OULUN AMMATTIKORKEAKOULU

Arttu Mäkelä

AUTOMATED SOFTWARE TESTING FOR ANTVIEW

AUTOMATED SOFTWARE TESTING FOR ANTVIEW

Arttu Mäkelä
Bachelor's Thesis
Autumn 2021
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Information Technology, Option of Software Development

Author: Arttu Mäkelä
Title of Bachelor's thesis: Automated Software Testing for AntView
Supervisor: Teemu Korpela
Term and year of completion: Autumn 2021 Number of pages: 30

This thesis was commissioned by Verkotan Oy. Verkotan Oy provides testing and consulting services in the wireless industry. AntView is a tool for visualizing, analysing and storing antenna measurement data via web user interface written with the JavaScript's React library. It uses the Django web framework and MongoDB in its backend.

The aim of the thesis was to provide Verkotan's AntView software with an automated testing pipeline.Test scripts were created with Python to test AntView's Python API and the underlying parameter calculation functions. An automation server using Jenkins and Docker was also set up for AntView to be built and tested. This server pulls, builds and tests the application by using Jenkins, and while deployed it can be used as a platform for manual testing and test development.

The result of the thesis was a functioning Jenkins automation server operating on a Debian 10 host along with Python test scripts testing AntView and its Python API. Test results are viewed via Jenkins' web user interface. This project was a good opportunity to learn about testing in software development, Linux, Docker, Jenkins and about the concept of continuous integration and continuous development.

Keywords:

Docker, Jenkins, software testing, Python

CONTENTS

1 INTRODUCTION

This thesis was commissioned by Verkotan Oy. Located in Oulu, Finland, Verkotan provides testing and consulting services in the wireless industry.

The aim of this thesis was to provide Verkotan's AntView software with an automated testing environment along with tests to test AntView's Python API and some of the core functionalities of the software through this API. AntView is a tool for visualizing, analysing and storing antenna measurement data via a web user interface written with JavaScript's React library. It uses the Django web framework and MongoDB in its backend.

The Python test scripts were made to test aspects, such as parameter calculation functions and data validation, so that the backend stores and sends data in an intended format, and general API functionality, such as login and sending or deleting measurements. All tests were testing the fully built software in an environment similar to a released product.

The thesis also included the setup of a Jenkins automation server on Debian 10 (a Linux distribution). This was implemented using Docker to install Jenkins and any required software to a Docker container. This container included Docker itself in order to launch AntView's containers from within the Jenkins container. Once Jenkins was set up on the server, a chain of jobs was created to pull, build and test AntView by using Jenkins' web GUI. This testing pipeline is executed by a button press on the GUI.

This pipeline was also partially tested using a pipeline script to run all the stages. In this method, all the information regarding the pipeline is contained in a single file. This file can then be deposited to version control and Jenkins can be instructed to pull this file and run it. This method becomes more useful with a more complex pipeline and larger teams.

2 TESTING IN SOFTWARE DEVELOPMENT

2.1 Testing Methods

Testing can be manual or automated. In automated testing, tests are executed by scripts. Most popular programming languages offer built-in testing frameworks that help developers to automate their testing. Testing is also categorized depending on the testing target. The target of a test can vary from a single function to testing of data flow between modules in a built software. (1)

2.1.1 Unit Testing

Unit tests aim to test the smallest individual components of a software from simple functions and methods to larger modules. A unit test expects a function with a known fixed input e.g., a string literal “expected input”, to return an expected value (2). Unit tests are relatively fast to execute and are always executed before the more taxing and time-consuming integration and system tests. The correct order of executing the different types of tests is illustrated in figure 1 below.

By executing the unit tests first in the testing pipeline, various errors can be caught before application is deployed to later testing phases, saving time and money (3). Without unit testing, a simple defect in later testing phases requires more time investment to debug. Implementing unit tests can give the developers more certainty that their functions and methods are working as intended. Sometimes a line of code which appears to be unrelated to a function somehow manages to break it. While unit testing cannot catch every possible error, with proper implementation it can give a good coverage.

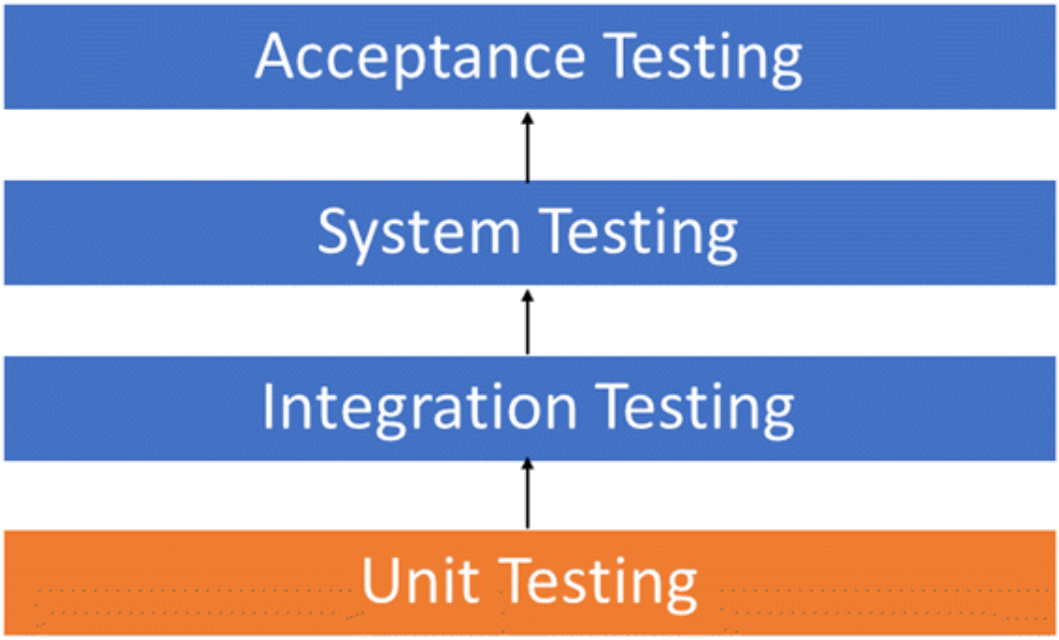

FIGURE 1. Unit testing scripts are always executed first in pipeline. (3)

In essence a unit test is an assertion, e.g. a simple add-function taking two variables a and b, then adding them together. To test this function, a unit test would assert that the inputs a and b return an expected value of the sum of a and b. Following a successful test, a simple message to indicate success is enough for information. In case of Python, a simple dot represents a successful test in standard testing framework. When a test fails, an error message is given usually in the form of an assertion error, meaning that the function did not return the output that was expected in the test.

A developer can write their code in a particular way to accommodate unit testing. Unit testing encourages developers to keep their functions short and focused. Errors in a large function that is not cut down into smaller pieces are often more time consuming to debug and the error reports from unit testing frameworks cannot be very specific on what section of the function failed. Another way to accommodate testing is to use Test-Driven-Development (TDD), where tests are written before actual implementations of functions. (2)

If a function to be unit tested requires external factors, such as a website, to be available for requests, a concept called mocking is required to test these functions properly. Mocking allows the developer to “mock” the wanted response from that website regardless of the website’s state. It is

important to test that the function sends a request to the website with the specified URL and behaves as expected, without needing to worry about factors outside of the developer's control. (5)

2.1.2 Integration Testing

Integration testing verifies that two or more individual components work together as expected. While unit testing tests individual functions, integration testing focuses more on the flow of data through the application and how the software modules work together (4). As an example, one of the tests for AntView sends a standard measurement to the database and requires the backend to be running. The data is then fetched back, and each metadata field is compared to the sent measurement, checking for a data type and value.

Integration testing is especially important in a large project, where each software module is developed by a different developer. The programming practices and logic of each developer can differ, which increases the likelihood of conflicts with the links between modules. (4)

2.1.3 System Testing

System testing is the process of testing the complete, fully integrated application with all its modules and possible peripherals in place. It is comparable to an end-user using the application. It is performed after unit testing and integration testing phases and before the software is released to the market. System testing is more disconnected from the source code than unit or integration testing. It focuses on usability, performance, recovery (e.g. from a crash) and functionality. (14)

Usability testing focuses on the end-user's experience with the software; UI testing and how easy or logical it feels to use, the appearance and responsiveness of the software and user friendliness. The UI efficiency is tested by evaluating the effort required to navigate to different parts of the UI. Usability testing is advised to start as soon as possible in development, since the feedback from this phase will help steer the development in the right direction and any glaring problems with the UI and its efficiency can be identified and fixed at the early phase. (14)

Load, performance and stress testing is done to ensure that the application works under real-life conditions and high stress. It is used to identify performance bottlenecks and the maximum concurrent users the application can support. It is especially relevant in client/server and web-based applications such as online games, where peak online users can vary heavily from the average. A release of a new online game can fail catastrophically if proper load testing has not been done, and the peak user amount has not been assessed correctly. (14)

2.1.4 Manual Testing

Certain parts of an application can be difficult to test automatically. Manual testing is usually a requirement to properly test some of the features, such as how the UI behaves, looks and feels on different devices. Certain features simply cannot be reliably tested through automation. Human perspective also helps spot usability issues within an application. Automated testing spots defects mainly based on an input/output behavior, whereas manual testing can spot more nuanced issues with the application such as how the UI components are aligned or how intuitive it feels to use. This kind of testing can be considered usability testing, which aims to evaluate a product from an end-user's perspective and will likely stay relevant regardless of how efficient automated testing becomes. (6)

Before starting to automate certain tests in AntView, manual testing of the software was done before starting thesis on automation. Test cases were written from an end-user's perspective and then executed. Based on the tests, bugs and usability problems were reported. This was helpful to get a good understanding of the application and to also appreciate both the manual and automated side of software testing. With comprehensive documentation, manual testing can also be outsourced. An outsider's view of the application often helps the developer to identify issues that never occurred before. Often the designer of the application uses it exactly as they intended, which is often not ideal when testing.

2.1.5 Security Testing

Security testing is a term that covers multiple different methods related to the security of a software and is especially important in today's web-application heavy environment. It focuses on identifying vulnerabilities in software and any architecture related to it such as networks and databases. Below are described the different security testing methods and some of the most prominent security threats today (Figure 2).

Vulnerability scanning is usually done with an automated scanning tool. It can identify known security loopholes in the targeted system. Penetration testing simulates a cyber-attack against the applications network system and is usually done by a cyber-security professional to expose the vulnerabilities of the system. It is effectively a form of "ethical hacking", where the hacking is authorized in order to provide information about potential security flaws. (16)

Risk assessment explores the security risks of the system and categorizes the threats based on the severity of the threats to low, medium and high levels. The goal is to get an idea of the most critical threats and proceed to develop a strategy to control them. (16)

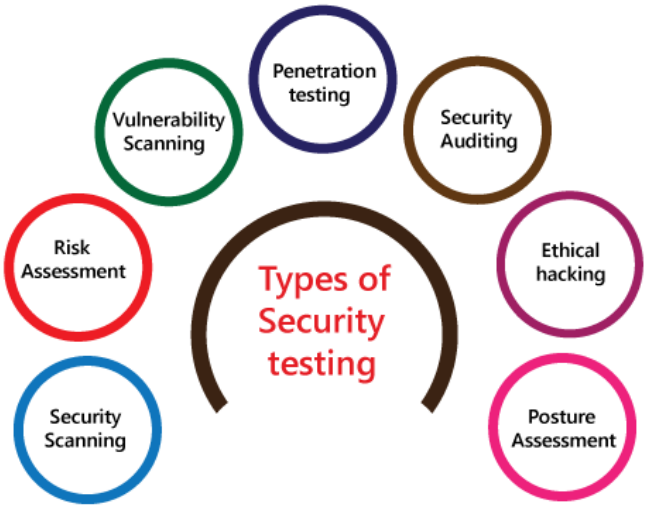

FIGURE 2. The different ways to test the security of software. (16)

The OWASP Foundation has compiled a list of the most threatening security vulnerabilities in today's web-applications (Figure 3). The most prominent ones are introduced below to get an idea of the threats that security testing is fighting against. (17)

An injection flaw is a vulnerability where an input (e.g. from a login page) is interpreted and processed as a query in the database. These injections can grant hackers access to private user data and compromise the entire database. Input validation and parameterized queries can be used to reduce the chance for injection flaws. (18)

Incorrectly implemented authorization and authentication functions can allow hackers to gain access to sensitive information such as user passwords and session tokens. Cross-site scripting can enable attackers to execute scripts in the user's browser that can hijack user sessions and redirect users to malicious websites. (18)

Taking these threats into account in all stages of software development and utilizing the appropriate testing methods and expertise will increase the security of applications.

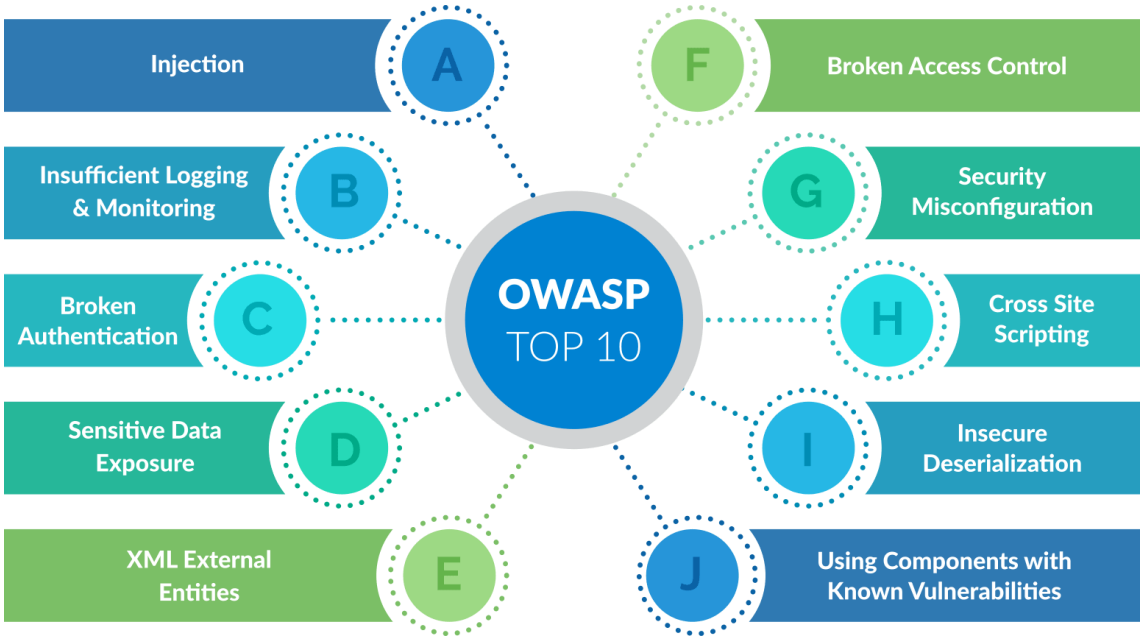

FIGURE 3. OWASP top security threats of web applications. (19)

2.2 Testing Practices

Each test function in a test class should be independent, designed to not interfere with or rely on any other tests in the class. The test runner component, which is the component that executes the tests and provides the results to the user, does not run the tests of a class in any particular order (7).

It is recommended to avoid any kind of randomness in a test. Test inputs should be fixed and isolated from outside factors and should not fail because of factors such as website availability or randomly generated numbers. (2)

Tests should ideally have only one assertion (2). If a single test has multiple assertions and a test fails, it is not immediately clear what part of the test failed. A test also stops executing when an assertion error is raised, which means that if the first assertion of a test fails, there will be no information on the subsequent assertions. This was initially an issue in a test class that was made. To test multiple data fields of a measurement, a list of fields was looped through and an assertion was made for each one. If there is an assertion error with multiple fields, with this looping method the test would have to run and fail multiple times to find each error. When these tests are separated and run independently, entire class can be tested in one execution and error reports will be more descriptive.

The fastest tests should be run first. If an application is deployed to a test environment ,such as a Jenkins server, it should execute any test scripts that can be run before building the application. If any of these tests fail, the build process is not even required. Overall, if tests are quick to execute, developers are more likely to run them. (2)

Naming convention of tests differs from normal functions. Since these test functions are never called in this code, their names can be longer and more descriptive. It should be known what is being tested from the function name itself. When a test fails, the name of that test is the first thing a user sees in the error report. The more descriptive the name of that test is, the easier it is to fix the problem. (8)

3 TECHNOLOGIES IN PROJECT

3.1 Python's unittest module

Automated testing for AntView was done using Python and its standard unittest module. Unittest is a testing framework inspired by Java's JUnit. It provides the user with several helpful tools such as setup, teardown methods and assertions to check for conditions. Each testcase created by this module is an instance of the TestCase class. Every test method starts with the word "test" to indicate that the method is a test. (7)

Multiple test methods often share a setUp method, which allows the user to specify instructions that are to be executed before each test method in that class. The test class itself can also have a setup, called setUpClass. This class setup is shared with each test method and is executed once. Both the setUp and setUpClass methods are accompanied by tearDown and tearDownClass methods, which are used to do any required cleanup after a test. Without setup methods the tests and their individual setups could easily become very repetitive. In one of the test classes that was made, every individual test needs to generate a unique device ID that does not already exist in the database. Running this in the setup method of the class makes the code a lot easier to read and maintain. (7)

The Unittest module provides many assertions to check for various conditions. The most common one is assertEqual, which checks two values for equality. AssertTrue (or False) checks that an expression is true. One of the more useful ones, specifically in testing AntView's Python API, was the assertRaises, which checks for proper raising of errors. Many of the tests in the project are checking that certain errors are raised, for example, when data is attempted to be sent in wrong format. Figure 4 shows an example of a test using assertRaises, which verifies that AntView's Python API returns the correct error message when attempting to delete an already delete measurement.

```
def test_delete_invalid_measurement(self):

    db_meas = self.c.insert_measurement(self.meas)
    self.c.delete_measurements([db_meas['id']])

    with self.assertRaises(RuntimeError):
        self.c.delete_measurements([db_meas['id']])
```

FIGURE 4. A test using assertRaises.

3.2 Docker

Docker is an open platform that packages software into containers which can run reliably in any Linux based environment. These containers run in their own loosely isolated environments, where each container has their own set of software and packages (9). These containers do not run in their own virtual OS. Instead they share the same Linux kernel, which is responsible for interacting with the hardware. This approach allows packaging, building and running software in an environment that is isolated enough for most developing needs, while also eliminating the need for setting up multiple virtual environments which take longer to start and require large amounts of host OS memory and overall resources. Figure 5 illustrates the difference between running software in containers and running separate virtual environments. (10)

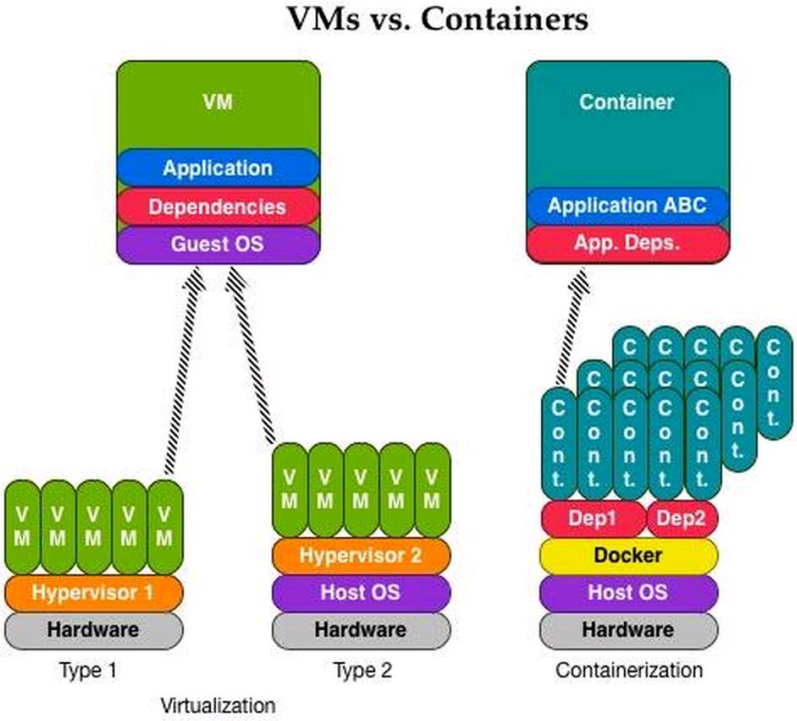

FIGURE 5. The differences between virtualization and containers. (11)

To run software in a docker container, the first requirement is an appropriate Linux OS, either in a physical or virtual environment. To build a container, a dockerfile and a docker image are also needed. A dockerfile is a list of instructions on what software and packages the container is supposed to have. This dockerfile is then built into an image using the “docker build” command. Images can then be run by the “docker run” command, becoming containers running in their own environment sharing the Linux kernel with other containers. (9)

Docker images can be pulled from and hosted in a public Docker repository called the Docker Hub. For example, it is possible to pull the base image of a Linux OS of our choice, use it as the base image in our own dockerfile and add more software on top of the base image. The Docker client communicates with the Docker daemon, which listens to Docker API requests (e.g. docker pull) and manages docker objects, such as images, containers, networks and volumes mapped to the containers. (9)

Docker Compose is a tool for defining, managing and running multi-container applications. Running and managing multiple containers and their run commands can get tedious very quickly. A Docker Compose file is used to organize common docker commands to a more structured and manageable format, where variables, such as the different services (e.g. a list of containers), networks, volumes, environment variables and ports etc. are separated. The entire application and all the required services can be started using this single file. (15)

Docker is effectively virtualizing the software without virtualizing the OS, eliminating possible version or package conflicts that often emerge in ordinary deployment of software. It allows the developer to focus more on developing rather than troubleshooting conflicts during deployment. Docker itself did not invent the concept of containers, but as a high-level tool accessing the Linux kernel, it made the containerization considerably easier and more accessible (10).

3.3 Jenkins and Continuous Integration

Jenkins is an open-source automation server capable of automating various tasks related to building, testing and deploying software. It is written in Java and can be installed on systems running the Java Runtime Environment (JRE) and also with Docker on Linux systems (12).

Jenkins is often an integral part of the continuous integration (CI) of applications. Continuous integration is the process of integrating changes in the project as often as possible. This approach keeps the project up to date and errors can be identified at an early phase of development. In the earlier days of software development, developers would work on their part of a project considerably longer before the changes would be combined in an integration server. This approach would very likely cause conflicts that take considerably more time to solve compared to using CI. CI also supports the popular Agile software development method since it allows teams to quickly react to requests from customers. (13)

When Jenkins server is installed, it produces a web GUI for the user to easily manage everything related to that Jenkins instance. Through this GUI it is possible to create jobs to automate specific tasks, such as pulling repositories from a Git server, execute Linux commands and automate installation of applications and test scripts. Below figure shows the opening page of AntView's Jenkins (Figure 6).

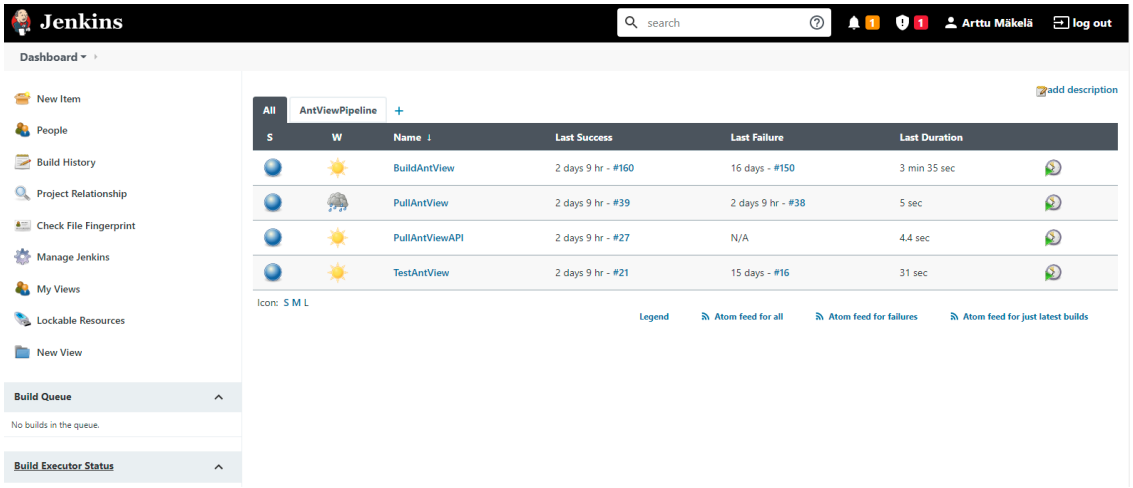

FIGURE 6. Opening page of Jenkins web GUI for AntView.

Jenkins' usefulness is extended through its extensive collection of plugins, which allows to customize Jenkins to better suit users' needs. The Build Pipeline plugin helps to get a better visual representation of the actual jobs that are being chained. Source control management plugins , such as Git can almost be considered part of standard Jenkins itself. There are also many variants of testing metrics and monitoring plugins that help to document and analyze the performance of tests. Many of the useful plugins, such as Git are recommended to be installed automatically after the Jenkins installation is complete.

3.4 Linux

Linux is an ecosystem of free and open-source UNIX base operating systems developed initially by Linus Torvalds in 1991. While in the beginning it was created mainly for use in PCs, it became the ideal operating system of choice for servers and embedded systems. Even Android OS is based on the Linux kernel. (20)

Linux is community driven and free to be modified and distributed. It is a highly secure operating system with a several popular distributions (e.g. Debian, Ubuntu) to choose from. It is flexible, lightweight and stable compared to Windows, making it an ideal operating system for programmers and servers. It does not hold the user's hand, which is an advantage in many cases, for example when servers need to be running constantly without interruptions. Updates and installations are easily done through the shell which communicates with the kernel, allowing users to also containerize applications with Docker. (20)

4 RESULTS

Debian distribution of Linux is used in this project for the Jenkins server and development. AntView is a multi-container, dockerized application which requires a Linux environment. For developing purposes, Debian is run as a virtual operating system using Oracle VM VirtualBox. The Jenkins server runs on a physical Debian installation and is accessed remotely via the SSH (Secure shell) connection.

4.1 Tests

4.1.1 Overview of the tests

The tests are written using Python's unittest module, importing AntView's Python API to carry out many of the requests to the software backend. NumPy, requests, JSON and CSV modules are used to help with testing and setting up data along with a more niche library called parameterized which enables to use the same test with different parameters. The scripts test login functionality, DUT (Device Under Testing) and measurement functions, and ensures that calculation functions are returning correct values when they are requested via API. Test data is parsed and loaded from CSV files to Python dictionaries. An additional CSV is used in order to give certain tests information on what values to use on parameter calculations. This CSV file contains file names, expected calculation values and additional information that is needed for some calculations. The tests that use this "master file" are designed in a way where extra test data and parameter calculations can be added without needing to edit the test scripts. Figure 7 shows the first few lines of this CSV file.

	column 1	column 2	column 3
1	Filename	basta_tilt_accuracy	basta_tilt
2	extra_info	{"basta_beam_width": 3}	{"basta_beam_width": 3}
3	test1.csv	1.0	-1.0583923606629195
4	test2.csv	1.0	-1.0583923606629195

FIGURE 7. A part of the CSV file that is used to setup many of the tests.

4.1.2 Inserting Measurements

The Test_InsertMeasurement class tests many scenarios related to sending measurements to a database. Measurements uploaded via AntView are required to follow certain rules and these tests check that the backend and the API enforce the rules. Along with the measurement data itself, a measurement file contains metadata, which is information, such as the name of test engineer and measurement date. When a measurement is sent to the database, it is then requested back and compared against the sent data. The measurement data, which is in a NumPy array, is tested using the NumPy library's own assertion that checks two NumPy arrays for equality. Metadata is tested using assertEqual, which checks that both the value and data type are identical.

When sending measurements, some metadata, such as the test engineer's name and measurement type, are required to have non-null values. There is also data that is always automatically generated by the backend, such as the name of the user which added the measurement. These types of scenarios have a test that checks that the correct error is returned when a user sends measurements breaking these rules. Many of the data fields also require a certain data type, and each of these cases is tested similarly to above.

4.1.3 Parameterization

Measurements can have dozens of different types of metadata. There are also multiple different parameters that can be calculated for measurements. Every scenario needs to be tested but the test scripts become increasingly hard to maintain if every scenario has its own test function. Initially these tests were done in a loop to solve the problem of maintaining multiple test functions with separate expected values, but this is a bad practice. When a test fails, the execution of the function stops. No information is received on the rest of the loop.

The Parameterization library solves this problem. To use this library, test data needs to be grouped to a list of tuples containing the test name, expected result of test and actual result of test. When the data to be tested is in this format, it is necessary to only use a decorator and pass a few of the variables in a tuple to the test as arguments. Figure 8 below showcases this method of grouping very similar tests under one function and executing them separately.

```python
class Test_Parameters(unittest.TestCase):

    disable_warnings(exceptions.InsecureRequestWarning)
    # get tuple list (name, expected value, calculated value)
    test_vals = t.get_parameters()

    @parameterized.expand(test_vals)
    def test_parameter(self, name, ev, val):
        self.assertEqual(ev, val)
```

FIGURE 8. Usage of the Parameterized library.

Placing test data to a list of tuples is done in another Python file (imported as "t" in figure 8). The "get_parameters" method returns that list. Below is an example of the output the "test_field" method produces when the above-mentioned decorator is used and a list of tuples is passed to it which contain the data to be tested. This level of verbosity in text output is usually not necessary when tests succeed but it is helpful when showcasing the tests. Figures 9 and 10 show the differences in test result outputs with different verbosity levels.

```
test_field_00_duts (__main__.Test_InsertMeasurement) ... ok
test_field_01_measurement_date (__main__.Test_InsertMeasurement) ... ok
test_field_02_channel (__main__.Test_InsertMeasurement) ... ok
test_field_03_peak_elevation (__main__.Test_InsertMeasurement) ... ok
test_field_04_peak_azimuth (__main__.Test_InsertMeasurement) ... ok
test_field_05_efficiency (__main__.Test_InsertMeasurement) ... ok
test_field_06_frequency (__main__.Test_InsertMeasurement) ... ok
test_field_07_bandwidth (__main__.Test_InsertMeasurement) ... ok
test_field_08_if_bandwidth (__main__.Test_InsertMeasurement) ... ok
test_field_09_beam_azimuth (__main__.Test_InsertMeasurement) ... ok
test_field_10_beam_elevation (__main__.Test_InsertMeasurement) ... ok
test_field_11_co_pol_eirp (__main__.Test_InsertMeasurement) ... ok
test_field_12_cross_pol_eirp (__main__.Test_InsertMeasurement) ... ok
test_field_13_co_pol_gain (__main__.Test_InsertMeasurement) ... ok
test_field_14_cross_pol_gain (__main__.Test_InsertMeasurement) ... ok
test_field_15_phi (__main__.Test_InsertMeasurement) ... ok
test_field_16_theta (__main__.Test_InsertMeasurement) ... ok
test_field_17_power_tapering (__main__.Test_InsertMeasurement) ... ok
test_field_18_gain (__main__.Test_InsertMeasurement) ... ok
```

FIGURE 9. Example test output in terminal with verbosity level set to 2

```
............................
----------------------------------------------------------------------
Ran 28 tests in 11.165s

OK
```

FIGURE 10. Same tests with verbosity level of 1, where dots indicate a passed test.

The parameterization module names the test automatically based on the test name, execution order and the “name” variable that was passed to the test. It is a convenient way to test all metadata fields and parameter calculations and requires minimal editing of the test script when new metadata types are added.

4.2 Test Automation Server

4.2.1 Test Environment

The Jenkins automation server runs on an Intel NUC mini-PC with the Debian 10 Linux distribution. The Jenkins instance runs in a Docker container, with AntView’s own containers launched from this Jenkins container. The Jenkins and AntView containers share the same Docker Daemon. Figure 11 below shows the command with all the necessary prefixes to start the Jenkins container.

```
docker run -d --name jenkins -p 8080:8080 -p 50000:50000 -v jenkins_home:/var/jenkins_home \
-v /root/laboratory:/laboratory -v /var/run/docker.sock:/var/run/docker.sock jenkinsdoc
```

FIGURE 11. Docker run command with port and volume mappings launching “jenkins” container based on “jenkinsdoc” image.

The “Jenkins_home” is a directory on the physical drive containing all the settings, jobs and user info related to our Jenkins. This volume is mapped to the Jenkins container with the “-v” prefix. Without this mapping, all the information would be wiped out when the container is stopped. To preserve information in a container, volume mapping is required. The second volume is mapped in order to pull AntView’s repository from the Git server. The third one maps the directory containing the “docker.sock” to the Jenkins container. This is the UNIX socket that the Docker Daemon listens to and is required to launch containers inside a container (21). Figure 12 illustrates this entire test environment from docker containers to the important volumes and version control.

This practice of launching containers from a so-called master container has potentially some serious security flaws since Docker by default launches all containers as a root user. This means that anyone with access to Jenkins can execute shell commands to gain access to sensitive information. This Jenkins server however is not hosted on an open network. (22)

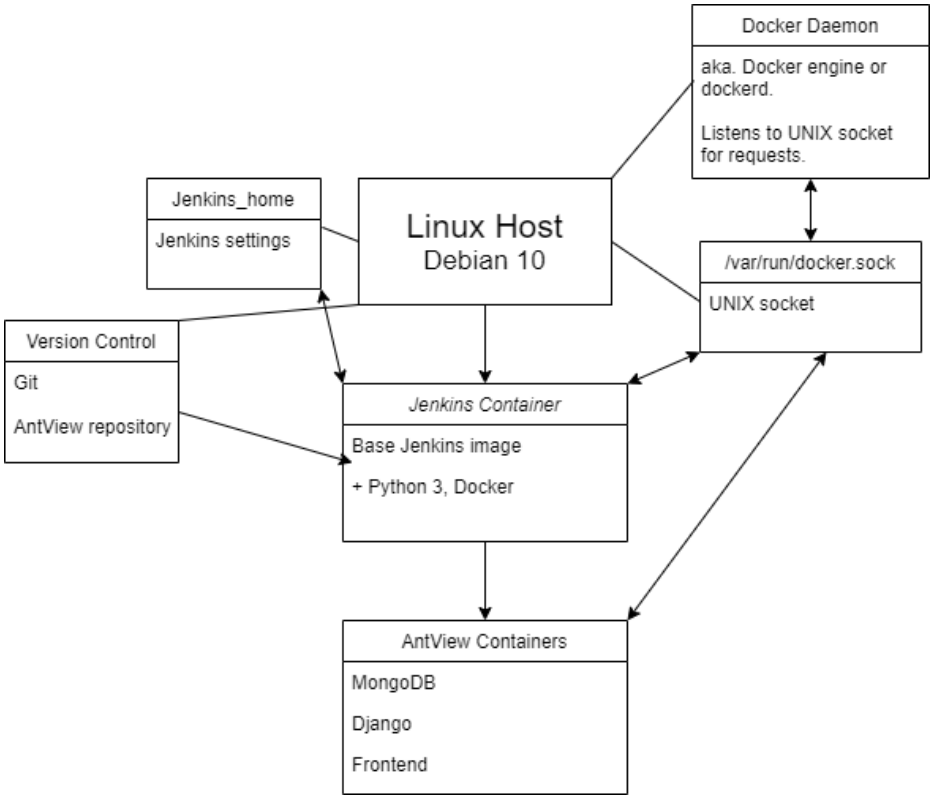

FIGURE 12. AntView's test environment.

The AntView instance built by the Jenkins pipeline is left running after the jobs are complete. This can be helpful when doing manual testing or test development. For doing quick tests for a specific build, there is the option of connecting to the server with SSH and using the server's AntView instance, bypassing the need to setup users' own AntView in a virtual Linux environment.

4.2.2 Jenkins Jobs

Jenkins jobs are specific tasks that can be chained together to form the testing pipeline of the software. In this project, there are four stages in the pipeline; pulling the latest AntView build, build stage, pull Python API and test scripts, and finally the execution of test scripts. This chain of jobs is started by a button press using the web GUI but can also be set up to be triggered by a commit to the version control. Currently in this project manual activation of the pipeline is deemed more practical than a trigger on every single commit. By using the Build Pipeline plugin, the process can be visualized clearly as shown in figure 13 below.

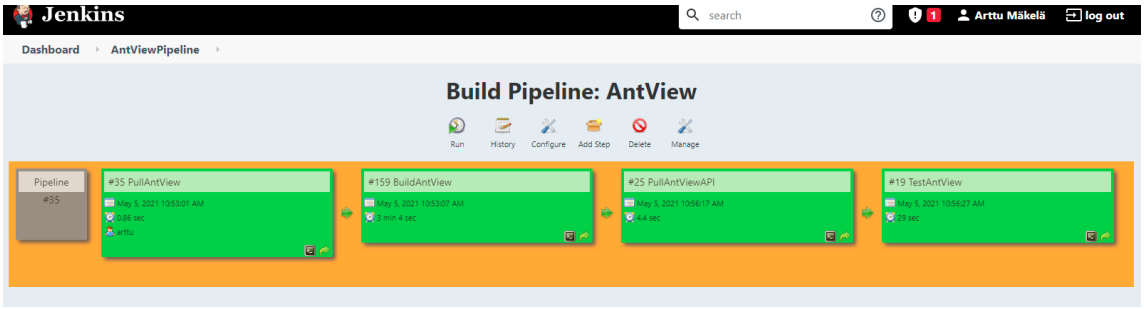

FIGURE 13. The four stages of AntView testing pipeline; pull and build AntView, pull API + test scripts, execute tests.

The “PullAntView” job is a simple task of pulling the master branch of the AntView project and a running script to zip the project folder. The zip file is then stored in Jenkins as an artifact for next job in the pipeline. All artifacts are saved in the Jenkins volume and can be reviewed through the web GUI.

The “BuildAntView” job uses the zip file and performs various commands to setup the environment for AntView to be correctly installed. AntView is then installed using Docker, creating and starting the three containers: frontend, mongodb and django.

“PullAntViewAPI” job pulls AntView’s Python API and the test scripts from the version control and copies a file from the Jenkins volume containing few variables so that tests properly connect to the AntView REST API. The environment at this point is ready to run the tests using the “TestAntView” job. This job installs any imports to the container if they are missing and then runs the test scripts.

4.2.3 Jenkinsfile and scripted pipeline

This project’s pipeline was done through the Jenkins Web GUI. The entire pipeline of software can also be put into a Jenkinsfile by using the Pipeline plugin (separate from the graphical Build Pipeline plugin shown in figure 13). Both the web GUI and Jenkinsfile pipelines can be written using slightly modified Groovy’s syntax. The Jenkinsfile can then be checked into the version control and is treated as part of the source code. Using this method, the Jenkins pipeline is part of

the project rather than a separate entity inside a Jenkins folder on the host computer, giving teams better access to review and edit the pipeline if necessary. (23)

The author experimented with using the scripted pipeline method to run the operations. This can be done somewhat easily without prior knowledge of the Groovy language by using Jenkins' built-in snippet generator. Users can use the tool to define various steps in the pipeline by picking a step from a list. The chosen step is then generated as lines of code in the script file. This is a convenient way to learn the basic syntax for scripted pipeline. Figure 14 shows an example of AntView's pipeline as a scripted pipeline without the build stage.

```
pipeline {
    agent any
    stages {
        stage('PullAntView'){
            steps{
                deleteDir()
                git changelog: false, credentialsId: 'ArttusGitCreds', poll: false, url: '/laboratory/Laboratory/Development/near
                sh './create_zip.sh'
            }
        }
        stage('PullAPI') {
            steps {
                git branch: 'unittests', changelog: false, credentialsId: 'ArttusGitCreds', poll: false, url: '/laboratory/Labora
                dir('/var/jenkins_home/workspace/') {
                    sh 'pwd'
                    sh 'cp auths.py /var/jenkins_home/workspace/jenkinsfile_test'
                }
            }
        }
        stage('Test'){
            steps{
                sh 'pip3 install -r requirements.txt'
                sh 'python3 test_login.py'
                sh 'python3 test_duts.py'
                sh 'python3 test_measurements.py'
                sh 'python3 test_parametercalcs.py'
            }
        }
    }
}

```

FIGURE 14. AntView pull, API pull, and test jobs in a scripted pipeline.

Without prior knowledge of the commands, the process of finding specific steps with the snippet generator is quite cumbersome. There is no search option and overall using the generator to create these pipelines is like using the GUI with extra steps. The requirement to wrap steps, such as shell commands, with "sh" might make some commands difficult to read especially if there are multiple quotations nested inside.

The pipeline plugin offers its own view of the completed pipeline. This view has some advantages over the standard UI which offers a basic console output for the executed job. For pipeline scripts, each separate shell command and their outputs and statistics can be viewed in a list format. The pipeline can also be started from a specific stage. The replay feature allows the user to modify the pipeline script and rerun it in the already existing pipeline instance, which is useful for proto-

typing and testing slight changes in the script. Figure 15 shows the Stage view plugin for a scripted pipeline, skipping the build stage of AntView and using an already built instance to run the tests.

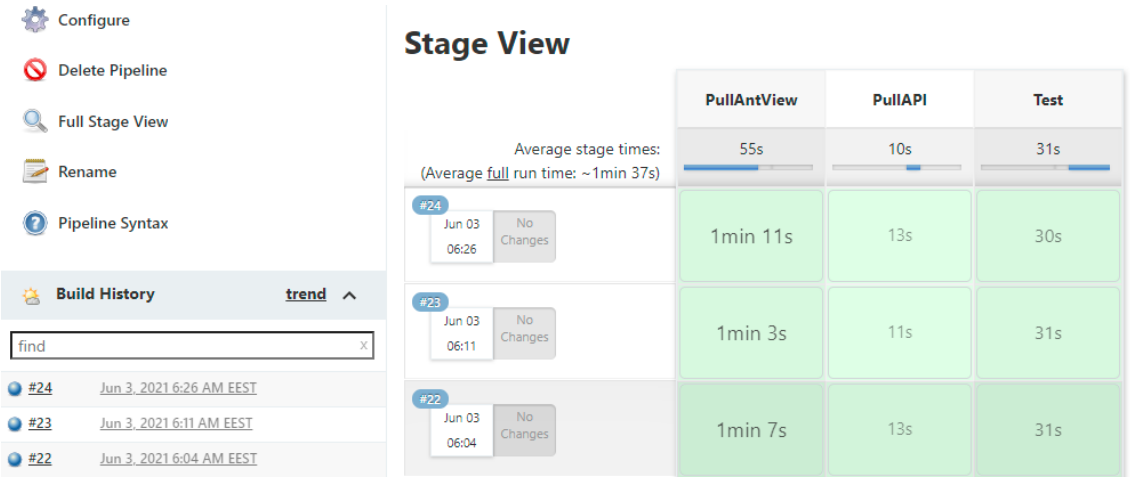

FIGURE 15. Stage view plugin for a scripted pipeline.

This pipeline view as shown in figure 15 provides more clarity than the Build Pipeline plugin shown in figure 13 The history of pipeline executions is clearly listed, and each stage (e.g. PullAntView) can be clicked to show more details of the steps in that stage. Overall, I think this method of writing the pipeline to a single file has enough benefits to it to be considered as a replacement for the GUI method, especially if the user becomes more used to the scripting language.

5 CONCLUSIONS

During this thesis, I was introduced to automated testing in software development. I learned the general Docker usage and how to setup a basic Jenkins pipeline to execute the tests. I learned how to implement automated testing to a project and got an idea of the good and bad practices when it comes to test creation. A previous experience with Python from smaller projects was helpful.

The tests in this project used the AntView's Python API to carry out the requests to the software backend. The tests are therefore not unit tests but tests testing the API and the fully built software. Some tests such as the ones requesting parameter calculations from the backend, are also indirectly testing the calculations functions themselves, making sure they return the values they are supposed to. These tests are useful in ensuring the API and the built software works as intended but requires a build phase in order to be executed, which means that running the tests after a change in codebase takes a few minutes compared to about 25 seconds if AntView is not rebuilt.

The test data is loaded based on a CSV file that lists each measurement file by name and related information such as expected calculation results and additional parameter information. This effectively allows adding additional calculation tests by editing this file if the parameter calculation option exists in AntView. This CSV file also gives a good documentation of what parameters and files are tested and with what values.

Learning Docker and Jenkins fundamentals were very useful parts of this thesis. Getting to know how dockerfile and Docker Compose work, how to setup containers with their required mounted volumes and port mappings will no doubt be useful to know in future projects. While the Jenkins implementation in this project was straightforward, the thesis overall has been a good introduction to the tool.

The testing pipeline was implemented using the Jenkins GUI, but I also experimented with the scripted pipeline method. It is a potential replacement for the GUI version, especially if the project pipeline complexity increases and different jobs start sharing variables or if the number of chained

jobs becomes hard to manage by the GUI. Since the pipeline script can be deposited as a Jenkinsfile to version control, it enables a better reviewing of the pipeline in larger teams.

This thesis also made me realize the usefulness of Linux and how little I know about the operating system. The reliability, speed and its open-source nature makes it a very powerful OS. I got to know the basics of it, but to fully utilize tools like Jenkins, I need to know my way around Linux and its commands better.

I believe that creating a more comprehensive testing plan at the start would have been good for the thesis. I had the general idea of what aspects to test, but once the more obvious test cases were made, a lot of time was spent on creating tests which were later completely redone or discarded. As an example, creating multiple test cases for parameter calculations, only to later figure out an entirely different way to do all of them could have been avoided with a better planning. There are benefits to a trial-and-error approach, but I think in the future I need to do better planning and research before actual implementation.

REFERENCES

1. Shaw, A. Getting started with testing in Python. Date of retrieval: 14.4.2021
   https://realpython.com/python-testing/

2. Harold, E. Effective unit testing by Eliotte Rusty Harold. Devoxx 2019. Date of retrieval: 14.4.2021
   https://www.youtube.com/watch?v=fr1E9aVnBxw&ab_channel=Devoxx

3. Unit testing tutorial. Guru99. Date of retrieval: 15.4.2021
   https://www.guru99.com/unit-testing-guide.html

4. Integration testing. Guru99. Date of retrieval: 15.4.2021
   https://www.guru99.com/integration-testing.html

5. Shafer, C. 2017. YouTube. Python tutorial: Unit testing your code with the unittest module. Date of retrieval: 15.4.2021.
   https://www.youtube.com/watch?v=6tNS--WetLI

6. Singh, R. 2020. Manual testing. Date of retrieval: 18.4.2021
   https://www.toolsqa.com/software-testing/manual-testing/

7. Unit testing framework. Python documentation. Date of retrieval: 18.4.2021
   https://docs.python.org/3/library/unittest.html

8. Testing your code. docs.python-guide.org. Date of retrieval: 19.4.2021
   https://docs.python-guide.org/writing/tests/

9. Docker documentation. Docker overview. Date of retrieval: 21.4.2021
   https://docs.docker.com/get-started/overview/

10. FreeCodeCamp. YouTube. A full DevOps course on how to run applications in containers. Date of retrieval: 21.4.2021
https://www.youtube.com/watch?v=fqMOX6JJhGo

11. Makam, S. 2014. Sreenivas Makam's Blog. Docker overview. Date of retrieval: 23.4.2021
https://sreeninet.wordpress.com/2014/12/28/docker-overview/

12. Jenkins user documentation. Date of retrieval: 27.4.2021
https://www.jenkins.io/doc/

13. What is continuous integration. CloudBees. Date of retrieval: 27.4.2021
https://www.cloudbees.com/continuous-delivery/continuous-integration

14. System testing. Guru99. Date of retrieval: 29.4.2021
https://www.guru99.com/system-testing.html

15. Docker documentation. Docker compose. Date of retrieval: 3.5.2021
https://docs.docker.com/compose/

16. Security testing. Javatpoint. Date of retrieval: 6.5.2021
https://www.javatpoint.com/security-testing

17. Top 10 web application security risks. OWASP. Date of retrieval: 18.5.2021
https://owasp.org/www-project-top-ten/

18. How to prevent SQL injection attacks. Positive technologies. Date of retrieval: 18.5.2021
https://www.ptsecurity.com/ww-en/analytics/knowledge-base/how-to-prevent-sql-injection-attacks/

19. Verma, G. 2018. Security risks to watch out for in your web application. Date of retrieval: 20.5.2021
https://startupglide.com/security-risks-to-watch-out-for-in-your-web-application/

20. Introduction to Linux operating system. GeeksForGeeks. Date of retrieval: 20.5.2021
https://www.geeksforgeeks.org/introduction-to-linux-operating-system/

21. Stack Overflow. Date of retrieval: 21.5.2021
https://stackoverflow.com/questions/35110146/can-anyone-explain-docker-sock

22. Benjamin, P. 2018. Docker security best practices. Date of retrieval: 24.5.2021
https://dev.to/pbnj/docker-security-best-practices-45ih#docker-engine

23. Jenkins documentation. Pipeline as code with Jenkins. Date of retrieval: 25.5.2021
https://www.jenkins.io/solutions/pipeline/