Tampere University Unit of Computing Sciences

TIEA4 Project Work (City centre campus)

TIETS19 Software Project Management Practice (City centre campus)

Software component manager

Test Report

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | 424990, Jyry Uitto | | 431258, Petri Nieminen  427770 Jani Aakio  439656, Kaisa Sormunen  K427803 Pekka Oinas  K427534, Sanni Onkalo  K439136, Denis Kole | |  |
|  |  |

Version history

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Description |
| 1.0 | 23.05.2021 | Jani Aakio | The report |

Contents

1 Introduction 4

1.1 Purpose and scope of document 4

1.2 Product and environment 4

1.3 Project constraints related to testing 4

1.4 Definitions, abbreviations and acronyms 4

2 Testing process 5

2.1 General approach 5

2.2 Testing roles 5

2.3 Test schedule 5

2.4 Test documentation 5

3 Testing Tools 5

4 Test cases and results 5

4.1 Test results 6

4.2 Unit testing 6

4.3 Integration testing 6

4.4 System testing 6

4.5 Special testing 6

4.6 Acceptance testing 6

4.7 Xyz (of your choice..) 6

APPENDIX A […Z] 6

# Introduction

## Purpose and scope of document

This document will discuss the methods that were used to tests the software component manager.

## Product and environment

The product is: Software component manager and it is run on docker-platform. The team uses basic docker-setup for the services and will deliver these docker-images to the customer who may use them in other types of environments.

## Project constraints related to testing

The product does have multiple constraints when it comes to testing:

* There is confidential information to be stored in the project database when it is taken into use. As such, we are not able to test with “real-data” and are constrained to use our own ideas on what the data would look like.
* We offer a UI to the customer but lack the time/resources to test graphical UI’s programmatically. As such we are restricted to basic user-testing with our graphical UI, which may lead to some bugs getting through easier. This issue was discussed with the customer and he gave us permission to limit our testing with our UI.
* Limited experience with docker and the programming-languages used being unfamiliar with the team-members also pose a problem. Hopefully this can be overcome during testing implementation.

## Definitions, abbreviations and acronyms

UI User interface

Docker Environment that virtualizes packages (containers) to be used as services

# Testing process

As the team has very little experience with the language used, we have no-one to give us guidance on these languages and as such, no specific style guides were used. Common sense was expected from the codebase.

## General approach

The team will try to do as much automated testing as possible. This is done due to the fact that such tests are more reliable (less human errors) and as a learning experience. There are three distinct services in the system and each will have a different approach:

* UI/View (Angular) container has very minimal automated testing due to the lack of resources. The testing will be done when pull requests are made. When a pull request is made, the one approving it should try the new or fixed implementations that they work correctly.
* Controller container (Scala) has the most automated testing. This includes unit testing of most high-level functions. These tests should try to test that the functions work properly and give errors if given bad data. Also should try to make sure that panics do not happen in the code. Integration tests that tests the service from the outside are to be created. These use the containers RESTAPI to test that the container works as it is expected to work when used with the RESTAPI
* Model (Postgres) container has very minimal testing. This is a pre-made component and only acts as a database for the service. As such it does not have much implementation to be tested. Schemas and such are tested within the Scala-containers tests and do not need to be re-tested.

When doing manual testing, the containers are always ready to be tested as the database is filled with some example data so the testing is very easy and almost no setup is necessary. Because of this test-data we can immediately see if there is some problem with displaying or inserting data.

## Testing roles

No specific roles have been assigned. Every developer is in charge of his/her own code and the code that the approve to be merged. This should encourage the developers to be careful with the code that they merge into the product

## Test schedule

As implementation is to be tested as it is implemented. No specific schedule is made for testing. It is expected that the code that is good when it is merged and testing should be done when it is being inspected.

It was mostly the manager Jani who approved the code to the repository. As such most of the end-testing was done by him. The code was expected to pass integration-tests and if it didn’t, it was discussed why this particular part was not passing.

UI-testing was done when a pull-request came in. The branch was pulled and the the UI was brought up in docker. The tester would then test the new implementation and the old to see that no problems were apparent.

## Test documentation

Github will show testing results for every pull request that was made.

# Testing Tools

No specific tools were used. An own testing docker file was built which ran integration tests automatically in our CI pipe. It used pytest to run tests on our API endpoints to see that it behaved as expected

# Test cases and results

## Test results

As discussed earlier, UI tests were done when a pull request was done. Often no problems were found and if there were problems, they were very quickly fixed.

The integration tests in CI pipe were sometimes not passing, but when this did happen, the fixes were usually done immidiatelly.

## Unit testing

Unfortunately, the unit tests have not yet been implemented at the time of writing this document. They are quite far but not yet ready for implementation

## Integration testing

Integration testing ran tests for every endpoint available. It sent a request for those endpoints and made sure by checking the database that the endpoint was giving correct information. (as most of the endpoints deal with database queries or insertions)