Valtteri Kukkola

Benefits of Test Automation Results
Data Visualization

Metropolia University of Applied Sciences
Bachelor of Engineering
Information and Communications Technology
Bachelor’s Thesis
06.11.2019
The objective of this thesis was to study the benefits of visualizing test automation results data by implementing a software that visualizes test results stored in a database. The thesis describes the development of an application which is a prototype of this type of software, as well as the reasons behind the decisions made in the process of developing the implementation. The thesis also presents different types of views implemented in the application, and the ideas how they could benefit people working with test automation.

The study was commissioned by Siili Solutions Oyj, for Software Automation Laboratory unit (SALabs). Software Automation Laboratory is a unit in Siili driving for creating the next versions of software automation.

The System Under Test for the development of the software will be one called KnoMe, internally developed by Siili. The test result data from KnoMe are stored and then later visualized by a dashboard application developed within the present study.

Keywords
Test Automation, DevOps, Continuous Integration
Contents

Abbreviations and Key Concepts

1 Introduction 1

2 Objective 2

2.1 Current State 2
  2.1.1 Managers 3
  2.1.2 Developers 3

2.2 Improvements 4
  2.2.1 More Visibility to Managers 4
  2.2.2 Faster Feedback Loops for Developers 5

3 Technologies and Methodologies 6

3.1 Robot Framework 6

3.2 Containers 8

3.3 Continuous Integration and Continuous Delivery 10

3.4 System Under Test, KnoMe 11
  3.4.1 Architecture 11
  3.4.2 Test Execution 12

4 Dashboard Application Implementation 13

4.1 Architecture 13

4.2 Data Storage 14
  4.2.1 Test Results and Metadata 14
  4.2.2 File Changes 17

4.3 Graphical User Interface 19
  4.3.1 API 19
  4.3.2 Manager View 21
  4.3.3 Single Test Run View 23
  4.3.4 History View 25

4.4 Back-End and Database 26

4.5 Testing 27

4.6 Environment 27

5 Future Improvements 28
### Abbreviations and Key Concepts

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS</td>
<td>Amazon Web Services, Cloud computing platform.</td>
</tr>
<tr>
<td>CI</td>
<td>Continuous Integration (CI) practice of merging code together to a specific branch in a short development cycle.</td>
</tr>
<tr>
<td>CD</td>
<td>Continuous Delivery (CD), A software engineering approach in which teams produce software in short cycle. Continuously testing, building and releasing versions in a high frequency.</td>
</tr>
<tr>
<td>Containers</td>
<td>Standard unit of software that packages up code and all its dependencies.</td>
</tr>
<tr>
<td>DevOps</td>
<td>Set of practices that automates processes between software development and operations, in order to build, test and release software faster and more reliably.</td>
</tr>
<tr>
<td>Docker</td>
<td>Set of platform as a service products that use OS-level virtualization to deliver software in containers.</td>
</tr>
<tr>
<td>Docker-Compose</td>
<td>Tool for defining and running clusters of Docker applications.</td>
</tr>
<tr>
<td>Docker hub</td>
<td>A hosted repository service provided by Docker.</td>
</tr>
<tr>
<td>End-to-End Testing</td>
<td>Technique used to test whether the flow of an application from start to finish is working as expected.</td>
</tr>
<tr>
<td>Git</td>
<td>Distributed version-control system for tracking changes in source code during software development.</td>
</tr>
<tr>
<td>Git commit SHA</td>
<td>Git hash-object, 40-character checksum hash from saved content-</td>
</tr>
<tr>
<td><strong>Gitlab</strong></td>
<td>A web-based DevOps lifecycle tool that provides a Git-repository manager.</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Gitlab CI</strong></td>
<td>GitLab Continuous Integration, A Web application with an API that stores its state in a database. It manages projects and builds and provides a graphical user interface.</td>
</tr>
<tr>
<td><strong>Gitlab Runner</strong></td>
<td>Open source project that is used to run jobs and send the results back to GitLab. Used to in conjunction with GitLab CI.</td>
</tr>
<tr>
<td><strong>Node.JS</strong></td>
<td>Open source, cross-platform, JavaScript runtime environment that executes JavaScript code outside of a browser.</td>
</tr>
<tr>
<td><strong>PostgreSQL</strong></td>
<td>Open Source relational database management system.</td>
</tr>
<tr>
<td><strong>Pug</strong></td>
<td>High performance template engine implemented with JavaScript for Node.js</td>
</tr>
<tr>
<td><strong>Python3</strong></td>
<td>Version 3 of the programming language Python. Used also as a command to specify desired Python version.</td>
</tr>
<tr>
<td><strong>Robot Framework</strong></td>
<td>Testing framework written with Python.</td>
</tr>
<tr>
<td><strong>Selenium Framework</strong></td>
<td>Portable framework for testing web applications.</td>
</tr>
<tr>
<td><strong>Selenium Grid</strong></td>
<td>A part of selenium. Suite that specializes in running multiple tests across different browsers, operating systems and machines in parallel.</td>
</tr>
<tr>
<td><strong>Shell Scripting</strong></td>
<td>Programming in Unix shell, Scripting used in software automation.</td>
</tr>
</tbody>
</table>
1 Introduction

Quality assurance has become a must in modern software development processes which follow agile methodologies. With modern test automation techniques and DevOps pipelines, developers can be sure that when they make changes to their software, the updated version will work as intended. The need for test automation rises especially in larger projects where there are thousands of tests that need to be run to ensure the quality of the software. Projects on this scale have to tackle different kind of problems with test automation. With large masses of automated tests, the benefits of test-automation begin to fade away. Feedback loops for developers increase with the time that it takes to execute all the tests. With large test masses it becomes difficult to read the test results and compare them to other test runs. Often there is only the possibility to read test results from single test execution and it can become very slow and inefficient if more test execution results need to be checked at once to understand test behavior. Tools that help test automation often lack visual representation of the history of test runs, which can be very valuable resource in fixing unstable test cases.

This thesis was done for Siili Solutions Oyj, for Software Automation Laboratory unit (SALabs). Software Automation Laboratory is a unit in Siili, which drives for creating the next versions of software automation. SALabs is a part of New Solutions department in Siili, which has a focus on developing new business solutions on top of traditional consultant business. The objective was to study ways of implementing software that would visualize data from stored test automation results. The thesis describes one implementation which is the first working version of this type of software, and the reasons behind making the decisions to come up with this implementation. The System Under Test for the development of this software is one called KnoMe, internally developed by Siili. The scope of the thesis is also to compare ways of implementing the desired functionalities with different approaches to research for the best ways of implementing this kind of a system.

The objective for SALabs is that the software development work will continue after this project even further and to make it open-source software for everyone to use. The present study is an important part of a wider project which aims at researching ways to use AI for test automation optimization.
2 Objective

This chapter will first describe the current state of test automation in various real-life projects. It will first go through known problems and issues from the perspective of manager and developer level employs, and then present the improvements sought after by this project work.

2.1 Current State

In development processes where quality assurance and continuous integration have been taken into count, companies can be certain that their software goes to production flawlessly. Modern ways of developing software include a delivery pipeline which makes sure that necessary steps have been done for a software version, before it goes to production. DevOps practices have started to play a crucial part in software development everywhere. In a well build DevOps pipeline a developer can start a pipeline with a push of a button or just making a commit to project repository. The pipeline ensures that everything works as intended and tries to catch possible bugs and unwanted behavior before merging the code with version control. This type of pipeline allows developers to build software in a more agile way and to get feedback of their changes faster. CI platforms often give users the possibility to access test reports created by testing frameworks from one test run or a job. The thing that is missing is a representation of the test results from multiple test runs, which could be valuable information in many different situations.

Different frameworks and tools for test automation have various ways of showing the results of executed tests. What they all have in common is that they do not have a proper ways of viewing test history, the necessary metadata from the test executions or a clear way of finding trends by comparing test runs together.

Robot Framework has one of the best reporting features from individual test runs [5.] It generates reports that are easy to understand, even without a deeper technical background. Other frameworks such as a popular web testing framework Cypress lack this kind of reporting functionality completely [7.] Although cypress offers an option to pay for their own dashboard solution, which implements things like taking screenshots, recording videos of the test runs, test parallelization and run grouping.
2.1.1 Managers

The group of people interested in the test results from test automation is not solely limited to developers and technical experts building the software. Often there are product owners and managers who are also interested in the results or status of tests either in software that is under development or software that is in production and has regression tests regularly. In many cases the only way of accessing test reports is from the CI environment itself.

In these situations, the person interested in accessing this data needs to have technical skills to find the reports from the CI environment, which might be inside a virtual machine in the cloud or in on-premises server. They also require access rights for the environment which might be limited only to developers and technical experts to prevent unwanted things from happening. These requirements often limit managers and non-technical experts from being able to access the reports, which leads to them relying on the technical experts whenever they want to access the data.

2.1.2 Developers

Many CI environments already provide a way to store the job artifacts data from individual job executions. These artifacts can include the test results data, and some environments even offer features to browse these artifact files in a browser view. Often the only option is to download the files to a local machine and go through them to find the desired information. If a developer would like to compare these results with another test execution, they would then need to go to a different pipeline job and download the artifacts. Needless to say, this takes time and effort and users need to go through the whole test reports to verify that they were the ones that they actually wanted to compare with another test runs.

As a result, the knowledge that could be used from previous test runs is often hidden together with all the other metadata when new test runs fill the pipeline of the CI environment.
2.2 Improvements

To further improve the benefits from software automation, a visual representation of test automation data can become an important asset for people working with projects that take use automation. To study how this kind of a solution would benefit everyone, Siili wanted to build software. The software is a unique prototype, thus making it hard to compare to any existing products.

The solution implemented here is a dashboard that visualizes all the data from test results enriched with important metadata from the test execution environment. With this dashboard, developers can compare different test runs in a more efficient way and find patterns and similarities from the test executions, which can be useful for troubleshooting. Also, as a very important objective, the application can provide a way to optimize test execution based on the results history. The implementation is designed for two groups of people in mind. The first group is managers and nontechnical users who benefit from intuitive and simple user interface to be able to get a high-level idea of the test results. The second group consists of developers and technical experts. They are the group that desire deeper dive in to the test results data and technical logs.

2.2.1 More Visibility to Managers

Managers and nontechnical experts desire simplified things. A single easily accessible place where they can find all the results they want, even from multiple different projects. In this kind of context, an easy to read graphical representation is a user-friendly way to understand the data even without much understanding of the context. It can be easy to use even with a very little understanding of the technical logic behind the automated testing itself.

With an application authenticated for instance with active directory credentials, the management level people could have an ease access to the software. This would give them information that they simply were unable to get from the CI environments before. With the application, they could keep up with the status and situation of the project. As a result, the improved visibility can benefit in better communications and overall results in a project.
2.2.2 Faster Feedback Loops for Developers

By storing test result data, file changes from version control and other metadata from the environments where tests are executed, dashboard software can help developers to optimize their test execution. Developers can see from test history that certain files have connections to certain test cases. This may sound simple, but in practice it can be very difficult to see which tests might be the most crucial to run first from a test set of thousands of tests, often written by other people than the developers themselves.

The dashboard is aimed to solve the problem with finding and opening multiple test reports, as it combines all that data together to a single place. Users also gain more visibility to the history of the test runs since all the data is stored instead of disposed. This can be valuable when some old test behavior reappears in the future. For example, after the development phase of software is done, the maintenance team might be different than the one who did the software. With the history of test data provided by the application, they would get beneficial knowledge that is helpful at fixing test cases and understanding test behavior.
3 Technologies and Methodologies

This section covers some of the most relevant technologies and frameworks used in the project development. There are plenty of other minor technologies used in the project but going through all of them would not be relevant here.

3.1 Robot Framework

Robot Framework is the testing framework used in the system under test that was used to develop the project at hand. Robot Framework is a Python based framework that works as abstraction layer for implementing keyword-driven testing. It is originally developed by Pekka Klärck and currently maintained by Robot Framework Foundation. It is commonly used for test automation but lately has been implemented with support for robotic process automation as well. [5.]

Robot Framework has a vast number of different libraries that can be used with it. The most commonly used one is Selenium Library which is an abstraction layer for portable automation framework Selenium. Selenium is a web automation framework that works as a core for multiple testing frameworks such as the Selenium Library for Robot Framework. [6.]

The end-to-end tests from KnoMe, were chosen as a test set to develop the dashboard software. These tests are meant to behave in an exactly similar way that a real user would use the software under testing. They use Robot Frameworks own built-in library and external robot framework libraries, Robot Framework Selenium library and Robot Framework Requests library.

Robot Framework produces test reports in different formats. It has its own graphical reports which are a very clear and easy to read representation. An example image is shown below in Figure 1. It also generates XML formatted file named output.xml that has all the test execution data. Developers using Robot Framework usually use the generated reports to view data from the tests. These reports have the information and logs from the test execution and can contain screenshots from test execution failures.
Figure 1. Robot Framework test results report.

This Figure 1 shows the actual test results gathered from single execution of the end-to-end tests in the KnoMe project. The username and address of the environment are taken away from the figure for preservation of anonymity. The hierarchy of elements in the report logs enables users to handle bigger test reports more easily. Users can look at the logs from a higher level or dive in to them as deep as they want. The report by default opens all the steps from test executions that have failed. In this case the figure shows entity migration tests that belong to their own group, which belongs to a higher level group of administration, which belongs to the highest level of group called “tests”.
3.2 Containers

Containers have become a popular way of implementing infrastructure in software development. They were designed to be the silver bullet that reduces problems that exist when developing or deploying software on different operating systems and platforms. Unfortunately, they do not solve all the issues there are, but they offer beneficial features of dealing with the common problems.

Docker container is a lightweight implementation to package all the necessities for an application to run. They store all the mounted files and volumes, work as a runtime for the environment and have a way of storing configurations. These containers can be built from readymade images that have certain libraries and tools preinstalled. For example, Python applications in this project are built from a Python 3.7 docker image [3.]

![Diagram](image)

**Figure 2.** Architecture of containerized applications. (Containers in Docker, 2019)

The Figure 2 shows an example of how the containerized docker applications work with the rest of the environment.
The Docker containers are in a sense small virtual machines that run in an isolated environment. They can be created to same network to enable communication from one container to another or exposed to outside world through desired networking ports. [2.]

Containers work especially well in testing environments. With containers it is convenient to ensure similar beginning conditions for each test execution. The atomic nature of containers makes them an excellent choice for modern automation project infrastructure.
3.3 Continuous Integration and Continuous Delivery

Continuous Integration and Continuous delivery are important parts of the software development process. There are multiple platforms for implementing continuous software development pipelines and they all offer very similar features. In most of them one can write the pipeline as code, hide your secret information such as passwords to environment variables, start pipeline executions which result in a new delivery of a software version and much more. They often have very similar way of creating the pipeline-as-code, and they only differ on the syntax.

Typical CI/CD pipeline has separate steps for building the application, specific test phases and a packaging phase in the end to create a new version of the software ready for deployment.

![Example Gitlab CI/CD pipeline](CI/CD pipeline, 2019)

Figure 3. Example Gitlab CI/CD pipeline. (CI/CD pipeline, 2019)

Figure 3 shows an example from a simple CI/CD pipeline. The pipeline in the picture builds the application, runs tests against it and deploys it to development environment. It also has an option to deploy to production.

The pipelines can be automated to execute all the steps from testing all the way to deploying the new version of the software. The pipeline execution can be triggered from a change commit to the code or started manually. For example, in development branches, a pipeline can be configured to run only the tests made for the project. The deployment parts can be configured to occur from a certain branch in the version control or to require a manual validation before deploying to production environment.

It is common to use a separate master or production branch that also takes care of packaging the software and possibly launches it somewhere. There can even be multiple specific branches that deploy the software to different environments such as quality assurance, development and production.
3.4 System Under Test, KnoMe

KnoMe is a competence management system developed to improve implementation of Siili vision of consulting business. The purpose for KnoMe is to help sales employees to find best projects for Siili consultants. The logic behind it is that KnoMe maps together the competence of Siili consultants with the interests that they have and this way sales can then reply to customer demands with the best consultants. In a simplified way to look at it, KnoMe is a web service that has CV types of profiles of all the employees at Siili. These profiles have data about technologies and skills of consultants, and as an important factor, the interest that they have towards a specific subject. These subjects can be anything from a specific technology to a field of business. KnoMe also has a vast number of other features such as giving different types of reports. The development of a second version of KnoMe is ongoing and this second version is also the one used to generate test result data for developing the application in this thesis.

3.4.1 Architecture

The architecture of KnoMe is implemented in microservices that run in container clusters [10.] The complete architecture of KnoMe is not in a scope of this thesis, but the parts of its CI pipeline are relevant. KnoMe project uses Gitlab CI as continuous integration environment. Gitlab CI uses runners for building the virtual environment where test execution happens. In this case, these runners are virtual machines hosted in AWS. In the CI pipeline inside virtual machines, the infrastructure is built from docker containers as a cluster with docker-compose. The CI environment in Gitlab CI is built from a YML type of file named gitlab-ci.yml. This file runs the scripts used for building the infrastructure and executing the tests. The file itself is also in the context of version control which allows all branches and forked versions to have their own pipelines. The CI/CD pipeline in KnoMe projects run a vast number of different kind of tests including unit tests, different end-to-end tests for different parts of the software and a lot more. The scope for the present study lies in the end-to-end tests that are done with Robot Framework.
3.4.2 Test Execution

The Robot Framework end-to-end tests are executed inside their own docker container in the pipeline. The execution happens with a shell script that runs in the Gitlab CI environment. The script starts docker-compose clusters that first builds the infrastructure for the robot testing. This infrastructure has two docker-compose clusters. One for the application itself and one for the cluster that runs robot testing infrastructure.

The robot tests docker-compose cluster consists of two services, seleniumgrid and the robot service. Seleniumgrid container is pulled from dockerhub and given few parameters such as the size of the browser window that it should use, the version of the container image to use, volumes for storing video recordings of test executions and few other parameters.

The container for robot is built from a docker file. The docker-compose file defines the dependencies for the robot service, volumes that can be accessed from the container, few environment variables for the container, and specifies the path for the Dockerfile that is used to build the container.

The robot Dockerfile creates a container that is build using Python 3.7 image as a base for it. In the docker file first users for the container and directory for it are created. Then all the necessary files are added so that they will be mounted for the container. A pip install command is then run to install all the requirements for the robot container to work. In this case these requirements consist Robot Framework with few libraries. The Docker file then creates few directories and gives the ownership rights to these directories to the “robot” user. Last thing that the docker file does, is adding entry points where to start the test execution once the container is build.

The first shell script ran here is named wait-for-selenium.sh. This script is generally used when interacting with selenium grid containers. The sole purpose of the script is to wait for the selenium grid service to be ready for interaction. Second shell script named run-tests.sh runs the Python commands for starting the robot test execution. The robot test starting command can take plenty of arguments, and these arguments will be useful for this project later on. A relevant feature in this implementation is an option of adding more metadata for the generated output.xml test report.
4 Dashboard Application Implementation

This chapter describes the solution done in the present study for achieving the results sought after. It goes through the different elements of the developed software and explains how and why they have been implemented. The chapter contains few code examples most relevant parts of the software, as well as figures of the architecture and graphical user interface views.

4.1 Architecture

The keyword in this whole project was experimenting. As “Software Automation Laboratories” is the name of the unit where this thesis is written at Sili, so experimenting sounded suitable for developing something new. With this idea in mind, the structure of the bigger system that implements the application is written with smaller replaceable pieces that are easy to test, deploy, manage and most importantly to this thesis, to experiment with. The system developed has separate front-end, back-end, and multiple databases. It is also designed to read specific data created by another open-source software that is integrated to the system under test. The complete structure of the software can be seen in Figure 4.

![Dataflow architecture](image)

Figure 4. Dataflow architecture.

The Figure 4 shows the elements of the whole system that achieves the visualization of test results history. The different steps of the architecture will be further described in their individual chapters.
4.2 Data Storage

The storing of data needed to be integrated to the system under test so well, that it would not take any additional steps from the developers. If this part would not have been automated with the test executions, it could have raised a high possibility that developers would have skipped it, especially in situations where they can see earlier that their test executions is going to fail. The most important data that can be acquired from the automated test in the perspective of this application, is when test executions fail due to changes in the source code. Like in everyday life where people can learn from failures, the test history data can also be used to learn, if it still is accessible later.

The storing of required data is done with an open-source tool created by Tommi Oinonen. The tool called TestArchiver is used to store the data from output.xml generated by Robot Framework tests. TestArchiver also has support for listener type of storing data which would store data even if the robot test execution breaks down without generating the output.xml file, and this way of storing the data should also work with the application created but is not properly tested. The TestArchiver project is also growing to have support for other types of test data than Robot Framework. TestArchiver is also developed in parallel with this project and is partly influenced by requirements and feedback from it. The TestArchiver project can be found from GitHub under the SALabs organization. [11.]

4.2.1 Test Results and Metadata

Gitlab CI has very useful environment variables that give access to important data that is required for the front-end application to visualize the test data with the history included. Robot Framework allows us to store more data in its output.xml file which was a good way of adding few more values to the test execution data.

The first value stored from Gitlab CI is the id of the pipeline job execution. This is a unique id given to the test run by the Gitlab CI. It is generated from the number of pipelines executed by the CI. The second value stored is the name of the branch used by git in the given pipeline. Branch name is necessary information in every situation, and it is also a plus, since this information cannot be seen from the normal robot framework log files. The third environment variable that is stored in metadata is the git commit SHA. This is a hash value that can be later used to compare changes from this commit to different commits or branches.
Gitlab CI has already access to these environment variables, but in order to store them in the output.xml file, it is needed to pass them to the docker file where the robot execution is run. Luckily docker has a handy way of passing environment variables to containers. When the run-e2e-robot.sh script is executed in the gitlab-ci.yml file, it is possible to add variables to the docker-compose script that builds the cluster where the Dockerfile for robot is. This is done with a following syntax in Code example 1.

```
Docker-compose.robot.tests.yml -e CI_JOB_ID=$CI_JOB_ID -e CI_COMMIT_REF_NAME=$CI_COMMIT_REF_NAME -e CI_COMMIT_SHA=$CI_COMMIT_SHA
```

**Code example 1. passing environment variables for docker-compose.**

After adding these variables to the docker-compose cluster, also the Dockerfile where robot container is built, needs to have these variables. They are added in the Dockerfile with dockers ENV operator syntax shown in Code example 2.

```
ENV CI_JOB_ID=$CI_JOB_ID
ENV CI_COMMIT_SHA=$CI_COMMIT_SHA
ENV CI_COMMIT_REF_NAME=$CI_COMMIT_REF_NAME
```

**Code example 2. Passing environment variables inside a Dockerfile.**

After this it is possible to use them in the shell script that is used to start the robot test execution inside the container with a shell script shown in the Code example 3. The script checks whether the execution is started in a local environment or in the CI environment. If it is started in CI environment, the results are embedded with more metadata from the environment. This metadata includes the environment variables passed to the docker container. The metadata is added to the execution command of robot framework since robot framework supports this kind of functionality. Most frameworks do not support metadata type of additions, which is why the TestArchiver tool itself has a way for adding metadata. Code example 3 shows the most relevant parts of the script that starts robot.
if [ ! -z ${CI_JOB_ID} ]; then
CHANGES=$(python3 getChangedFilesWithApi.py --token $KNOME_RE-
DUX_GITLAB_API_KEY --sha $CI_COMMIT_SHA)
echo "Sleep $ROBOT_START_SLEEP seconds..."
sleep $ROBOT_START_SLEEP
python -m robot --outputdir ./logs/ 
  --variablefile variables.py 
  --metadata "cipipelineid:$CI_PIPELINE_IID" 
  --metadata "series:$CI_COMMIT_REF_NAME" 
  --metadata "branch:$CI_COMMIT_REF_NAME" 
  --metadata "commitsha:$CI_COMMIT_SHA" 
  --metadata "changedfiles:$CHANGES" 
  --metadata "joburl:$CI_JOB_URL" 
  --metadata "cijobid:$CI_JOB_ID" 
  --metadata "username:$GITLAB_USER_NAME" 
  --metadata "environment:CI" 
  --metadata "testframework:robotframework" 
  ${ROBOT_ARGS} 
./robot/tests
EXITVAL=$?
else
  echo "Sleep $ROBOT_START_SLEEP seconds..."
sleep $ROBOT_START_SLEEP
python -m robot --outputdir ./logs/ 
  --variablefile variables.py 
  --metadata "environment:locally" 
  ${ROBOT_ARGS} 
./robot/tests
EXITVAL=$?
fi
chmod u+x /home/robot/test/logs/output.xml
echo "Storing Data to SALabs Testing Purposes"

  git clone https://github.com/salabs/TestArchiver.git

  python3 TestArchiver/test_archiver/output_parser.py 
  --database knomearchive 
  --host url-of-the-database 
  --user username 
  --pw ********** 
  --dbengine postgresql 
  /home/robot/test/logs/output.xml

  echo "Done Storing data, thank you!"

  exit $EXITVAL

Code example 3. Snippet from a shell script that starts code execution inside docker in 
local and CI environments.

The end of Code example 3 shows how the TestArchiver tool is cloned and the Python 
program ran to store the results data of the tests. It is also important to notice that the 
extit value needs to be stored from the test execution script to give correct information 
from the test execution.
4.2.2 File Changes

Gitlab CI did not offer file changes as environment variables, so another kind of solution needed to be found to store them. Like the environment variables it was an efficient way to store file changes in the robot output.xml file, and Git itself offers few good ways of comparing the file changes to different commits and branches [4.]. The only issue was that inside gitlab-ci.yml container, normal git was not installed, which made it impossible to just use git diff command to get file changes in the pipeline. This kind of solution would also have lacked the possibility to store changed files data, when the tests were executed with local machines.

Gitlab API offered a way of comparing even local commits to remote branches, which was the more appropriate way of implementing this in this environment. A simple Python program that made a get request to the API was able to get the necessary information about changed files. The script uses the latest commit made and compares it to the remote develop branch, which in this project is the main branch where all developing branches are forked from. Locally it can use the latest local commit, and in the pipeline the script can use the environment variable that was already passed to the container where the tests are ran.

The Gitlab API required a special user authentication token. As in the SUT environment many of the secret tokens and variables were stored in the AWS Secret Manager, and pulled to the project when it was built, this approach was also tried for storing the token. After trying for few times with different scripts to pull the token inside the container where the Python script was, it became clear that this created unnecessary steps for this functionality, as the other AWS secrets were not needed inside containers only with the first step where the infrastructure was built. Gitlab CI like many other CI tools also offered a way to add its own environment variables, which became the best place to store this secret token.

With the token properly stored the script could easily be called in the same script that started robot execution and stored other environment variables. The script parsed the input from the API to make a string that had all the names of changed files, which was the most important thing to map test behavior with code changes in a higher scope. Code example 4 shows how the program was done with Python.
import urllib.request
import json
import sys
import argparse

URL_BASE = "https://<the-url-of-desired-gitlab/api/v4"

parser = argparse.ArgumentParser(description="Lists changed files between de-
velop and given branch.")
parser.add_argument('--token', type=str, help='API token.')
parser.add_argument('--sha', type=str, help='git sha or branch name')
args = parser.parse_args()

if args:
    url = URL_BASE + "/projects/xxxx/repository/compare?from=develop&to=" +
    args.sha
    headers = {'Private-Token': args.token }
    req = urllib.request.Request(url, headers=headers)
    response = urllib.request.urlopen(req)
    data = json.loads(response.read().decode('UTF-8'))
    AllInOneString = ""
    for diff in data['diffs']:
        AllInOneString += (diff['new_path'] + ",")
    print(AllInOneString)

Code Example 4. The Python program that interacts with Gitlab API to store changed files.

The same functionality now done with a Python program, could have been done without the need of an API just using git. This would be an efficient approach when running tests on a local machine or in a CI environment that has git installed on the execution level. Approach with using git to get file changes with Python is shown in Code example 5. This approach works only if the project git history is accessible at the execution level of this program.

REFERENCE_BRANCH="origin/master"
COMMIT_PREV=$(git rev-parse ${REFERENCE_BRANCH})
COMMIT_CURRENT= $BUILD_SOURCEVERSION
CHANGES=$(git diff --name-only "${COMMIT_PREV}..${BUILD_SOURCEVERSION}" | tr '\n' ,)

Code Example 5. Python program parsing file changes from a local git environment.

After the file changes have been mapped to a variable by the script shown in Code example 5, they could be added to the TestArchiver in a similar way as shown in the Code example 3.
4.3 Graphical User Interface

The front-end of the application was done with Node.JS [9.] It uses Express framework for implementing the REST interface that makes queries to the back-end, and Pug templating engine for implementing the static web pages [8.] Reasons behind these technical decisions were previous experience with them and the idea to build something effectively working without using too much effort on it. This is addressed later in this thesis in the future improvements part.

The dashboard of the application is designed for multiple purposes. It is built to be a tool for developers and test automation professionals, and to be an easy-to-understand visualization of otherwise harder to acquire data for non-technical personnel. As a result, it has a few different kinds of views that differ a bit from each other.

The idea is that when an organization or a team starts using the application, they can select which views they desire and use them with a little modification to suit their needs. For example, they can choose how many runs of tests they can view in the page that contains “X” number of the latest test results.

4.3.1 API

The REST structure that is used to communicate with the Python back-end was implemented with Express. It also uses other Node.JS middleware such as request-promise and promise which are required for synchronous calls to the database. The API was implemented for a complete CRUD structure, but it mainly uses GET-requests, since its sole purpose is to get data from the back-end so it can be visualized.

The requests when changing a view, make a few parametrized calls to the back-end, depending on the state of the program, and refactor the results json data to be passed to pug as JavaScript objects. Code example 6 is an example get function that retrieves the data from latest run of test execution.
The function at Code example 6 makes three different requests to the back-end asynchronously with “Promise.all” method, maps the results and returns a json data which is passed on for res.render method. The requests are asynchronous, but the logic is handled in a synchronous way with callback functions. Res.render method then tells pug which view to render and then maps data to variables defined in that view file. In this case these variables are the title of the page which is a hardcoded string. The rest of the values for testData, lastData and serieName are acquired from the requests made for back-end.

```
router.get('/lastrun', function (req, res, next) {
    let url1 = SERVER_URL+'/data/series/'+allSeriesID+'/results/?last=1';
    var url2=SERVER_URL+'/data/series/'+allSeriesID+'/suite_status_statistics/?last=1';
    var url3=SERVER_URL+'/data/series/'+allSeriesID+'/test_status_statistics/?last=1';
    Promise.all([url1,url2,url3].map(value => {
        return rp({
            uri: value,
            json: true
        })
    })).then((data) => {
        console.log(data);
        res.render('lastrun', {
            title: 'Last test run',
            testData: data[0],
            lastData: data.slice(1,3),
            serieName: lastSerieName
        });
    }).catch((err) => {
        console.log(err)
        res.render('error')
    });
});
```

Code example 6. GET function that fetches all the needed information to build a view of one test ran.

The rest of the GET requests used for communicating with the back end have similar principles with the addition of having parameters when searching for specific data series. The URL used for the back-end calls comes from an environment variable as it needs to be changed depending on the environment where the code is executed.
4.3.2 Manager View

Manager view is designed to have easy-to-read charts and most relevant information to people who want to keep an eye of the test execution status. It could be used as a lava-lamp kind of board that shows current status of last test runs. If a developer team sits on the same room, they could have a monitor showing this view, and they would always know the status of their test executions. A screenshot example of the manager view is shown in Figure 5.

![Manager View Screenshot]

Figure 5. Screenshot of the top part of the Manager view.

The manager view shows the result from last test execution with an option to toggle between tests and suites. It also has a similar donut graph for the whole project history data. Below them is an option to dive in to the last test execution data and a list of the most fragile test cases. On the left side of the screenshot shown in Figure 5, there is a navigation bar that gives access to all the different views on the software, which is similar with all the different views.
Scrolling the page down the manager view more gives access to other diagrams. The first one from the bottom left in Figure 6 below shows the pass/fail ratio from all the test runs in the project. The number of tests shown can be limited to specific situations depending how many builds the user want to see. This kind of graph gives the user a fast way to see the trends from all the test executions. The second graph is similar but from a single branch. The application automatically shows this kind of diagrams for the 5 most active branches that are used in the project in this manager view.

Figure 6. Screenshot of the lower part of the Manager view.

Similar kind of graphs can be used for automated tests that are ran in a production environment. If tests are running daily or for instance overnight, this kind of dashboard can instantly tell viewer if some of the tests overnight have failed and some action is needed for repairing the software under test. For management level people this kind of information could be impossible to get without similar kind of solution as this.
4.3.3 Single Test Run View

Single test run is a view that shows the data of individual test build. This view can be sometimes more useful than traditional test reports from native reports of testing tools for two reasons. First it is very likely a lot easier and faster to find a specific view from history with this kind of application. Second benefit comes from the ability to show users a complete test build, even if the build is divided and ran in smaller parts. Running tests in parallel is often even recommended but the problem is that one might end up with increased number of different reports from the test executions and thus being unable to properly read them or compare them to other test results.

Proper implementation of integrating this application to the system under test, allows the usage of either build numbers from CI, or other metadata to combine test executions together to one build. It can then be viewed in the applications perspective as one singe build, like it fundamentally is. Also, with a proper implementation the application will also be able to tell the user that a specific test build was combined from smaller parallel test executions. This gives users the ability to compare test builds together whether they are run all in the same execution or split between smaller groups of tests. The screenshot of a single test run view can be seen in the figure 7.

![Screenshot example of a test run view](image)

Figure 7. Screenshot example of a test run view
The graph on the right corner in Figure 7 is similar to the ones in manager view and shows the total test pass percentage. The left div element on the page contains all the metadata stored with the test data. It shows the build number, the number of test suites in the build and the branch that the tests are executed against. It also shows other CI specific metadata such as cijobid, cipipelineid, and the URL to the pipeline where the tests were executed. In this example the testing environment was Gitlab CI, so the link would give users straight access to the pipeline execution on Gitlab side. Users can also see other metadata such as if the tests were executed locally or in the CI pipeline, the commit-SHA from the git commit and who the user was that executed the tests. A lot of this data is nice to know at this point, but it can become very valuable in some cases when customizing the software for specific purposes.

It is noteworthy to mention that the software is built in such a way that it shows all the metadata that is embedded with the test results. This way users of the software can add any metadata that they desire to the results and can see them at this view. There are various kind of projects that all have their own important metadata that can affect test execution or otherwise be useful information to have later.

Below these two containers there is a complete list of all the test suites executed in the build. The list shows the status of the suites, their names, the names of the tests inside these suites, time elapsed in executing these suites and tags given to the tests or suites. This is a very easy to read presentation of the data that already is a lot more than most of the test frameworks natively report.

After the software was ready in the scope of the present study there have been additional views implemented in the application. One of the most important of them was a single test view that was designed during the project. The single test view was designed to have all the information of a single test case that possible can be found in one place. This can mean screenshots from the test execution and whatever metadata is wanted to present there, together with test execution times and obviously the status of the test. The single test view extends this single build view further and can be accessed by pressing one of the executed tests on the list in the single test build view.
4.3.4  History View

History view is the single most unique feature of the application. It is a tool for test automation personnel to get all the necessary data that could result in affecting test behavior in one view. This kind of representation of test data cannot be found from the readymade reports of any test frameworks. This view shown in Figure 8 allows developers to find trends and similarities from different test builds, to identify unstable test cases that inconsistently change their status, and to dive in to more details of a specific test build faster than with any other solution. This is all build from the data that all modern continuous integration environments have but have not visualized in a history-based view to achieve these benefits.

Figure 8. Screenshot of the History View.

From the top-left button shown in the Figure 8, users can open a list of branches and change the content to get branch specific history views, by default it shows the “main” branch of the project which in this case is develop since the project is not in production yet. By clicking from the build columns in the page users can open the single test build view from that specific build to get more details.
4.4 Back-End and Database

The back-end, database and testing of the software is briefly described in this section. The aim of this thesis was more in the visual elements of the software, but the system would not have been able to work without a back-end and a database.

The application uses a server written with Python to work as a back-end for the dashboard application. This approach was used to make it easier to change from one service of the software combination to another and keep the fundamental idea of having all the logic separated to smaller pieces. Separating the logic allows another type of front or back-ends to be switched during the development more easily. The back-end server uses Tornado web server for performing SQL queries to the database. It handles asynchronous events with generator-based coroutines. They work in similar kind of a way like promises with JavaScript and are often easier to read than chaining call-back functions.

The application uses PostgreSQL database as a storage for the test data from the system under test. The database is hosted with AWS web service called RDS (Relational Database Service). RDS gave very efficient tools for creating and monitoring the database and was overall very convenient fit for the needs of this project. PostgreSQL together with AWS also offer easy scalability for the database which might come in handy when the amount of test data will increase in the future. [12.]

The schema for the database is designed on the TestArchiver project. It was also partly changed on the TestArchiver project side to meet all the needs of the visualization project at hand. The schema of the database and the decisions made for creating it can be found from the GitHub repository of the project. [11.]
4.5 Testing

The application itself uses Robot Framework for different kind of tests and the idea is that when it will be released as an open source project, people can use the test-set of the application to contribute on the development of the project. The testing happens in a container environment with an open source tool called Zalenium.

Zalenium is a tool developed by Zalando designed to ease interacting with Selenium grid. It creates selenium nodes that are used as a test execution environment for Selenium tests, which in this project are further abstracted with Robot Frameworks selenium library. Zalenium offers features such as video recording with a dashboard that collects your test execution videos, live preview of the GUI while the tests are executed inside containers and more. Zalenium uses docker-selenium so users do not have to think about mapping corresponding web drivers with their browsers since docker-selenium takes care of that. Zalenium supports Chrome and Firefox and has an option to redirect tests for other browsers to specific cloud testing providers, which are not free to use like Zalenium itself. [13.]

4.6 Environment

The stack of the application can be started with a docker-compose cluster, as individual docker containers or without docker at all. If started with docker, the prerequisites needed are only docker and docker-compose. The stack works with any operating system that supports docker and does not require any additional installations this way. All the required prerequisites of Node.JS and Python dependencies are automatically installed to their own containers to ensure that they are always correct and up to date. The application was designed to also work without docker. This need comes from real life situations where some customers might be unable to use docker or cloud environments.
5 Future Improvements

For the scope of this thesis the first version of the software is ready, but it will be further developed in many ways by the Siili employers and hopefully everyone else who develops an interest in it, once it is open sourced.

5.1 Open-Source

The idea is to open-source the software and get public interest around it to raise the awareness of the benefits of storing and visualizing test data, to spark new ideas of how this type of data could be useful, and to develop the software even further. The complete structure of the application is implemented with the idea in mind that it will be open sourced, thus made easy to replace parts of it to fit in a specific situation. The plan is to release the application in GitHub, although a few additions will be made before it is opened to the public.

There is an idea to develop a test set for the application which could be used for the development of it. The application would visualize its own test data and would be independent of other systems to test, which would ease the work to start contributing for the project. This way there would also be an example of the metadata storing part in the same project repository, which currently is implemented only in the internal software KnoMe that was used for development.

The testing environment of the software currently supports Unix based operating systems, as it uses bash scripts to start docker-compose stacks, and before open sourcing the application, a solution to easily run it on windows machines needs to be implemented.
5.2 Prioritizing

The internal goal at Siili is to develop an advanced version of a prioritizing server for the project, which would efficiently handle prioritizing of tests to be executed when certain parts of the system under test is changed. This prioritizing server work has already been started and will most likely be fully implemented in the version open sourced in the future. The idea is to benefit from the file changes stored from the test automation data to map them together with the test behavior on the same execution.

5.3 Restructuring

The goal of the present study was to create a working solution for this specific type of use case, and to study the best ways of implementing it. The project was able to reach the desired state with the technologies decided at first but depending on future requirements there might be some parts of it that will change.

The graphical user interface which was implemented with pug was a sufficient enough solution with the minimum viable product requirements set for the project. If the complexity of the user interface will increase in the future, it could require usage of some modern JavaScript frameworks such as React or Vue.JS. Both of them would also have their own libraries for converting the software from web application to native mobile applications, should a desire for that kind of functionality arise. Handling the state of the application would also become easier with libraries such as Redux and Vuex.
6 Feedback

The original idea was to group possible users of the software in to two different groups, but after the actual project, it was concluded that there were three different user groups the software is targeting. The roles in which technical experts work in projects can vary quite a lot, which resulted in dividing them to two smaller groups, Developers and Automation specialists.

The feedback section is written based on verbal feedback from various people that have attended one of the presentations where the project was presented or based on other verbal conversations about test data visualization needs in some of the customer projects where the author has worked before.

6.1 Feedback from Management Level

There has been a clear interest from management level people in this kind of software that would be easy for them to use. The feedback has come from different types of organizations that do test automation in a variety of different scopes. The application currently works for one project, and a clear need has risen from the feedback for support for multiple projects from the same dashboard.

To summarize the feedback from management level people, there is a genuine interest for the software, and they are eagerly waiting for its next versions.

6.2 Developers

There is an increasing number of developers who are not necessary experts in test automation but are still in a position in their projects where they also work with tests. From the feedback it can be understood that this group of people want easy ways of seeing what went wrong if a test fails to be able to quickly fix either the solution they have made or the tests. In some cases, this can mean screenshots of the test executions or just seeing the test logs from different tests in a more efficient way than currently.
6.3 Automation Specialists

The feedback from automation specialists has been very straightforward. All agree on the difficulties of finding what causes some tests to become flaky. Everyone that have heard about the idea of combining all the necessary information into one single page where it is possible to compare them to other test runs have had a clear interest for the concept.

7 Conclusion

The application created as a solution to study ways to solve the problems of modern automation projects was a success on its scope. As a result, the first working version of this kind of dashboard software has been now been used for representing the benefits that can be gained from visualization of automation data outputs and metadata bundled together. The project has been a very good first version that enables other people to work on improving it and creating next versions of it with less knowledge of the context that would have been required without it.

The project has been presented at a few internal events at Siili Solutions and at one public event during Robot Framework meet-up at Helsinki. The feedback has been very promising, and there clearly is an interest to this kind of software in the market. Team SALabs at Siili has gained a lot of valuable information from this project, and thus intends to continue developing it to meet the personal requirements for Siili consultants to use it on their customer projects, while also increasing the visibility of the project. The more visibility the project gets, the closer we are at re-inventing the practices used for regression testing.
8 References


Figure links:
