UNIVERSITY OF TARTU
Institute of Computer Science
Conversion Master in IT Curriculum

Janeli Õun

Testing Estonian Scientific Computing Infrastructure Self-Service in Cypress Framework

Master's Thesis (15 ECTS)

Supervisors: Ilja Livenson, MSc
Viktor Mirieiev, MSc

Tartu 2022

Testing Estonian Scientific Computing Infrastructure Self-Service in Cypress Framework

Abstract:

Academic communities and research organisations are increasingly in need of powerful computational resources. In Estonia, scientists and researchers can use the ETAIS self-service portal to access computing and storage resources. To improve the user experience and the quality of the user interface, the mistakes made in the development of the software should be prevented from reaching the users. This can be done by finding the errors through testing.

This thesis aims to extend the automatic test coverage for the ETAIS self-service portal. The thesis includes the introduction of ETAIS and the best practices of software testing. The architecture of the portal is documented, automatic integration tests are planned and executed using Cypress software. The results are presented, discussed and suggestions for further actions are given.

Keywords:

Estonian Scientific Computing Infrastructure, Cypress framework, automated testing

CERCS: P170 Computer science, numerical analysis, systems, control

Eesti Teadusarvutuste Infrastruktuuri iseteeninduse testimine Cypress raamistikus

Lühikokkuvõte:

Akadeemilised kogukonnad ja uurimisorganisatsioonid vajavad järjest võimsamat arvutusjõudlust. Eestis on teadlastel ja uurijatel võimalik pääseda ligi arvutus- ja mäluresurssidele ETAIS'i iseteenindusportaali kaudu. Portaali kasutajakogemuse ja kasutajaliidese kvaliteedi täiustamiseks peab takistama arendusfaasis tehtud vigade jõudmist kasutajateni. Seda saab ennetada, leides probleemid testimise teel üles.

Selle töö eesmärk on suurendada ETAIS'i iseteenindusportaali testidega kaetavust. Töö sisaldab ETAIS'i ja tarkvaratestimise parimate tavade tutvustust. Portaali arhitektuur dokumenteeritakse, planeeritakse automaatsed integratsioonitestid ja teostatakse need Cypress tarkvara kasutades. Esitletakse tulemused, arutletakse nende üle ja antakse soovitused tulevaste tegevuste kohta.

Võtmesõnad:

Eesti Teadusarvutuste Infrastruktuur, Cypress raamistik, automaattestimine

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Acknowledgements

I would like to thank my supervisor Ilja Livenson for providing me with the fascinating topic for this thesis and his guidance throughout this project. I am also deeply grateful to my colleague Ivo Klaas for relentlessly helping me through the obstacles encountered during the practical test-writing process. Finally, I would like to thank Sander Kuusemets for all the support, encouragement and assistance I received from him, not only during writing this thesis but throughout the whole journey toward my Master's degree.

Table of contents

Abbreviations and Terms **6**

1. Introduction **7**

2. Background information **9**

2.1. Estonian Scientific Computing Infrastructure Self-Service 9

2.1.1. The need for scientific computing 9

2.1.2. About Estonian Scientific Computing Infrastructure 10

2.1.3. The concept of the self-service portal 10

2.1.4. The architecture of the self-service user interface 11

2.1.5. ETAIS development process 15

2.1.6. The significance of using the Waldur platform 16

2.2. Software testing 17

2.2.1. Need for software testing 19

2.2.2. Testing levels 20

2.2.3. Best practices of software testing 21

2.2.3.1. Automated testing 21

2.2.3.2. Continuous testing 22

2.2.3.3. Smoke testing 23

2.2.3.4. Regression testing 23

2.2.4. Previous testing in the ETAIS self-service 23

2.2.6. Choosing testing metrics 23

2.2.7. Thesis testing goal 25

3. Planning the tests **26**

3.1. Scope 26

3.2. Testing software 27

3.3. Best practices for tests 28

3.4. Selecting test suites 29

3.5. Selecting the testing data 34

4. Results **35**

4.1. Structure of the testing environment 36

4.2. Overview of the created tests 36

4.2.1. Discovered errors 39
4.3. Running the tests 39
4.4. Integrating the tests 41
5. Discussion **43**
6. Summary **46**
References **47**
Appendix **51**
I Testing commands 51
II Licence 52

Abbreviations and Terms

Back-end	Relating to the operational server-side (the data access layer) of a website.
Continuous integration (CI)	The practice of frequently merging all developers' isolated code changes to a shared mainline.
Continuous delivery (CD)	The practice of automatically preparing code changes for a release to production.
Development and operations (DevOps)	Set of practices, philosophies and tools that combine processes between software development and IT operations.
Estonian Scientific Computing Infrastructure (ETAIS)	A project that provides infrastructure and application support services to the research communities and R&D companies in Estonia.
Front-end	Relating to the graphical user interface (the presentation layer) of a website.
High-performance computing (HPC)	The practice of processing data and performing complex calculations at high speed, usually by aggregating computing power.
HTTP request	A request made to host by a client to request access to a resource on the server.
Test case	A set of preconditions, inputs, actions and expected results based on the test conditions.
Test suite	A collection of related test cases with a similar goal.
TypeScript	High-level programming language that is most well-known as a scripting language for Web pages.
User interface (UI)	User interface is the means that are used for a human to interact with software or hardware. This includes display screens, the appearance of web pages, keyboards, mouses etc.

1. Introduction

The purpose of this thesis is to create automatic tests for the Estonian Scientific Computing Infrastructure self-service portal in order to discover the errors in software faster, decrease the loss of resources used to fix them and improve the quality and reliability of their services.

Researchers and scientists are increasingly in need of powerful computational resources. With the rise of artificial intelligence, machine learning and big data analytics, more academic communities and research and development organisations are searching for ways to access High-Performance Computing (HPC) services [1]. In Estonia, an institution called Estonian Scientific Computing Infrastructure (ETAIS) combines all major Estonian HPC providers [2] to offer computing and storage resources for the Estonian scientific community [3].

To greatly improve access to the resources, regardless of the users' location, ETAIS has implemented a self-service portal that can be accessed by both service providers and customers [4]. The portal acts as a marketplace, where service providers can offer resources that the customers can use for their projects. The portal is based on the Waldur service brokerage platform, which is open-source software built for managing computational resources [5]. This platform is not only used by ETAIS but also by governments, enterprises and research infrastructures all over the world.

To ensure the quality of the self-service portal, software errors have to be found as soon as possible. Fixing mistakes later in the development lifecycle is costly and customers may not be satisfied with errors in their product, harming the reputation and credibility of the software. In order to find errors quickly, testing should be done often, which is easier to achieve with automatic tests. Automated testing can be run after every change to validate that the functionality of preexisting code is not harmed by the new update. This does not ensure there will be no errors. Still, depending on the thoroughness of the created test cases, it will significantly improve the probability of finding a mistake before deploying the code into production. The automated tests created for the ETAIS self-service portal can be developed in a way that they could be used on other Waldur-based portals as well, to help improve the quality of computational service provision all over the world.

Prior to the writing of this thesis, there were only a few automatic tests available for Waldur that did not cover a lot of the use cases[1]. Because of that, the ETAIS self-service developers did not find many errors via automatic tests and relied on log traces and users reporting the

[1] Summary of code coverage is provided in Section 2.2.5. Previous testing in ETAIS.

errors found in the live environment. To improve the situation, it was first necessary to map the architecture and functionalities of the self-service portal, as the previous information about the structure was outdated and there was no documentation of the technical requirements used to create and develop the portal. Subsequently, a strategy was needed to decide on how to start the process of extending the test coverage and the scope of this thesis.

The main body of the thesis at hand is divided into four chapters. The first chapter introduces ETAIS and the self-service platform in greater detail while also describing the tenets of software testing that have to be taken into account in planning the testing strategy and creation of the tests. The second chapter concentrates on planning the tests, and the third describes the results of the work conducted. The fourth chapter consists of a discussion of the process and results. The thesis ends with a summary.

After the main body of the thesis, there are appendices consisting of the guide for running the tests and the licence outlining the intellectual property rights of this thesis.

2. Background information

This chapter gives a general overview of the Estonian Scientific Computing Infrastructure (Eesti Teadusarvutuste infrastruktuur – ETAIS) self-service and the best practices of software testing. The first part of the chapter describes the need and significance of the services provided by ETAIS, as well as the architecture and the development process of the self-service portal. The second part of the chapter gives a brief overview of the goal of software testing and the types and methodologies available for conducting it. The existing testing practices of ETAIS and the metrics for measuring testing effectiveness are also described. Based on the information presented in this chapter, the testing goals for this thesis are set.

2.1. Estonian Scientific Computing Infrastructure Self-Service

According to the ETAIS webpage [3], the goal of the ETAIS project is to increase the competitiveness of the Estonian computing and data-intensive research disciplines by providing access to a new and modern scientific computing infrastructure. The primary way to access this infrastructure is through the ETAIS self-service portal.

ETAIS self-service portal is a single entry point for provisioning and managing computational and storage resources shared by ETAIS consortium members and public cloud providers. It is aimed at research groups affiliated with Estonian research and development institutions from both the public and private sectors [6].

2.1.1. The need for scientific computing

The primary need for the ETAIS services stems from the scientists' need for accessible and powerful computational resources. According to an article by Agrawal and Choudhary [7], while there are empirical and theoretical scientific discoveries that can be performed by gathering observations through experiments and developing general theories without the assistance of computers, there are also systems that are too complex for humans to research without technological support. Computational and data-driven science use computing power to process those immense workloads, like when creating simulations of complex real-world phenomena or extracting patterns from massive data sets.

To perform this kind of scientific research, scientists often need much higher firepower than traditional computers and servers can provide. The practise of aggregating computing power to deliver on that need is called High-Performance Computing (HPC). It is a way of

processing huge volumes of data at very high speeds using multiple computers and storage devices as a cohesive cluster. By dividing significant computational problems into small, simple, independent tasks and running them simultaneously, scientists can get their results faster, with less wasted time and money. In some cases, physical testing might not be feasible, leaving computer-created simulations as the only viable way of gathering insight [8]. These HPC clusters are the primary resources available through ETAIS self-service [9].

2.1.2. About Estonian Scientific Computing Infrastructure

Estonian Scientific Computing Infrastructure (ETAIS) belongs to the Estonian Research Council's (ETAG) roadmap of research infrastructures providing computing and storage resources for the Estonian scientific community [3].

ETAIS is a nationwide unified infrastructure of scientific computation, consisting of scientific computation centres, the computation clusters and data repositories inside these centres, central services connecting the centres and resources, software engineers, and the training of end-users. The services are available to all R&D institutions and research-based enterprises. ETAIS is managed by the Council of the ETAIS consortium, while the infrastructure's daily work is supported by the IT departments of the relevant institutions or special departments dedicated to scientific computation [10].

2.1.3. The concept of the self-service portal

The ETAIS self-service portal allows users to access the resources regardless of their location or operating system used. The use of the self-service portal significantly improves access to computing and data storage (including mid-term archiving) capabilities, as well as customer support [4]. This offers research groups a way to collaborate on using and sharing research infrastructure to minimise the bureaucracy of negotiating access, quotas and payments.

According to ETAIS Self-Service Guide [6], there are three user groups in the portal – service providers, organisations as the customers, and their end-users. The portal is built on the concept of a marketplace, where service providers can offer resources to customers' projects, as visualised in Figure 1 below.

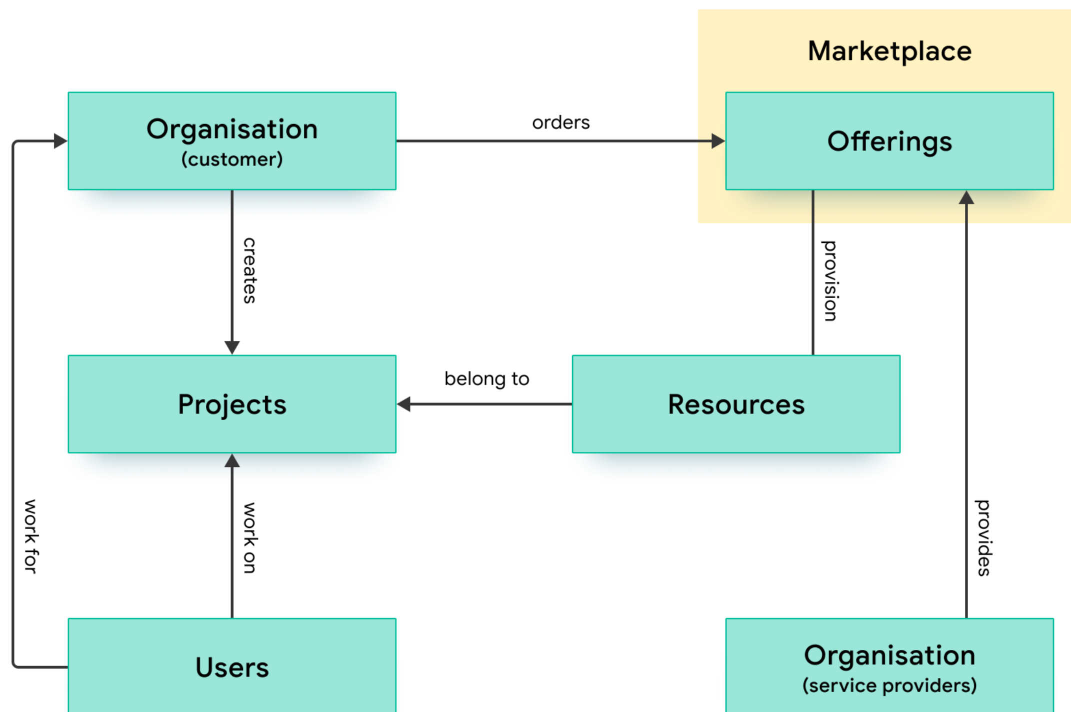

Figure 1. Main use case and relations of the ETAIS self-service portal.

Customers are organisations that use the portal to access resources for their projects and manage the system's end-users (humans or robots), who will work on those projects. Projects are entities that a customer can use to combine and manage their resources and teams. Service providers are the organisations that provide offerings – concrete services available for ordering, such as High-Performance Computing Clusters, Virtual Machines (servers with network connectivity), and Block Devices (persistent volumes for data storage).

To date, approximately 1500 users[2] from Estonia (mainly researchers and students) have used the self-service portal with a steady growth pattern. As such, the stability of the user interface is becoming more important for providing a good user experience.

2.1.4. The architecture of the self-service user interface

For the purpose of setting the scope and planning the test cases later in the thesis, it is important to understand the structure of the self-service portal. The documentation about this is incomplete and outdated, which is why the architecture had to be discovered by manually browsing with different permissions activated, mapped out and re-documented during this phase of the thesis. A visualisation of the pages, organised by permissions needed to visit them, can be seen in Figure 2.

[2] According to the statistics provided by ETAIS self-service operators. For more information, contact support@hpc.ut.ee.

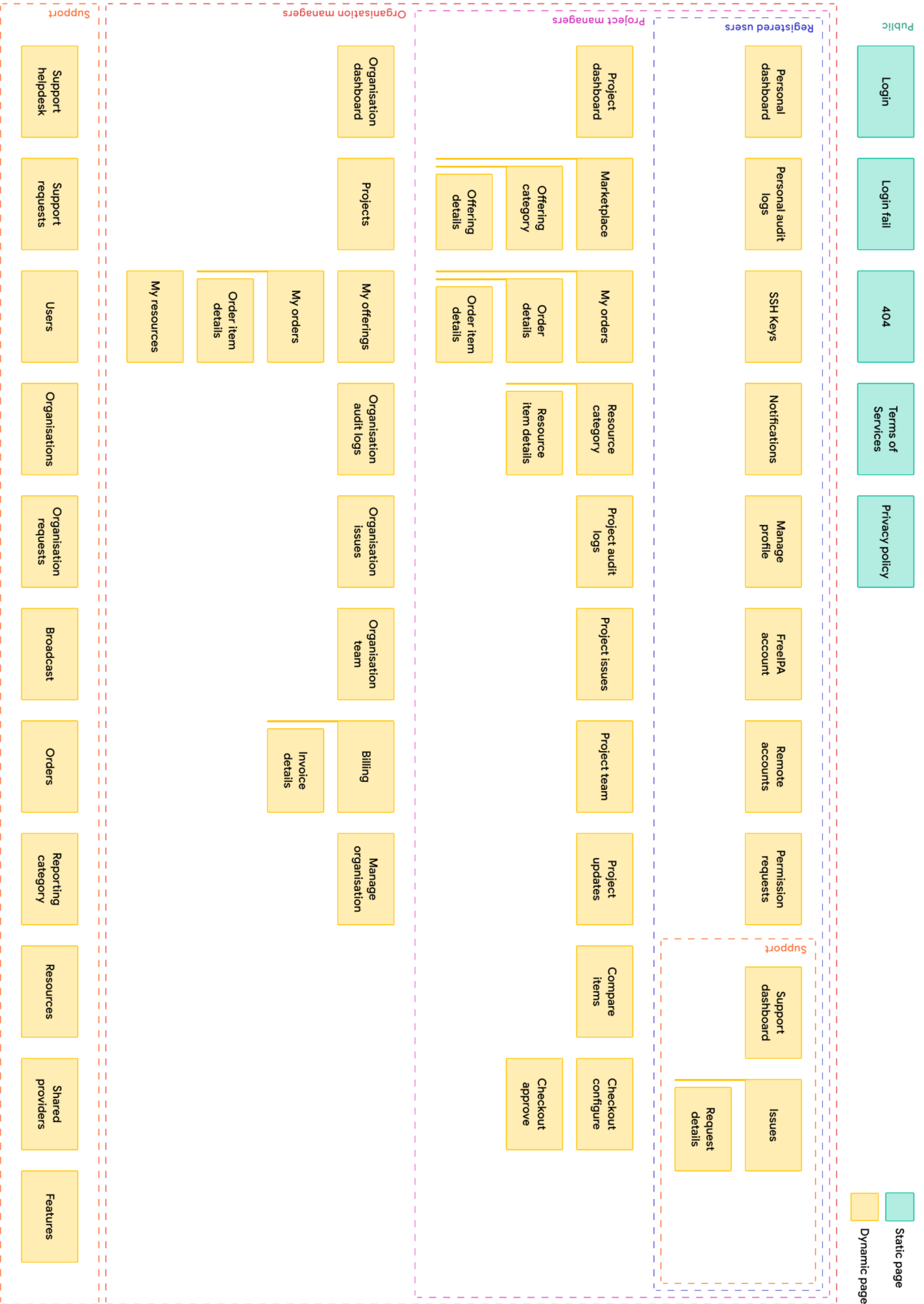

Figure 2. The visualisation of the pages, organised by permissions needed to visit them.

ETAIS's self-service user interface consists of 50 unique dynamic main pages and five static pages. The static pages remain the same for all the users, and they display fixed content. Those pages are *Login*, *Login failure*, *Terms of Service*, *Privacy policy* and *404* (missing content) page. The dynamic pages display content that adjusts to the changes in the database and the permissions and actions of the user. Each user has a hierarchical role – organisation owner, project manager or system administrator. Each role has different permissions that allow the user to see corresponding dynamic pages. There are additional roles of staff (global read-write) and support (global read-only) that fall outside of the hierarchy of the other roles and can be assigned separately to any user. These last roles are intended for the operators of the self-service portal.

All users have access to pages intended for overview and management of the user's personal profile, which are as follows:

1. Personal dashboard (overview of organisations and projects where the user participates);
2. Personal audit logs (overview of events related to the user);
3. SSH keys (management of the user's public SSH keys);
4. Notifications (management of the user's notification);
5. Manage profile (management of the user's profile details);
6. FreeIPA account (management of a common Unix account trusted by members of ETAIS - UT, TalTech and KBFI);
7. Remote accounts (overview of accounts created for the user by remote service providers, e.g. LUMI super-computer);
8. Permission requests (management of user requests for new permissions in projects and organisations);
9. Support dashboard (overview of support tickets created by user and form to create new support ticket);
10. Support issues (overview of issues created by the user);
    10.1. Support request details (detailed information about the issue and ability to communicate with operator's support).

Users with permissions in projects can access all pages mentioned before, as well as the following pages connected to the management of the IT infrastructure and the project's day-to-day work:

11. Project dashboard (overview of the project resources and latest events);
12. Resources (management of the provisioned resources by category);
    12.1. Resource item details (management of the details of a specific item);
13. Marketplace (overview of offerings available for provisioning);
    13.1. Offering category (overview of offerings available by category);
    13.2. Offering details (overview and available configurations of the offering);
14. My orders (overview of the orders made by the user);
    14.1. Order details / Checkout – review (overview of the order);
        14.1.1. Order item details (overview of details of a specific order item);
15. Project audit logs (overview of events related to the project and its resources);
16. Project issues (management of trackable requests to the operator of the project);
17. Project team (management of the project team);
18. Project updates (approval of project information modifications for remote offerings);
19. Compare items (comparison of details of selected offerings);
20. Checkout – configure (management of shopping cart, submitting orders);
21. Checkout – approve (approval of submitted orders).

Users with permissions in organisations have access to all aforementioned pages with the addition of pages dedicated to the overview and management of the organisation's projects, members and subscriptions to resource providers. Such pages are:

22. Organisation dashboard (overview of managed resources and projects);
23. Projects (management of the projects);
24. My offerings (overview of offerings owned by the organisation);
25. My orders (overview of the history of requests for specific offerings);
    25.1. Order item details (overview of specific order item);
26. My resources (overview of resources used in projects under the organisation);
27. Organisation audit logs (overview of events related to organisation, projects and resources);
28. Organisation issues (management of trackable requests to operators of projects under the organisation);

29. Organisation team (management of the organisation members and their project affiliations and user roles);
30. Billing (overview of resource usage accounting information);
    30.1. Invoice details (overview and accounting management of details of specific invoice);
31. Manage organisation (management of the organisation details).

There is an additional support role assigned to users who should deal with the requests and reports. All registered users get access to the support dashboard, issues and request details pages, but only the support role has permission to access the following pages:

32. Support helpdesk (overview of all trackable requests on the portal, creating new requests);
33. Support requests (overview of all trackable requests on the portal and customer support satisfaction overall rating);
34. Users (management of all users on the portal);
35. Organisations (management of all organisations on the portal);
36. Organisation requests (management of requests for new organisations);
37. Broadcast (creation and history of mass user notifications);
38. Orders (management of all orders on the portal);
39. Reporting (reports of user activities, financial transactions and service offerings);
40. Resources (management of all resources on the portal);
41. Shared providers (dedicated report for OpenStack shared service providers);
42. Features (activation and deactivation of features available for the portal);

Short descriptions of all the functions available for each page were included in the documentation of all the pages found during the process of discovering the structure of the portal and the result was sent to the developers. It can be referenced for creating automatic tests, understanding support tickets, looking for the instances of a specific feature and explaining the structure and features to users.

2.1.5. ETAIS development process

ETAIS self-service is based on the open-source product called Waldur, which is co-developed by the University of Tartu. Development adopts DevOps principles and uses automation tools

for running automated tests on different steps: commits to feature branches, releases, regular linting of source and artefacts.

The scope of the thesis is on the front-end part (aka Waldur Homeport) of the platform, where the automated testing happens mainly during the code development phase, i.e. before the application is packaged into containers and shipped to the registry. A more detailed description of the Waldur development and operations process is provided in Sergei Zaiaev's MSc thesis [11]. Below, the main steps of building the front end of software are displayed (Figure 3).

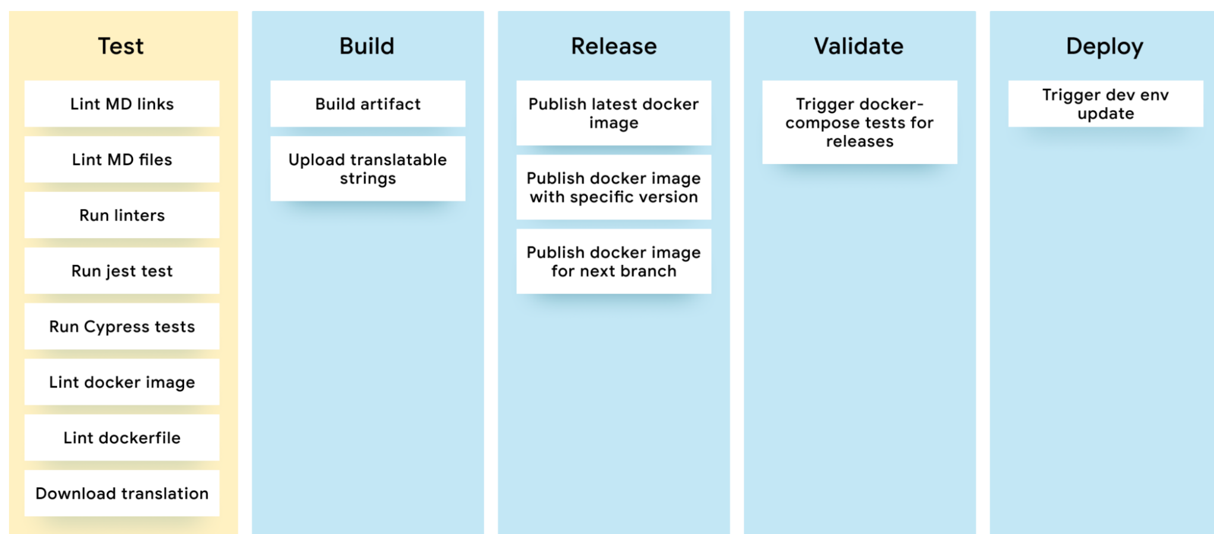

Figure 3. The visualisation of the CI/CD pipeline of the Waldur Homeport. The tasks performed during the test phase are highlighted with a yellow background.

The tests are run by CI/CD pipeline on each commit, so it is important that they are reasonably quick and self-contained.

In addition to manual and automated testing, ETAIS is collecting user issues automatically via Sentry[3] from production environments. While the latter is beneficial, it still means that a user has had a negative experience due to a crash in the system. Hence, improvements in automated testing across multiple systems are considered to be essential for ETAIS.

2.1.6. The significance of using the Waldur platform

Waldur service brokerage platform, on which the ETAIS self-service portal is based, is an open-source platform built for managing private clouds, public clouds and HPC Centres [5]. Since its code is open source and is used by clouds and HPC centres all over the world, it

[3] sentry.io application monitoring and error tracking software

would be useful to create tests robust enough that they would benefit not just the ETAIS self-service portal, but other portals based on Waldur as well. An example of such benefactors would be the Puhuri portals [12] used by the LUMI supercomputer. The code of tests formed in this thesis can be made publicly available to other computing infrastructure providers, who might benefit from similar tests, reducing the workload of their test developers. There are already some tests created for Waldur that can be used as a reference and are improved on during this thesis.

Puhuri Portals are deployed from January 2022 for larger organisations and communities. At the moment of writing, the number of unique users was approximately 420. The number is expected to increase significantly after the go-live of LUMI’s main partition LUMI-G. The users behind the numbers are primarily researchers from different European countries.

2.2. Software testing

Software testing is a set of activities in the software development cycle, which are done by executing the software with a selected set of inputs and checking whether the program behaves in an expected way, as defined in *Handbook of Software Engineering* by Fraser and Rojas [13]. It can roughly be categorised into two processes: validation and verification.

In *Software Testing and Continuous Quality Improvement, Third Edition,* Lewis [14] claims that software validation evaluates whether the product meets the customer’s needs. It traditionally happens at the end of the project to assess if the software is what the users expect it to be. Software verification determines whether the product adequately and correctly performs all intended functions without performing any functions that degrade the performance of the system. It ensures the quality of software products, and when incorporated into testing, software verification occurs throughout the development cycle.

To ensure the quality of a software product, it is important to first understand what it is composed of. According to ISO/IEC 25010 standard [15], the software quality model comprises a broad range of characteristics such as functional suitability, usability, security, maintainability, etc. (visualised in Figure 4). Each characteristic has multiple attributes that can be evaluated to gain an understanding of the level of quality for those characteristics.

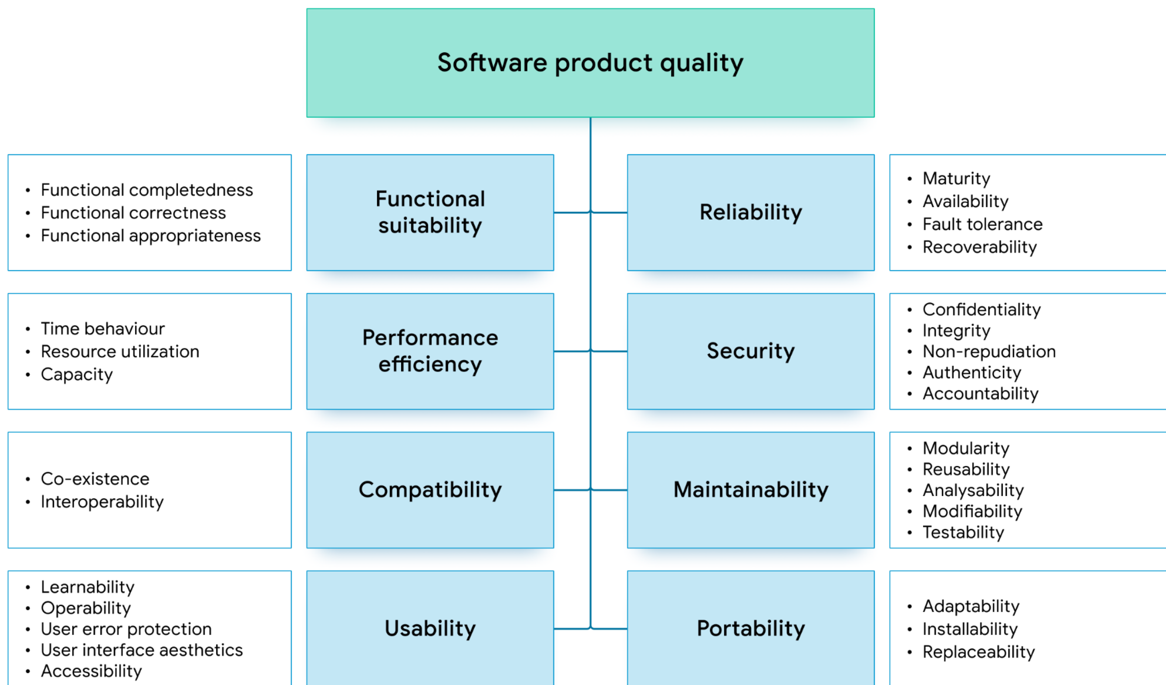

Figure 4. The characteristics of software product quality.

Since those characteristics are very diverse and extensive, it is unfeasible to test for them in a similar fashion. As these quality aspects stem from different sources such as front-end, back-end and database, Mohan [16] suggests also combining these characteristics into three stacks of the user interface, services and database layers (Figure 5).

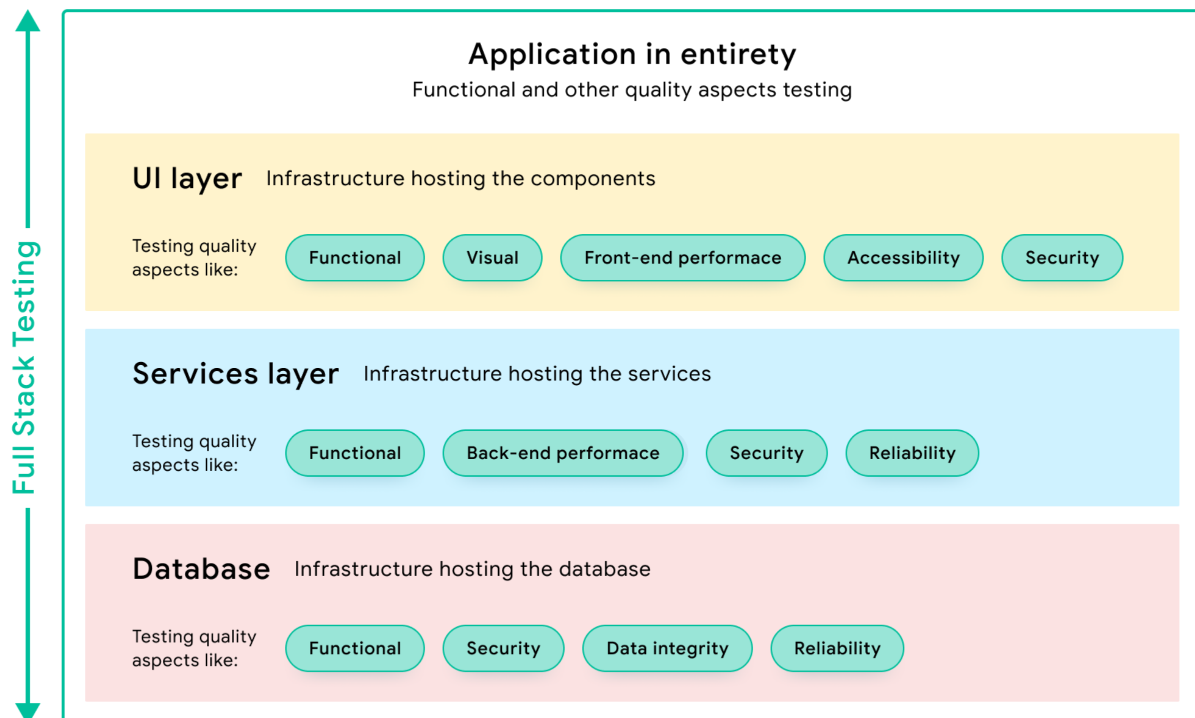

Figure 5. The characteristics of software product quality, combined into three testable stacks.

Testing by the described stacks allows the tester to target specific data sources with testing strategies that work best on them and find failures of similar origin, making it easier to locate the error and fix it. This makes it important to take this into account when defining the scope of the test scenarios later.

2.2.1. Need for software testing

The primary reason for software testing is to detect software failures so that the defects may be discovered and corrected. As explained by IEEE [17], if a programmer makes a human error, it can manifest as a fault – an incorrect statement in software code. Executing the faulty code might produce wrong outputs in certain situations, which can cause failures – observable incorrect behaviours in the software.

Those failures have to be found as soon as possible as the cost in resources, reputation and in some cases even user safety is much lower when the failure is corrected earlier rather than later. According to Krasner [18], when the failure is found before it is launched, it can still cause expenses due to waste of resources on the faulty outcome, potential costs for delays, the need to redo the work and conduct a failure analysis to establish the initial cause to prevent it from happening again.

Still, those costs are significantly higher once the software has been deployed, as there are much more steps needed to fix the mistake and the costs can now include reimbursing the purchases, customer complaint handling, managing downtime, support teamwork, legal costs, wasted marketing costs, drop in sales, not to mention brand damage and tarnished reputation. Quadri and Farooq [19] claim that generally, if a defect costs one unit to fix after finding it during the requirement review, the same mistake would cost 10-100 units to fix after the product has been released.

Even though the testing can prevent unnecessary expenses, it is not cheap itself. In 2019, a panel of CIOs and senior technology professionals reported that around 23% of their organisation's annual IT budget was allocated to quality assurance and testing [20]. Since it would be needlessly expensive and, according to Spillner and Linz [21], even impossible to exhaustively test for all combinations of input data and their preconditions, the scope of the tests has to be clearly defined on the goals of the testing process. There is no set way to select the tests, as the same authors claim that testing is also context-dependent, meaning that every software has a different purpose, environment and therefore testing needs. The cost of testing has to be balanced with the cost of risks associated with not testing.

2.2.2. Testing levels

Since testing is most cost-efficient when done as early as possible, Lewis [14] suggests that testing should be parallelly integrated into the development process, whether it happens iteratively or linearly. The development phases may change depending on the specific software, but the typical testing levels that verify those phases are unit, integration, system, and acceptance testing (Figure 6). These levels will be described in more detail based on the writings of Morgan, Samaroo, Thompson and Williams [22].

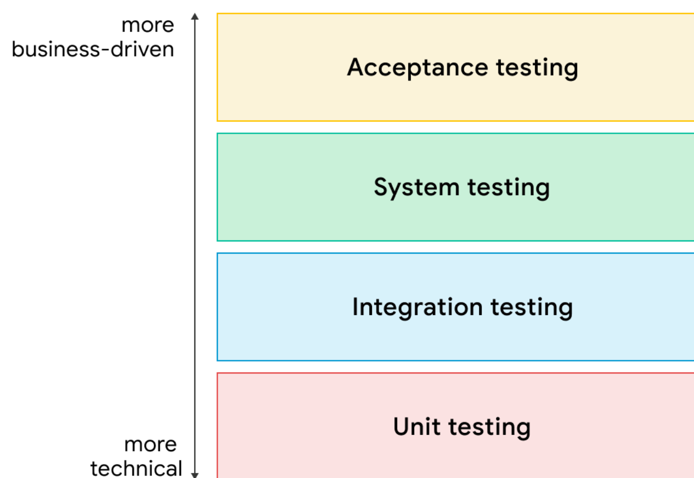

Figure 6. The hierarchy of the typical software testing levels.

Unit testing is the most granular level of software testing. The code for a software product is generally written in smaller components – units. They are usually constructed in isolation, to be integrated at a later stage. Prior to that integration with other units, each unit can be tested for meeting its quality requirements. The tests can be created after the code or, in a process called Test-Driven Development, the tests can be written first. The code is built and tested and changed until it passes all the tests. Unit testing assists developers in finding errors without a need for formal defect management.

The next level is called integration testing. Once multiple units have been integrated into a larger system, integration testing should be run to expose faults in interactions between integrated components that may not be found at the unit testing level. It is considered best practice to test integrations incrementally, not after combining all units together (known as the "big bang" approach), since it allows integration issues to be located more easily and reduces the risk of discovering problems later in the process. This level can be separated into component integration testing and system integration testing. Component integration testing is focused on the integration between projects, while system integration testing deals with interactions between systems, services and external organisations.

The third level of testing is system testing. Since the tests described above are not taking into account the conditions of the live environment or the user interactions across the system as a whole, they are unlikely to reveal any failures resulting from those aspects. System testing runs in a representative live environment and searches for defects in complete functions that the system must perform, such as searching and browsing for specific content across multiple pages. In addition, system testing should cover non-functional requirements like usability, performance, stress handling and recovery, that are describing the behaviour of software in both regular and extreme operating cases.

Acceptance testing is performed to confirm to end users that the software will meet their expectations. Its main goal is to demonstrate that the software is ready for use and that it functions in conformance with the business needs, processes, contracts and regulations.

2.2.3. Best practices of software testing

There are multiple ways of conducting software testing, depending on the specifications and needs of the current project. Over time, some practices have proven to be more popular in the testing community due to the ease of use and reliable results that are in balance with the expenses.

2.2.3.1. Automated testing

Since defects cost less to fix when they are discovered earlier, and every time the code is updated new faults might be introduced, it can be reasoned that it is best to test as soon as possible after every change. Mathur [23] explains that doing this manually would be unfeasibly labour-intensive as the routine execution of many tests can be tiring and prone to errors. Test automation can aid in both faster and more reliable completion.

Automated testing has a large upfront cost due to the time spent on planning, creating, and maintaining the automated tests, which can be significantly longer than giving instructions to manual testers. Also, it has to be taken into account that the automation tools might have an additional cost and that humans working on automated testing scripts are usually specialised and expect much higher salaries than manual testers. However, the more times the automated tests can be executed, the less time and human effort running the tests take in comparison to the manual, and the greater the return on investment, making it generally very beneficial for any medium and long term project.

According to Kramer and Lageard [24], automated tests can be realised by interacting with the software through the graphic user interface (GUI), using API or directly at the code level.

This means that the same system can be tested by imitating user inputs such as mouse clicks, querying the API endpoints directly, or automating each code component with unit tests.

It is important to note that automatic tests should not be used without combining with other types of testing, including manual, so that it would be possible to catch errors outside the predefined tests' framework. Spillner and Linz [21] stress that the testing can only show us the presence of defects, not their absence and that it is impossible to test for the scenario. This means that a low number of failures revealed in the automatic test might mean that the test cases used are ineffective.

Automated testing can also be run to analyse the code itself, without executing the application. This can be done by using tools called linters, which detect simple logic defects and deviations from the standards of code style. Tómasdóttir, Aniche and Deursen [25] investigated the adoption of linters by developers. All interviewed developers claimed that they used linters. The main reasons cited were catching errors early in the process, highlighting typos that are hard to notice, enforcing rules that make the code more readable, keeping the code consistent, speeding up code review, learning about the language used, etc. Linters are considered fast and easy to use and are therefore often the first automated test used in a project.

2.2.3.2. Continuous testing

The benefits of automated testing shine the most when used alongside continuous integration (or CI). Continuous integration is considered the best practice among development teams. It involves each developer committing their code to the main source code regularly so that their updates would have only small differences from the codebase and there would be fewer integration conflicts. Since every addition and change carries a risk of breaking something, it is important to validate that the new code works as expected as soon as possible. A key part of CI is running the test every time. This is called continuous testing and it should be done automatically, as it is repetitive, time-consuming and needs consistency on small details [26].

2.2.3.3. Smoke testing

Dustin, Rashka and Paul [27] suggest that the first test run on any new software version or build should be smoke testing, also known as build verification testing. It assesses if the main functions of software work: e.g. whether the program runs when accessed, the UI opens, or if the main feature reacts to user input. Smoke testing first ensures that no effort is wasted in testing an incomplete build.

2.2.3.4. Regression testing

Expanding on the smoke test, regression testing includes checking all existing functionalities already proven viable before. It focuses on finding regressions in working software and defects such as degraded or lost features, including old bugs that have come back. Experience shows that modifying an existing program is a more error-prone process than writing a new program [27]. Therefore, regression testing should occur after each update. Since it is usually lengthy even with automatic testing, it is a good idea to select the specific regression testing procedures based on the highest probability of detecting the most errors.

2.2.4. Previous testing in the ETAIS self-service

According to ETAIS developers, their self-service portal has been in development for many years. It follows an API-first design, where the main business logic is implemented in the back-end part of the solution. The developers report that the test coverage of the back end is high, however, the web client has been developed in a less structured way. Prior to this thesis, the following testing has been performed for the front-end:

- Unit tests – jest[4]-based tests for specific components of the user interface. The current code coverage is ~65%.
- Integration tests – scenario components using Cypress, which currently consist of 88 test cases. The thesis at hand aims at extending these tests. Mapping the existing test cases to the functionalities found while documenting the architecture, the integration tests cover about 8.5 unique pages, which is ~15.5% of all the unique pages found.
- System testing – performed manually in the development and testing environment.
- Acceptance testing – not used.

Both unit and integration tests are run for every change in the development.

2.2.5. Choosing testing metrics

To measure how effective the applied testing has been, it is important to gather metrics before and after the implementation of the new tests. Metrics are quantitative measures used to evaluate and estimate the attributes like quality or progress of a process [28]. According to Bose [29], when the markers of success are established early in the planning stage, it becomes easier to monitor the progress and evaluate the state and adequacy of the final result.

[4] jestjs.io testing framework for JavaScript and TypeScript

Publications by IEEE, as concluded by RTI International [30], have presented numerous potential metrics that can be used to test each software quality characteristic. Since there are so many possible metrics to follow, it is important to choose the ones that are the most informative and useful in the current case. To be able to quantify the success of the testing process, the metrics should be derivative, meaning that the effectiveness of testing can be divided by a resource used. This way, the metrics gathered before changes in testing and after them can be compared.

A selection of the derived metrics that can be used to evaluate the success of the conducted tests, chosen based on the suitability for previous work on ETAIS self-service, are as follows:

1. Test effort measures how much there is to show for the effort put in. The metrics to measure this can be how many tests have been run, created, or reviewed per time unit or how many defects have been found per test.
2. Test effectiveness reveals how successful each of the designed tests is. It calculates from the total amount of faults found the percentage that every single test contributed, which should give an indication about the quality and value of the test.
3. Test coverage indicates how much of the software is being tested. It can be found based on the percentage of requirements or the amount of code covered by tests. If measuring the extent of code covered, this metric can be called code coverage. This gives an understanding of whether there are enough tests in the test suite or if there is a need for more to be added.
4. Test economy metrics help understand how much has been spent on testing and can help to plan the allocation of resources in the future. These metrics can be found by comparing the time or cost allocated with the actual expenses. These differences can be found per project, test case, bug fix or time period.
5. Defect distribution measures which defects are found, which helps with monitoring the errors and planning their fixes. The distribution could be divided by cause, feature, severity, type, etc.

The choice of the primary metric for this thesis is based on the cost of conducting the measurements. As it has previously been established that the testing process is costly and the resources used should not exceed the resources potentially risked, it is logical to use what metrics can already be found readily available. Since test coverage can be easily calculated by selecting units (like lines of code, statements, loops, requirements, pages) of software and comparing the amount covered with tests with the full amount, it can be used without having

to gather any additional data. Some testing environments offer automatic code coverage for unit tests, which can keep the cost of using code coverage as a metric at a minimum. Focusing on the test coverage also helps in evaluating whether the amount of tests created is sufficient or is there a need for higher coverage amount or coverage on different units of software.

2.2.6. Thesis testing goal

The goal of this thesis derives from the best practices and work already done on ETAIS self-service. The best practices call for continuous testing and the existence of smoke and regression tests. While the ETAIS self-service portal has smoke tests, the overall test coverage of the software is currently low. Therefore the goal of this thesis is to plan and create the automatic regression tests of ETAIS self-service for continuous integration and increase the test coverage.

Full coverage is usually not considered necessary, as the balance of cost-benefit ratio is usually the best at 80% [31]. It is not suggested to push for high test coverage from the beginning, since it is more important to cover the most critical functionalities first than to try and test for every line of code. As mentioned before, creating automatic tests takes up a lot of time and resources so the goal is to allocate those resources incrementally, gather the results and make the following decisions based on those results. The growth of the percentage of test coverage depends on what can be achieved by the workload expected for a master's thesis.

As the time required to set up the testing environment and increase the test coverage by 1 percentage point is unknown at that point, the goal would have to be set by assessing the current coverage and significantly improving it. Before this thesis, 88 test cases have been added incrementally. Therefore it would stand to reason that doubling that amount in a single implementation would be a significant result and an acceptable goal to stop on to plan further steps.

3. Planning the tests

This chapter describes the process of planning the tests. It sets the scope, defines the requirements for the tests and contemplates the choice of software. The choices made here are based on the information collected throughout the previous chapter. The planning process ends with the creation of the test scenarios.

3.1. Scope

Since the goal is to create regression tests that the developers could run after implementing every update, the testing process should be fast and directed to the portion of the software most impacted by the changes. A significant portion of the development work at this point is being done on the front-end functionalities of the platform, which have the most obvious impact on the user experience since users interact with the platform through it. Therefore it would make sense to limit the scope of the tests to only the user interface stack. Mock data will be used for the back-end to decrease the runtime for the automated tests and to be in control of the testing conditions.

As the unit tests are best written by developers while creating the units and those should be tested before running the regression tests, an external tester can best work from the integration level forward. While there already is a substantial test cover for the unit tests, there are currently a limited number of functional tests created for the ETAIS self-service portal and no tests for non-functional quality aspects. Before focusing on higher levels of testing, system testing and acceptance testing can be done, it is reasonable to have substantial coverage on the integration level.

Having decided on the main metric of success for this project being test coverage, it is important to set a goal of the test coverage growth percentage. This can be decided based on multiple aspects of code – the amount of statements, lines, requirements, etc. covered. ETAIS self-service portal does not have requirements specified. During the research phase of this thesis, while documenting all the pages of the portal, all the functionalities of each page were documented as well. Since the functionalities on each page repeat throughout the portal and can be difficult to count without proper definitions and requirements, they are best grouped by the page they exist on, since each page also represents one core functionality that they are created to convey. Therefore, the requirement coverage could be replaced by page coverage, when making sure that all the functionalities on each page are covered.

The goal of this thesis was to create automated regression tests to improve the test coverage. Manually investigating the page architecture revealed 55 unique types of pages, which would mean that creating tests that cover the integrations on 6 unique previously uncovered pages would improve the page coverage by more than 10 percentage points, being significant enough to satisfy the goal. In order to find which pages to prioritise, it can be argued that the most critical features are the ones that are most often used. The developers of the ETAIS self-service portal have gathered data about page visits via Google Analytics that will be analysed and the most visited pages will be selected.

After creating tests for 6 untested pages, the number of new test cases can be compared with the number of tests created before this thesis. If the amount of total test cases has doubled, the goal has been reached. If not, tests will be created for interactions of the pages next in the list of most-visited pages, until the statement coverage meets the set goal.

3.2. Testing software

The integration tests already available for the Waldur platform have been created using Cypress testing software. Mobaraya and Ali claim in their article [32] that the industry standard for conducting automated tests on software's functionalities has been using Selenium, however, they consider Selenium to be somewhat outdated and to have some limitations because of that. Comparing Selenium with Cypress, they note that Cypress is a little bit faster in executing tests, and uses significantly fewer lines of code to achieve that result, making Cypress more efficient. Cypress does support fewer languages, browsers and frameworks [33] but since those limitations do not affect the testing process in this specific case, there is no reason to consider changing the testing environment and having to redo the tests created before this thesis.

Cypress is known for its ease of setup, since it combines a testing framework, assertion library, mocking and stubbing tools in one software, removing the need for additional dependencies [34]. According to its documentation [35], Cypress has multiple features useful for improving both the test writing and the testing process including the following:

- Debuggable snapshots of each step of the testing process;
- Automatic waiting for commands to be executed before starting the next command, removing issues stemming from asynchronisation;
- Automatic reloads of tests whenever a change in test code is saved;
- Waiting for an aliased process to finish instead of waiting for a fixed time;

- Ability to control the network traffic and intercept the server responses with mock data;
- Automatic screenshots and videos of entire test suites and failure points to avoid having to rerun the test to find the issues.

Even though Cypress was created for end-to-end testing, the features available make it a great tool for integration and regression testing as well. The tests created with Cypress can be used for continuous integration, as Cypress is compatible with all popular CI providers [36], including GitLab, the DevOps software used by the ETAIS development team.

3.3. Best practices for tests

The goal is to generate tests that could be used on other portals based on Waldur, without needing significant modifications. For that, it is important to make sure that the tests are not specific to only the ETAIS self-service instance by being dependent on the specific content, setup or local code variations.

Cypress's best practices guide [37] is also directed toward making the automatic tests more scalable. Their main instructions include the following suggestions for scalability:

- Making each test independent from others;
- Controlling the state of the environment and data used for content;
- Targeting testable elements via selectors that would not change throughout the development process;
- Avoiding interaction with third-party sites and servers that the tester has no control over;
- Avoiding repeating unit tests on the integration testing level;
- Waiting for an aliased process to finish instead of waiting for a fixed time.

While most of these suggestions can be implemented without an issue, there can be some difficulties in targeting the elements via selectors in the manner Cypress recommends. They instruct to avoid targeting elements based on HTML classes, id's or changing content. Instead, they suggest editing the code to add specific testing attributes. This change would make the selection process simple and foolproof but would require the tester to make changes to the portal's code, which contradicts the goal of creating tests that can be applied to the existing platform used in many similar instances all over the world. Therefore, this best practice can not be followed as suggested, and the identification of a specific element across

multiple instances of Waldur-based portals has to be ensured in a more complicated manner. Depending on the situation, the next best options are to select HTML elements by unique content, structure or class that does not change over the different platforms.

3.4. Selecting test suites

Test suites are a collection of related test cases with a similar goal, whether it is testing connected features, functionalities on the same page or similar quality aspects. Morgan, Samaroo, Thompson and Williams [22] define a test case as a set of preconditions, inputs, actions and expected results, based on the test conditions. This means that a test case consists of the description of conditions under which the test is run, the steps that are taken in order to reach the end result and the expectation of what the result should be. If the expectation matches the outcome, the test is considered to be a success. If the outcome is different from the expected result, the test has failed.

For creating the test suites, the most-visited unique web pages were selected from data gathered via Google Analytics. If there were multiple instances with different content of the same page type, only the first instance was considered.

All functionalities of the page were defined as test cases, sometimes combining multiple similar functionalities into one case to avoid repeating unnecessary steps and code lines. If a functionality repeats multiple times throughout the pages, it may be tested fewer times to keep the runtime of tests lower. Some features formed test cases that were similar to each other but different from others in the same suite. Those were planned into a separate test suite, to optimise the resources needed to run them and to keep the individual test suites shorter and cleaner.

The test suites and the related test cases are described in the table below (Table 1). The first column describes the popularity ranking of the page. The second column lists the page group, the name of the page and under those, the name of the test suite. The last column lists the test cases in each test suite. Each test case consists of the expected result and the steps needed to take to reach it. The expected outcome condition is listed as the name for the test case. Some of the test cases were created for the Waldur platform before this thesis, those are marked as "pre-existing".

#	Page group / Page / *Test suite*	Test cases
1	Public / Login / *Login page functionalities*	● Should accept cookies ● Should open Privacy Policy page ● Should open Terms of Service page ● Should log in via username-password ● Should display error with incorrect username-password
2	Users / Personal dashboard / *User dashboard functionalities*	● Should render title ● Should render user event count and graph ● Should report a security incident ● Should not report security issue without selecting the type of incident ● Should open Marketplace category ● Should open Organisation dashboard ● Should open Project dashboard ● Should export Projects as an *Excel* file ● Should open Event types modal ● Should expand Audit Log item and read details ● Should export Audit Logs as a *PDF* file ● Should navigate Audit Logs via pagination ● Should display Audit Logs search results
3	Users / SSH keys / *SSH Keys functionalities*	● Should render the title and key list ● Should expand items and copy the key to clipboard ● Should export the list as a *CSV* file ● Should not submit null key input ● Should display error messages for invalid key input (preexisting) ● Should display error messages for existing key input (preexisting) ● Should add an SSH key with appropriate inputs (preexisting) ● Should be able to delete an SSH key (preexisting)
4	Projects / Resources / *Marketplace category page functionalities*	● Should render the title and resource list ● Should expand the resource and render details ● Should import resource ● Should open resource item page

	Projects / Resources / *Resource actions*	• Should edit resource • Should move resource • Should synchronise resource • Should stop resource • Should restart resource • Should update resource security groups • Should update resource floating IPs • Should destroy resource • Should unlink resource
5	Projects / Resources / Resource item details / Security groups / *Resource item page functionalities*	• Should render the title and overview • Should open offering details modal and render the content of it and its tabs • Should open plan details modal and render the content • Should open each tab and render contents • Should create security group • Should synchronise groups • Should open security group details modal and render the content • Should open security group detail view
	Projects / Resources / Resource item details / Security groups / *Resource item actions*	• Should edit resource item • Should request direct access to resource item • Should synchronise resource item • Should change resource item plan • Should change resource item limits • Should terminate resource item • Should unlink resource item
	Projects / Resources / Resource item details / Security groups / *Security group actions*	• Should edit security group • Should set rules to security group • Should destroy security group • Should synchronise security group • Should unlink security group
6	Organisations / Organisation dashboard / *Organisation dashboard functionalities*	• Should render title • Should render monthly cost count and graph • Should render team size count and graph • Should add project • Should render invitation modal

		• Should report an issue • Should expand resource items and render content • Should open plan details modal and render its contents • Should open attributes modal and render its contents • Should open resource category page • Should open resource item detail page

Table 1. The test suites and test cases planned, based on the most visited pages of the ETAIS self-service portal.

With each user permissions type, there are functionalities on the navigation sidebar that are present on all the pages related to those permissions. There are also functionalities on the header and footer of each page. Tests for changing the active permissions set have already been created as a full test suite and will not need any new test cases. Separate test suites were formed in order to test the remaining multi-page functionalities without relating them to any specific pages (Table 2). Very similar test cases are combined into one bullet point and not written out separately to keep the table easier to read.

Page group / *Test suite*	Test cases
Users / *User navigation*	• Should go to each page (4 new, 4 pre-existing test cases) • Should go to the dashboard via logo • Should validate navigation via dropdown • Should validate logout function via dropdown
Projects / *Project navigation*	• Should go to each page (7 test cases) • Should go back to organisation • Should validate navigation to profile pages via dropdown
Organisations / *Organisation navigation*	• Should go to each page (11 test cases) • Should validate navigation via dropdown
All internal / *Header and footer navigation*	• Should go to support • Should change language • Should go to Compare items page • Should go to Checkout page

	• Should not display Compare and Checkout without rights • Should display version info • Should display back-end health info • Should open Privacy Policy page • Should open Terms of Service page • Should log out

Table 2. The test suites and test cases planned, based on the functionalities of different permissions.

For the context of this thesis, 13 test suites were planned for the automated regression tests. These test suites contain a total of 104 new test cases. This test plan covers 10% of the most visited web pages, as well as functionalities related to the permissions necessary to access them. The number of test cases will increase by more than twofold, from 88 to 192. The changes are visualised in the figure below (Figure 7).

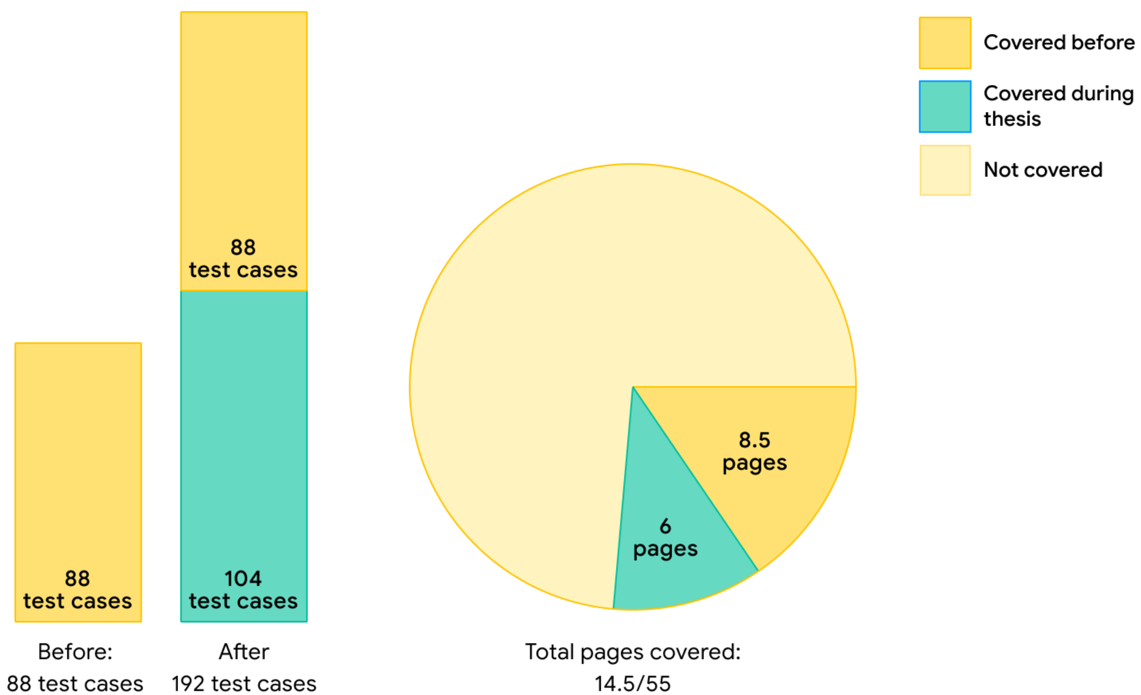

Figure 7. The improvement of the page coverage percentage and number of cases.

It should be noted that previously the tests were not created by following any documented order of importance. As such, the new tests cover situations that the users might experience more often. Some functionalities covered will repeat throughout the platform, decreasing the need to repeat those test cases for those pages.

3.5. Selecting the testing data

Most of the pages selected for testing are dynamic, meaning that they get their data from structured data sets. This can result in not only varying amounts of content visible depending on the data accessed, but also different sections and functionalities might be available to the user accessing the page. Because of that, it is very important to select the data used to populate the pages to be useful for the purposes of the tests. Poorly selected test scenarios can cause some test cases to not be properly tested, damaging the quality of the software.

According to software tester Thomas Hamilton [38], there are multiple ways to generate test data. It can be done either manually, by copying data from other environments, copying data from legacy client systems or using automated test data generation tools. Since generating test data manually or even by using automated tools is very time-consuming, it can be more cost-beneficial to use data from other environments of legacy systems.

Therefore, for the purposes of this thesis, it was decided to use the most representative data set possible and copy the test data from the production environment of the ETAIS self-service portal. To avoid leaking any private information, the data was anonymised. The resulting mock data was later manually modified during the test writing process when needed to adjust for covering more test cases.

The provided data mostly supports the so-called “happy path” testing strategy. Vance [39] describes the happy path as the use case that achieves the main purpose of the functionalities of the software. It is the foundation that has to be achieved, as alternative tests are built upon the context established by the happy path testing. Happy path tests can act as documentation and capture regressions. The alternatives to testing data following the optimal path include, according to Bernstein [40], testing for null values, incorrect values and various validation errors. While these take into account the possibility of human error happening in the data, they make the scope of the tests a lot bigger and test for rare occurrences. They should be added when aiming for maximum test coverage, but they are not the first priority, which is why they are out of scope for this thesis.

4. Results

This chapter describes the results of the testing process. It begins with describing the structure of the testing environment and the tests created. After that, the reasoning behind the selection of the datasets is outlined. The chapter concludes with the description of running the tests and integrating them into the development process.

4.1. Structure of the testing environment

There already exist some Cypress tests for the Waldur platform, that are organised in accordance with the Cypress guide about writing and organising tests [41]. Due to the complexity of the platform, the suggested structure has some extra layers to better categorise and organise tests and the testing data. The test and data files are combined into folders based on the structure and functionalities of the modules they are testing. The architecture of the Cypress testing folder is as follows:

1. Fixtures (directory for data files with mock data to fill the content of the testable pages);
   1.1. Customers (mock data files related to organisations);
   1.2. Dashboard (mock data files of content to fill the dashboard);
   1.3. Group invitations (mock data files of content for the group invitations module);
   1.4. Invitations (mock data files of content for the invitation sending module);
   1.5. Marketplace (mock data files of content to fill the marketplace pages);
   1.6. Offerings (mock data files related to offerings);
   1.7. OpenStack (mock data files related to resources);
   1.8. Projects (mock data files related to projects);
   1.9. Support (mock data files of content to fill the support pages);
   1.10. Users (mock data files related to users);
2. Integration (directory for test files);
   2.1. Common (tests for functionalities that are shared between all users, such as link persistence, content on header and footer, login pages);
   2.2. Customer (tests for functionalities of managing the organisations);
   2.3. Group invitations (tests for group invitation module functionalities);

2.4. Invitations (tests for invitation sending functionalities);

2.5. Marketplace (tests for functionalities that are related to using the marketplace);

2.6. OpenStack (directory for test files of different types of resources);

2.6.1. Instance (tests for functionalities of virtual machine type resources);

2.6.2. Tenant (tests for functionalities of private cloud type resources);

2.6.3. Volume (tests for functionalities of storage type resources);

2.7. Project (tests for functionalities of managing the projects);

2.8. Support (tests for functionalities related to support features);

2.9. User (tests for functionalities of managing the user profile);

3. Plugins (directory for plugin files, to add features to Cypress);
4. Support (directory for files that include commands that are used often throughout the tests).

This architecture has been created and added to by developers who created tests for the platform before this thesis. There are hierarchical decisions that could be done differently, such as separating fixtures and testing directories for invitations and group invitations, but in order to make the integration of new tests easier, it is reasonable to keep using the same structure and adding to it without editing the positioning of the existing directories. Each of the mock data files and the tests added during this thesis could be logically placed into existing directories, causing no reason to add any new folders.

4.2. Overview of the created tests

The tests were created according to the test plan and the results (code for tests, mock data used, files created during the process) are available in the GitHub repository at the following link: github.com/6unli/waldur-homeport/tree/thesis/cypress. The link directs to the branch of Waldur platform source code, which has files edited or added during the thesis by the author. It is possible to see the lines of code added or modified by the author by clicking on the commit name *add Cypress tests for most visited pages*[5].

The test suites can be found in the *integration* directory, where each suite is contained in a TypeScript file. Following a common convention, the files have the suffix of **.spec.ts* to differentiate the testing files from the source files that have the suffix of **.ts*.

[5] github.com/6unli/waldur-homeport/commit/ec25e1db7aab37f83165f27d642e709ab6d471ff

The suites begin with *describe()* function that encompasses the following functions and shortly describes the general context of all the test cases in this suite. It can be followed by definitions of constants used throughout the suite. The next method is usually *beforeEach()*, which is run before each test case and is mostly used to set up the testing context, choose parameters under which the tests are conducted and define which mock data files to use in case of specific HTTP method requests. An example of the structure of a test suite can be seen below (Figure 8).

```
describe( title: 'Header and footer', fn: function () {
  beforeEach( fn: () => {
    cy.mockUser( userName: 'admin').setToken() Chainable<any>
      .intercept( method: 'GET', url: '/api/customers/*', {fixture: 'customers/andersen.json'...})
      .intercept( method: 'GET', url: '/api/projects/**', {fixture: 'projects/project-2.json'...})
      .visit( url: 'projects/6628f38de458475baad6aa146f2ab406/') Chainable<AUTWindow>
      .get( selector: '.loading-title').should( chainer: 'not.exist') Chainable<Subject>
      .waitForSpinner();
  });

  it( title: 'Should go to support', fn: () => {...});

  it( title: 'Should change language', fn: () => {...});

  it( title: 'Should go to Compare items page', fn: () => {...});

  it( title: 'Should go to Checkout page', fn: () => {...});

  it( title: 'Should not display Compare and Checkout for User permissions', fn: () => {...});

  it( title: 'Should display version info', fn: () => {...});

  it( title: 'Should display backend health info', fn: () => {...});

  it( title: 'Should log out', fn: () => {...});
});
```

Figure 8. The structure of a test suite to validate the functionalities of the header and footer.

The main body of test suites consists of test cases, defined inside the function *it()*. The first argument is the name of the test case, which the author filled with the description of the expected outcome so that the function and test case name read as a sentence (e.g. *"It should export the list as .csv"*). The second argument consists of the steps taken to reach the end result. Then the expected outcome is stated in an assertion. Examples of three test cases in one test suite are presented below (Figure 9).

```
it( title: 'Should export Projects as an Excel file', fn: () => {
  cy.get('h5') Chainable<JQuery<HTMLHeadingElement>>
    .contains( content: 'Managed projects').parent().parent() Chainable<JQuery<HTMLElement>>
    .find( selector: '.btn-group > .dropdown') Chainable<JQuery<HTMLElement>>
    .contains( content: 'Export as') Chainable<JQuery<HTMLElement>>
    .click() Chainable<JQuery<HTMLElement>>
    .get( selector: '.open > .dropdown-menu > li > a') Chainable<JQuery<HTMLElement>>
    .contains( content: 'Excel') Chainable<JQuery<HTMLElement>>
    .click();
});

it( title: 'Should open Event types modal', fn: () => {
  cy.get('.table-buttons').contains( content: 'Event types').click();
  cy.get('.modal-title').contains( content: 'Event types').should( chainer: 'exist');
  cy.get('button').contains( content: 'Cancel').click();
});

it( title: 'Should expand Audit Log item and read details', fn: () => {
  cy.get('h5').contains( content: 'Audit logs').should( chainer: 'exist').waitForSpinner();
  cy.fixture('dashboard/events.json').then( fn: (events) => {
    const message = events[0].message;
    const eventType = events[0].event_type;
    const userName = events[0].context.user_full_name;
    const ip = events[0].context.ip_address;
    cy.get('.dataTable').contains(message).prev().click();
    cy.get('.event-details-table') Chainable<JQuery<HTMLElement>>
      .should( chainer: 'contain', userName) Chainable<Subject>
      .and( chainer: 'contain', ip) Chainable<Subject>
      .and( chainer: 'contain', eventType);
  });
});
```

Figure 9. The structure of three test cases validating the functionalities available on the Personal dashboard page.

The end result is compared with the state described in the assertion and if the outputs are the same, the tests pass. If the outputs are different, the comparison is tried again until a predefined timeout period (4000ms by default) ends and the outputs are either the same or the test case fails. A test case can contain multiple assertions if needed. The test case also fails if a step can not be executed as described and the retries do not succeed during the timeout period.

The test cases are run in order of appearance, each running individually from the other cases, so that the failures of previous tests do not affect the process or results of the following test cases.

4.2.1. Discovered errors

During the process of writing automated tests, 5 errors were found and reported as issues to the support of the platform. The errors were found by manually interacting with the portal because getting the expected outcomes required manual interaction with the live environment due to a lack of specifications. The test writing itself is a manual process and therefore the errors appeared before the automation was completed. The issues reported were:

- The visibility of notifications in the header in scenarios where they should be hidden;
- Unintended information displayed instead of the 404 page in the cases of incorrect identifiers in URL links;
- Incorrect HTTP request path;
- Incorrect heading and title of a page;
- Unreasonably long loading times for views related to specific organisations.

Even though the issues were reported to the support team, it is important to note that an error that has manifested once has the odds of happening again in the future. To make sure that the fixed problems are noticed when they reappear, it is good to check whether the automatic tests would catch these types of errors. It is considered part of regression testing. In the tests created, the automatic tests include checking for excess information visible in the header, correctness of HTTP request paths and contents of titles and headings. Concerning the other errors, Cypress automatically fails tests when they time out after a set period of time but the 404 page was out of scope for the tests created, as it was not one of the most-visited pages.

4.3. Running the tests

To run the test, it is possible to follow the commands located in *waldur-homeport* directory file *.gitlab-ci.yml* under the “Run E2E tests” section or the guide added to the thesis at hand as Appendix I Testing codes. The tests can be run with or without the graphic user interface, depending on what commands are used to run the program from the terminal. As the user interface adds a lot of useful information, the tests were run with the UI most of the time during this thesis.

When opening the Cypress user interface, a list of all the available tests is presented, grouped by their directories, as seen in Figure 10. They can be run individually or all together in a web browser selected by the user. The browsers available are all compatible browsers on the user’s machine and Electron (Chromium-based browser that runs in headless mode).

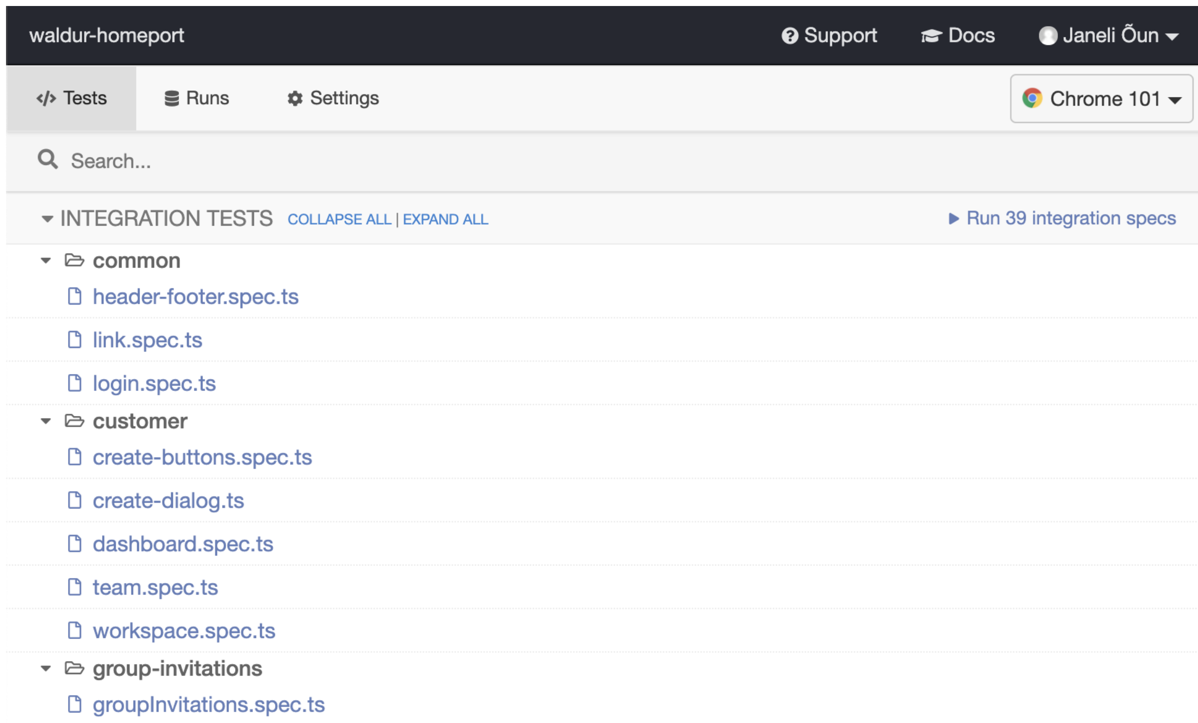

Figure 10. Landing view of Cypress user interface.

When the tests are run in the graphical user interface, the right side of the UI consists of the interface of testable software (Figure 11). The viewport of the interface can be defined in the code for the individual tests or in the global configuration file. The left side of the interface displays the test cases selected for this test run. The cases are marked as successful or unsuccessful as they are being completed. Above the list is the total number of tests succeeded, tests failed, time spent on the tests and a button for rerunning all the tests listed.

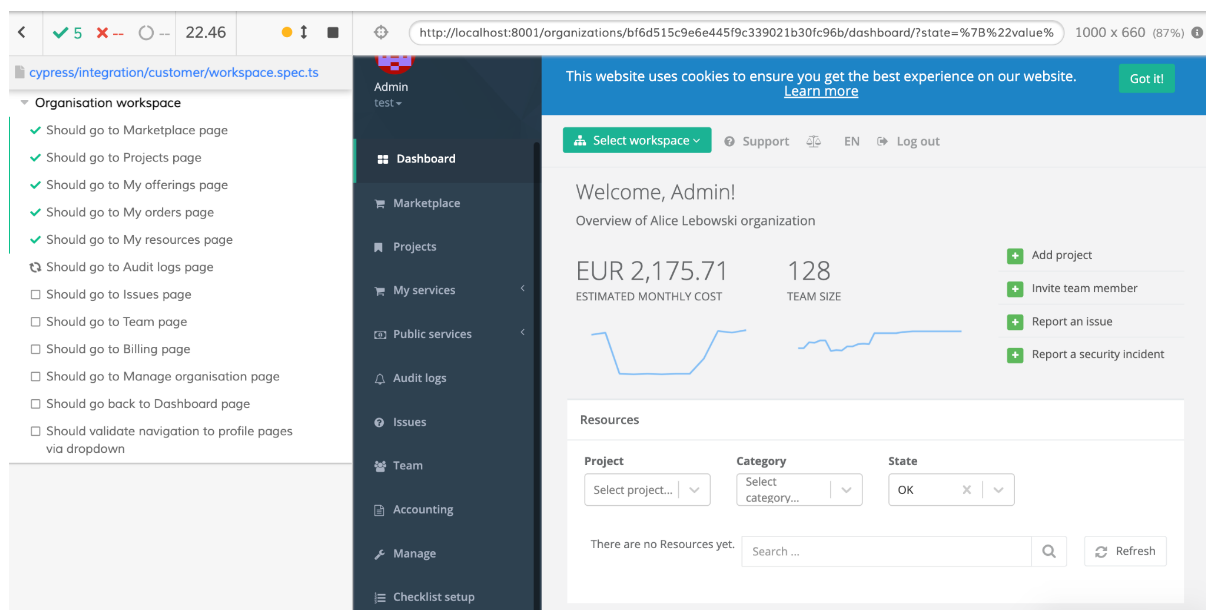

Figure 11. Test suite running in Cypress.

Each test case can be expanded to see detailed information about HTTP methods called and steps completed before, during and after the test cases. The steps can be individually selected to be visualised on the right side of the interface. When an error occurs, the reason for failure, relevant information and the test code for the failing step are outlined to improve the debugging process, visible in Figure 12.

Figure 12. Example of the information displayed in case of error during the testing.

To see the visualisation of test runs without having to clone the repository and set up the environment, a *videos* folder is included in the directory with the video files automatically generated by Cypress during the test runs. Also, a *screenshots* directory is available, which includes the screenshots made automatically in cases of test failures. Furthermore, there is a directory *downloads* that contains the files downloaded during the automated testing process.

4.4. Integrating the tests

The created tests should be run as often as possible. According to continuous integration standards, the tests should be integrated into the codebase to be executed automatically every time the base code is updated. ETAIS development team uses GitLab DevOps software to manage their codebase, which is where the tests would be integrated. The visualisation of the automatic tests running in GitLab can be seen below in Figure 13.

```
743   ├──────────────────────────────────────────────────────────────────────────────┤
744   │ ✔  user/manage.spec.ts                  00:02      1      1      -      -      - │
745   ├──────────────────────────────────────────────────────────────────────────────┤
746   │ ✔  user/ssh-keys.spec.ts                00:16      4      4      -      -      - │
747   ├──────────────────────────────────────────────────────────────────────────────┤
748   │ ✔  user/workspace.spec.ts               00:03      4      4      -      -      - │
749   ├──────────────────────────────────────────────────────────────────────────────┤
750   │ ✔  workspace_selector.spec.ts           00:05      5      5      -      -      - │
751   └──────────────────────────────────────────────────────────────────────────────┘
752     ✔  All specs passed!                    02:47     88     87      -      1      -
753 INFO: Gracefully shutting down. Please wait...
754 Done in 488.02s.
756 Saving cache for successful job
757 Not uploading cache develop due to policy
```

Figure 13. Visualisation of finished test run as a part of continuous integration.

To ease the integration of the new automatic tests into the current codebase, the author made sure that the tests are created in accordance with Cypress's best practices [37]. The created code itself was tested with linter, which is configured to automatically check whether the syntax follows the standards.

5. Discussion

The growing member count of the ETAIS self-service portal has increased the importance of providing a good user experience to all the users by improving the stability of the user interface. In order to find the potential errors before they reach the users, it was needed to improve the test coverage of the portal.

In order to do that, it was first necessary to understand and document the architecture and the functionalities provided by the self-service portal at hand. There was no documentation of front-end requirements available. These requirements are usually the basis on which the test cases are created. Having to find the functionalities to cover with automatic tests by interacting with the live environment, made this thesis more laborious.

While the features found on each of the pages were outlined in the related documentation, they were not formed into requirements that could be used as guides to design and develop the software. The tests created up to this point could be used as a kind of documentation of specifications. If ever needed to edit or add to the existing source code, the tests can be used to check for the fulfilment of all the requirements. Basing the development of new code on tests created beforehand follows the logic of Test-Driven Development. While it is a method proven to be useful in some cases, it is important to be aware of the limitations of having to rely on tests for requirements.

The author believes that the main concern would be that the time needed to create and edit the tests compared to the requirements can make any editing process slower and less agile. Another problem is that it may move the focus from getting all the requirements tested to solving technical issues and therefore leaving blind spots not covered by either. Hence, the author would suggest continuing with the creation of the tests, but also documenting the requirements in a quick but comprehensive way to have a solid basis for creating both software and tests.

The open-source platform Waldur, on which the self-service is based, is used in multiple international projects. Previously, for the front-end part of Waldur, there existed unit tests covering about ~65% and some integration tests, both were run for every change in the deployment. During this thesis, a goal was set to increase the amount of integration test cases twofold, aiming to extend the number of pages covered by automated regression tests.

From the most-visited pages of the ETAIS self-service portal, the first 10% of unique page templates were selected to be covered with automatic integration tests. The functionalities found on these pages were described as test suites of test cases to be executed as automatic

tests. To encompass all the functionalities of the selected pages, the navigation menus of each permission type, header and footer were also covered with tests.

The major decisions made by the author were related to the scope of the tests created during the thesis. The test types were limited to creating automated regression tests on the integration level for the purpose of using them in the continuous integration process. The tests were created for only the UI layer of the system. The decisions done were based on the evaluation of balancing the resources used with the potential benefits gained. The lack of integration-level tests was considered to be the main problem to be solved, as it can be reasonably done by an external tester and is necessary to complete to a substantial level before higher-level tests can be created. The user interface layer was chosen due to its stability issues having the most obvious impact on the visiting users' experience.

While there were some test cases created to test negative scenarios (cases where something should not be able to be done), the mock data used to fill the pages with content was mostly valid with no purposeful null or incorrect values added. This decision was made to cover a lot of regular scenarios that might happen more often than edge cases born out of users' human errors. Testing for these erroneous scenarios takes a lot of resources but has a low probability of actually occurring during user interactions. Therefore, while they are important to cover in the future to improve the user experience, they are not among the highest-priority tests that should be created before.

Another decision with a large impact was the choice of the framework chosen for creating the test cases. This decision was relatively easy for the author since the previous testing of the same type had been done on software called Cypress. While not considered to be the long-term industry standard, it has a few drawbacks and multiple benefits, making it a great tool for the current needs. It also helped the author to have a preset style for the architecture of the testing environment and the code to follow, as they had no previous experience in writing tests and could imitate the style and commands used in previous tests.

The decisions made aided the author in significantly increasing the quality and stability of the ETAIS self-service portal. The sustainability of the performed work is an important aspect that was considered throughout the test creation process. The completed process acts not only as a logical continuation of test extension for the ETAIS self-service portal and the whole Waldur platform but also as a basis for continuing the work. It was important to create tests that are as future-proof as possible since the integration test coverage should be extended further and higher levels of testing should be built upon the existing levels in the near future.

At first, the author wanted to use a different metric than the one finally chosen. As Cypress supports a plugin for code coverage, the author had hoped it can be used to automatically calculate code coverage by branches, lines, statements or functions. While the code coverage plugin itself worked when installed as instructed, another module was needed to make the feature work as intended. The module had to prepare code in a way that its lines can be counted when the tests activate them, in a process called instrumentalisation. But this plugin was not created to work with the React framework which is used as the basis for the ETAIS self-service front-end application and the lines, statements, etc. could not be counted. The author had to use a fallback metric, which was page coverage, which is not as accurate as code coverage.

The greatest challenges posed to the author stemmed from the lack of experience in automated testing and understanding how browsers execute network calls. The visual user interface and user-friendly debugging options of Cypress were beneficial in overcoming both of the issues. The author could draw information from studying the existing code, reading the well-documented guides for Cypress and interacting with the UI of the software. The testing proved to be an underestimated and interesting field. The knowledge and experience gained throughout this process will be useful in the new endeavours of the author.

6. Summary

ETAIS is a project that provides infrastructure and application support services to the Estonian research communities and R&D companies. The main goal of the thesis was to extend the automatic test coverage for its self-service portal. This included the information gathering, documentation, planning and execution of the automated tests. The thesis introduced and described the need for both ETAIS and its self-service portal. Furthermore, the basics and necessity of software testing were outlined with the processes that are considered to be best practices.

Based on that information, the scope for the tests created during this thesis was laid out and the basic testing plan was generated. Following that plan, automated integration level tests were created for the UI layer of the open-source platform Waldur, which is implemented as a self-service portal for ETAIS. The tests covered 10% of the most-viewed pages and functionalities related to those pages. The number of test cases was more than doubled, significantly improving the test coverage.

As the next possible steps, the integration test coverage at the user interface layer should be extended even further. The errors reported by users and found via the Sentry monitoring system should be added to be tested for by the automatic tests. The data used for testing could be diversified by adding extreme cases, errors and null values, to test how the system handles these situations. After improving the tests on the integration level, system and acceptance level tests can be planned and produced.

References

[1] Europa Science Ltd., “Supporting science with HPC,” *Scientific Computing World*, Sep. 13, 2021. https://www.scientific-computing.com/feature/supporting-science-hpc (accessed May 10, 2022).

[2] “NCC Estonia - EuroCC ACCESS,” *EuroCC Access*, Feb. 18, 2022. https://www.eurocc-access.eu/about-us/meet-the-nccs/ncc-estonia/ (accessed May 10, 2022).

[3] “About ETAIS,” ETAIS. etais.ee/about/ (accessed Apr. 18, 2022).

[4] “About ETAIS,” ETAIS. etais.ee/goals/ (accessed Apr. 18, 2022).

[5] “Cloud and HPC Management Platform,” Waldur Open-Source Cloud Marketplace. waldur.com/ (accessed Apr. 18, 2022).

[6] “ETAIS Self-Service Guide,” ETAIS. etais.ee/self_service/ (accessed Apr. 18, 2022).

[7] A. Agrawal and A. Choudhary, “Perspective: Materials informatics and big data: Realization of the ‘fourth paradigm’ of science in materials science,” APL Materials, vol. 4, no. 5, Apr. 2016, doi: 10.1063/1.4946894.

[8] “Cloud,” Oracle. www.oracle.com/cloud/hpc/what-is-hpc/ (accessed Apr. 18, 2022).

[9] “ETAIS resources,” ETAIS. etais.ee/resources/ (accessed Apr. 18, 2022).

[10] Estonian Research Council, “Estonian Scientific Computing Infrastructure,” Estonian Research Infrastructure Roadmap 2019, p. 42, 2019. [Online]. Available: www.etag.ee/wp-content/uploads/2019/06/ETAg_Research_Infrastructure_Roadmap_2019.pdf

[11] S. Zaiaev, “A Modern CI/CD Pipeline for Cloud Native Applications,” Master’s Thesis, UNIVERSITY OF TARTU Faculty of Science and Technology Institute of Computer Science, 2021. Accessed: May 16, 2022. [Online]. Available: https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=72359&year=2021

[12] NeIC, “Puhuri Documentation,” Puhuri. http://puhuri.neic.no (accessed May 16, 2022).

[13] G. Fraser and J. M. Rojas, *Handbook of Software Engineering*, Software Testing. Springer, 2019, pp. 123–192.

[14] W. E. Lewis, Software Testing and Continuous Quality Improvement, Third Edition. CRC Press, 2016.

[15] "ISO 25010," ISO 25000 Portal. iso25000.com/index.php/en/iso-25000-standards/iso-25010 (accessed Apr. 20, 2022).

[16] G. Mohan, Full Stack Testing: A Practical Guide for Delivering High Quality Software. O'Reilly Media, Inc., 2022.

[17] IEEE Computer Society, "IEEE Standard Classification for Software Anomalies," IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993), Jan. 2010, doi: 10.1109/ieeestd.2010.5399061.

[18] H. Krasner, "The Cost of Poor Quality Software in the US: A 2018 Report," Sep. 2018. Accessed: Apr. 20, 2022. [Online]. Available: www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-report/The-Cost-of-Poor-Quality-Software-in-the-US-2018-Report.pdf

[19] S. M. K. Quadri and S. U. Farooq, "Software Testing – Goals, Principles, and Limitations," *International Journal of Computer Applications*, vol. 6, no. 9, pp. 7–10, Sep. 2010, doi: 10.5120/1343-1448.

[20] L. S. Vailshery, "QA and testing budget allocation 2012-2019," Statista, Oct. 2019. www.statista.com/statistics/500641/worldwide-qa-budget-allocation-as-percent-it-spend/ (accessed Apr. 20, 2022).

[21] A. Spillner and T. Linz, Software Testing Foundations, 5th Edition: A Study Guide for the Certified Tester Exam. Rocky Nook, 2021.

[22] B. Hambling (editor), P. Morgan, A. Samaroo, G. Thompson, and P. Williams, Software Testing: An ISTQB-BCS Certified Tester Foundation Guide. BCS, The Chartered Institute for IT, 2019.

[23] A. Mathur, Foundations of Software Testing. Addison-Wesley Professional, 2014.

[24] A. Kramer and B. Legeard, Model-Based Testing Essentials - Guide to the ISTQB Certified Model-Based Tester: Foundation Level. John Wiley & Sons, 2016.

[25] K. F. Tómasdóttir, M. Aniche, and A. van Deursen, "Why and how JavaScript developers use linters," Oct. 2017. Accessed: May 16, 2022. [Online]. Available: http://dx.doi.org/10.1109/ase.2017.8115668

[26] JetBrains s.r.o., “Automated Testing for CI/CD,” JetBrains. www.jetbrains.com/teamcity/ci-cd-guide/automated-testing/ (accessed Apr. 22, 2022).

[27] E. Dustin, J. Rashka, and J. Paul, Automated Software Testing: Introduction, Management, and Performance. Addison-Wesley Professional, 1999.

[28] Guru99, “Software Testing Metrics: What is, Types & Example,” Mar. 07, 2020. https://www.guru99.com/software-testing-metrics-complete-tutorial.html < (accessed May 11, 2022).

[29] S. Bose, “Essential Metrics for the QA Process,” *BrowserStack*, Sep. 02, 2020. www.browserstack.com/guide/essential-qa-metrics (accessed Apr. 22, 2022).

[30] RTI International for G. Tassey, The Economic Impacts of Inadequate Infrastructure for Software Testing. National Institute of Standards and Technology, 2002.

[31] Atlassian, “Introduction to code coverage,” *Atlassian*. https://www.atlassian.com/continuous-delivery/software-testing/code-coverage (accessed May 11, 2022).

[32] F. Mobaraya and S. Ali, “Technical Analysis of Selenium and Cypress as Functional Automation Framework for Modern Web Application Testing,” Dec. 2019. Accessed: May 07, 2022. [Online]. Available: http://dx.doi.org/10.5121/csit.2019.91803

[33] E. Kinsbruner, “Cypress vs. Selenium: What’s the Right Cross-Browser Testing Solution for You?,” *Perfecto by Perforce*, Jan. 10, 2022. https://www.perfecto.io/blog/cypress-vs-selenium-whats-right-cross-browser-testing-solution-you (accessed May 07, 2022).

[34] “End to End Testing Framework,” *JavaScript End to End Testing Framework | cypress.io testing tools*. https://www.cypress.io/how-it-works/ (accessed May 07, 2022).

[35] “Why Cypress?,” *Cypress Documentation*. https://www.cypress.io/features (accessed May 07, 2022).

[36] “CI Provider Examples,” *Cypress Documentation*. https://docs.cypress.io/guides/continuous-integration/ci-provider-examples (accessed May 12, 2022).

[37] “Best Practices,” *Cypress Documentation*. https://docs.cypress.io/guides/references/best-practices (accessed May 06, 2022).

[38] T. Hamilton, “Test Data Generation: What is, How to, Example, Tools,” *Guru99*, Apr. 16, 2022. https://www.guru99.com/software-testing-test-data.html (accessed May 12, 2022).

[39] S. Vance, *Quality Code: Software Testing Principles, Practices, and Patterns*. Addison-Wesley, 2013.

[40] C. Bernstein, “happy path testing,” *TechTarget*, Aug. 09, 2019. [Online]. Available: https://www.techtarget.com/searchsoftwarequality/definition/happy-path-testing (accessed: May 15, 2022).

[41] “Writing and Organizing Tests,” *Cypress Documentation*. https://docs.cypress.io/guides/core-concepts/writing-and-organizing-tests#Folder-structure (accessed May 10, 2022).

Appendix

I Testing commands

To run the tests, it is first needed to set up the environment. First, if not already installed in the system before, it is needed to install Node.js and npm (Node Package Manager). Node.js is an open-source server environment that allows the user to run JavaScript outside a web browser. It can be installed from its official website[6] with npm already included. After the installation, npm can be used to install, update and uninstall Node.js packages from the command line.

The first package to install is Yarn dependency management[7]. Installing and updating Yarn can be done by running the following command:

```
npm install --global yarn
```

When Yarn is installed, the user can navigate to the local project, inside the *waldur-homeport* directory, to install the project's dependencies by simply entering:

```
yarn
```

The previous steps have to be run only once to set everything up. To run the server, the user can enter the command:

```
yarn ci:start
```

Once the server is running, run the tests with the visual user interface by entering:

```
yarn run cypress open
```

Alternatively, for running the tests without the visual user interface, there is no need for starting the server separately and instead of the last two commands, the following command can be run:

```
yarn ci:test
```

The testing interface can be closed by closing the Cypress UI window or by pressing `Ctrl+C` on the command line where it was opened.

[6] nodejs.dev/download/
[7] classic.yarnpkg.com/lang/en/docs/install/

II Licence

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Janeli Õun**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose **"Testing Estonian Scientific Computing Infrastructure Self-Service in Cypress Framework"**, mille juhendajad on **Ilja Livenson, MSc** ja **Viktor Mirieiev, MSc**, reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.

2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 3.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.

3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.

4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Janeli Õun

17.05.2022