

**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

BACHELOR THESIS

Michal Kyjovský

**A configuration system for automated
testing**

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: doc. RNDr. Petr Hnětynka, Ph.D.

Study programme: Computer Science

Study branch: Software and Data Engineering

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In date
Author's signature

I would like to express my sincere gratitude to my supervisor, doc. RNDr. Petr Hnětynka Ph.D. for his willingness to supervise this thesis and all his assistance. Also, I want to thank all my former colleagues from the Mantis project who brought me into the topic of Big-Data, Software testing and automation.

Title: A configuration system for automated testing

Author: Michal Kyjovský

Department: Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: doc. RNDr. Petr Hnětynka, Ph.D., Department of Distributed and Dependable Systems

Abstract: Software testing is an essential part of the software development life cycle. Before there was extremely rapid progress in development methodologies and tools, it was harder to fulfil requirements relating to validation and verification. One of the most applied approaches is the automation of the testing processes. There is a plethora of technologies for automated software testing, such as Tosca, Selenium or the Robot Framework. Unfortunately, in practice is very common that testing is not taken as seriously as the development phase. As a result of this lack of consideration, it often transpires that the execution of tasks related to software testing is conducted by employees who may not be appropriately qualified; therefore, they can find the configuration and utilisation of these tools to be difficult. This thesis aims to design, implement, and test the web-based application, which will provide the tester with an easy to use environment for the configuration and execution of automated tests implemented in the Robot Framework. The objective of this thesis is to provide an environment that will prevent the formation of defects caused as a result of human error.

Keywords: Software testing, Python, Robot Framework, Automation

Contents

1	Introduction	2
1.1	Goals	2
1.2	Outline	3
2	Technologies	4
2.1	Software Testing	4
2.2	Automation Process	7
2.3	Robot Framework	8
3	Functional Requirements	12
3.1	Technologies used	16
3.2	Model Requirements	18
3.3	View Requirements	19
3.4	Controller Requirements	20
3.5	Supported Features	20
4	Nonfunctional Requirements	22
4.1	Integration	22
4.2	Extensibility	22
4.3	Security	22
4.4	Testing	22
4.5	Continuous Integration	23
4.6	Code Quality	23
4.7	Development Tools	23
5	Implementation	24
5.1	Architecture	24
5.2	Plugins	33
6	Case Study and Evaluation	34
6.1	Example of usage	34
6.2	Related Work	47
6.3	Unsupported Features	48
7	Conclusion	50
7.1	Future Work	50
	List of Figures	55
A	Attachments	56
A.1	Contents of the Attachment	56

1. Introduction

Since software and information technologies have become common in almost every industry sphere, software requirements continue to evolve. Whilst most software applications will have been developed with a specific objective(s) in mind, it is critical that end-users can rely on the software to act as they expect. As demands from software applications can increase significantly, and thus software developers implement changes regularly, it is extremely important that any development does not affect the expected behaviour of the system or application.

Software testing is used throughout each stage of development to validate that the software behaves as expected.

Usually, testing is not a popular discipline, not for the developers when they should cover their implementations with unit tests, nor for the actual testers, who should provide unbiased test scenarios for multiple stages of the system testing.

In the following text, we primarily focus on the Software testing performed by the unbiased tester, hence someone who did not contribute to the code, because if the testing would be the only developer's responsibility, we would possibly encounter more bugs than if it would be tested by someone who does not have the inner view into the source code. The crux of this approach is that the testers are usually limited by time and resources, not to mention the knowledge base. The issue with the resources means that in larger companies, the environment employees use is pre-defined, thus if the tester needs some additional tool to finish his/her job, he/she must proceed higher authority to obtain approvals, then he/she must set the new environment to perform the validation process finally. This causes delays in the testing, and it is a regular phenomenon. When we speak about the knowledge base, we mean that the tester must be introduced to and understands the application or software he/she supposed to test. This is crucial in order to provide solid test coverage. Finally, there is a time limitation. When agile techniques are blooming, companies start to move from waterfall-like development practices to agile patterns. However, this is a difficult task, and in many cases, it ends in the setup that is somewhere between those two approaches. In most cases, this affects the testing process that is left to the ends and thus squeezed in time allocation.

Due to the fact that testing produces limitless ways how to verify a system behaviour, which cannot be done only by the manual testing, automated tests were introduced. In the past few years, the automation has grown in importance. Testing is now moved to the build servers, and it is usually part of some complex tasks or pipelines. Of course, we could say that it solves our problem, but the opposite is true. These automation test still needs to be developed and tested before they can be deployed into a more critical process.

1.1 Goals

The main goal of the thesis is to design and implement the extendable and scalable web application that will help beginners, and also advanced testers simplify the software testing process. The application will remove the complexity of configuration of the Robot Framework and related libraries as well as the needs

of system dependencies on those libraries from the tester's machine. The only thing that the user needs to do is upload the test scripts and let the application do the hard work.

1.2 Outline

In the next chapter, we focus on the software testing process in general, then the automation process description follows, its relation to the software testing and the software development. Finally, the Robot Framework is introduced in connection to previous sections. We describe what the Robot Framework is, how it works, what its structure looks like, and its usage.

Chapter 3 is dedicated to functional requirements on our application. We state all needs that the system must apply based on its domains. In the subsections are described formats of inputs and outputs and how the system is configured. Finally, we define the supported features.

Chapter 4 is related to the nonfunctional requirements. There we discuss demands on the development process, best practices that should be followed and the expected utilization of our application.

Chapter 5 describes the implementation of the application. We begin with the high-level architecture of the system and explain the needs for and structure of individual environments the project possesses. Section 5.2 describes the modular idea of the architecture as well as the intended extensibility that is supported or is planned to be developed.

Chapter 6 provides the example of usage of the application for each user role. In three separate sections, we explore options available to the Tester, how Admin can manage the application, and how the Reviewer can observe the executions performed in his/her project. At the end of the section we summarize the related work.

The last chapter summarises our efforts and concludes this thesis with the plans for the future development.

2. Technologies

In this chapter, we strive into the Software Testing topic. We go through the base concept and categorization, introducing the techniques and approaches conducted on a regular basis in the industry. Right after, we dive into the idea of automated testing, what the areas of expertise are, and the benefits and disadvantages. At the end of this chapter, the analysis of the Robot Framework take place, focusing on its purpose, usage and how it can help us to combine the topics mentioned earlier in this chapter.

2.1 Software Testing

Software testing is based upon the fundamental principle that humans make mistakes; therefore, the application developed by humans may contain defects too. Of course, it would not be fair to attribute the mistakes only to the developer since unexpected software behaviour causes can originate from the foundation of the software development process - requirement engineering. Requirements can be, for instance, ambiguous, wrong or misunderstood. To avoid such discrepancies, the software testing integrates the investigative procedures of software validation and verification into the Software Development Life Cycle (SDLC).

Irrespective of how basic the previously stated principle is the software testing is a challenging activity. The testing, among others, does not incorporate only the actual testing of software but also the following activities:

- Understanding the system being tested
- Understanding the requirements
- Design the test scenarios

Building the idea of what software testing is about here, we would like to remind one of the famous Dijkstra's observation: "*Testing can never prove the absence of bugs, it can only show their presence.*". Admitting this idea, we get the principal purpose of the Software Testing, and that is: "*Finding the subset of the possible inputs to a system, that gives us a reasonable certainty that we have found the major defects.*" [37]

Focusing on the testing approaches, three scenarios can happen when we are deciding which approach to choose. Before we dive into the exploration, the following question should always be answered: "*What sources of information are available for test generation?*" [37]. This is important since it puts us in the position to prepare our grounding to test and therefore re-allocate time resources based on the activity's importance. For instance, if we have missing user documentation, we should spend more time on the tested software exploration. The general rule that applies is that the more information sources we have, the better the selection of appropriate test coverage can be made.

Summarizing the previous paragraph, we can converge into one of the following situations originating from the assets provided for the testing process:

- White Box testing

- Black Box testing

Bringing out the first option - *White Box* testing, we are facing the best scenario we can since we have access to the source code, programmer's documentation and related resources. In practice, we do not have access to complete source code, for instance, in the case of externally used packages or web services. Another option listed above - *Black Box* testing denotes the scenario when we select a testing approach based on the intended functionality of the software under the test captured by the requirements. In the end, although not explicitly stated, we may face the scenario where neither functional and design specification nor the source code are provided.

Speaking about the essential resources for the testing, we cannot omit the other kind of resources that subject matter.

Firstly we need to answer the question: Who should perform the testing? The answer to this vary based on the approach applied in particular project development and through time as the techniques in the Software Development process continuously evolves. In practice, many developers apply the Test-Driven development [37] approach, which means that tests are written before any implementation starts based on a software specification and implemented features follow up to make these test pass [37]. In the past few years, it has become a shared conviction that external testers should be preferred over the developers who put their own efforts into the application development and are oriented in their own code. It results in the situation where developers disregard the weak places in their code or because the testing is intended to destroy their solution, the testing methods can be more benevolent. Therefore, the testing is currently, in most cases, covered by a specialized QA team which can be, in some cases, also the external company, which can initiate in the third situation of testing approaches we were discussing - outsourced. The better/preferred approach depends mainly on the particular situation and the system under test. Test-driven and Behaviour-driven development offer techniques that can be sufficient for test coverage; therefore, the testing responsibility can be split into more stages.

A second question to be answered: when to conduct testing? Generally, as soon as possible. The reason for this is simple; once the bugs occur in the production environment affecting the customers of particular software, it is more expensive and challenging to provide a bug fix. Here are the advantages of Test-driven and Behaviour-driven development; however difficult these approaches are to apply in practice. When a defect occurs in the production environment, moreover observed by a customer, the stakeholders need to be informed, the version of the software based on the situation withdrawn and the fixture and another testing must be conducted, which in most cases can be expensive. [37]

In the end of this section we should emphasize the major limitation of the testing - the economic decisions. However contradictory it sounds, testing is an expensive activity either on time or on a budget. Therefore the general strategy of any testing team is to test based on maximum efficiency using minimum resources.

Concepts

Having the overall idea of the Software Testing discipline, we should provide the basic formalism starting with the actual definition of Software Testing which

is clearly defined as "*an investigation procedure conducted to provide stakeholders with information about the quality of the software product or service under test*" [21]. The testing procedure then resolves in the reports justifying the software or service examination results remarking the software *defects* or *bugs* if any. As mentioned the previous paragraphs, a fault can be caused due to errors in the implementation or misunderstood user requirements.

Before we start with the description of the individual testing disciplines, we should define some basic terms:

- **Test Case** - A single unit representing the software or service examination. It uses vast amount of inputs in various forms, such as flat files, user interaction, or simply randomly generated values. Each *test case* maps the user requirement or set of user requirements.
- **Test Step** - Represents the individual verification of the user requirements or its part. *Test steps* can be also categorized on those that contains the *Verification Point*, and the regular step of a *Test Scenario*.
- **Verification point** - A *Test Step* that denotes the expected value or behaviour based upon the given input.
- **Test Scenario** - A functionality defined by the requirements that can be tested. It is a collective set of *test cases* that determines the positive and negative characteristics of the application.
- **Test Suite** - Compresses the collection of *test cases*.

Another concept to be discussed is the level of testing. To avoid ambiguity with the previously defined stages at the beginning of this chapter, the test levels are related rather than to the resources to the detail of the *test scope* and the authority responsible for the test execution.

- **Unit Testing**

- The *Unit Testing* represents the lowest level of the software testing discipline. It aims to provide individual test coverage for the software's or service's units such as classes, methods or functions. It is usual that when creating unit tests then some of the elementary units are not implemented yet. This is when mocking helps. Mocking is based on the *stubs* which technically simulates the non-implemented object behaviour for the purposes of testing consisting of trivial object handling. There is a stream of developers following the idea of so-called *pure* unit testing where there are no dependencies among the units, and all dependencies are then simulated using the *stubs*. However reasonable it looks, this approach also has a significant disadvantage in the form of mocks maintenance. Therefore in practice usually an individual unit interacts with others. [37].

- **Integration Testing**

- We are stepping further from the *Unit Tests* when we aim to test the interaction between the individual units rather than their behaviour covered by the previous level. In the scope of the *Integration Testing*, an elementary unit would be classes, modules or packages. Technologies used for this level of testing will not vary from the *Unit Testing*; however, there might be more specialized tools that can be used.
- **System Testing**
 - When it comes to testing the entire software or service as a unit, we speak about the *System Testing*. There are various ways to provide test coverage in terms of the entire system; however, it mostly depends on a particular software. We can imagine the GUI under test providing the buttons that have dedicated behaviours. This particular case used to be an example of manual testing; nevertheless, there are technologies and frameworks capable of automating the process if the system under test is in the desired state. Responsibility for test execution or configuration lies with the QA team.
- **Acceptance Testing**
 - The final piece of our testing pyramid is the *Acceptance* testing. The main difference between the *System Testing* and the *Acceptance Testing* is who performs the tests; otherwise, these processes are similar to each other. In this particular case, the executor and observer is the customer who uses the system or service under test.

2.2 Automation Process

With the knowledge of the fundamental principles of the *Software testing* described in the previous section, let us introduce the process of *Automation* in the Software Development Lifecycle and how it can improve the *Software Testing*.

In the past few years, automation has increased significantly in the software industry. This is due to the major benefit it brings - reducing costs. Nowadays, it is a booming trend that the tasks that humans, such as reporting, had performed are now effectively automated, so teams' occupancy can be reduced or re-allocated.

The use-case of automation that meets the interest of *Software Testing* is the **Automated Testing**. What benefits the automation in this scope brings:

- Repetitive executions
- Errors caused by human factor reduction
- Efficiency in speed
- Less testers needed
- Process transparency

Considering the advantages listed above, the *repetitive executions* gives us **Regression Testing** which assures us that even if the source code of the particular unit was not changed, the dependencies changes did not affect the verified behaviour, or the previously fixed defects did not appear again. The Regression testing can be performed on any of the mentioned testing levels, and in practice, it is part of the CI.

What can be considered a disadvantage is that the architect of the automation test must be more experienced than the manual testers. Another downfall is that if all test executions conduct the automation process and the thorough test coverage consists of a vast amount of tests, we may face computational limitations. Therefore a prioritization of individual test suites must be introduced, taking into account the priority and necessity for regression.

2.3 Robot Framework

Combining the sections above and remembering the goal of our project, we got to the introduction of the automation testing framework developed in Python - the **Robot Framework**.

When we were deciding which automation tool to use, there were no hesitations. The Robot Framework is nowadays a quickly evolving Framework widely used by companies running on Python. This is due to the huge popularity of the Python programming language which is nowadays one of the most popular programming languages [40]. Furthermore, writing the test cases in the Robot Framework offers a natural workflow, including test preconditions, action, verification. Additionally, it allows use of natural language in the description of the particular Robot Framework units that are propagated to the report, so even a non-technical person can quickly get oriented.

Introduction

"Robot Framework is a Python-based, extensible keyword-driven automation framework for acceptance testing, acceptance test-driven development (ATDD), behaviour-driven development (BDD) and robotic process automation (RPA). It can be used in distributed, heterogeneous environments, where automation requires using different technologies and interfaces." [32]

The definition above is taken from the official documentation and clearly describes the tool we use in our solution. The motivations for selecting this particular technology were:

- Easy-to-use tabular syntax
- Custom keywords that are human readable
- Platform and application independence
- Provides well arranged outputs capturing the test execution results
- Provides level setup for test cases and suites
- Easy integration with version control systems

- Provides CLI
- Vast amount of libraries and framework for various use-case scenarios

Surely, there is many more benefits and also disadvantages starting with the limited tools support that gives us pretext for this project.

Architecture

Figure 2.1 demonstrates the high-level architecture of the Robot Framework. The *Test Data* box denotes the easy-to-edit tabular format, continuing to the *Robot Framework* box that demonstrates the core unit on its start processes the data, executes test suites and test cases and generates the output files. *Libraries* provide additional functionalities to the framework, so it can make lightweight testing anytime. *Libraries* then can use the API of the application directly under test or via low-level specialized tools denoted by the *Test Tools* unit.

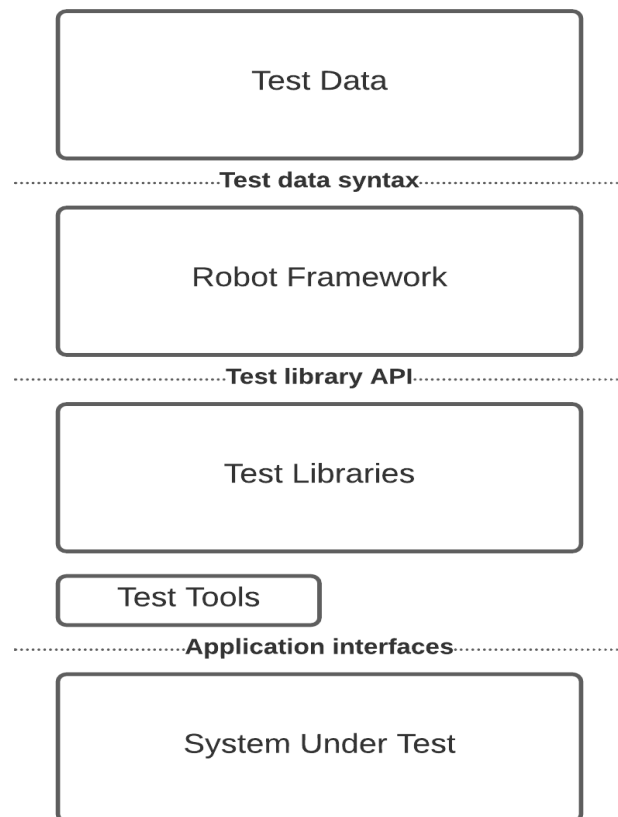


Figure 2.1: Robot Framework - Architecture

Creating Test Data

In order to arrange test cases, the following structure needs to be followed:

- Test cases are placed in the *test case file*
- A test case file is automatically wrapped by a test suite

- A directory that contains *test case files* forms a higher level test suite
- A test suite directory can contain recursively other test suite directories
- A special initialization file for the test suite directory configuration can be used

In addition to the structure above, there are also the following specializations:

- **Libraries** containing the auxiliary low-level keywords
- **Resource files** with variables and higher-level user-keywords
- **Variable** files to provide more flexible ways to store variables

The **Resource** files usually follow the Robot Framework syntax as opposed to the **Libraries** and **Variables** files that use the syntax of the programming languages supporting the Robot Framework, hence in most cases - the Python and Java.

Supported file formats

The common structure of the Robot Framework data is the *space separated format* where units of data or keywords are separated by at least two or more spaces, eventually a TAB character. We can substitute the previous approach with *pipe separated format* which replaces the numerous space separators by a single '|' pipe character surrounded by spaces. Both these approaches bring the advantage of test data file formatting being as easy-to-read as possible.

Robot Framework implicitly expects the *.robot* extension for an execution. This behaviour can be changed by the '-extension' command-line option, which result in additional usually used file extensions in practice such a *txt*, *rst* or *rest*. In the case of the *reStructuredText* format, the Robot Framework data must be embedded into code blogs. In the obsoleted versions of the framework was also common to use the *HTML* and *TSV* formats. Those are not supported now; however, they can be processed in the form of plain text by the version of the framework 3.2 or newer. [32].

Test Data Example

The example below (Figure 2.2) demonstrates a simple Robot Framework test suite. We can observe that the definition of the header has the format in the form of the table name prefixed and suffixed by the '***'. Describing the code, the *Settings* table is the place where the imports of the Libraries, Resources and Variables files come as well as the documentation and the settings command that applies for the entire suite and should be executed only once. The *Variable* section defines all global variables and constants relevant for the particular test suite. The *Test Cases* table consists of the individual test cases that will be in order (if not changed by the CLI options) executed. Each test case is built from the keywords defined in the *Keywords* table. Additionally, if we do not intend to use the Robot Framework for testing but the automation task generally, we could in the example, and also generally substitute the *Test Cases* table by *Tasks* table.

```

1 *** Settings ***
2 Documentation      Example using the space-separated-format.
3 Library            OperatingSystem
4
5 *** Variables ***
6 ${MESSAGE}        Hello, world!
7
8 *** Test Cases ***
9 My TC-Example
10     [Documentation]  Example TC.
11     Log             ${MESSAGE}
12     My Keyword      ${CURDIR}
13
14 Another TC-Example
15     Should Be Equal  ${MESSAGE}    Hello, world!
16
17 *** Keywords ***
18 My Keyword
19     [Arguments]     ${path}
20     Directory Should Exist    ${path}

```

Figure 2.2: Space separated format example

Custom Libraries

Speaking about the Libraries files implemented in Python or other language supporting the Robot Framework, we have a vital tool for creating custom keywords that might be more efficient or specialized to a specific purpose. The only thing that needs to be followed is the naming convention. Regarding Python, we mean the *snake-case* convention as common practice for Python programming language according to a PEP-8 specification [24]. The functions from the particular Python module are then automatically transformed into a *space separated format*.

Test Execution

The Robot Framework has a vast range of command-line options that are well documented. This means that we execute the framework using its CLI or as an imported library in case of using internally within the Python code. In the case of the library usage within the Python module, the CLI options are mapped onto a *run* method executing the tests specified by a path to a particular test suite.

3. Functional Requirements

The following chapter describes in further detail the functional requirements questioned by our application. Firstly we will focus on the use-case scenarios based on the user role they are dedicated to. Our application intends to provide three users roles; this can be, however, changed in the future. These roles are:

- Reviewer
- Tester
- Administrator

The following diagrams illustrate the use-case scenarios which are expected for a particular user to be supported. Based on further comments to each of the diagram below, we will define the application's domain upon which the requirements for our application will be defined.

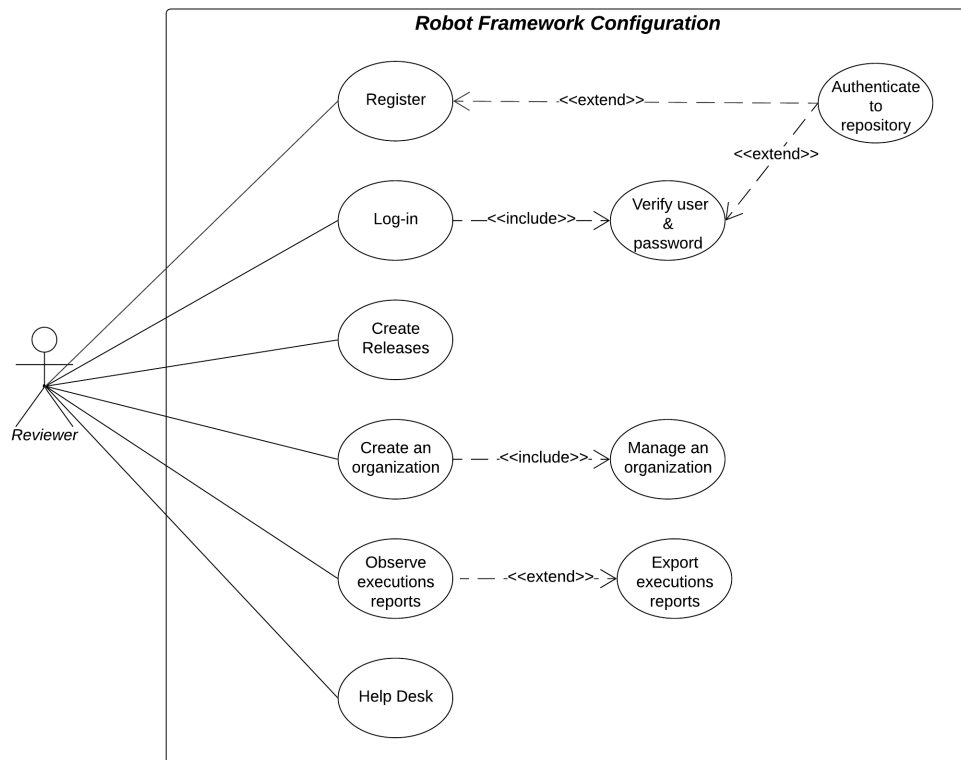


Figure 3.1: Reviewer - Use-Case Diagram

Figure 3.1 displays the use-case scenarios associated with the **Reviewer** role who is responsible for reviewing and approving the test executions. Let us describe in more detail the particular use-case scenarios from the Figure 3.1:

- **Register** - The user can register to the application either by the regular form or by the third-party application.

- **Log-in** - The user can log into the application either by the regular form or by the third-party application.
- **Create releases** - User can create *releases*. The **release** is the marking structure that consists of author and tag that enables additional test categorization.
- **Create an organization** - The user can establish an *organization*. Each user can be owner of one *organization*, but can be member of multiple *organizations*.
- **Observe executions reports** - The user is eligible to access the results of the test executions made by members of the user-related organizations or tags he/she authored. Additionally, the user can export all results he/she has access to.
- **Help Desk** - The user can approach the application's help desk team.

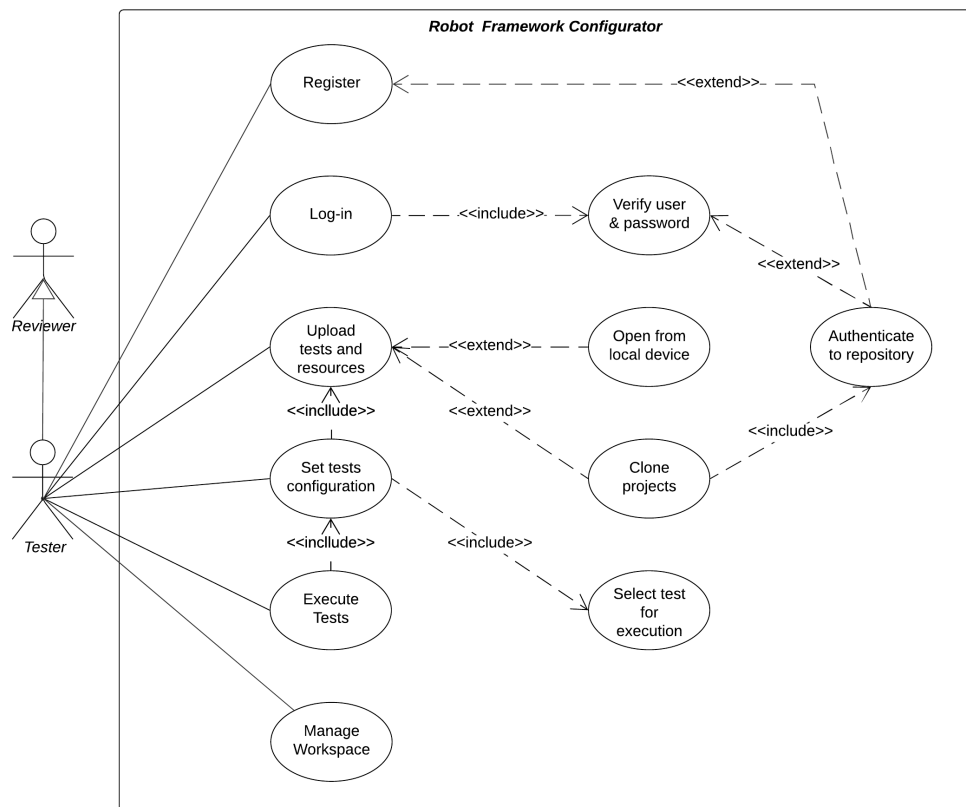


Figure 3.2: Tester - Use-Case Diagram

Use-cases associated to the **Tester** role are built on top off the **Reviewer** role. Figure 3.2 displays the particular use-cases with the following descriptions:

- **Upload tests and resources** - The user can upload the test-scripts either from his/her local file system or by cloning from the Git repository.

- **Set tests configuration** - User can set the execution options as defined by the Robot Framework CLI [32].
- **Execute Tests** - The user can based on the configuration from the *Set tests configuration* scenario trigger the test execution.
- **Monitor running jobs** - The user can monitor the jobs that are currently running or have status failed.
- **Manage workspace** - The user can control his/her workspace and files it contains.

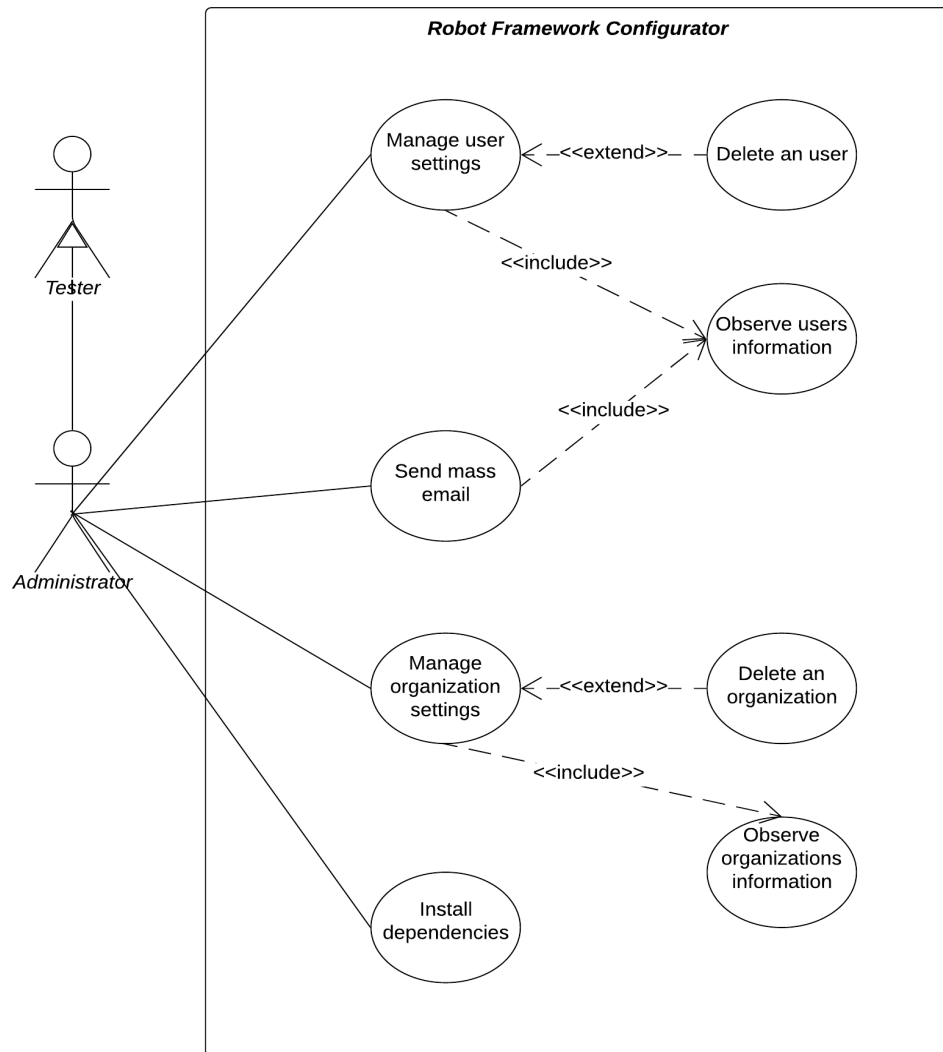


Figure 3.3: Administrator - Use-Case Diagram

The last use-case scenario we will describe is dedicated to the **Administrator** role. As Figure 3.3 denotes, the administrator possesses all abilities of the **Tester** and **Reviewer** roles; therefore, he/she can perform the tester's job as well as the reviewer's. Speaking about extensions that the **Administrator** role brings based on the use-cases in Figure 3.3:

- **Manage user settings** - The user can manage users of the application including changes of rights or deleting the user.
- **Send mass email** - The user can send notifications to the application's users.
- **Manage organization settings** - The user is eligible to manage organizations settings including the deletion of an organization.
- **Install dependencies** - The user can install missing dependencies that are required for the test execution.

Having introduced all use-cases we should now focus on the choosing of the platform that could handle these scenarios. Firstly we need to serve multiple users simultaneously, hence the parallel access. As stated previously, we want to take the responsibility of the dependencies resolution out of user's machine, hence the application should run on the remote server or a cloud. We need to have access to the users information; hence to have a persistent data storage - database that could be accessed by the administrator regardless of the time or place. Another crucial attribute is the code base and platform independence, which is easier to obtain by developing the web applications. Finally we want to application to perform its tasks in parallel, therefore we need additional machine that will compute the user's execution asynchronously so the user can use the application while the execution is being performed.

Based on the summary above, we end with the requirements that could be covered by the web application. Developing a web application is a complex task, since there are vast amount of technologies that could be used. Nevertheless, the architecture pattern are in the most cases same or at least similar. In recent years the Model-View-Controller [15] based patterns have been heavily used for the web-application. It splits web application into the three components:

- **Model** - Represents the real-world entities that are represented by a database objects. In our case we speak about the application's users, executions, organizations and their attributes.
- **View** - This component is responsible for the content that is being displayed to the user.
- **Controller** - Finally the *Controller* orchestrates all requests coming to the application and delegates tasks to the other components.

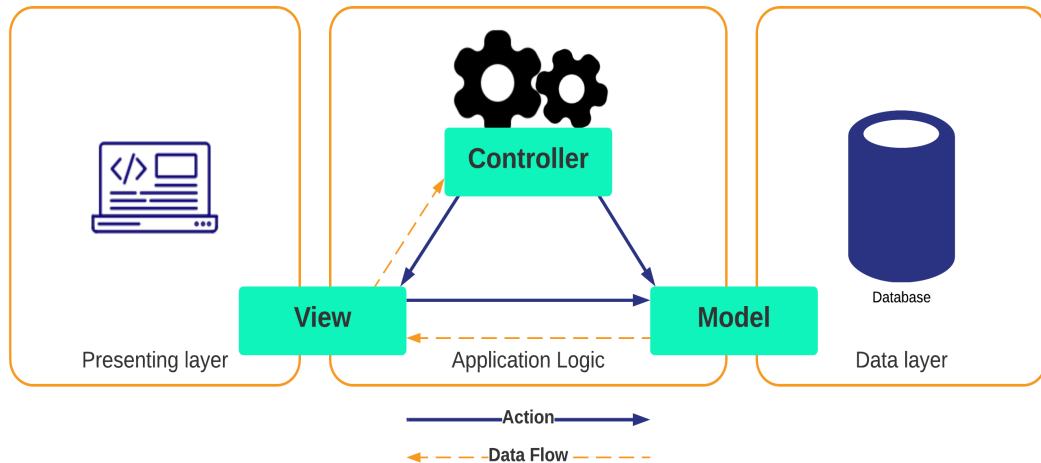


Figure 3.4: Model View Controller

The MVC pattern in its pure form however is not being used that often [15], due to the divergence it gained through the time. However, the selected technologies should bring the benefits that the pattern offers and the core idea remains same.

Based on the decision made earlier let us move to the selection of the technologies that are used with the justification of the particular selection.

3.1 Technologies used

When it comes to the choice of technologies to implement the application, we must consider the following questions. Where will the application be hosted? Are we going to support on-premise platforms, clusters or cloud strictly? How about the language support on the regular servers? Should we use some Javascript framework over the separation of back-end and front-end? Furthermore, there are many more questions to be solved, but the leading one is what technologies we should use to avoid many dependencies in our project. The following subsection will answer those questions with the additional justification of the particular choice.

- Python 3+

Even though the Python represents a minor part of the servers default support [41] as opposed to PHP for instance, the argument for this choice is that most of the Linux-like system comes with the default Python pre-installed; moreover, the testing framework chosen as a crux of this application is implemented in the Python as well. Thus, to simplify the integration between the web application and the part of the system responsible for the test configuration, execution and evaluation, Python version 3 is considered the right choice over the concurrent platforms such as PHP, Java, .NET or Node.js.

- Flask

The Python programming language has gained significant growth in popularity in recent years; therefore, many web frameworks were introduced to supply market needs. From all possible frameworks that Python offers, we chose the Flask microframework [10]. It is a lightweight option compared to Django, Dash or TurboGears, for instance, that suits our needs ideally and does not bring any additional functionality that would increase the implementation complexity. Even though we build our application based on the MVC pattern, which Flask does not implement due to its lightweight architecture as opposed to Django, we have tools to faithfully approximate the approach.

- Docker

When we want to develop an application that should work regardless of the platform or provided host machine; virtualization and containerization are the right choices. The original intention of approaching this topic was to create a virtual box for each environment. However, this approach is highly insufficient since we would have to create a virtual machine for three environments, which would make our development more difficult, not only for ourselves but also for the machine we develop on. This is where Docker comes to word. It is an efficient containerization platform that uses lightweight containers that exploit the host machine instead of the virtual boxes.

- Docker Compose

The challenging part of our containerized architecture is the orchestration, configuration and interconnection of the containers. We need to solve the precedence of the build process. Also, dependencies among the containers take non-trivial attention. The cutting edge technology to solve this problem nowadays is the Kubernetes [22]. The Kubernetes, however, is intended for larger systems supporting clustering. Our application is not intended for such architecture, and using Kubernetes for a single node does not make much sense. Rather than Kubernetes, we will be using the Docker Compose as it perfectly matches our needs.

- Redis

To make our application run fast, so the user is not limited when it comes to the execution of tests, Redis will be used. Redis [28] is the fast in-memory database, cache and queue that enables us to run asynchronous tasks, so the user can use the application without waiting for the HTTP response.

- RQ

The Redis Queue [34] is the Python library that connects our application to the Redis server, and it can establish the tasks that suppose to be executed and deliver the results via intuitive API. Another use-case the RQ is used for is the monitoring of the tasks. This comes in handy when the task fails or takes significantly more time than it is supposed to and thus needs to be cancelled. RQ is, however, not the leading API in the case of communication

with Redis; this belongs to Celery [4], on which the RQ is inspired. However, when it comes to the cognitive complexity of code, the RQ won since it was easier for later integration than the Celery.

- PostgreSQL

Our application will be using the specialized Docker container with configured Postgres [25] database to provide permanent storage of data. Postgres is the open-source, lightweight, and highly effective database server, which perfectly fulfil our use case. Compared to other popular databases such as MongoDB or MySQL, Postgres possesses reliable ORM support in Python and its framework; thus, we do not have to implement it by ourselves. Furthermore, this ORM support comes integrated directly in the Flask microframework; therefore, we reduce the project dependencies.

- NGINX

NGINX [23] is nowadays one of the most popular web servers that can also be used as a reverse proxy and load balancer. Oppose to the Apache, and it is built on top of event-driven architecture, which in some instances can be more efficient than the process-driven architecture used by the Apache [2]. Even though Apache is still leading in the case of deployed web servers (it comes pre-installed on Linux servers), NGINX showed its potential in the past few years and had a more innovative approach to the webserver configuration.

- Gunicorn

Gunicorn [17] is the Python WSGI HTTP server used for running our application and serving our application to the user based on incoming HTTP requests. What may be considered a limitation when we talked about a portable web application is that the Gunicorn is for Unix-like systems only. This was limiting when docker was not used, and the application was developed on a Windows machine. Omitting this fact, since our application is containerized in various environments, it does not bother us anymore. One final note, the Gunicorn was the second choice. The application initially ran directly via uWSGI, but let us face it, rather than another configuration file, we can use the specialized API.

Gleaned from the categorization made up upon the MVC architecture we classify the application's requirements into the components they relate to.

3.2 Model Requirements

The following section describes the functional requirements for the *Model* component. Requirements state the way the data are treated as well as the design and technologies the model uses. The **Model** consists of the database storage for the application's data and the API that communicates with the **Controller** component. Finally is contains the in-memory database for our asynchronous tasks.

- Entities representation
 - The database model will be built on top of SQLAlchemy as stated in the Section 3.1 in order to simplify the implementation, ability to change database, overall complexity and increase the security. .
- Database server
 - The database will be running on the standalone server different to the one that hosts our application.
- Database provider
 - Referring to the Section 3.1 - the database provider will be PostgreSQL.
- Database security
 - The database will have one maintainer and will be password protected.

3.3 View Requirements

Requirements placed upon the *View* component describes the expected behaviour from the application in terms of the user interaction. Furthermore they specify requirements on API communicating with the *Controller component*.

- Responsive design
 - In order to allow users to work and collaborate regardless of what device they use the application must provide GUI capable of adapting to a device preferred by the user. To achieve this goal the Bootstrap [3] will be used to simplify the implementation and to have a reliable service that is widely used in the web development.
- Browser agnostic
 - To allow to the user work from any browser he/she either has to, or is willing to use the application must be independent on the browser. This means to avoid any browser specific implementations of the GUI that would tend to the functional limitation of the application.
- Asynchronous operations
 - The application will use priori asynchronous calls to allow user to use the application while the appropriate task is being performed. The responsibility for the asynchronous operations will be taken by AJAX library.
- Proxy
 - The Web Server will hold connection with the client and the application over the proxy.

3.4 Controller Requirements

Finally, we move into the *Controller* component to define the remaining requirements. What needs to be covered in this section is the format of the inputs and outputs the application allows or supports. Afterwards, we want to define requirements for the fundamental functionalities and interactions with other components.

- **Supported file extension for uploading**
 - When the user uploads test scripts using local file system, only pre-configured file extensions are allowed.
- **Workspace**
 - Each user has a dedicated workspace, where the uploaded and generated files are stored.
- **Remote Repositories**
 - The user can use for script uploading his/her Git-based repository, which supports OAuth 2.0 protocol.
- **Executable file recognition**
 - Only the files that meet the Robot Framework specification are provided as an option to the user for the execution.
- **Executable file validation**
 - All files that meet the basic criteria for Robot Framework Configuration and Execution will be validated against Robot Framework header definition before being provided to the user for configuration option.

3.5 Supported Features

This section aims to state all supported features that our application possess in the context of this project development. Features that are not explicitly listed in the list below will not be implemented or required.

- Test execution of Robot Framework test scripts in supported formats defined by the Robot Framework specification [32]
- The application will allow the user to configure the Robot test scripts in the same range as the Robot CLI offers.
- The user can download the results of the test execution in the original format generated by the *rebot* facility.
- The user has dedicated workspace, that cannot be affected by others users actions.

- The user is allowed to upload the test scripts either in the archive format or the multiple line format directly from his/her workstation.
- The user can authenticate into the application using Github, GitLab or Bitbucket and upload their scripts directly from their remote repositories.
- Export of user's executions into the CSV format.
- Filtration of the files that can be used for the test execution.
- The user can edit his user information, more precisely its username, email, password and the profile image.
- Users can create organizations and span their test execution under the particular organization.
- The user can tag their execution, so they are grouped under the tag id.

4. Nonfunctional Requirements

This chapter will focus on the non-function requirements of our application that impact the installation, deployment, architecture, and development process. This chapter and Chapter 3 are not final since the intention is to continue with the development of this platform. However, they are final in the scope of this project scope.

4.1 Integration

The application should be capable of running on any environment supporting Docker and Docker Compose or the Linux-like system, with pre-installed software dependencies described in Chapter 3. The application is not intended for the cluster architectures; thus, it is limited to a single node. The reason for the requirement to use containerization using the Docker platform is to have a portable solution. Therefore the hosting machine needs only the Docker installed. Furthermore, these days the Docker is the standard for the containerization that the most companies use.

4.2 Extensibility

Due to the wide range of use cases that could be explored in the Software Testing process, our application will not fulfil all projects or companies' needs. The scanner in the scope of this project suppose to support the back-end test execution and results export. The scanner should follow the modular architecture to extend the functionality without significant changes to the application. For instance, when we talk about supported git repositories, we might like to extend the currently supported repositories or completely change them. Also, the format of execution export could be adjusted so it fulfils user needs. All supported features can be observed in Section 3.5 as well as the list of unsupported features in Section 6.3.

4.3 Security

Requirements on security are always the most difficult part of the software development and maintenance process, as threats are continuously evolving, and therefore so do the requirements on the application's security. Nevertheless, the application is not intended as a public service, but rather a custom solution running on an inner network, with limited access. However; we cannot be careless on the subject of security. Security is described in more detail in Chapter 5.

4.4 Testing

Every application should be tested, and this application is no exception even though it is intended for *Software Testing*. Testing of a web application is not

trivial. Our unit test coverage will be focusing on the endpoints that provide service for asynchronous calls and the package dedicated for the extraction and robot execution. Those are modules that can be unit tested and verified. In terms of integration testing, it will not be covered in the current project scope.

4.5 Continuous Integration

In order to make the development process organized and manageable, we will be using the Git repository for versioning our codes. The development process will follow the GitFlow [16] git branch model. This means that the development of an individual feature will be performed under the feature branches, and when the feature's development is completed, it can be merged into the mainstream branch.

The ability to build and the code quality will be checked via the continuous integration pipelines. Before any feature branch can be merged into the mainstream branch, it must pass each of the pipeline checks, so the code in the mainstream branch presume stable.

4.6 Code Quality

The quality of the application's source code will be controlled via an automated CI pipeline within the git repository as well as during the build process on the production environment. This ensures that no unstable code is delivered to the customer. The Lint [26] tool will be used for quality checking purposes.

4.7 Development Tools

The developer shall utilize the tools that conform to requirements related to the code quality and continuous integration.

5. Implementation

The following chapter describes the architecture, individual components, and the technical solution of the application.

The application is built on top of the Docker containers. Container architecture varies between the environments the application runs on; those are the Test and Production environment. We also have a development environment that can be run locally; however, it requires all dependencies and tools to be installed on the developer's machine. The purpose of this environment is the GUI development, where the advantage of the Flask build server can be used in full range, rather than within a container.

Container structure for the **Test** environment consists of container hosting the Postgres database, Redis container responsible for running asynchronous tasks and providing in-memory database so that the users can interact with the application without any limitation of application usage. In contrast, the job is running, Worker secures communication hence the task queueing between the Web application and the Redis, and finally, the Web Application container, which holds the web server and the application itself.

Speaking about the **Production** environment, all we need to do is extend the architecture of the **Test** environment by the specialized web server, which will be enriched by the reverse proxy and will monitor and manage the loads. The **Production** environment represents the final product that will be used by the customers.

Focusing on the critical use case of our product, the test execution of the application workflow will be then as follows. The user will interact with the authentication page. He/she may register as a regular user or via the Git repositories. All user accounts are stored in the database, and also, all data that relates to the particular user are stored there as well. When the registration is done, the user may upload his/her test scripts in a specified format, configure the execution via the intuitive GUI, providing the hints of each parameter and expected format of arguments. Each user has a dedicated workspace; thus, no conflicts affect other user executions when it comes to collaboration. All executions are performed asynchronously in the specialized Worker container, so the user is not limited by the execution time waiting on the HTTP response. Finally, after the execution proceeds, the user may list its results, download the files produced by the Robot Framework or export its execution into the CSV. Users can categorize their test runs by the release tags or based on organizations they belong to.

5.1 Architecture

The following section describes the architecture of our application. Referring to the Figure 5.1 there are four main components the application can be categorized.

- **Web Server** - Takes responsibility for serving the content to the user. Serves as a proxy and finally load balancer.
- **Web Application** - Web application consists everything the user can interact with in scope of our application. This means the GUI, Back-End

including the database.

- **Robot Executor** - Pluggable component providing the service of test configuration, execution and output generation.
- **Database** - Persistent data storage for user related data.

Our architecture is designed for three different environment as described above - *Development*, *Test*, and *Production* environment. The reason for this categorization is to make the process of development, deployment and testing of our application (Section 4.4) easier and transparent. Components *Web Application* and *Robot Executor* on the physical level of the architecture mingle. This interleaving can be observed later in this chapter on the schemes dedicated to the particular environment.

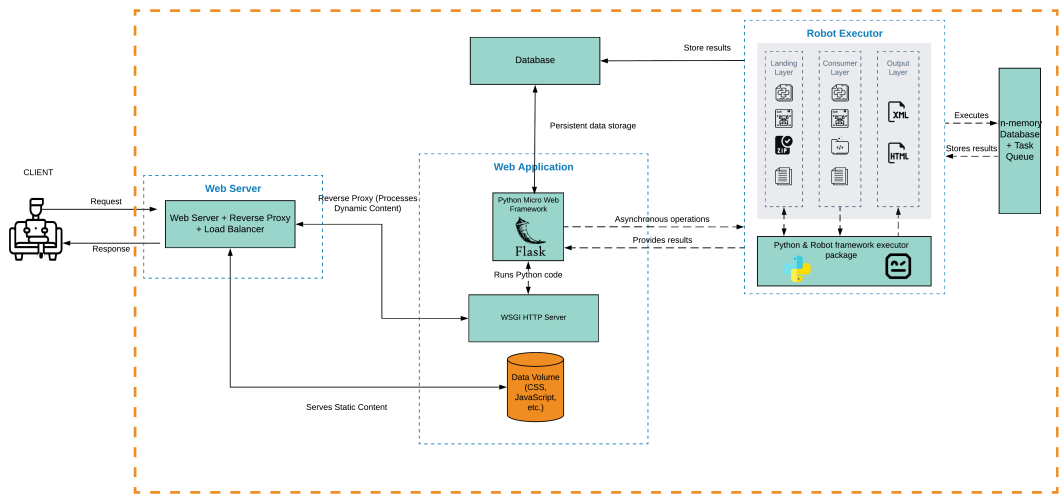


Figure 5.1: High-level architecture Diagram

The description will start on the development environment, where we define the fundamentals of our project, starting with the database model, through the Blueprint usage and thus the overall application structure that forms our *Web Application* component visualized in the Figure 5.1. In other sections, we will be extending our application by specialized services until the final stage - the Production environment.

The development environment is the environment where the application has a limited course of action. The developer must have pre-installed most of the used technologies on the machine he/she develops on. To run the application, a Flask build server is used, which possesses the debug mode and thus expedites our development significantly. Instead of the Postgres database, we use SQLite; the file-based database created automatically on the application's first build. Initially, this environment does not use any other web or HTTP server; thus, we do not take care of additional configuration; however, to secure support for authentication through a third-party provider, the environment variables holding the application OAuth id and secret must be set.

Developers are expected to use this environment when it comes to developing the front-end and facilities related to the view of the application. This is intended, due to the propagation of the rapid changes, whenever any source file is changed.

Further subsections describe the *Web Application* component from the high-level architecture described in Figure 5.1.

Database Model

To begin, let us define the database domain model of our application. This will not change in later stages due to the utilization of SQLAlchemy [35] for ORM purposes. According to the SQLAlchemy documentation [35] there are no differences between usage SQLite in the current environment and Postgres in the next stages. Figure 5.2 shows the domain classes defined in our solution.

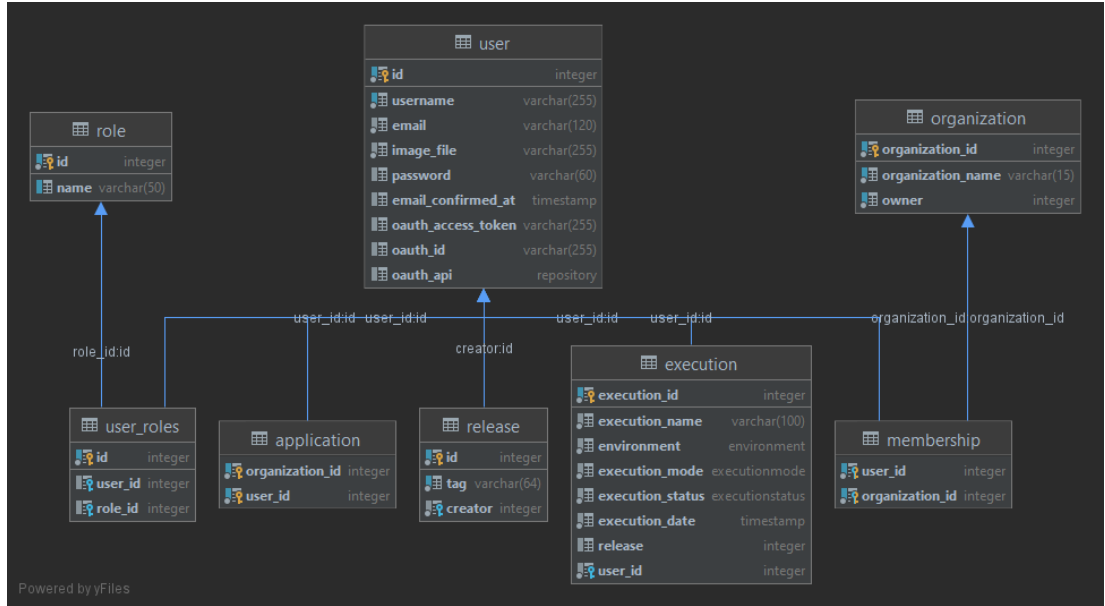


Figure 5.2: Database Model - Class Diagram

The **User** class represents the user of our application. The user has an ID serving as a Primary Key, username and email provided with a password during the registration process. The *image_file* stands for the storage of the user's profile image in our file system. Attributes marked with "oauth" prefix are used in case of the authentication using remote repositories to store the access tokens, user's ID, and finally, what kind of repository is used.

The **Execution** class stands for the single test execution performed by a particular user. Therefore, there is 1:n relationship between the **User** and the **Execution** class, denoting that each execution has just one executor. This is secured using the user's ID as a Foreign Key for the Execution. Using the advantages of the SQLAlchemy [35], we use the so-called "lazy" loading [36] as a kind of optimization that pre-builds the list of all executions associated with a certain user only when this field in the User class is accessed. Among others, this class possess a name attribute, **Environment** attribute, which denotes the execution environment, and due to the **Enum** superclass, it can be modified without changes into the relation. **Status** attribute marks the state of the Execution result and similarly as the last attribute has a base in the **Enum** class, therefore without loss of generality, it can be adjusted even after deployment. The last of the configurable classes is the **Execution Mode** that provides a pallet of

the execution modes representing the intention of a particular run; for instance, *Dry-Run* means that the specific run was the assuring one. The last attribute from this class is **Release** which stands for standalone class, providing an additional key for test execution categorization.

Omitting the Enum-based classes mentioned in the previous paragraph, we will move to the description of the Organization-related classes. Our application offers to users ability to create and manage an organization. The **Organization** consists of the organization's ID, name, and owner. The **Membership** relation denotes that the particular **User** instance is a member of the certain **Organization**. This relation is built on top of primary keys from both classes and altogether creates the primary key for this relationship.

Speaking about the user roles, we could follow the approach of the **Enum** based classes. However, this seems like a reasonable option, and the model was initially designed in this way; to secure the views for the user with certain rights, we needed to rework it from scratch. Using the Flask-User [14] library, we could easily integrate the role verification over each view that requires it under the condition that the database model for the user's role will follow the library specification (Figure 5.3). Implementing this scheme gives us a powerful and extensible tool for decorating views that have access restrictions.

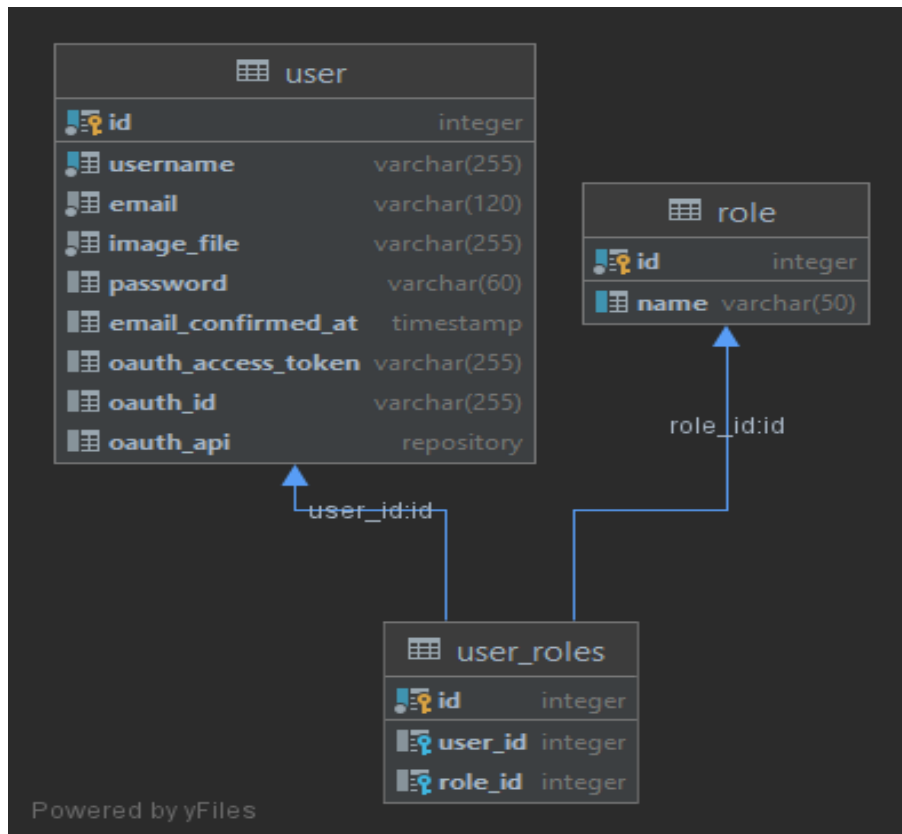


Figure 5.3: Role Based Authorization - Class Diagram

The last class from the Figure 5.2 is the **Application** class. This class represents the user's candidacy to a certain organization.

Application Structure

In order to allow future extension, and to support the logical categorization of subdomains and collaborative development, we want to have a modular structure to our project. This can be done using the *Flask Blueprints*. Figure 5.4 shows the individual packages within our application. Each package represents a particular blueprint except the *roco* which represents the application itself.

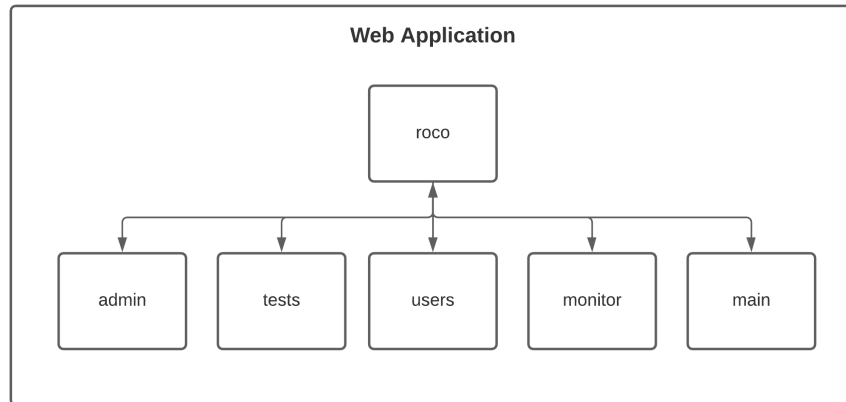


Figure 5.4: Web Application Component - Domain Diagram

Flask Blueprints is a concept of application modularization using common design patterns. The Blueprint objects generally act like a Flask Application object, but it is not the same. When we speak about the application object, the routing and configuration needs to be separated per application and handled on the level of the uWSGI. This approach is also followed in practice; however, it is not required to fulfil our purpose. Rather than that, we will use blueprints that objectively act the same as the previous approach but are being handled by Flask, more precisely the Flask Application. We will construct our application using the Factory pattern [11], and during the construction of our application object, we will register our blueprints. Usage of a Blueprint means that we need to split our source code into several packages to dedicate each package to a specific purpose. In our project, we have chosen the following division of packages based on the registered blueprints:

- **admin** - Stores all views and functions that are related to the *Administrator* role.
- **main** - Stores all views and functions that are accessible from the main navigation panel after the user is logged-in.
- **monitor** - Stores all views and functions that are associated with the *Worker* monitoring and management.
- **tests** - Stores all views and functions associated to the test configuration, executions and results exploration.
- **users** - Stores all views and functions that are dedicated to the user and organization management.

This package categorization also makes it easier to adapt the concepts of Aspect-Oriented-Programming (AOP). This has already been achieved by the modularization using the blueprints and creating a logging system for each individual package subject to the modular architecture.

In addition to the listed packages that are subject to blueprint division, we have the **model** package, which consists of definitions of database classes and the querying API that simplifies the operations over the database and strictly follows the DRY principle.

When we started the modular scheme of our project, let us dive into more detail within the particular packages starting with the root **roco** package encapsulating our application core.

Package - roco

The **roco** package is the Flask Application package, that contains the application configuration and starts all sub-components that are in its subtree.

We use the Flask-CLI [12] utility that enables us to define the commands that can be used when building or running the application.

Package - admin

Extending the short description from the listing above, the **admin** package take care of serving content that the only user with the **Administrator** role possesses. Referring back to the Figure 5.3 all views provided within this package protected so only the user with the **Administrator** rights can access them.

Package - main

This package represents the views and endpoints accessible from the main navigation panel and, at the same time, are accessible by each of the role specified within our application.

Package - model

Oppose to the blueprint packages, this package does not follow the standard structure defined above. This package provides the database model definition build on the Object-Relational-Mapping framework - SQLAlchemy. Another function that this package offers is the API for communicating with the database from the application with respect to the application context [13]. This is a convenient approach since when the application is being built using the Factory pattern, and we would try to access the database object before the application context is fully built, we would fail. By encapsulating the database querying into the specialized object, we can avoid such issues in the build process or the running environment. The last feature this package provides is the configuration for the third-party application authorization.

Package - monitor

This package aims to provide the monitoring facility over the **Worker** and the **Redis**. The user then has the capability to monitor the running tasks and cancel

them or, in a case of a job failure to re-rerun the failed job. This package is built on top of the rq-dashobord [33] project that provides the monitor display over the registered RQ Worker under the specialized blueprint. Because this library defines its user interface, which was not pluggable into our solution, we have extracted necessary back-end methods and integrated them into our platform.

Package - tests

To categorize all views related to the process of the configuration, execution, and result observation, the tests package was introduced. It gathers all entities such as views, endpoints, forms, and functions necessary to prepare test configuration that is then redirected to the **Robot Executor** (Figure 5.1) package for processing.

Package - users

This package spans all views, forms, classes and functions dedicated to the users; this means the authentication and authorization process as well as the user's settings. It also integrate the pluggable solution for the third-party application authentication and authorization.

Package - robot executor

Referring to the high-level architecture (Figure 5.1) last component that needs to be described is the *Robot Executor*. Figure 5.5 visualizes the inner structure of the component. Extending the brief definition at the beginning of this section, we can add how the component works. Starting with something we call *Landing Layer* in our architecture, which is nothing more than the directory assigned to the certain user, which accepts the raw files uploaded to the server, which are validated against their file extension at this very first layer. The process of the validation, extraction and sanitization is secured by the *Landing Processor* module.

Right after the files that are in the *Landing Layer* are being selected and extracted to the *Consumer Layer*. Files from this layer that match the Robot Framework [32] executable definition are listed to the user in the *Configuration tab*. The module responsible for the tasks performed over the *Consumer Layer* is depicted in the Figure 5.5 as a *Consumer Processor*.

The execution process over this layer is responsibility of the *Executor* module (Figure 5.5). After the execution it produces the regular robot outcomes as well as the state of the execution and propagates them to the user. Generated outputs are pushed into the *Output layer* dedicated to the particular user.

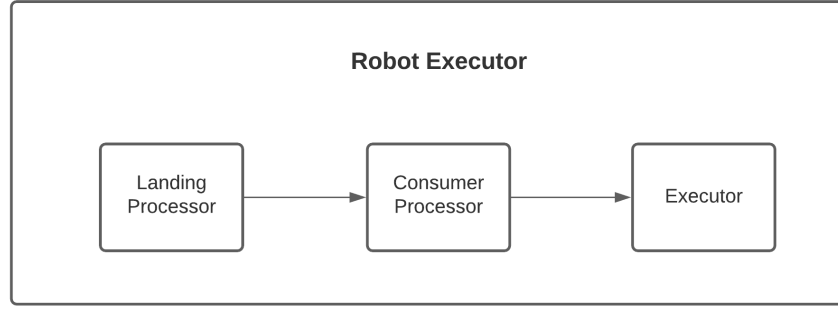


Figure 5.5: Robot Executor Component - Domain Diagram

Moving from the **Development** environment where the module skeleton and domain model was introduced, let us focus on bringing the concept of containerization. Oppose to the **Development** environment we will use PostgreSQL database placed into the specialized Container (Figure 5.6 - *Database (PostgreSQL) Container*). This separation is due to security, configuration independence, and avoiding having one robust container that would have to serve multiple purposes. The database is constructed whenever the container is built; this means that it is not intended for permanent data storage.

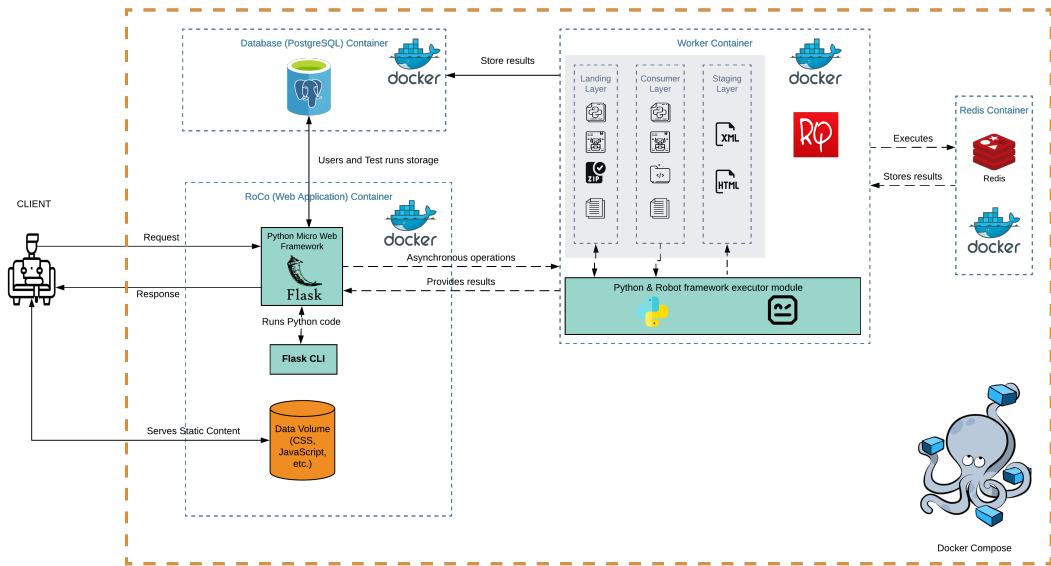


Figure 5.6: Test Environment - Architecture Scheme

The Figure 5.6 defines also the *RoCo (Web Application) Container* and the *Worker Container*. These two containers shares so-called volumes [7], which are the shared storage among the containers, and both containers can be accessed without knowing that another container has the right to do so. This is exploited during the file uploads, so when test scripts are pushed to the server and land in the *Landing Layer* (visualized in the Figure 5.6). This is when the data comes to the *RoCo* container; however, they are processed by the *Worker* container by the process explained in the previous section. Once the execution workflow is done, and the robot outputs are produced into the *Staging Layer*, they are accessible

from the *RoCo* container and served to the user. The *Redis* container servers as an in-memory database providing the fast storage and data structures to perform the test execution asynchronously and independently on the running web application.

Configuration of the individual container is kept to default settings in the current environment. Only self-configured container is the *RoCo* container built on top of the Python [8] container, which is configured using the regular Dockerfile [6], more precisely *Dockerfile.test*. At the beginning of the process, updates and git is installed. Afterwards, all requirements defined in the *requirements.txt* are installed, the project directory is copied into the container, and finally, the entry point script is run.

All containers described in the scheme above are orchestrated via Docker-Compose [9], which takes care of the precedence of builds as well as for dependencies between the containers, eventually the external communication in terms of ports. For the configuration is required, the *.env.test* that stores the environment variables that are then loaded by Flask into the application's configuration.

The final problem we need to solve is the web server and HTTP server. We do not integrate these into the **Test** environment to reduce the complexity; furthermore, there are no benefits in integrating those into this environment since there are no application-dependent configuration or core functionality required not to be handled by the Flask build server.

We are moving to the last stage of our development, to the **Production** environment intended for actual use; we can build on the fundamentals already introduced in the previous two stages. Figure 5.7 shows that we extended our platform by an additional *NGINX* container that serves as a web server, reverse proxy and load balancer. We use the defined docker image for building the *NGINX* container, but we do modify its configuration to fulfil our needs.

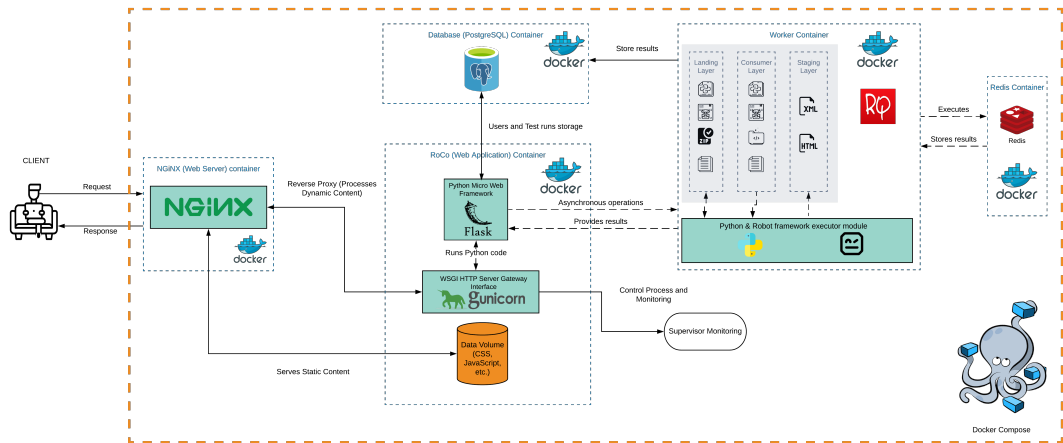


Figure 5.7: Production Environment - Architecture Scheme

From this moment, the user will interact with the proxy listening on the port specified in the Docker-Compose configuration and redirect the requests to the port of the *RoCo* container.

Another change we have made is the HTTP server. In the current environment, the Gunicorn [17] will be used rather than the Flask build server, which is not created for the **Production** environments. The Gunicorn is the superstructure over

the uWSGI accessible using Python, which again brings certain advantages, mainly reducing the configuration complexity by using Python instead and supporting process monitoring.

We are returning to the database, which was built on every application's build. The data are kept between the individual builds; this is where the advantage of the standalone Docker container.

In Section 4.3 we were speaking about the security required by our application. This is a complex topic regarding the fact that we use containers over the existing system. Since our solution aims to be used internally rather than in public service, we did not encapsulate or sandbox the execution of the test scripts. If we implemented public service, there would be a potential security risk due to the logic Docker works on in accessing the host machine resources or the resources of other containers. An attacker might get outside the application context. This vulnerability is caused due to the usage of the *volumes* that allows file sharing among the containers. In this particular environment, we have created a non-root user with minimal rights to the outer scope; however, this solution is sufficient in terms of our application's goal; it must be innovated in the further extensions.

Summarizing the paragraphs above, we have developed a fully working solution for the configuration, execution and reporting of the automation tests written in the Robot Framework. This solution is scalable and can be deployed on any environment supporting Docker and the tool for container orchestration, such as Kubernetes, which is considered an industry standard these days.

5.2 Plugins

The modular architecture of our application gives us two leading advantages. First, we can easily extend the repositories or the services for OAuth authentication by adding a configuration object into the *config.py* module within the **model** package. After that, the endpoints for the authorization and authentication with the callback URL needs to be implemented. The solution is designed in the way that the providing application can be exchanged with minimal source code modification.

The second advantage mentioned is the *robot_executor* package providing the classes that take care of processing the robot test scripts. Omitting that this project intends to support Robot Framework, this package can also be replaced or extended by other technologies such as Cucumber, which works on a similar principle. Therefore we can support two automation testing framework at once or separately. Extending the application by another view is dedicated to the configuration of the concurrent technology.

Both of these abilities give us flexibility for future development when the intention is to support various applications accepting the test execution results or support another technology for automated testing.

6. Case Study and Evaluation

This chapter is devoted to the explanation of the usage of our application based on the user's role that our system offers. Section 6.1 will simulate how the application can be managed on basic users scenarios and what options it offers.

6.1 Example of usage

As was stated at the beginning of Chapter 3, our application in the current version offers three types of user roles. Those are Tester, Reviewer and Admin. Subsections below will describe major use-cases with respect to the role that the user has.

6.1.1 Reviewer Role

The intention for creating the **Reviewer** role was to introduce authority that could review the test results. It is a common practice, especially in the companies subject to audit verification, that there are Subject Matter Experts to each of the development environments that either approves or declines the test execution.

In our system, we do not provide the reviews options as, for instance, are provided in the ALM VERA [1], where the Subject-Matter-Expert (SME) can review the test execution and add his/her insights.

Example - Basic Utilization

The following example demonstrates the primary usage of our application. We will proceed with the usual registration process; then, we will describe the tabs related to the primary user and organization management and options specific for the **Reviewer** role.

Starting with the *Login Page* shown in the Figure 6.1 we can navigate to the *Registration Page* using the *Sign-up* link highlighted by number 1 in the figure. The *Login Page* also offers authentication using the 3rd party application presented in the Section 6.1.2. The last option that user in this view has is to reset his/her password if forgotten.

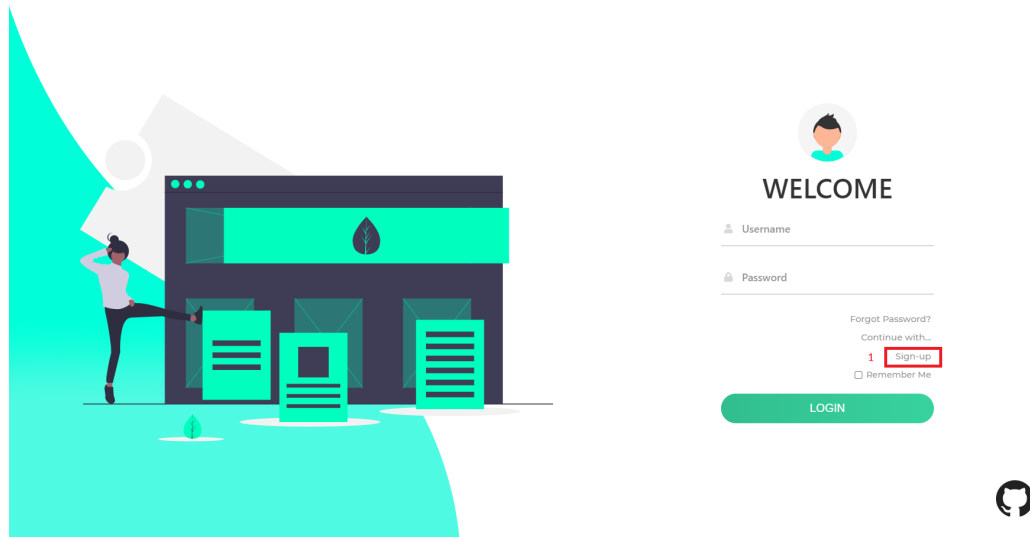


Figure 6.1: Login Page

The *Registration page* in the Figure 6.2 asks for basic user information such as username, email, password and finally the role. For the purposes of this demonstration, we will choose the **Reviewer** role and submit the form.

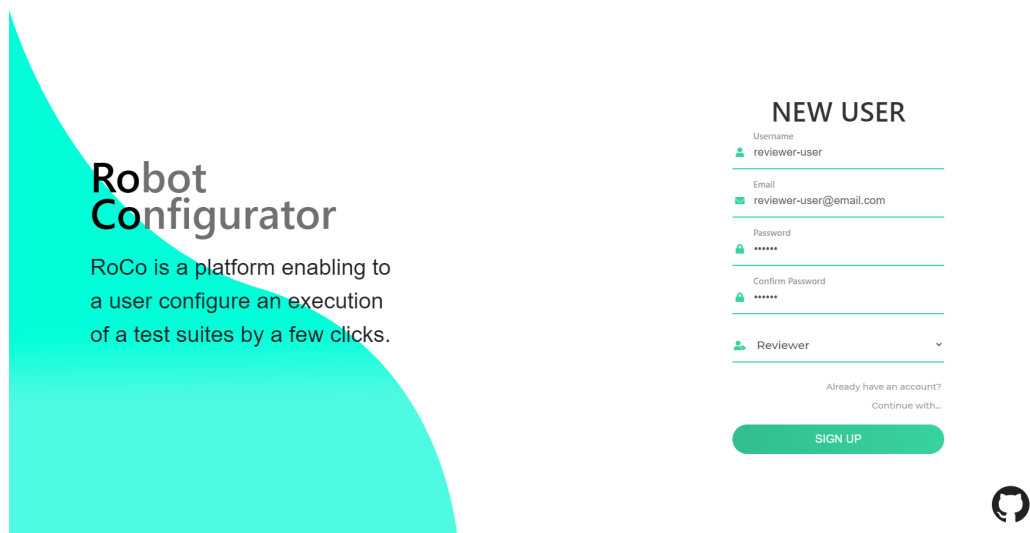


Figure 6.2: Registration Page

Immediately after the registration is done, the *Welcome Page* is displayed to the user (Figure 6.3), capturing the current version changelog as well as the important messages that should be shown to users when they log into the application.

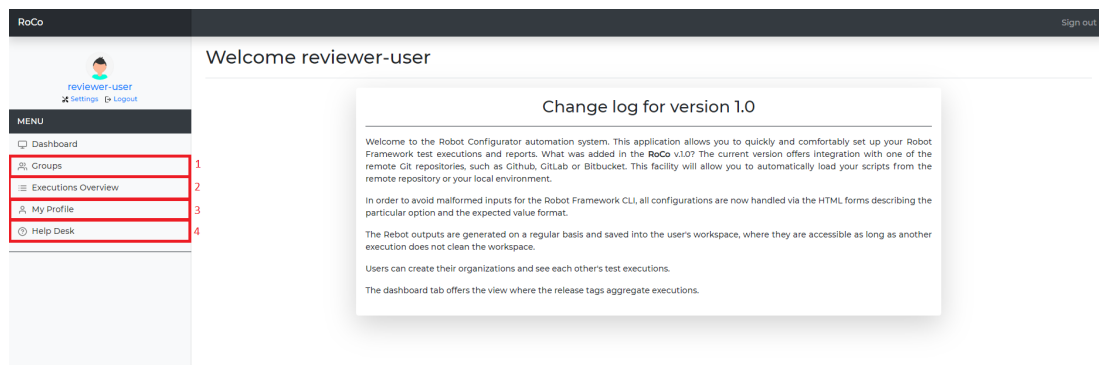


Figure 6.3: Welcome Page - Reviewer

The user now has several options where to go. In this example we will focus on tabs *Groups*, *Execution Overview*, *My Profile*, and *Help Desk*, that are highlighted in the same order in the Figure 6.3.

We are starting with the *Groups* tab, which allows the user to create and manage an Organization or Join organizations. Let us then create the organization called "Organization". Figure 6.4 shows the form that requires only the organization name. Right after the form submit we can see that we were redirected to the page displaying all user's memberships. (Figure 6.5).

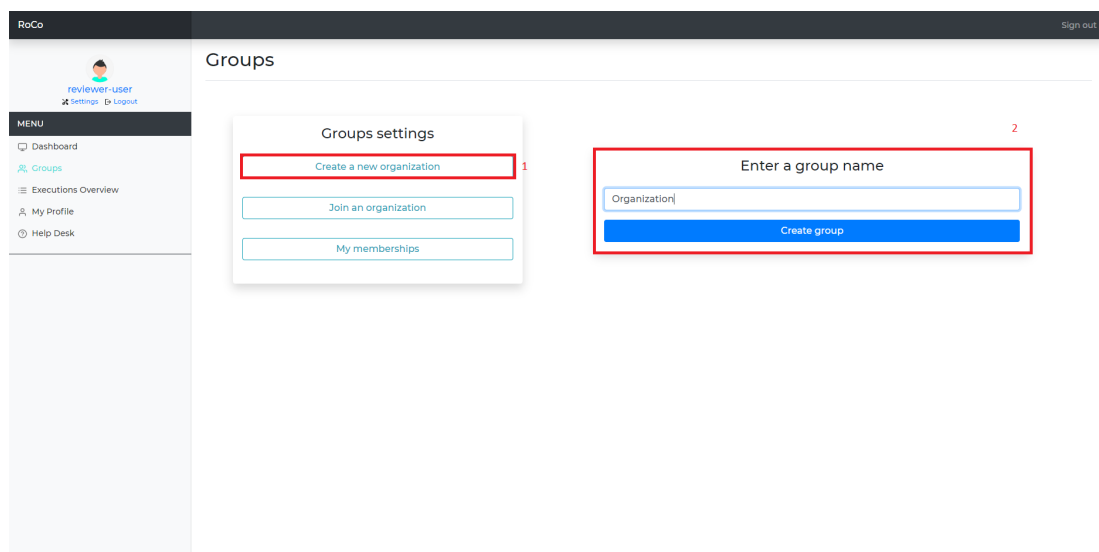


Figure 6.4: Create a new organization page

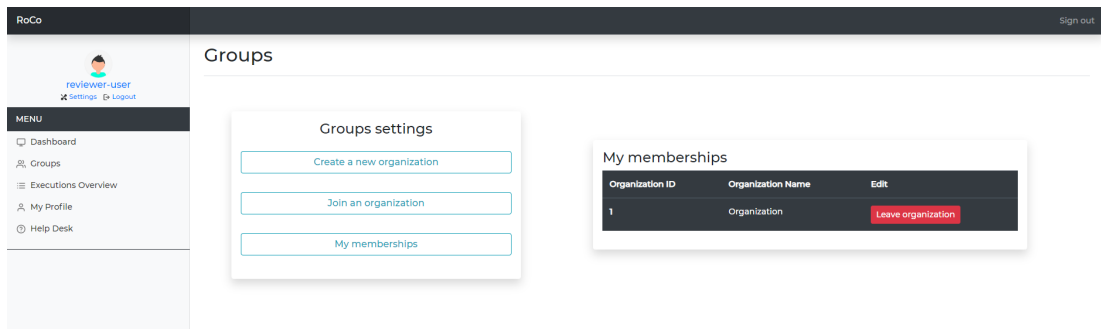


Figure 6.5: Manage the memberships page

From now and forward, users can request to join our organization. The join request will be displayed immediately in the Organization management panel (Figure 6.6), and the owner of the organization will also receive the email notification. The organization's owner can add users by pressing the *Add* button as shown in the Figure 6.6 by Highlight 1.

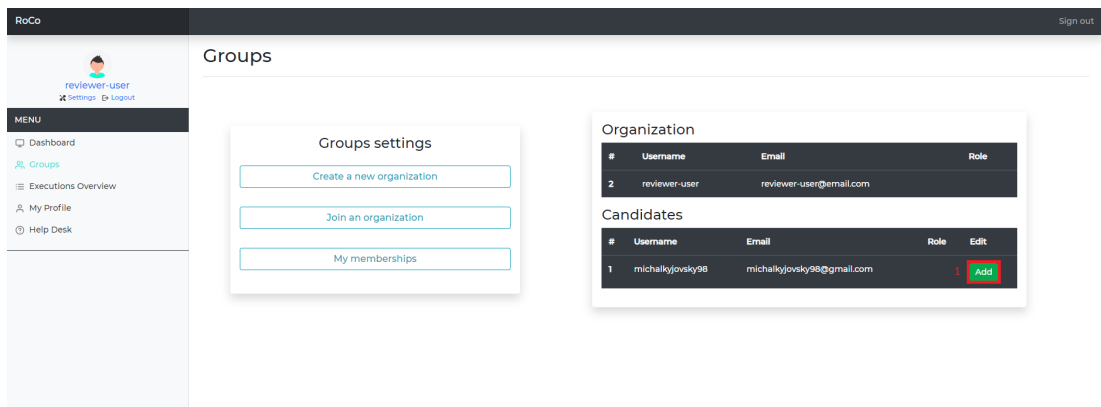


Figure 6.6: Manage the organization page - A

Memberships and the applications can be as shown managed from the *Create a new organization* pane, where all members and applicants for the user's organization are listed. Figure 6.7 shows that our organization has two members, including its owner.

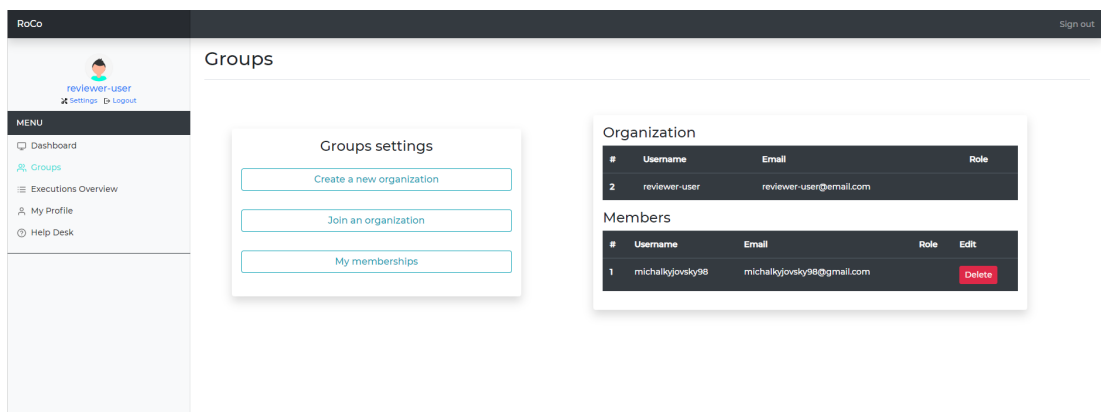
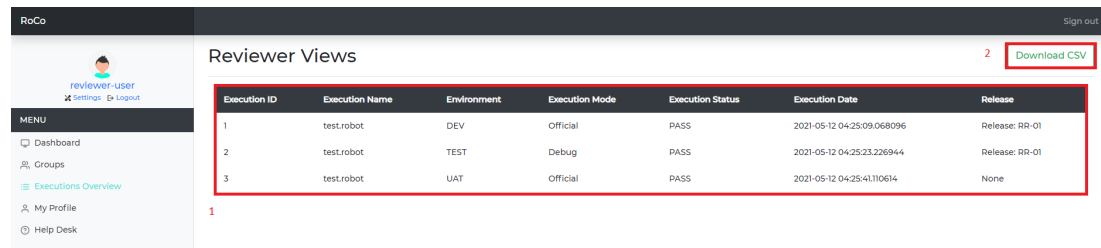


Figure 6.7: Manage the organization page - B

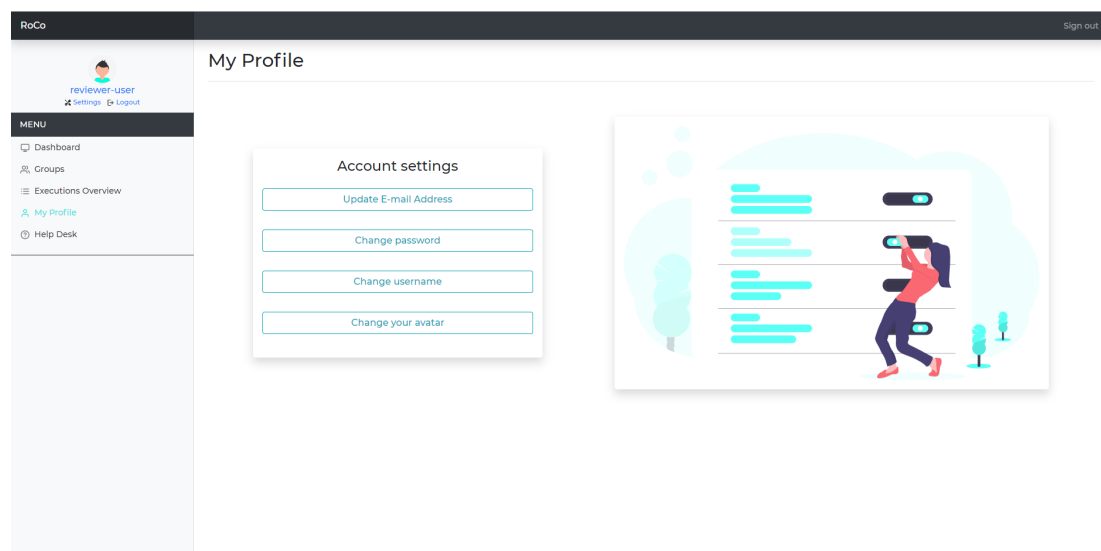
Now that we have a tester in our organization let us move into the *Executions Overview* tab to check performed executions. Figure 6.8 shows all executions related to the organizations that our user has established or is a member of and all **Release tags** he/she has authored. The user can also download the table in the figure by clicking the *Download CSV* button.



Execution ID	Execution Name	Environment	Execution Mode	Execution Status	Execution Date	Release
1	test.robot	DEV	Official	PASS	2021-05-12 04:25:09.068096	Release: RR-01
2	test.robot	TEST	Debug	PASS	2021-05-12 04:25:23.226944	Release: RR-01
3	test.robot	UAT	Official	PASS	2021-05-12 04:25:41.110614	None

Figure 6.8: Executions overview page

Moving from the crux of the Reviewer role, let us move to the *My Profile* tab to observe what user settings are possible in our application. Figure 6.9 shows that the user can change his/her password, username, e-mail address or avatar. These user-related settings facilities are allowed only to users who registered into our application the traditional way, hence the *Sign-up* form exposed in the Figure 6.2. Users registered via the OAuth service are limited only to the avatar change since other users data are intended to be synchronized with the remote repository.



Account settings

Update E-mail Address

Change password

Change username

Change your avatar

Figure 6.9: My Profile page

The last page we did not cover yet in this tutorial is the *Help Desk* page. The *Help Desk* tab is used for requests for package installation or a regular thread for communication with administrators of our application. The mail that is being sent has a predefined structure, and it requires a header, problem classification and the body. The Figure 6.10 shows the e-mail form.

Figure 6.10: Help Desk page

6.1.2 Tester Role

The **Tester** role is the default role used in our application. The user with this role can configure his/her test scripts, execute them and observe the result. Testers are also eligible to create organizations or join others user organizations. Each user has assigned his workspace that he/she can manage.

Example - Test Configuration and Execution

Let us have a simple test script presented in Figure 6.11 implemented in the Robot Framework to demonstrate the use-case scenario of test configuration and execution. The script below is a trivial test implementation that uses only built-in libraries and posses two test cases.

```

1  *** Settings ***
2  Library      String
3
4  *** Test Cases ***
5  Stable test
6      Should Be True      ${True}
7
8  Unstable test
9      ${bool} =  random_boolean
10     Should Be True      ${bool}
11
12 *** Keywords ***
13 random_boolean
14     ${nb_string} =  generate random string 1 [NUMBERS]
15     ${nb_int} =  convert to integer  ${nb_string}
16     Run keyword and return  evaluate  (${nb_int} % 2) == 0

```

Figure 6.11: Example of simple Robot Framework test script

The *Stable test* will always end with status **PASS**. The *Unstable test* on the other hand, acts unpredictably. The script in Figure 6.11 will be stored in our remote GitLab repository; therefore, we will be in the next steps using authentication via a 3rd party application and cloning the directory rather than uploading the locally stored files.

We have our inputs ready, so move on to the application. Figure 6.12 shows the login page, more specifically authorization into the application using the remote Git repository. This way of authorization implicitly expects that the user will be registered with the **Tester** role; since we do not expect that **Reviewer** would use the Git repositories, this can be, however, changed later by an **Admin**.

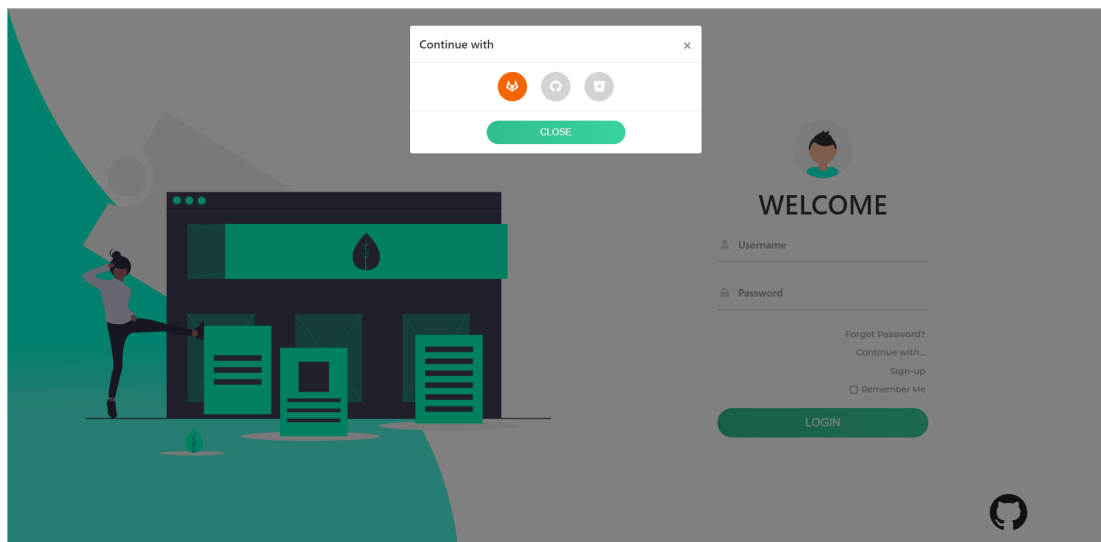


Figure 6.12: Login Page - OAuth

In the following figure, we have an application welcome page, already introduced in the Subsection 6.1.1. Let us navigate to the *Dashboard* tab to create the Release tag that we will use for the execution annotation.

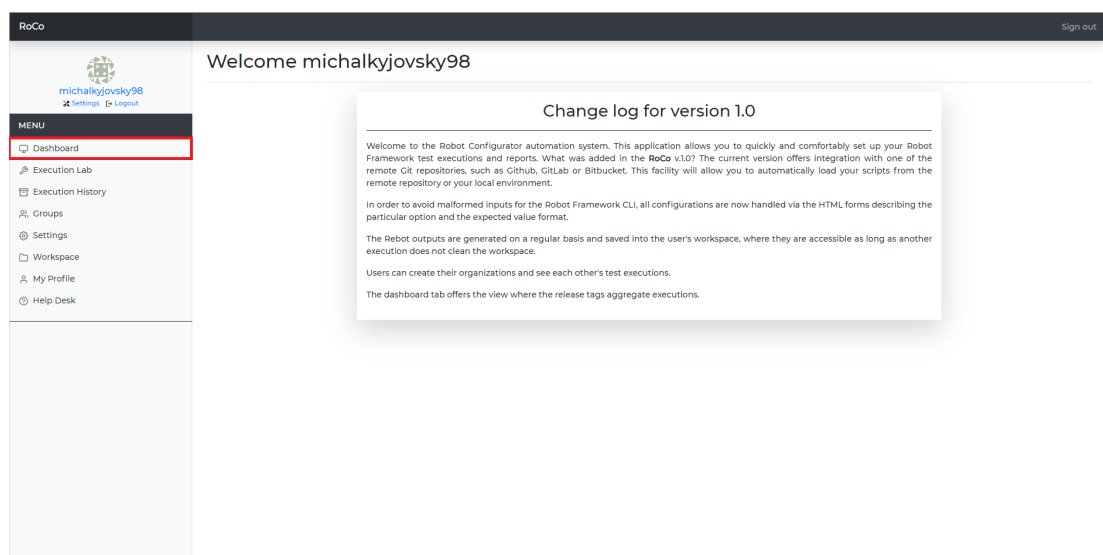


Figure 6.13: Welcome Page

The *Dashboard* view offers to create so-called "*Release Tags*". It is an optional feature that enables to the user create a *Tag* that serves as an aggregation key for the individual execution. In incremental software development, the are software versions called releases when they are being developed. Our application took that idea in order to simplify an execution categorization to the user. Figure 6.14 shows how to create a new release tag with the highlighted steps. Each release has its information panel denoting the success rate of the execution, hence the ratio between the failed and passed tests. The table next to the information panel lists executions annotated by the particular release tag.

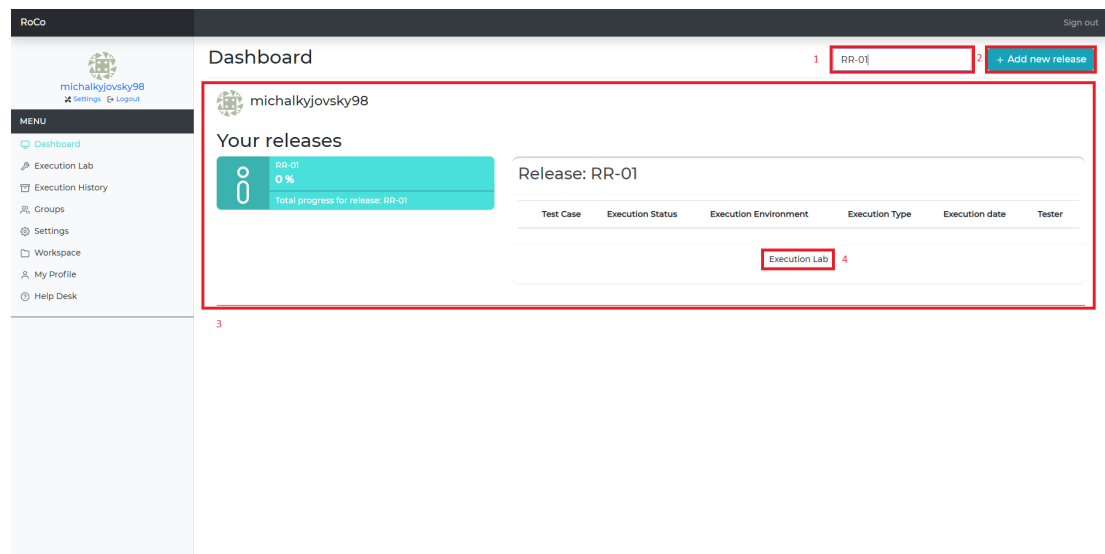


Figure 6.14: Dashboard

The user now can navigate into the *Execution Lab* view. There can be observed two ways how to get there. The first option is to use the main navigation panel adjacent to the left side of the viewport. The second option is a shortcut marked in the Figure 6.14 by number 4.

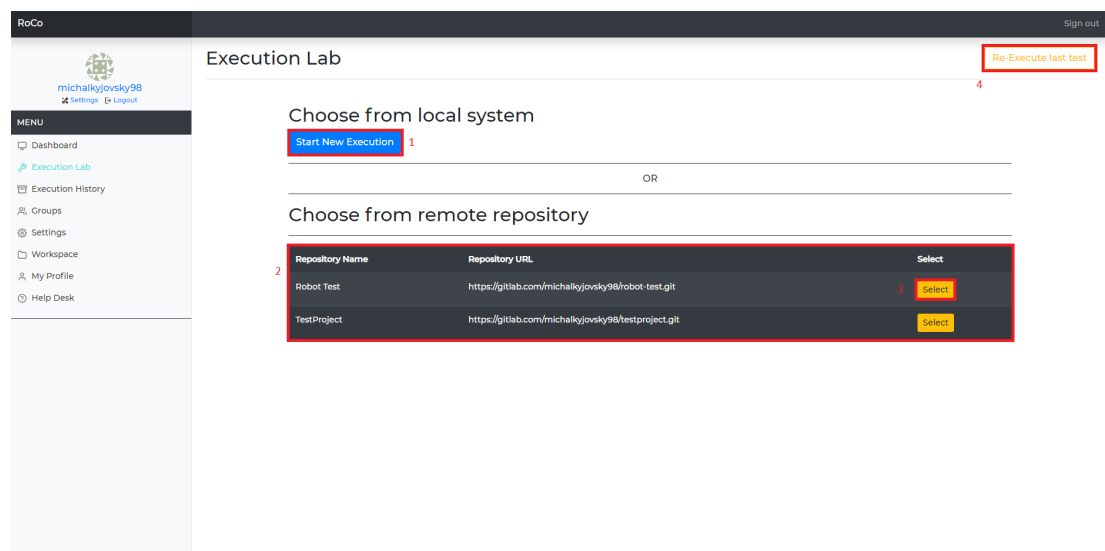


Figure 6.15: Execution Lab

In the Dashboard view displayed in the Figure 6.15 we can observe that the user can upload test scripts using the local file-system by using the *drag and drop* option highlighted by number 1 in the figure or by regular file explorer denoted by Highlight 2. The prompt requiring the files to upload will be triggered by uploading from a local file system, as shown in Figure 6.16.

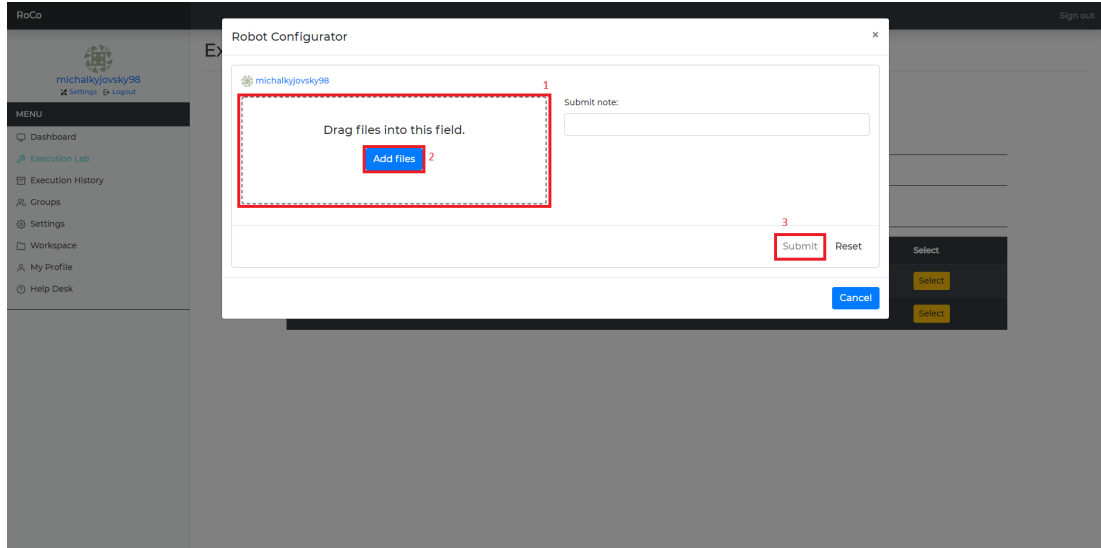


Figure 6.16: Upload from local file system form - A

When files are uploaded and staged to be sent to the server, a Front-End validation of allowed file extension is performed, this can be seen in the Figure 6.17. Files that fulfil configured criteria are marked by green success mark. The red cross mark signs that file cannot be uploaded and the "Submit" button is disabled. The user can delete staged files by pressing the bin icon as exposed in Figure 6.17 on highlight 2.

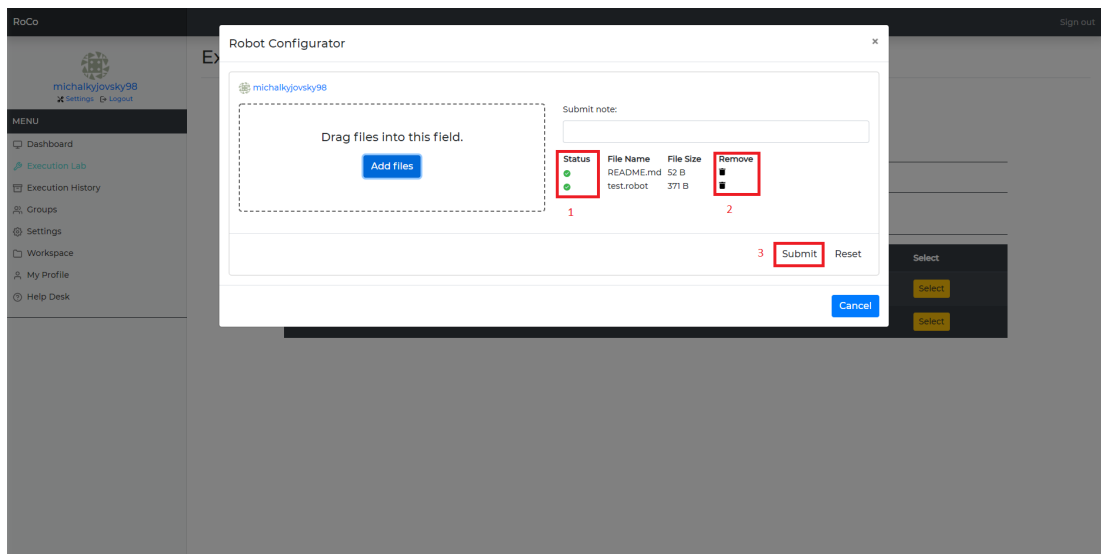


Figure 6.17: Upload from local file system form - B

Another option that our solution offers is to use already uploaded files in the user's workspace. This is possible by pressing the button *Re-execute last test* as

shown by highlight 4 in the Figure 6.15. Using this feature, already prepared files will be used, so the user can easily choose the script he/she willing to use.

The last option is using the user's remote repository. In this example, we use the GitLab repository, where our example project is stored. The only operation the user needs to perform is to select the appropriate repository and press the "Select" button as shown in Figure 6.15 on highlight 4.

We are now in the "Configuration Panel" page. This page allows to the user easily configure the Robot Framework execution as well as he/she would use the Robot Framework CLI. The Figure 6.18 demonstrates the possible configuration of the execution. We selected only one file that could be used for the execution, as shown in Highlight 1. The release tag that we created at the beginning of this tutorial is now available in the select box element highlighted by number 2. User can select from a pre-configured environment and kind of execution as shown by highlights 3 and 4. The configuration of the Robot Framework option is represented by the form boxes below. We selected the option for *Generic automation mode*, and we can select only the particular Test Case, for an instance the "Stable test" from the Figure 6.11. When all necessary is setup, we can click on the *Configure* button denoted as the Highlight 7 in the figure.

Figure 6.18: Configuration panel - editable

By clicking on the *Configure* button, all fields will be marked as read-only. This facility was introduced to provide the user with an option for self-check or eventual screenshot capture if required a verification point. Figure 6.19 demonstrates the configuration prepared for run. By selecting the *Run* button, the execution will proceed.

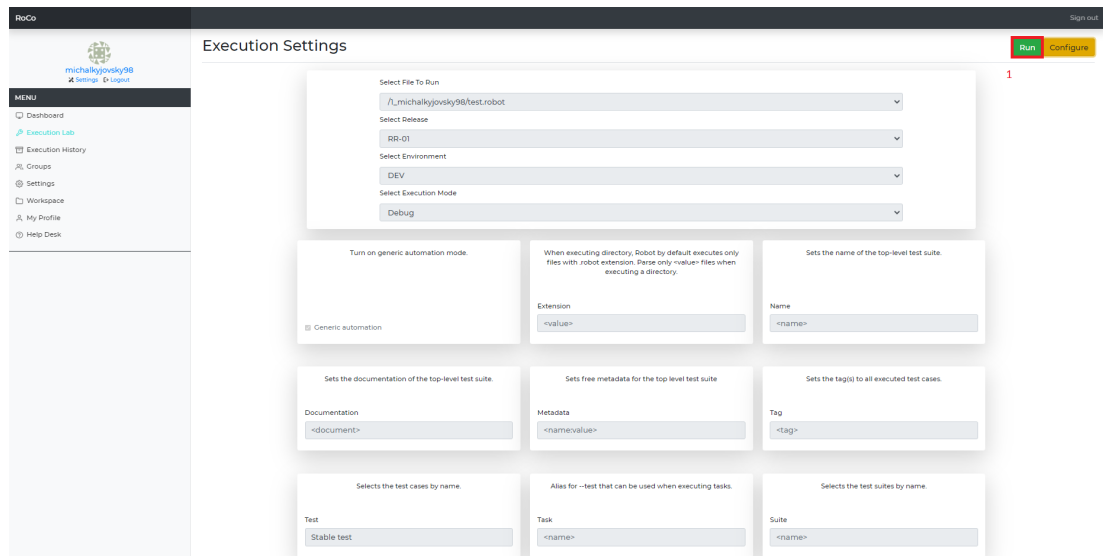


Figure 6.19: Configuration panel - read-only

Now, when execution is triggered, the user will be moved into the *Execution results* view, where he/she can observe the results of the execution and the output files generated by the Robot Framework, including the archive wrapping all generated outputs. Links for downloading the generated outputs can be observed in Highlight 1 in Figure 6.20. The user can also download these files directly from his/her output workspace by clicking the *Workspace* tab in the main navigation panel, as shown by Highlight 3. In the end, the user can go to the *Execution History* tab denoted by Highlight 2 and observe all his/her past executions.

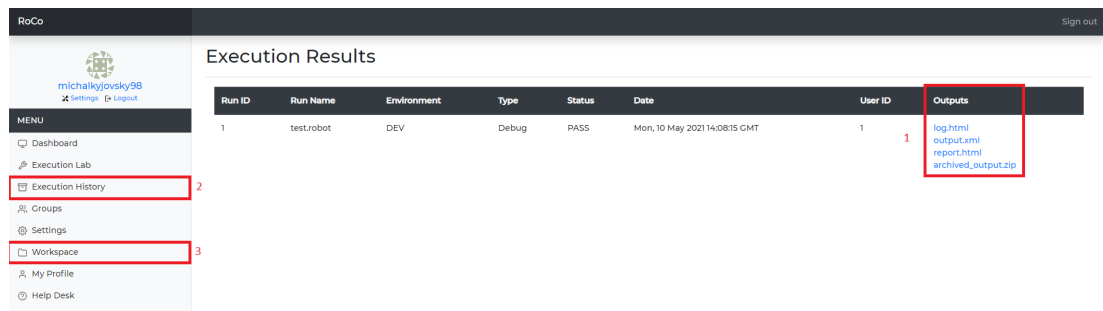


Figure 6.20: Execution Results view

Figure 6.21 we can see past executions dedicated to the current user. Furthermore, as Highlighted as number 1, the user can export our execution history into the CSV format.

Execution ID	Execution Name	Environment	Execution Mode	Execution Status	Execution Date	Release
1	test.robot	DEV	Debug	FAIL	2021-05-11 05:53:06.065261	Release: RR-01
2	test.robot	DEV	Debug	PASS	2021-05-11 05:56:15.757064	Release: RR-01

Figure 6.21: Execution History view

The *Workspace* page represented by the Figure 6.22 offers to download the content of the output directory dedicated to the current user, which holds the generated output files from the last test execution.

#	File Name	Download
1	archived_output.zip	Download
2	log.html	Download
3	output.xml	Download
4	report.html	Download

Figure 6.22: Workspace view

6.1.3 Administrator Role

Combining previous roles, we can finally introduce the **Administrator** role. The user with administrators rights posses all abilities of the **Tester** and the **Reviewer** enriched by the *Administrator Panel* that enables to the user manage users of the application as well as organizations. Moreover, the Administrator can install Python Packages from PyPI directly from the application and send mass emails to the application's users.

The Administrator is created during the installation process. Only the Administrator can promote the other user to the **Administrator** role.

The very first ability that **Administrator** has is mass mailing to all users. This is useful, especially when users need to be notified, for and instance, about system patches, upgrades or application unavailability. Figure 6.23 shows the email form for the mass mailing.

Figure 6.23: Notification Email page

Another option that the **Administrator** possesses are to manage individual users. He/she can see the information about the individual user, change their role or delete them from the application if necessary.

#	User ID	Username	E-Mail	Memberships	Role	Edit
1	1	michalkyovsky98	michalkyovsky98@gmail.com	Organization	Tester	Change Delete
2	2	reviewer-user	reviewer-user@email.com	Organization	Reviewer	Change Delete
3	3	admin	michalkyovsky98@gmail.com	N/A	Admin Account	Admin Account

Showing page 1 of 1

Figure 6.24: Users Management page

As well as the **Administrator** can take care of users management, he/she can manage the organization in the same way.

#	Group ID	Group Name	Members	Edit
1	1	Organization	michalkyovsky98 reviewer-user	Delete

Showing page 1 of 1

Figure 6.25: Groups Management page

The last extension that this role offers is to install Python packages if required. Packages are fetched only for certificated Python Package Index [39], so the user can be assured that the package is validly prepared and can be installed into our environment.

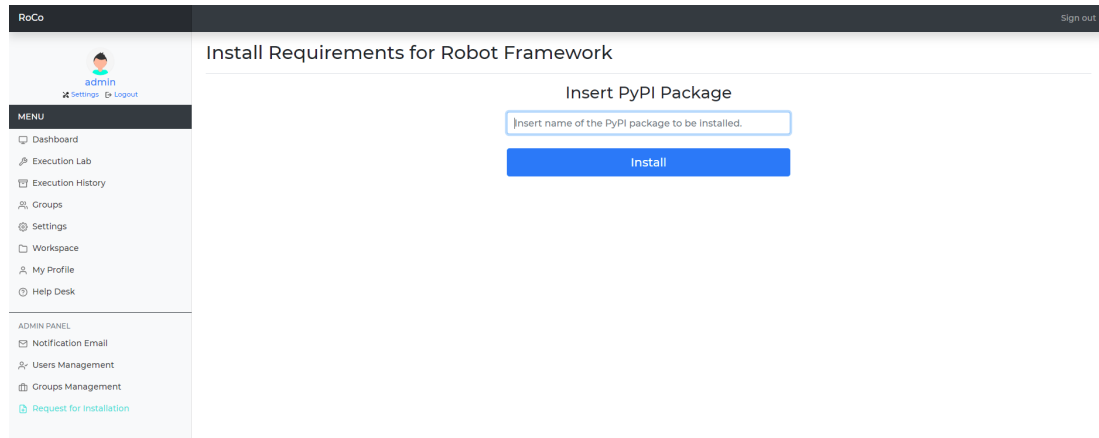


Figure 6.26: Request for Installation page

6.2 Related Work

This section is dedicated to analysing the related work, hence applications and systems that serve a similar or the same purpose. In the following paragraph, we will break our major targeted use-cases and the alternate applications that serve the particular purpose.

Speaking about the more straightforward configuration of the Robot Framework execution, we will not be successful in a search for such a tool. However, we may find the plugins into the IDEs used among developers and testers that make test configuration and execution less laborious. For instance, we can assume the Plugins into the IntelliJ products such as Robot Plugin [18] or specialized IDEs such as RIDE [29] which is the original editor for Robot Framework or the RED [27], an editor based on the Eclipse IDE. There are many other environments and plugins that help the user with Test Script development and, in some instances, also with the configuration and execution. For instance, one such custom solution could be Robocop [30], the Pycharm plugin developed by the author of this thesis providing the syntax highlighting and the configuration and execution integrated directly into the IDE. In any case, they do not provide the convenience of the easy setup and execution for the non-experienced user in terms of specialized IDE.

The idea of taking the environment where a test is being run was inspired by the usage of build servers such as Jenkins [19] or CircleCI [5]. Our intention has a practical surrounding. In larger companies, where the employees receive their work-station usually happens that the employee cannot install all the tools required by his current job. He/she then needs to approach the service desk or the other authority to get the dependencies installed. This is the real crux, especially when the subject matter expert who should give his/her approval upon his decision does not work in the place the employees does, furthermore is located in a different country. If the test execution is moved to the machine configured independently, we may reduce this bureaucracy process. Oppose to the tools

```

1 $ robot -V Config/Presto/presto_test.py \
2     -v environment:TEST \
3     -v db_type:ATHENA \
4     --outputdir results_SQL_queries \
5     --listener "Convertor; -F/robot/;-c Config/ALM/config.
6     properties;-R validation;--cycle_name Release_XY;--domain XYZ" \
        "/robot/tests/test_SQL_queries.robot"

```

Figure 6.27: A complex command to execute a Robot Framework script

named at the beginning of this paragraph, software brought this idea a step further, so it is cloud-based and fully dedicated to the Robot Framework. We are talking now about Robocorp [31] solution. This solution makes the crux of the mentioned problem among the other benefits behind it. However, this solution is commercial and for more advanced users, which is not our goal.

The last targeted use-case that should be mentioned is taking the CLI configuration into the web forms. As much as the Robot Framework is an appreciated tool, the CLI configuration tends to be not user-friendly. Demonstrating on the example (Figure 6.27), where the configuration command can have multiple lines combining the separators and quotation:

Our application brings the *Configuration Panel* that provides user the easy configuration with the CLI option description and format of the parameters. This facility is completely unique among the other tools that help with the *Robot Framework* configuration.

6.3 Unsupported Features

The following section summarizes all features that have been taken out of the current scope. Each of the listed feature contains justification for the decision to omit the particular feature.

- Testing of Front-End using Selenium or similar technologies.
 - We omitted this facility since the Front-End testing must be performed on the user's side, which collides with the primary goal of this project - to move the complexity of machine setup from the user.
- External build server for running the tasks.
 - Instead of having the our-own specialized server dedicated to the heavy-task performance, we would prefer the solution, where this is could be at least partially moved to the build servers.
- Additional synchronization of the regular user with the repository.
 - What could be definitely improved is to additionally synchronize with the Git repository. This was not implemented due to the low importance.

- Support for an output archiving such as Jira, Confluence, TestLink, and others.
 - This feature would accomplish the testing process to its perfect form providing testers ability to automatically transform their execution report in the reporting tool they use.
- Selection of the particular branch for the test scripts upload.
 - In testing it is usual, that a probationary test execution are not performed against the mainstream branches. Therefore it would be improving to select the git branch that tester would like to use.
- Synchronization of an user accounts with its third party account in the way of user info settings.
 - When users change their personal data in the remote repository, we would like to implement the web-hooks to propagate such information to our platform.

7. Conclusion

In this project, we have developed a scalable web-based configuration application for the Robot Framework. The application can accept user inputs in various formats, validate and post-process the input, configure the Robot Framework test execution, and execute the test cases. Despite that the simplification of the testing process was the original intention of this project, our application offers more. Based on the testing regular testing processes, users can group into the organizations or projects; furthermore, users may tag their test runs according to a release or any other desired key, so the reporting is easier for them.

To develop such an application, a robust and reliable architecture needed to be introduced. We needed to analyze the Robot Framework as a standalone program and an API that can be used within the Python programming language. If we want to talk about automation testing or software testing in general, we had to analyze the software testing is. The typical signs could be used in agile environments, so the application will not be based on a single approach. To support the vast span of the hosting technologies nowadays used, we had to develop a modular and scalable containerized architecture that is not limited by its hosting environment.

Software testing process, these are not just the testers, but also stakeholders such as subject matter experts (for instance, a Technical Unit, Business or Quality units), release managers or projects owners. These roles are not intended for the testing process in the execution process, but they want to see the results that are immutable and relative to a particular moment. For this purposes, a role system was introduced, coming with the Reviewer role.

Every application needs its administrator or a maintainer, and nor this role was forgotten. Admin user is being created during the installation phase. Admin user through the admin panel is eligible to manage the users and the organizations as well as to install missing packages and inform users about important notices.

7.1 Future Work

Developing this kind of application tends to be more complex in time, mainly when a single person develops it. This project, as it is of now, is only a tiny bit of the intended system.

As for now, users can use the application without any limitation; however, certain facilities could be introduced. First but not least is the pluggable support for the outcome of the execution. The flat files are not how the companies hold their testing history in most cases, especially when audit commissions control the software testing process. In such a case, we would like to export our results into more permanent storage such as Jira [20], ALM Vera [1] or TestLink [38]. The problem is that these applications have a minimum in common. Therefore the design and implementation of the elegant solution were out of the scope of this project.

We have developed the ideal solution for the single tester or the small teams in software testing and automated testing. Nevertheless, when we dive more into the automated testing purpose, we will figure out that some scheduler performs it on a build server or a cluster such as Jenkins. The application could also exploit this,

and rather than a custom worker integrated with the solution, we could provide a plugin for the build servers. Thus we would provide only the configuration and schedule details that could be delegated on an outer machine.

Quite a significant limitation when it comes to the Robot Framework use cases is the automated Front-End testing; this is an excellent feature of this framework; however, it is not possible to perform it on the server-side. A possible option would be to trigger the clients' execution, yet this collides with the original idea to move package installation and configuration out of the user machine to the hosting machine.

The idea of pluggable solutions for archiving and reviewing applications could be brought the level up. We could extend the Reviewer role by feature to approve or decline the executions. The tester would be then notified about the decision. This would be preceded by a request for approval that could be optionally set up in the initial test configuration, so we would not limit users who would not like to use this approach.

Bibliography

- [1] *ALM VERA*. URL: <https://www.tx3services.com/platform/vera-for-alm>. (accessed: May 25, 2021).
- [2] *Apache vs Nginx: Practical Considerations*. URL: <https://www.digitalocean.com/community/tutorials/apache-vs-nginx-practical-considerations>. (accessed: May 25, 2021).
- [3] *Bootstrap Documentation*. URL: <https://getbootstrap.com/docs/4.6/getting-started/introduction/>. (accessed: May 25, 2021).
- [4] *Celery-Project Documentation*. URL: <https://docs.celeryproject.org/en/stable/>. (accessed: May 25, 2021).
- [5] *CircleCI User Documentation*. URL: <https://circleci.com/docs/>. (accessed: May 25, 2021).
- [6] *Docker - Dockerfile*. URL: <https://docs.docker.com/engine/reference/builder/>. (accessed: May 25, 2021).
- [7] *Docker - Volumes*. URL: <https://docs.docker.com/storage/volumes/>. (accessed: May 25, 2021).
- [8] *Docker Official Images - python*. URL: https://hub.docker.com/_/python. (accessed: May 25, 2021).
- [9] *Docker-Compose*. URL: <https://docs.docker.com/compose/>. (accessed: May 25, 2021).
- [10] *Flask*. URL: <https://flask.palletsprojects.com/en/2.0.x/>. (accessed: May 25, 2021).
- [11] *Flask - Application Factories*. URL: <https://flask.palletsprojects.com/en/2.0.x/patterns/appfactories/>. (accessed: May 25, 2021).
- [12] *Flask - Command Line Interface*. URL: <https://flask.palletsprojects.com/en/2.0.x/cli/>. (accessed: May 25, 2021).
- [13] *Flask-SQLAlchemy Context Documentation*. URL: <https://flask-sqlalchemy.palletsprojects.com/en/2.x/contexts/>. (accessed: May 25, 2021).
- [14] *Flask-User v1.0*. URL: <https://flask-user.readthedocs.io/en/latest/>. (accessed: May 25, 2021).
- [15] Martin Fowler. “Model View Controller”. In: (July 2006). URL: <https://martinfowler.com/eaaDev/uiArchs.html#ModelViewController>.
- [16] *GitFlow*. URL: <https://datasift.github.io/gitflow/IntroducingGitFlow.html>. (accessed: May 25, 2021).
- [17] *Gunicorn documentation*. URL: <https://gunicorn.org/#docs>. (accessed: May 25, 2021).
- [18] *IntelliJ IDEA Robot Plugin*. URL: <https://plugins.jetbrains.com/plugin/7430-robot-plugin>. (accessed: May 25, 2021).
- [19] *Jenkins User Documentation*. URL: <https://www.jenkins.io/doc/>. (accessed: May 25, 2021).

- [20] *Jira Software*. URL: <https://www.atlassian.com/software/jira>. (accessed: May 25, 2021).
- [21] Cem Kanner. “Exploratory Testing”. In: (Nov. 2006). eprint: <http://www.kaner.com/pdfs/ETatQAI.pdf>. URL: <http://www.kaner.com/pdfs/ETatQAI.pdf>.
- [22] *Kubernetes Documentation*. URL: <https://kubernetes.io/docs/home/>. (accessed: May 25, 2021).
- [23] *NGINX Documentation*. URL: <https://nginx.org/en/docs/>. (accessed: May 25, 2021).
- [24] *PEP 8 – Style Guide for Python Code*. URL: <https://www.python.org/dev/peps/pep-0008/>. (accessed: May 25, 2021).
- [25] *PostgreSQL 13.3 Documentation*. URL: <https://www.postgresql.org/docs/13/index.html>. (accessed: May 25, 2021).
- [26] *Pylint User Manual*. URL: <http://pylint.pycqa.org/en/latest/>. (accessed: May 25, 2021).
- [27] *RED*. URL: <https://github.com/nokia/RED>. (accessed: May 25, 2021).
- [28] *Redis Documentation*. URL: <https://redis.io/documentation>. (accessed: May 25, 2021).
- [29] *RIDE*. URL: <https://github.com/robotframework/RIDE/wiki>. (accessed: May 25, 2021).
- [30] *RoboCoP*. URL: <https://github.com/MichalKyjovsky/RoboCoP>. (accessed: May 25, 2021).
- [31] *Robocorp*. URL: <https://robocorp.com/>. (accessed: May 25, 2021).
- [32] *Robot Framework Documentation*. URL: <https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html>. (accessed: May 25, 2021).
- [33] *RQ Dashboard project repository*. URL: <https://github.com/Parallels/rq-dashboard>. (accessed: May 25, 2021).
- [34] *RQ Documentation*. URL: <https://python-rq.org/>. (accessed: May 25, 2021).
- [35] *SQLAlchemy Documentation*. URL: <https://docs.sqlalchemy.org/en/14/>. (accessed: May 25, 2021).
- [36] *SQLAlchemy Documentation - Relationship Loading Techniques*. URL: https://docs.sqlalchemy.org/en/14/orm/loading_relationships.html. (accessed: May 25, 2021).
- [37] Richard N. Taylor Sungdeok Cha and Kyochul Kang. *Handbook of Software Engineering*. First Edition. Seoul, Irvine, Pohang: Springer, 2019. ISBN: 978-3-030-00261-9.
- [38] *TestLink Github repository*. URL: https://github.com/TestLinkOpenSourceTRMS/testlink-code/tree/testlink_1_9. (accessed: May 25, 2021).
- [39] *The Python Package Index*. URL: <https://pypi.org/>. (accessed: May 25, 2021).

- [40] *TIOBE Index*. URL: <https://www.tiobe.com/tiobe-index/>. (accessed: May 25, 2021).
- [41] *Web Programming Languages Usage (2021)*. URL: https://w3techs.com/technologies/overview/programming_language. (accessed: May 25, 2021).

List of Figures

2.1	Robot Framework - Architecture	9
3.1	Reviewer - Use-Case Diagram	12
3.2	Tester - Use-Case Diagram	13
3.3	Administrator - Use-Case Diagram	14
3.4	Model View Controller	16
5.1	High-level architecture Diagram	25
5.2	Database Model - Class Diagram	26
5.3	Role Based Authorization - Class Diagram	27
5.4	Web Application Component - Domain Diagram	28
5.5	Robot Executor Component - Domain Diagram	31
5.6	Test Environment - Architecture Scheme	31
5.7	Production Environment - Architecture Scheme	32
6.1	Login Page	35
6.2	Registration Page	35
6.3	Welcome Page - Reviewer	36
6.4	Create a new organization page	36
6.5	Manage the memberships page	37
6.6	Manage the organization page - A	37
6.7	Manage the organization page - B	37
6.8	Executions overview page	38
6.9	My Profile page	38
6.10	Help Desk page	39
6.12	Login Page - OAuth	40
6.13	Welcome Page	40
6.14	Dashboard	41
6.15	Execution Lab	41
6.16	Upload from local file system form - A	42
6.17	Upload from local file system form - B	42
6.18	Configuration panel - editable	43
6.19	Configuration panel - read-only	44
6.20	Execution Results view	44
6.21	Execution History view	45
6.22	Workspace view	45
6.23	Notification Email page	46
6.24	Users Management page	46
6.25	Groups Management page	46
6.26	Request for Installation page	47

A. Attachments

A.1 Contents of the Attachment

The structure of the attachment is as follows:

- The *figures* - the directory containing the the figures from the thesis for the lucid orientation.
- The *source-code* - the directory that contains the source of the application, unit tests, configuration files, and installation scripts. There are highlighted installation two scripts - "*install.sh*" for Linux-like operating system and "*install.bat*" for windows. The "*README.md*" provides advanced documentation for the application's installation and structure.
- The *docs* - the directory that contains the programmers documentation sources. It has two sub-directories - the "*doctrees*" that contains the auxiliary file for the web and the "*html*" directory with the "*index.html*" as an entry-point to programmers documentation.
- The *tex-source* - directory which contains the TeX source code used for the thesis compilation.
- The *thesis.pdf* - the PDF version of the thesis.
- The *manual.pdf* - the user documentation for our project.

