball.net/projects/markdown/.

means, that along with updating the project's repository with the reports of test executions, the server would also push these results (for a given student), into a corresponding repository, into a subdirectory dedicated for the project. The students would only need to have read access to their repositories, which would allow them to view detailed test logs (including their older versions), without the necessity to request these from the teacher. This could be a nice improvement compared to the current situation, where the reports are either sent out by email or posted in the faculty's information system.

As it is difficult to predict the difficulty of implementing a unified testing process for various courses, which utilizes FITest, my personal hopes as the author are, that the service will continue to be expanded and improved. The potential of such a service is great, not only because of the automation but also due to the possibility of the service being usable by the students directly, providing immediate results and grading, and with multiple attempts, similarly to the Marmoset Project, mentioned in Subsection 3.1.1. In the future, the service can be published as an open-source project, meaning a wider circle of teachers, or students could be assisted and also included in the creative process. This approach could also bring new ideas and experiences, and out-of-the-box solutions and expansions to FITest.

Bibliography

[1] Amazon Web Services. *Amazon Web Services (AWS) - Cloud Computing Services.* 2019. [Online; visited 02.09.2019]. Available at: https://aws.amazon.com.

[2] Babu, A., Hareesh, M., Martin, J. P., Cherian, S. and Sastri, Y. System performance evaluation of para virtualization, container virtualization, and full virtualization using xen, openvz, and xenserver. In: IEEE. *2014 Fourth International Conference on Advances in Computing and Communications.* 2014, p. 247–250.

[3] Bender RBT Inc.. *Requirements Based Testing Process Overview.* 2009. [Online; visited 23.10.2019]. Available at: http://benderrbt.com/Bender-Requirements%20Based%20Testing%20Process%20Overview.pdf.

[4] Chacon, S. and Straub, B. *Pro Git.* Apress, 2014. ISBN 978-1484200773.

[5] Cholewa, T. *10 most important differences between OpenShift and Kubernetes* [online]. 2019 [cit. 2019-11-10]. Available at: https://cloudowski.com/articles/10-differences-between-openshift-and-kubernetes/.

[6] Codeboard.io. *Codeboard - the IDE for the classroom* [online]. [cit. 2019-11-04]. Available at: https://codeboard.io/.

[7] Docker, Inc. *About Docker Engine - Community* [online]. 2019 [cit. 2019-11-07]. Available at: https://docs.docker.com/install/.

[8] Docker, Inc. *About images, containers, and storage drivers* [online]. 2019 [cit. 2019-11-11]. Available at: https://docs.docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers/.

[9] Docker, Inc. *Docker Kubernetes Service (DKS)* [online]. 2019 [cit. 2019-11-07]. Available at: https://www.docker.com/products/kubernetes.

[10] Docker, Inc. *Docker overview* [online]. 2019 [cit. 2019-11-07]. Available at: https://docs.docker.com/engine/docker-overview/.

[11] Friis, M. *Announcing Docker Enterprise Edition* [blog post]. Docker Blog, 2017 [cit. 2019-11-07]. Available at: https://www.docker.com/blog/docker-enterprise-edition/.

[12] Google Inc.. *Deployment* [online]. 2019 [cit. 2020-01-12]. Available at: https://cloud.google.com/kubernetes-engine/docs/concepts/deployment.

[13] Help, S. T. *What Is Software Testing Life Cycle (STLC)?* 2019. [Online; visited 24.10.2019]. Available at: https://www.softwaretestinghelp.com/what-is-software-testing-life-cycle-stlc/.

[14] Henry, W. *Podman and Buildah for Docker users* [blog post]. 2019 [cit. 2019-11-08]. Available at: https://developers.redhat.com/blog/2019/02/21/podman-and-buildah-for-docker-users/.

[15] Hooda, I. and Chhillar, R. S. Software test process, testing types and techniques. *International Journal of Computer Applications*. Citeseer. 2015, vol. 111, no. 13.

[16] Jorgensen, P. C. *Software testing: a craftsman's approach*. 3rd ed. Auerbach Publications, 2013. ISBN 9780429184574.

[17] Katherine, A. V. and Alagarsamy, K. Conventional software testing vs. cloud testing. *International Journal Of Scientific & Engineering Research*. Citeseer. 2012, vol. 3, no. 9, p. 1.

[18] Lexico. *API: Definition of API by Lexico* [online]. Lexico Dictionaries [cit. 2019-10-27]. Available at: https://www.lexico.com/en/definition/api.

[19] Massey, V. and Satao, K. J. Comparing Various SDLC Models And The New Proposed Model On The Basis Of Available Methodology. *International Journal*. 2012, vol. 2, no. 4.

[20] Menage, P. *CGROUPS* [online]. [cit. 2019-11-06]. Available at: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[21] Meszaros, G. *XUnit test patterns: Refactoring test code*. 1st ed. Pearson Education, 2007. ISBN 978-0131495050.

[22] Microsoft. *Microsoft Azure Cloud Computing Platform & Services*. September 2011. [Online; visited 02.09.2019]. Available at: https://azure.microsoft.com/en-us/.

[23] Mimir. *Mimir Classroom* [online]. [cit. 2019-11-04]. Available at: https://www.mimirhq.com/.

[24] Myers, G. *The Art of Software Testing*. 1st ed. John Wiley & Sons, Inc., 1979. ISBN 0471043281.

[25] npm, Inc.. *Npm | build amazing things* [online]. [cit. 2019-11-05]. Available at: https://www.npmjs.com/.

[26] Pallets. *Flask Documentation* [online]. 2010 [cit. 2020-4-21]. Available at: https://flask.palletsprojects.com/en/1.1.x/.

[27] Petazzoni, J. *Anatomy of a Container: Namespaces, cgroups & Some Filesystem Magic - LinuxCon* [online]. 2015 [cit. 2020-4-21]. Available at: https://www.slideshare.net/jpetazzo/anatomy-of-a-container-namespaces-cgroups-some-filesystem-magic-linuxcon.

[28] Python Software Foundation. *Openshift 0.10.1* [online]. [cit. 2019-12-31]. Available at: https://pypi.org/project/openshift/.

[29] Python Software Foundation. *Python 3.7.6 documentation* [online]. [cit. 2019-12-31]. Available at: https://docs.python.org/3.7/.

[30] Python Software Foundation. *Virtual Environments and Packages* [online]. [cit. 2019-11-05]. Available at: https://docs.python.org/3/tutorial/venv.html.

[31] Red Hat, Inc.. *Chapter 1. Introduction to Control Groups (Cgroups)* [online]. [cit. 2019-11-06]. Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01.

[32] Red Hat, Inc. *Creating Images: Guidelines* [online]. 2019 [cit. 2019-11-12]. Available at: https://docs.openshift.com/container-platform/3.11/creating_images/guidelines.html.

[33] Red Hat, Inc. *Developer guide: Build Inputs - Builds* [online]. 2019 [cit. 2019-11-12]. Available at: https://docs.openshift.com/container-platform/3.11/creating_images/guidelines.html.

[34] Red Hat, Inc. *Developer guide: Builds - Triggering Builds* [online]. 2019 [cit. 2020-04-17]. Available at: https://docs.openshift.com/container-platform/3.11/dev_guide/builds/triggering_builds.html.

[35] Red Hat, Inc. *Developer guide: How Deployments Work* [online]. 2019 [cit. 2019-11-12]. Available at: https://docs.openshift.com/container-platform/3.11/dev_guide/deployments/how_deployments_work.html.

[36] Red Hat, Inc. *Developer guide: Projects* [online]. 2019 [cit. 2019-11-12]. Available at: https://docs.openshift.com/container-platform/3.11/dev_guide/projects.html.

[37] Red Hat, Inc. *Developer guide: Quotas and Limit Ranges* [online]. 2019 [cit. 2019-11-12]. Available at: https://docs.openshift.com/container-platform/3.11/dev_guide/compute_resources.html.

[38] Red Hat, Inc. *Image Registry* [online]. 2019 [cit. 2019-11-12]. Available at: https://docs.openshift.com/container-platform/3.11/architecture/infrastructure_components/image_registry.html#integrated-openshift-registry.

[39] Red Hat, Inc. *Kubernetes on CoreOS: Overview of a Service* [online]. 2019 [cit. 2019-11-11]. Available at: https://coreos.com/kubernetes/docs/latest/services.html.

[40] Red Hat, Inc. *Okd: The Origin Community Distribution of Kubernetes that powers Red Hat OpenShift.* [online]. 2019 [cit. 2019-11-12]. Available at: https://www.okd.io/.

[41] Red Hat, Inc. *Okd Minishift: Develop Applications Locally in a Containerized OKD Cluster* [online]. 2019 [cit. 2019-11-12]. Available at: https://www.okd.io/minishift/.

[42] Red Hat, Inc. *OpenShift Container Platform 4.2 Documentation* [online]. 2019 [cit. 2019-11-11]. Available at: https://docs.openshift.com/container-platform/.

[43] Red Hat, Inc. *OpenShift Container Platform architecture* [online]. 2019 [cit. 2019-11-11]. Available at: https://docs.openshift.com/container-platform/4.2/architecture/architecture.html.

[44] Red Hat, Inc.. *What's a Linux container?* [online]. 2019 [cit. 2019-11-06]. Available at: https://www.redhat.com/en/topics/containers/whats-a-linux-container.

[45] Red Hat, Inc. *Developer guide: Build Inputs* [online]. 2020 [cit. 2020-04-26]. Available at: https://docs.openshift.com/container-platform/3.11/dev_guide/builds/build_inputs.html#source-code.

[46] Refsnes Data. *Python Introduction* [online]. [cit. 2019-12-31]. Available at: https://www.w3schools.com/python/python_intro.asp.

[47] Rosen, R. *Linux Containers and the Future Cloud* [online]. 2014 [cit. 2019-10-10]. Available at: https://www.linuxjournal.com/content/linux-containers-and-future-cloud.

[48] Shaw, K. What is a hypervisor? [online]. Network World. 2017, [cit. 2019-11-05]. Available at: https://www.networkworld.com/article/3243262/what-is-a-hypervisor.html.

[49] Sikeridis, D., Papapanagiotou, I., Rimal, B. P. and Devetsikiotis, M. A Comparative taxonomy and survey of public cloud infrastructure vendors. *ArXiv preprint arXiv:1710.01476*. 2017.

[50] Skoković, P. and Rakić Skoković, M. Requirements-based Testing Process in Practice. *International Journal of Industrial Engineering and Management (IJIEM)*. 2010, vol. 1, no. 4, p. 155–161.

[51] Spacco, J., Pugh, W., Ayewah, N. and Hovemeyer, D. The Marmoset project: an automated snapshot, submission, and testing system. In: Citeseer. *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 2006, p. 669–670.

[52] Tahat, L. H., Vaysburg, B., Korel, B. and Bader, A. J. Requirement-based automated black-box test generation. In: IEEE. *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*. 2001, p. 489–495.

[53] Techopedia. *What is the Software Development Life Cycle (SDLC)?* [online]. 2019 [cit. 2019-10-27]. Available at: https://www.techopedia.com/definition/22193/software-development-life-cycle-sdlc.

[54] The Containers Organization. *Podman* [online]. 2019 [cit. 2019-11-08]. Available at: https://podman.io/.

[55] The Kubernetes Authors. *Jobs - Run to Completion* [online]. [cit. 2019-12-08]. Available at: https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/.

[56] The Kubernetes Authors. *Concepts* [online]. 2019 [cit. 2019-11-12]. Available at: https://kubernetes.io/docs/concepts/.

[57] The Kubernetes Authors. *Kubernetes concepts: Images* [online]. 2019 [cit. 2019-11-12]. Available at: https://kubernetes.io/docs/concepts/containers/images/.

[58] The Kubernetes Authors. *Kubernetes concepts: Persistent Volumes* [online]. 2019 [cit. 2019-11-12]. Available at: https://kubernetes.io/docs/concepts/storage/persistent-volumes/.

[59] The Kubernetes Authors. *Kubernetes concepts: Service* [online]. 2019 [cit. 2019-11-11]. Available at: https://kubernetes.io/docs/concepts/services-networking/service/.

[60] The Kubernetes Authors. *Kubernetes concepts: Volumes* [online]. 2019 [cit. 2019-11-12]. Available at: https://kubernetes.io/docs/concepts/storage/volumes/.

[61] The Kubernetes Authors. *Pod Overview* [online]. 2019 [cit. 2019-11-11]. Available at: https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/.

[62] The Kubernetes Authors. *Production-Grade Container Orchestration* [online]. 2019 [cit. 2019-11-08]. Available at: https://kubernetes.io/.

[63] The Kubernetes Authors. *What is Kubernetes* [online]. 2019 [cit. 2019-11-08]. Available at: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[64] University of Maryland. *The Marmoset Project* [online]. [cit. 2019-11-03]. Available at: http://marmoset.cs.umd.edu/index.shtml.

[65] xUnit Patterns. *Test context* [online]. [cit. 2019-12-11]. Available at: http://xunitpatterns.com/test%20context.html.

[66] xUnit Patterns. *Test fixture (in xUnit)* [online]. [cit. 2019-10-29]. Available at: http://xunitpatterns.com/test%20fixture%20-%20xUnit.html.

Appendix A

Contents of the Attached Storage Device

The attached storage device contains the following directory structure:

```
/
├── fitest/ ..... Directory containing the service's source code and configurations.
├── report_src/ ..... Directory containing all source files of this technical report.
├── README.md ..... Formatted information about the contents of the folders.
├── readme.txt ..... Non-formatted information about the contents of the folders.
└── report.pdf ..... This technical report, in PDF format.
```