TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Nikolai Ogonkov 206486IABB

Charon Moodle plugin end-to-end test automation

Bachelor's thesis

Supervisor: Bahdan Yanovich
BSc

Tallinn 2024

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Nikolai Ogonkov 206486IABB

Charon Moodle pistikprogrammi läbivtestimise automatiseerimine

Bakalaureusetöö

Juhendaja: Bahdan Yanovich
BSc

Tallinn 2024

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Nikolai Ogonkov

20.05.2024

Abstract

"Moodle is a Learning Management System (LMS) designed to provide educators, administrators and learners with a single robust, secure and integrated system to create personalised learning environments" [1]. Moodle has built-in support for custom plugins, which allows for the integration of supplementary features to meet the requirements of the end user.

Testing is crucial for ensuring service continuity and quality in a production environment, it is one of the most important phases of the product release cycle. Test automation facilitates seamless handling of the testing stage of the product release cycle by not requiring a human presence to test and report on failed and passed tests.

The goal of this paper is to explain how end-to-end (E2E) testing and test automation for the Charon Moodle plugin were conducted. This paper also goes into detail why certain techniques, tools were used and describes the procedures employed for mitigating test flakiness.

This thesis is written in English and is 37 pages long, including five chapters, 13 figures.

Annotatsioon

Charon Moodle pistikprogrammi läbivtestimise automatiseerimine

“Moodle on õpihaldussüsteem, mis on loodud selleks, et pakkuda haridustöötajatele, administraatoritele ja õppijatele ühte kindlat, turvalist ja integreeritud süsteemi personaalsete õpikeskkondade loomiseks” [1]. Moodle'il on sisseehitatud tugi kohandatud lisandmoodulitele, mis võimaldab integreerida täiendavaid funktsioone, et vastata lõppkasutaja vajadustele.

Testimine on oluline, et tagada teenuse järjepidevus ja kvaliteet tootmiskeskkonnas, see on üks tähtsamaid etappe toote väljalasketsüklis. Testi automatiseerimine hõlbustab toote väljalasketsükli testimise etapi tõrgeteta käsitlemist, kuna ei nõua inimese kohalolekut testimiseks ja ebaõnnestunud ja läbitud testide kohta aruannete koostamiseks.

Läbivtestimise eesmärk on jäljendada inimese suhtlemist tarkvaralahendusega, mis võimaldab testimist lõppkasutaja vaatenurgast. Lisaks aitab läbivtestimine saavutada suuremat testide katvust, võimaldades testida tarkvara funktsioone, mille puhul ei ole ühik- ja integratsioonitestide loomine võimalik.

Käesoleva töö eesmärk on selgitada, kuidas Charon Moodle'i pistikprogrammi läbivtestimine ja testide automatiseerimine viidi läbi. Selles dokumendis käsitletakse ka üksikasjalikult, miks kasutati teatavaid tehnikaid ja vahendeid ning kirjeldatakse menetlusi, mida kasutati *flaky* testide vältimiseks.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 37 leheküljel, viis peatükki, 13 joonist.

List of abbreviations and terms

E2E	End-to-end.
CI/CD	Continuous integration and continuous delivery or deployment.
TalTech	Tallinn University of Technology.
E2E testing environment	An environment in which E2E tests may be executed.

Table of contents

1 Introduction 10
1.1 Problem description 10
1.2 Proposed solution 11
1.3 Goals and expected results 11
1.4 Thesis structure 12
2 Methodology 13
2.1 Background 13
2.1.1 Moodle 13
2.1.2 Charon Moodle plugin 13
2.1.3 Moodle Docker implementation 14
2.1.4 GitLab CI/CD 14
2.1.5 GitLab runners 14
2.1.6 Docker images, containers, and networks 14
2.2 Object description 15
2.3 Framework and language selection 16
2.3.1 Playwright 16
2.3.2 TypeScript 16
2.4 Repositories 17
2.5 Development process description 17
2.6 Test flakiness mitigation strategy 17
3 Solution 19
3.1 Solution requirements 19
3.2 Test case selection 20
3.3 E2E testing project architecture and test creation 21
3.3.1 MariaDB API 21
3.3.2 GitLab API 22
3.3.3 Global variables, enums, interfaces and functions 23
3.3.4 Test implementation 24

3.3.5 Test example 24
3.3.6 Playwright configuration and environment variables 26
3.4 Deployment of E2E tests 30
3.5 E2E testing environment for development tests 31
3.5.1 Creation of custom docker-compose files 31
3.5.2 Script for assembling the E2E testing environment for development tests 33
3.5.3 Script for disposing previously generated Docker elements 37
3.5.4 “charon-initialization.sh” script execution as part of the CI/CD pipeline 37
3.6 E2E test execution 38
3.6.1 Script for executing E2E tests 38
3.6.2 “test-initialization.sh” script execution and result collection 39
4 Analysis of the solution 41
4.1 Compliance with the solution requirements 41
4.2 Summary of the final solution 42
4.2.1 E2E testing project 42
4.2.2 Integration of the E2E testing project into the CI/CD pipeline 43
4.3 Solution testing 44
4.4 Encountered problems 45
4.5 Areas of potential improvement 45
5 Summary 47
Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis 49
Appendix 2 – E2E testing project structure 50
Appendix 3 – “niogon/E2E-tests” CI/CD pipeline’s “build_image” job 51
Appendix 4 – “ained/charon” CI/CD pipeline’s “build_moodle”, “test_moodle” jobs 52
Appendix 5 – “ained/charon” CI/CD pipeline’s “test_prod _moodle” job 54

List of figures

Figure 1. "Tester settings functionality" test. 26
Figure 2. playwright.config.js file. 28
Figure 3. .env file. 29
Figure 4. "build_image" job of the "niogon/E2E-tests" CI/CD pipeline. 30
Figure 5. Dockerfile for building the "niogon/e2e-tests" image. 30
Figure 6. Initial docker-compose file. 32
Figure 7. First part of the "charon-initialization.sh" script. 34
Figure 8. Second part of the "charon-initialization.sh" script. 36
Figure 9. "build_moodle" job of the "ained/charon" CI/CD pipeline. 38
Figure 10. "test-initialization.sh" script. 39
Figure 11. "test_moodle" job of the "ained/charon" CI/CD pipeline. 40
Figure 12. Results for development environment testing within the CI/CD pipeline. ... 44
Figure 13. Results for production environment testing on local machine. 44

1 Introduction

This chapter provides an overview of the problem at hand, the proposed solution, the goals and expected results, and the structure of this thesis.

1.1 Problem description

Charon, a Moodle plugin developed by Tallinn University of Technology's (TalTech's) Department of Software Science, builds on existing Moodle features and plugins, allowing teachers to create a more engaging learning environment for students.

As Charon has evolved, an increasing array of features has been integrated, significantly expanding the scope of website functionalities requiring testing prior to release. Testing plays a crucial role in product delivery, as it contributes to ensuring end user satisfaction. A prevalent practice among many software development teams is to write unit and integration tests for newly developed code.

Given the extensive functionalities provided by the Charon plugin, writing unit and integration tests for plugin features or components has become exceptionally challenging. Frequently, website components are tightly interconnected, making it difficult to decouple them for testing purposes, as one component may rely heavily on another to function properly. Decoupling the components would necessitate a significant allocation of resources to refactor a large portion of the Charon plugin codebase, making it impractical to pursue. Consequently, developers are compelled to manually test certain website components before deploying new versions by interacting directly with the website.

The Charon development team lacks a defined set of rules and best practices for manually testing the solution in the staging environment. This absence increases the likelihood of bugs appearing in the production environment, particularly as certain critical website components may remain untested before deployment. The issue is further exacerbated by

the fact that the Charon plugin is largely developed by students, leading to constant changes in the development team. This dynamic environment can result in business-critical functionality being underemphasized during the transfer of knowledge.

Bugs in the production environment typically have a significant impact on the end user experience, as a separate team is responsible for solution deployment and production version control. Consequently, even if the Charon development team implements a fix in a timely manner, a discovered bug may remain unaddressed for extended periods of time.

Certain Charon plugin features require testing in the production environment due to technical constraints. Consequently, some tests must be executed promptly after deploying a new version, typically during off-peak hours, such as at night. During these times, human testers are more likely to be fatigued, increasing the risk of errors.

1.2 Proposed solution

Developing E2E tests for critical website functionalities that cannot be tested through unit or integration testing will enhance the overall test coverage of the application.

Automating Charon plugin E2E testing is a great way to alleviate developers from the burden of manually setting up a testing environment and initiating test execution for their software solution. Test automation is a widespread practice adopted by many companies and small teams as an integral component of their CI/CD pipeline.

Writing E2E tests and automating test execution demands significant time and commitment, but its maintenance thereafter is straightforward and cost-effective, thereby allowing developers to dedicate more time to developing new features and enhancing the existing codebase.

1.3 Goals and expected results

The goals of this thesis were to create an E2E testing project and to configure CI/CD pipelines in a manner that enables E2E test execution and subsequent result reporting in GitLab.

The expected outcomes of this work were an E2E testing project and modified CI/CD pipelines. The E2E testing project needed to encompass a variety of tests that cover the

primary functionalities provided by the Charon plugin. It also had to include a framework of tools to streamline test creation and offer multiple configuration capabilities. In contrast, the modified CI/CD pipelines had to enable remote, automatic test execution and test result reporting.

1.4 Thesis structure

The thesis consists of three parts:

1) Methodology

This chapter provides a comprehensive explanation of essential concepts and details the planned development process, including procedures for reducing test flakiness. Furthermore, it outlines the establishment of solution requirements. Additionally, within this chapter, the author presents the rationale behind the selection of specific tools and frameworks to achieve the established objectives.

2) Solution

This chapter includes an overview of selected test cases. It also delineates the essential strategies and coding techniques employed in crafting the final solution, offering a thorough examination of its components.

3) Analysis of the solution

This chapter provides a comprehensive overview of the accomplishments achieved during the writing of this thesis and the structure of the E2E testing solution. Additionally, it presents the establishment of solution compliance with the specified requirements. At the conclusion of this chapter, the results from solution testing are provided, along with a detailed account of the challenges encountered by the author during the thesis writing process and areas identified for potential improvement.

2 Methodology

This chapter provides an overview of Moodle and the Charon Moodle plugin, along with details about the tools, frameworks, and languages utilized in creating the E2E testing project. Additionally, it outlines the development process and strategies implemented to mitigate E2E test flakiness.

2.1 Background

This section presents a detailed overview of Moodle and the Charon Moodle plugin, for which E2E tests and test automation were conducted. This section also provides detailed descriptions of the key tools, frameworks, and technologies utilized by the Charon development team, which facilitated the implementation of the E2E testing solution.

2.1.1 Moodle

Moodle, an open-source learning management system, stands as a favoured solution among many educational institutions and learning platforms [2]. “Moodle provides educators, administrators, and learners with a single robust, secure, and integrated system to create personalised learning environments” [1]. Key features of Moodle include progress tracking, peer and self-assessment, competency-based marking, advanced grading, multimedia integration, and numerous others. These features offer considerable utility in establishing a fully customized learning environment; however, they may not be suitable for all user requirements.

Plugins further the customization capacity of Moodle applications. Users may integrate plugins installed from the Moodle directory or create custom in-house plugins.

2.1.2 Charon Moodle plugin

Charon is a plugin developed by TalTech’s Department of Software Science. Charon incorporates many features developed over the years by students and teachers alike. These features are primarily written using PHP for backend logic, and JavaScript along with the Vue.js framework for frontend logic.

The Charon plugin serves as a crucial component of TalTech’s Moodle implementation, offering key features that enhance functionality. These include providing more detailed

performance overviews for course participants, enabling the submission of solutions to coding problems, facilitating the registration of labs, and various other critical features.

2.1.3 Moodle Docker implementation

A Moodle program can be launched as a multi-container Docker application, utilizing the Bitnami Docker images accessible via the Docker hub. The Dockerized version of Moodle consists of two containers: MariaDB and Moodle. The MariaDB container serves as a relational database for storing Moodle-specific data, while the Moodle container hosts web components and plugins required for launching the Moodle website [3]. Additionally, Charon development team utilizes the Adminer container alongside these base containers for database administration tasks.

2.1.4 GitLab CI/CD

CI/CD is an abbreviation for continuous integration and continuous deployment. At its core, CI/CD involves using a range of tools and methods to test code changes for bugs in the CI phase and then deploying those changes across different environments in the CD phase. In GitLab, CI/CD is facilitated via pipelines, which are composed of user-defined stages. Each stage comprises a series of customized jobs, executed by the GitLab runner to accomplish the tasks (building, testing, deploying, etc.) assigned to that stage [4].

2.1.5 GitLab runners

GitLab runners are applications used for executing GitLab jobs. GitLab runners have the capability to utilize multiple executors, each serving as distinct environments for the execution of jobs. GitLab offers developers the option to utilize GitLab-hosted runners, providing a convenient means of configuring a CI/CD pipeline. Additionally, there is the alternative of deploying self-hosted GitLab runners. Generally, self-hosted runners afford enhanced security, are more cost-effective compared to GitLab-hosted runners, and offer greater customization capabilities [5].

2.1.6 Docker images, containers, and networks

A **Docker image** can be viewed as an executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings [6]. An image can encapsulate a software component such as a database or statistics API, backend or frontend program layers or even whole programs. Docker

images can be generated using Dockerfiles, which are essentially text documents containing a series of instructions. Once a Docker image has been created, it can be instantiated.

Docker containers are instances of Docker images and share similarities with virtual machines. However, unlike virtual machines, Docker containers leverage the host machine’s operating system kernel, making them more lightweight. Another distinctive characteristic of Docker containers is their ability to run on various operating systems, differing from that of the host machine. This versatility is facilitated by Docker engine's utilization of hypervisor technology, which serves as an intermediary for communication between the host machine's kernel and the Docker container [7].

Docker networks provide the ability to isolate Docker containers. Networks can be created through the Docker client, and Docker containers can subsequently be incorporated into these networks. When added to a Docker network, each Docker container is assigned an IP address. Docker containers within the same Docker network can communicate with each other using container IP addresses, names, or even exposed ports [8].

2.2 Object description

In the framework of this thesis, the author has developed an E2E testing solution consisting of two principal components: the E2E testing project and its integration into the Charon GitLab repository's CI/CD pipeline.

The E2E testing project was predominantly built using the TypeScript programming language and the Playwright framework for E2E test development. Its design emphasizes scalability, maintainability, and configurability, as evidenced by the organizational structure featuring distinct files for tools, configurations, and tests. Separate test suites were created for both production and development environments.

Integration of the E2E testing project into the Charon GitLab repository's CI/CD pipeline was achieved through the utilization of custom shell scripts, Dockerfiles, docker-compose files, and modifications to the .gitlab-ci.yml files within both the Charon plugin's and E2E testing project's repositories.

2.3 Framework and language selection

The selection of suitable tools, languages, and frameworks is paramount for ensuring project maintainability and scalability. While highly subjective, the author had attempted to maintain objectivity throughout the decision-making process, striving to minimize the influence of personal experience on the outcome.

2.3.1 Playwright

Playwright is a framework for web testing and test automation that supports languages like C#, JavaScript, Python and Java. The framework is developed by Microsoft and its source code is openly available on GitHub, offering transparency and collaboration opportunities to its users.

The Playwright framework is a great choice for writing E2E tests because it works out of the box and has a lot of useful features that many of its competitors do not have. Some of Playwright's exceptional features, such as built-in mobile device emulation, support for testing Firefox, Chromium and Webkit-based browsers, auto-waiting implementation for many functions, distinguish it from similar frameworks like Cypress and Selenium [9].

2.3.2 TypeScript

TypeScript is a superset of JavaScript, which provides users the ability to write type safe JavaScript code. Type safety is vital for improving codebase maintainability and scalability.

Explicit typing provides important information pertaining to what kind of data is being altered and stored or returned by functions. This information can be utilized by the developer to better understand the overall code structure. TypeScript additionally performs type error checks both during compile-time and runtime, significantly enhancing the debugging process [10].

TypeScript and JavaScript combined have a large and active community of users. Both languages are well-documented and are used by many teams for client-side scripting, making these languages a popular choice among Playwright developers [11].

2.4 Repositories

The author utilized the `niogon/E2E-tests` and `ained/charon` GitLab repositories in the writing of this thesis. The `niogon/E2E-tests` repository served as the storage and maintenance platform for the E2E testing project.

All modifications necessary to integrate the E2E testing project into the `ained/charon` repository's CI/CD pipeline have been implemented in the `e2e-test-automation-2` branch. At the time of writing, development of the Charon plugin was ongoing, with continuous additions of new features. To focus the scope of this paper, the author had chosen to develop E2E tests specifically for version 1.8.5 of the plugin. The commit made on 23.01.2024, marking the release of version 1.8.5, served as the parent commit for the `e2e-test-automation-2` branch.

2.5 Development process description

The workflow primarily involved the writing of code, complemented by research, analysis, and miscellaneous tasks such as defining requirements and solution testing.

The author and supervisor reached a mutual agreement to hold weekly meetings aimed at reviewing the author's progress and outlining tasks for the ensuing week(s). This iterative and incremental approach facilitated the tailoring of the solution to align with the requirements of the Charon development team. Moreover, the feedback provided by the supervisor proved invaluable in identifying and rectifying issues pertaining to code style and logic. These scheduled sessions were conducted via Microsoft Teams and typically lasted for an hour.

2.6 Test flakiness mitigation strategy

Test flakiness can be defined as a characteristic of returning inconsistent and incorrect test results. Flaky tests can lead to false-positives or false-negatives arising within the testing cycle, potentially misreporting the state of the application. It is in the developer's best interests to avoid writing flaky tests and to mitigate their impact on the overall outcome of the testing cycle.

There are numerous factors that can contribute to E2E test flakiness:

- Popups or other notifications hindering click actions
- Faulty webpage element locators (selectors)
- Neglecting to ensure the complete loading of page elements
- Issues with testing logic [12]

Testing for flakiness is a good practice for determining whether testing logic requires to be altered and should always precede test deployment. At times, it may be prohibitively expensive or even technically unfeasible to rewrite tests to fully eliminate flakiness, particularly due to API or framework-related issues. In such cases, executing multiple iterations of the same test can substantially mitigate the effects of flakiness.

E2E testing frameworks utilize locators and locator abstractions for interacting with webpage elements. Locators can be defined as strategies for locating HTML elements. Locator complexity can span from specifying the unique value of the element's id attribute to describing the entire path of a tag in an HTML document tree. To mitigate flakiness when writing E2E tests, it is important to choose optimal strategies for selecting HTML elements. Short and precise strategies are preferable, with id attribute values providing the utmost precision in selecting webpage elements. It is important to avoid writing long and imprecise strategies. Imprecision can lead to multiple page elements getting selected, while long strategies are particularly susceptible to changes, leading to frequent failures and increased maintenance overhead [13].

The Playwright framework incorporates auto-waiting functionality within many of its functions designed for interacting with webpage elements. When attempting to interact with a webpage element, Playwright executes a series of checks to ensure consistent behaviour during test execution. Developers can specify a timeout period during which these checks are conducted. If the HTML element successfully passes all checks before the timeout expires, test execution continues. However, if any check fails, an error is triggered. The specific checks performed depend on the type of interaction being executed. While this framework feature is effective in reducing flakiness, it may not always suffice. To further minimize flakiness, developers can implement a custom timeout function to ensure consistent test results.

3 Solution

In the “Solution” chapter, the solution requirements are established, the overall development process of the E2E testing solution is described, and an overview of its key components is presented. This chapter also includes an example illustrating the process of E2E test creation.

3.1 Solution requirements

Solution requirements play a pivotal role in setting goals, ensuring that the development process remains focused and purposeful. Moreover, they provide the author with clear guidelines regarding code style and structure. Furthermore, these requirements can serve as reference points for analysing results, facilitating the assessment of the solution's success.

Before beginning the development process of the E2E testing solution, the author outlined the following solution requirements:

- The E2E testing solution needs to be versatile and configurable, necessitating the establishment of a foundation with scalability and future changes in mind. The E2E testing project will serve as this foundation, housing tests and tools/modules for test execution.

- To automate the testing process effectively, integrating the E2E testing solution into the Charon repository’s CI/CD pipeline is essential. This integration will ensure that E2E tests can be conducted against the most recent updates in the repository. Additionally, the E2E testing solution must have the capability to accommodate testing across multiple environments.

- The work presented in this paper must strictly adhere to clean code principles, thereby enabling future developers to efficiently expand upon the author's solution and maintain the codebase. Adhering to principles such as Don't Repeat Yourself (DRY) and Keep It Simple, Stupid (KISS) should help minimize logical interdependence.

- Additionally, test creation must conform to specific guidelines to ensure test maintainability and decrease flakiness. The procedures outlined in the “Test flakiness mitigation strategy” (see section 2.6) section of this paper effectively details the approach the author must employ in creating E2E tests.

- E2E tests must be organized into distinct groups, each associated with a specific testing environment. This will ensure that tests belonging to different environments are not executed concurrently.

3.2 Test case selection

The potential for writing thousands of E2E tests to cover every conceivable Moodle and Charon feature across various scenarios and environments exists. However, the time investment required for test creation rendered achieving comprehensive test coverage within the scope of this paper impractical due to time limitations. Consequently, careful test case selection became crucial to ensure that the most impactful areas of the website would receive sufficient coverage through E2E testing.

Upon thorough examination of the Charon plugin, the author and supervisor jointly determined that tests for both development and production environments need to be created. Specifically, the following functionalities were identified for testing across WebKit, Firefox, and Chromium browsers: login/logout, course creation, course settings, student overview, teacher overview, Charon settings, tester settings, solution submission via the built-in website editor, and solution submission via GitLab.

While login/logout and course creation functionalities are not exclusive to Charon, they play integral roles and enable the testing of other Charon features. Hence, they were included in the selection for testing.

It is important to highlight the significant differences between production and development environments, which greatly influence how testing is conducted.

In the production environment, strict security measures prevent the E2E testing project from accessing the Moodle database. This means that setting up E2E testing environments either requires manual intervention or must be integrated into the testing process.

Consequently, any test failures in a production E2E testing environment may necessitate manual intervention for resolution.

In contrast, in the development environment, the Moodle application lacks access to the submission tester. Therefore, scenarios involving solution submission via the built-in website editor and solution submission via GitLab can only be addressed within the production environment.

3.3 E2E testing project architecture and test creation

To facilitate the execution of JavaScript applications and the utilization of Node Package Manager (npm) for acquiring frameworks and tools, the author installed the Node.js runtime environment on their machine. Having the Node.js runtime environment is also a system requirement for running Playwright applications.

The TypeScript and Playwright packages were installed using the newly installed Node Package Manager. During the Playwright installation, the author was prompted to select the programming language, and TypeScript was chosen. Once the Playwright configuration was completed, a TypeScript-based Playwright demo application was generated, providing the foundation for the E2E testing project.

To achieve enhanced code readability, decoupling, reusability, and scalability, multiple application modules were defined. Modules are sets of related classes, functions, and variables that provide logical boundaries for the codebase [14].

3.3.1 MariaDB API

In an E2E testing environment, a container containing the Moodle database is created. This container can be communicated with by using the MariaDB client for Node.js, which can be installed as a code package via Node Package Manager (npm). Once installed, the MariaDB module allows for the creation of a connection pool, which acts as storage for multiple connections. These connections can then be retrieved from the pool and used to execute calls to the MariaDB database. After a process finishes using a connection, it is returned to the pool, ready for reuse [15]. This approach minimizes the performance overhead associated with frequent disconnections and reconnections to the database.

Communication with the MariaDB database is facilitated through SQL queries, which are specified as strings and passed to a connection from the connection pool. Upon execution of SQL queries, connections return Promise objects, which can be parsed into data and utilized across various sections of the project.

Harnessing the MariaDB client in the creation of E2E tests provided several benefits, including the randomization of test data, the minimization of flaky tests' impact on test results, enhanced debugging capabilities, and comprehensive testing coverage.

The `dbContext.ts` file was developed to streamline communication with the MariaDB database. It includes over 20 exported and reusable SQL queries in the form of functions, such as `getRandomTesterType`, `getTesterTypeCount`, `getRandomUniqueDBString`, `getRandomCourseId`, and `getRandomEnrolledCourseParticipantId`, among others, along with the implementation of database communication logic. This module is extensively utilized in a variety of development environment tests.

For instance, the `getRandomUniqueDBString` function ensures the generation of a random string that does not match any fields within the data entries of tables passed as arguments. This functionality effectively prevents test failures caused by non-unique random strings. Conversely, the `getRandomCourseId` function enables the selection of a random course id for testing purposes, ensuring that test outcomes are not reliant on specific input data.

3.3.2 GitLab API

The `gitbeaker/rest` code package facilitates the usage of the GitLab REST API. An instance of the GitLab class can be instantiated using the `@gitbeaker/rest` module by providing the host URL and a GitLab account access token. The GitLab class encompasses various functions for interacting with GitLab, including creating and pushing commits, fetching repositories, managing issues, modifying repository members, and more [16].

In the context of writing E2E tests for the Charon plugin, the GitLab API serves as a tool for simulating and testing answer submissions for various programming assignments. In many instances, the grading process is automated. The code committed via GitLab is retrieved from the student’s repository and executed against multiple tests to assess

compliance with solution requirements. Upon completion of the tests, the student's solution is evaluated, and points are allocated accordingly.

The `gitContext.ts` file was developed to encapsulate logic related to communication with GitLab. The functions defined within this file facilitate the submission of various solutions for distinct programming assignments. By specifying both the programming assignment and the points allocation for a solution, the suitable solution, stored as a text file, is retrieved using the `assignmentPointsMapping.json` file and subsequently pushed to the GitLab repository. This module's functionality is utilized in production tests, where the solution submission tester is configured and operational.

3.3.3 Global variables, enums, interfaces and functions

To enhance code reusability and maintainability, several variables, enums, interfaces, and functions were relocated to a dedicated `helpers` folder. Within this folder, the `globalFunctions.ts`, `globalInterfacesEnums.ts`, and `globalVariables.ts` files were established to store specific entities as indicated by their respective file names.

Within the `globalVariables.ts` module, twenty variables have been encapsulated within a read-only object named `globalConstants`, which can be imported in other parts of the project. These variables encompass table names, month abbreviations, various versions of the Moodle URL, and environment-specific values.

Enums and interfaces from the `globalInterfacesEnums.ts` file are exported individually, fulfilling crucial roles despite their limited quantity. Interface implementation enables TypeScript to detect compile-time errors in the codebase and enhances code readability. Meanwhile, enums facilitate the encapsulation of variables related to specific categories.

A total of 15 functions have been created to perform various tasks, such as generating random, unique strings and interacting with webpage elements like sliders. Each function is defined and exported separately to maintain clarity and manageability, as consolidating them into larger entities would result in excessively long sections of code.

The modules stored within the `helpers` folder are employed across the codebase. Some functionality is shared among these modules and even utilized within the Playwright configuration file.

3.3.4 Test implementation

In the `tests` folder, projects such as `auth.setup.spec.ts`, `dev.tests.spec.ts`, and `prod.tests.spec.ts` were established. Furthermore, the `reusableTestFunctions` subfolder was created to contain the `devFunctions.ts` and `prodFunctions.ts`| files. These files serve to store functions that can be reused in development and production tests, thereby minimizing duplicate code and enhancing test readability.

The `auth.setup.spec.ts` setup project is executed initially, preceding the `dev.test.spec.ts` or `prod.tests.spec.ts` testing projects, depending on the configuration. This project is tasked with logging into the Moodle website, closing any necessary popups, and storing the session cookies for subsequent E2E tests. Running the setup project before the testing project is essential to prevent tests from being conducted with expired session cookies and to ensure that unaddressed popups do not hinder the execution of clicks.

3.3.5 Test example

The process of writing E2E tests adheres to a set of fundamental principles. Initially, the `test` function from the `@playwright/test` module is invoked. Following this, the test name and body are provided as function arguments. Within the test body, interactions with the Moodle website are facilitated using the `page` instance alongside locators. Subsequently, upon executing various actions within the test, the `expect` function is employed to assess the state of the webpage, webpage elements, or the Moodle database.

If an assertion proves false or an interaction with the Moodle website fails to initiate, Playwright notifies the user of the test failure and indicates the line of code responsible for the test's termination.

To enhance comprehension of the test body structure, an example of a basic test designed for testing tester settings functionality is presented below.

Testers serve as containers for automated evaluations aimed at assessing student-submitted solutions for coding assignments. Upon evaluation completion, points are allocated to the student based on their solution, contributing to their overall course grade. A specific tester type is assigned to each course assignment to ensure appropriate testing for the corresponding programming language.

The primary objective of the “Tester settings functionality” test is to verify the capability of creating and deleting tester types through the Moodle website.

Firstly, a random course id is retrieved from the Moodle database, and a URL is generated to access the “charonSettings” endpoint. Upon accessing the webpage via the URL and accessing the tester settings functionality, a new tester type is generated using a random string not found in other tester type entries within the Moodle database. Following the creation of the new tester type, a `sleep` command is executed to allow time for the database to update the existing entry list. The argument “5000 milliseconds” represents an arbitrary timeout duration and can be adjusted as needed.

After confirming the increase in the number of database entries, the newly created tester type is deleted through a button-click action on the website interface. Subsequently, the database is given five seconds to update before verifying whether the deletion process was successful.

```
test('Tester settings functionality', async ( { page } ) => {
  const courseId : number = await db.getRandomCourseId();
  const charonEditUrl : string =
`${globalConstants.url}/mod/charon/courses/${courseId}/popup#/charonSe
ttings`;
  const initialTesterCount : number = await db.getTesterTypeCount();

  await page.goto(charonEditUrl);

  const testerName : string = await
db.getRandomUniqueDBString(globalConstants.testerTypeTable);
  await page.getByLabel('Tester type').fill(testerName);
  await page.getByRole('button', { name: 'Add' }).click();

  await sleep(5000);

  const newTesterCount : number = await db.getTesterTypeCount();
  expect(initialTesterCount == newTesterCount).toBe(false);

  await
page.getByText(testerName).locator('..').getByRole('button').click();

  await sleep(5000);

  expect(await db.getTesterTypeCount() ==
initialTesterCount).toBe(true);
});
```

Figure 1. "Tester settings functionality" test.

3.3.6 Playwright configuration and environment variables

The auto-generated Playwright `playwright.config.js` file provides numerous configuration options for executing E2E tests, collecting test results, and defining projects. The author has customized the Playwright application's configuration to align with the project requirements and the implementation of the solution.

In the custom Playwright configuration, parallelization has been disabled by setting the `workers` option to 1. While parallelization can enhance test performance, its use may also introduce potential race conditions and test flakiness, and it requires more RAM to be utilized effectively.

For debugging purposes, the author has configured a test report to be generated upon test completion. Additionally, videos of test execution and traces of actions performed during

test execution are saved on the first retry after a test failure. The generated report, videos, and traces are stored in the `playwright-report` and `test-results` folders.

A total of 4 projects have been defined in the Playwright configuration. The `setup` project is tasked with logging into the Moodle website and saving session cookies, which are subsequently utilized by other testing projects through the specification of the cookie-containing file in the `storageState` option. The `chromium`, `firefox`, and `webkit` projects are responsible for executing the same set of E2E tests across different browsers to identify browser-specific issues.

```
const { defineConfig, devices } = require('@playwright/test');
import { globalConstants } from './helpers/globalVariables';

module.exports = defineConfig({
  timeout : 220_000,
  testDir: './tests',
  fullyParallel: false,
  forbidOnly: !!globalConstants.isProduction,
  retries: 3,
  workers: 1,

  reporter: [['html', { open: 'never', outputFolder: 'playwright-
report' }]],

  use: {
    trace: 'on-first-retry',
    video: 'on-first-retry',
    baseURL: globalConstants.url,
  },

  projects: [
    { name: 'setup', testMatch: /.*\.setup\.ts/ },

    {
      name: 'chromium',
      use: { ...devices['Desktop Chrome'],
      permissions: ['clipboard-read', 'clipboard-write'],
      storageState: 'states/userState.json' },
      dependencies: ['setup'],
    },

    {
      name: 'firefox',
      use: { ...devices['Desktop Firefox'],
      storageState: 'states/userState.json' },
      dependencies: ['setup'],
    },

    {
      name: 'webkit',
      use: { ...devices['Desktop Safari'],
      storageState: 'states/userState.json' },
      dependencies: ['setup'],
    },
  ],
});
```

Figure 2. playwright.config.js file.

To configure non-Playwright-specific elements of the solution, the author chose to create a `.env` file for storing user-configured environment variables.

The `IS_PRODUCTION` variable may have the values 0 or 1, indicating whether tests for the development or production environment should be executed.

The `MOODLE_URL`, `MOODLE_PORT`, and `MOODLE_PROTOCOL` variables are combined to create an endpoint for accessing the Moodle website.

The `MOODLE_USERNAME` and `MOODLE_PASSWORD` variables serve as login credentials used to authenticate and log into the Moodle website. These credentials enable the execution of E2E tests from the perspective of a logged-in user.

The `HOST`, `DB_USERNAME`, `DB_PASSWORD`, `DATABASE`, and `CONNECTION_LIMIT` environment variables are utilized for creating connection pools, which are then used to establish connections with the Moodle database.

The `GITLAB_ACCESS_TOKEN`, `GITLAB_PROJECT_ID`, and `GITLAB_URL` variables are necessary for interacting with the GitLab API.

```
# Changes the tests that will be run (in production we can't alter
database)
IS_PRODUCTION=0
# Moodle socket address configuration
MOODLE_URL="localhost"
MOODLE_PORT="80"
MOODLE_PROTOCOL="http"
# Moodle developer login configuration
MOODLE_USERNAME="dev"
MOODLE_PASSWORD="dev"
# Database configuration
HOST="localhost"
DB_USERNAME="root"
DB_PASSWORD="root"
DATABASE="bitnami_moodle"
CONNECTION_LIMIT=5
# Git configuration
GITLAB_ACCESS_TOKEN=""
GITLAB_PROJECT_ID=""
GITLAB_URL="https://gitlab.cs.ttu.ee/"
```

Figure 3. .env file.

3.4 Deployment of E2E tests

Establishing one-way resource sharing from `niogon/E2E-tests` to the `ained/charon` repository was crucial for executing E2E tests against the `ained/charon` implementation of Moodle. For resource sharing purposes, the decision was made to utilize the GitLab container registry, largely due to the necessity of testing against a multi-container application.

```
stages:
  - build

build_image:
  stage: build
  image: docker:latest
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD
$CI_REGISTRY
    - docker build -t gitlab.cs.taltech.ee:5050/niogon/e2e-tests .
    - docker push gitlab.cs.taltech.ee:5050/niogon/e2e-tests
```

Figure 4. "build_image" job of the "niogon/E2E-tests" CI/CD pipeline.

CI/CD pipeline for `niogon/E2E-tests` was configured using the auto-generated `.gitlab-ci.yml` file. In this file, the `build` stage and `build_image` job within the `build` stage were defined. As part of the `build_image` job, a script for building and pushing the `niogon/e2e-tests` container image, containing the `niogon/E2E-tests` repository's code, to the GitLab container registry was outlined.

```
FROM mcr.microsoft.com/playwright:v1.43.0-jammy

COPY . /

RUN npm ci
```

Figure 5. Dockerfile for building the "niogon/e2e-tests" image.

A Dockerfile was utilized to specify the instructions for building the `niogon/e2e-tests` image. The official Microsoft Playwright image was chosen as the base image for the `niogon/e2e-tests` container image, primarily because it contains all the necessary libraries and dependencies required to run Playwright applications. At the conclusion of the Dockerfile, the command `RUN npm ci` was employed to retrieve all specified versions of frameworks and tools outlined in the `package.json` file of the application.

The TalTech runner with the Docker executor was used to fulfill the `build_image` job. Within the Docker executor, a Docker container with Docker runtime is created as specified in the `.gitlab-ci.yml` file. This setup facilitates the use of the Docker client for building and pushing the container image to the GitLab container registry.

3.5 E2E testing environment for development tests

Assembling and automating the creation of the testing environment within the `ained/charon` repository's CI/CD pipeline is crucial for executing E2E development tests pulled from the GitLab container registry.

The `ained/charon` repository leverages a private, self-hosted runner named *AinedRunner*, which is utilized for executing CI/CD pipeline jobs. The runner was previously equipped only with a shell executor, which impeded the author's ability to assemble the E2E testing environment. The runner also lacked any defined tags, meaning pipeline jobs could not be configured to run solely on *AinedRunner*. This limitation could have resulted in jobs running on other public runners with unknown configurations, leading to unexpected failures. The Charon development team addressed these issues at the author's request by installing a Docker executor and adding the `docker`, `dind`, and `ained` tags to *AinedRunner*.

3.5.1 Creation of custom docker-compose files

The Charon development team employed a custom `docker-compose.yml` file for the instantiation of Moodle. Initially, the Moodle website was created using the `docker-compose.yml` file without the Charon plugin, in accordance with Moodle's technical specifications. Subsequently, after the instantiation of Moodle, the `docker-compose.yml` file was manually modified to include a volume for loading the Charon plugin into the `charon-moodle` container. Following the modification of the `docker-compose.yml` file, the website was rebuilt with the Charon plugin installed.

This approach to instantiating a Moodle website with the Charon plugin within the CI/CD pipeline was unsuitable in the context of E2E test automation. The `adminer` container had no function in the instantiation of Moodle or the execution of E2E tests, as it is exclusively utilized for database administration. Moreover, the process of manually editing the `docker-compose.yml` file cannot be replicated within the CI/CD pipeline. The author

also encountered challenges with utilizing the `./..:/bitnami/moodle/mod/charon` volume due to the technical file sharing restrictions within the CI/CD pipeline.

```
version: '2'
services:
  adminer:
    container_name: adminer
    image: adminer
    ports:
      - "8190:8080"
  mariadb:
    image: 'bitnami/mariadb:10.6'
    container_name: charon-db
    environment:
      ## environment variables
    volumes:
      - 'mariadb_data:/bitnami/mariadb'
    ports:
      - "3306:3306"
  moodle:
    build:
      context: .
    container_name: charon-moodle
    environment:
      ## environment variables
    ports:
      - '80:80'
      - '9717:443'
    volumes:
      - 'moodle_data:/bitnami/moodle'
      - 'moodledata_data:/bitnami/moodledata'
      - './..:/bitnami/moodle/mod/charon'    ## Keep this line
commented out during initial Moodle install!
    depends_on:
      - mariadb
volumes:
  mariadb_data:
    driver: local
  moodle_data:
    driver: local
  moodledata_data:
    driver: local
```

Figure 6. Initial docker-compose file.

To address the aforementioned issues, the author decided to split the initial `docker-compose.yml` file into `adminer-docker-compose.yml`, `modded-docker-compose.yml`,

and `unmodded-docker-compose.yml` files. This solution eliminated the need for the `adminer` container to be utilized for instantiating the Moodle website, removed the necessity of manually altering the `docker-compose.yml` file, and allowed for the avoidance of utilizing the `./..:/bitnami/moodle/mod/charon` volume for the Charon plugin installation.

3.5.2 Script for assembling the E2E testing environment for development tests

Because of the technical constraints of YML type files, the author opted to create a separate shell script, distinct from the `.gitlab-ci.yml` file, to accurately define the steps required for Moodle instantiation and Charon plugin installation.

At the beginning of the shell script, the `HOSTNAME` and `ADMINER_NEEDED` environment variables have been defined. Users can alter the default values of the environment variables by specifying new values in the command-line script execution query. To establish an E2E testing environment, the `ADMINER_NEEDED` variable is set to “false”.

The first step to establishing an E2E testing environment within the CI/CD pipeline is the creation of an unmodded instance of Moodle using the `unmodded-docker-compose.yml` file. Following the instantiation of the `charon-db` and `charon-moodle` containers within the `docs_default` Docker network, the runner container with the exposed 2375 and 2376 TCP ports and dynamically generated name is added to the `docs_default` network. The runner container is tasked with executing steps defined in the job descriptions. By default, the runner container lacks access to the containers within the `docs_default` network since it is not part of it. Consequently, without access to the `charon-moodle` container, there is no ability to reach the Moodle website.

```
#!/bin/sh
ADMINER_NEEDED="${ADMINER_NEEDED:=true}"
HOSTNAME="$(docker ps --format "{{.Names}}" --filter expose=2375-
2376/tcp)"
UpdateStatus () {
  docker network inspect docs_default
  docker ps -a
  if [ "$ADMINER_NEEDED" = true ]; then
  RESPONSE="$(curl -Is http://localhost:80 | head -n 1)"
  else
  RESPONSE="$(curl -Is http://charon-moodle:80 | head -n 1)"
  fi
  echo "$RESPONSE"
  RESPONSE_PART=${RESPONSE#*HTTP}
  STATUS=$(echo "$RESPONSE_PART" | cut -d' ' -f2)
}
BuildUnmoddedMoodle () {
  if [ "$ADMINER_NEEDED" = true ]; then
  docker-compose -f unmodded-docker-compose.yml -f adminer-docker-
compose.yml up -d
  else
  docker-compose -f unmodded-docker-compose.yml up -d
  fi
}
BuildModdedMoodle () {
  if [ "$ADMINER_NEEDED" = true ]; then
  docker-compose -f unmodded-docker-compose.yml -f modded-docker-
compose.yml -f adminer-docker-compose.yml up -d
  else
  docker cp ./.. charon-moodle:/bitnami/moodle/mod/charon
  docker restart charon-moodle
  docker exec charon-moodle sh -c "cd /bitnami/moodle/mod/charon &&
ls"
  fi
}
BuildUnmoddedMoodle

if [ "$ADMINER_NEEDED" != true ]; then
docker network connect docs_default "$HOSTNAME"
fi
```

Figure 7. First part of the "charon-initialization.sh" script.

After instantiating the containers required to launch the Moodle website, it is crucial to ensure that the containers have finished building before proceeding with the installation of the Charon plugin. The state of the containers can be determined by pinging the “charon-moodle:80” socket address and checking the returned status code. A status code

of 200 indicates that the processes responsible for launching the Moodle website have been completed.

The Charon plugin is installed by copying all files from the `ained/charon` repository inside the runner container to the `/bitnami/moodle/mod/charon` directory within the `charon-moodle` container, followed by restarting the `charon-moodle` container. Once the “charon-moodle:80” socket address returns a status code of 200, confirming the successful installation of the Charon plugin, the Moodle database is populated with database entries using database seeding. Additionally, adjustments are made to the file permissions of the `charon-moodle` container to enable the utilization of specific website functionalities.

If any of the aforementioned steps in creating an E2E testing environment are not completed, exit code 7 is thrown, causing the job responsible for executing the script to fail.

```
CheckModdedMoodle() {
  ATTEMPTS=0
  UpdateStatus
  while [ "$STATUS" != "200" ] && [ $ATTEMPTS != 10 ]
  do
    ATTEMPTS=$((ATTEMPTS+1))
seconds
    sleep 0.5m
    UpdateStatus
  done
  if [ "$STATUS" = "200" ]; then
    docker exec charon-moodle sh -c "cd /bitnami/moodle/mod/charon &&
sh dev-setup.sh && php artisan db:seed --class=DevEnvDataSeeder"
    docker exec charon-moodle sh -c "cd /bitnami/moodle && chmod -R
777 ."
    docker exec charon-moodle sh -c "cd /bitnami/moodledata && chmod -
R 777 ."
    else
    exit 7
  fi
}
UpdateStatus
ATTEMPTS=0
while [ "$STATUS" != "200" ] && [ $ATTEMPTS != 10 ]
do
  ATTEMPTS=$((ATTEMPTS+1))
  sleep 1m
  UpdateStatus
done
if [ "$STATUS" = "200" ]; then
  docker exec charon-moodle sh -c "cd /bitnami/moodle && chmod -R 777
."
  BuildModdedMoodle
  sleep 10s
  UpdateStatus
  if [ "$STATUS" != "200" ]; then
    BuildModdedMoodle
    docker exec charon-moodle sh -c "cd /bitnami/moodle && chmod -R
777 ."
    docker exec charon-moodle sh -c "cd /bitnami/moodledata && chmod -
R 777 ."
  fi
  CheckModdedMoodle
  else
  exit 7
fi
```

Figure 8. Second part of the "charon-initialization.sh" script.

3.5.3 Script for disposing previously generated Docker elements

By default, images, containers, volumes, and networks generated as part of the E2E testing environment and during the execution of other jobs remain persistent after completion of the CI/CD pipeline. If left unmanaged, this persistence could lead to failures in future pipeline jobs.

The author opted to create a separate script for deleting all Docker-generated elements. This script is executed prior to assembling an E2E testing environment, ensuring that previously generated Docker images, containers, volumes, and networks do not affect future testing cycles.

3.5.4 “charon-initialization.sh” script execution as part of the CI/CD pipeline

In the `.gitlab-ci.yml` file, the `build_moodle` job was defined as part of the `E2E testing` stage. The official Docker image uses the Alpine Linux operating system. To enable the `charon-initialization.sh` script to function properly, Client URL (cURL) needs to be installed via the Alpine Linux package manager (APK) in the `before_script` section of the job. Within the runner container, navigation to the `DOCKER_CONTEXT_PATH`, where the `clean-docker.sh` and `charon-initialization.sh` scripts are located, is performed. Subsequently, file permissions for the scripts are adjusted to allow for their execution. Firstly, the `clean-docker.sh` script is executed to prepare the foundation for generating an E2E testing environment, followed by the execution of the `charon-initialization.sh` script.

After successfully generating the E2E testing environment, instances of the `charon-moodle` and `charon-db` containers are saved and pushed to the GitLab container registry for future debugging purposes.

```
variables:
  DOCKER_CONTEXT_PATH: docs
  DOCKER_TEST_RESULTS_PATH: test-results
  DOCKER_PLAYWRIGHT_REPORT_PATH: playwright-report
stages:
  - E2E testing

build_moodle:
  stage: E2E testing
  image: docker
  tags:
    - docker
    - dind
    - ained
  before_script:
    - apk add --update curl && rm -rf /var/cache/apk/*
  script:
    - cd $DOCKER_CONTEXT_PATH
    - chmod 777 ./clean-docker.sh
    - chmod 777 ./charon-initialization.sh
    - ./clean-docker.sh
    - ADMINER_NEEDED=FALSE ./charon-initialization.sh
    - echo "$CI_REGISTRY_PASSWORD" | docker login "$CI_REGISTRY" -u
"$CI_REGISTRY_USER" --password-stdin
    - docker commit charon-moodle
gitlab.cs.taltech.ee:5050/ained/charon/charon-moodle
    - docker image push gitlab.cs.taltech.ee:5050/ained/charon/charon-
moodle
    - docker commit charon-db
gitlab.cs.taltech.ee:5050/ained/charon/charon-db
    - docker image push gitlab.cs.taltech.ee:5050/ained/charon/charon-
db
```

Figure 9. "build_moodle" job of the "ained/charon" CI/CD pipeline.

3.6 E2E test execution

The `niogon/e2e-tests` container image and the E2E testing environment, created during the `build_moodle` job, facilitate the execution of E2E tests defined in the `niogon/E2E-tests` repository against Moodle with the Charon plugin, as part of the `ained/charon` repository's CI/CD pipeline.

3.6.1 Script for executing E2E tests

To execute all necessary steps for running E2E tests, the author created the `test-initialization.sh` shell script. Initially, the `niogon/e2e-tests` container image is

pulled from the GitLab container registry. Subsequently, the `playwright-tests` container is instantiated using the obtained image and incorporated into the `docs_default` network to gain access to the "charon-moodle:80" socket address. Following the addition of the `playwright-tests` container to the network, E2E tests are executed within it using the specified environment variables and command options.

```
#!/bin/sh
IS_DEV="${IS_DEV:=true}"
echo "$CI_REGISTRY_PASSWORD" | docker login "$CI_REGISTRY" -u
"$CI_REGISTRY_USER" --password-stdin
docker container stop playwright-tests
docker rm playwright-tests
docker pull $CI_REGISTRY/niogon/e2e-tests:latest
docker run --name playwright-tests -t -d $CI_REGISTRY/niogon/e2e-
tests:latest
docker network connect docs_default playwright-tests
PLAYWRIGHT_COMMAND="npx cross-env "\
## command variables
"npx playwright test tests/auth.setup.spec.ts tests/dev.tests.spec.ts
--repeat-each=5 --retries=2 --workers 1"
if [ "$IS_DEV" != true ]; then
PLAYWRIGHT_COMMAND="npx cross-env "\
## command variables
"npx playwright test tests/prod.tests.spec.ts --repeat-each=1 --
retries=1 --workers 1"
fi
docker exec playwright-tests sh -c "$PLAYWRIGHT_COMMAND"
```

Figure 10. "test-initialization.sh" script.

3.6.2 "test-initialization.sh" script execution and result collection

After the completion of the `build_moodle` job in the `ained/charon` repository's CI/CD pipeline, the `test_moodle` job is initiated. As part of this job, the permissions of the `test-initialization.sh` script are adjusted to enable its execution. Subsequently, the script is executed to run E2E tests inside the `playwright-tests` container. Upon completion of the E2E tests, the `test-results` and `playwright-report` folders, containing the generated reports, traces, and videos, are copied from the `playwright-tests` container to the developer-defined file directory `DOCKER_CONTEXT_PATH`. These files are then copied from the runner container's `DOCKER_CONTEXT_PATH` directory and stored as artifacts. Compressed artifacts can later be downloaded from GitLab for debugging purposes.

```
test_moodle:
  stage: E2E testing
  image: docker
  tags:
  - docker
  - dind
  - ained
  needs: ["build_moodle"]
  variables:
    IS_DEV: "true"
  artifacts:
    when: always
    paths:
      - $DOCKER_CONTEXT_PATH/$DOCKER_TEST_RESULTS_PATH
      - $DOCKER_CONTEXT_PATH/$DOCKER_PLAYWRIGHT_REPORT_PATH
  script:
   - cd $DOCKER_CONTEXT_PATH
   - chmod 777 ./test-initialization.sh
   - ./test-initialization.sh
  after_script:
    - docker cp playwright-tests:/$DOCKER_TEST_RESULTS_PATH
$DOCKER_CONTEXT_PATH
    - docker cp playwright-tests:/$DOCKER_PLAYWRIGHT_REPORT_PATH
$DOCKER_CONTEXT_PATH
```

Figure 11. "test_moodle" job of the "ained/charon" CI/CD pipeline.

The script execution process within the `prod_test_moodle` job, created by the author, closely resembles that of the `test_moodle` job. However, notable distinctions exist between these jobs. Specifically, the `prod_test_moodle` job does not require the establishment of an E2E testing environment within the pipeline. Additionally, differences arise in the command variables passed during the execution of the script.

4 Analysis of the solution

In this chapter, a comprehensive analysis of the E2E testing solution is presented. The author establishes compliance with the solution requirements, provides an extensive overview of the work done in the scope of this paper, presents a detailed report on the testing of the solution, describes the problems encountered, and identifies areas for potential improvement.

4.1 Compliance with the solution requirements

Throughout the writing process, the author made numerous adjustments to the solution across two repositories, resulting in significant differences between the initial and final versions of the E2E testing solution. Adhering to the solution requirements allowed the author to remain focused on addressing the most critical issues and to maintain a consistent code style and testing logic implementation.

During the development of the E2E testing project, significant effort was dedicated to writing clear and purposeful code aimed at enhancing readability and minimizing code complexity. To achieve this, the author organized code into separate modules, prioritizing logic decoupling and enabling code reuse. The avoidance of class utilization and a module design pattern were adopted due to JavaScript's dynamic nature and limitations in class-based design features, such as the absence of language-level accessibility modifiers, having no namespace functionality for class grouping, and having weak class inheritance capabilities.

While JavaScript lacks certain object-oriented programming (OOP) functionalities found in languages like C# and Java, the use of prototypes and other workarounds can emulate key OOP language features. In hindsight, employing classes and a more object-oriented approach might have been a preferable design choice for better code abstraction.

To streamline the creation of selector strategies for website components, the author leveraged the built-in browser selector utility and the Playwright Codegen tool. The generated selector strategies were then refined using XPath, CSS selectors, and Playwright selector abstractions to minimize flakiness and exposure to changes. The modified strategies proved robust during testing, demonstrating high reliability.

Prior to strategy utilization, certain actions are performed to ensure that webpage loading speed or popups do not hinder strategy execution, in alignment with the procedures outlined in the "Test flakiness mitigation strategy" section of this paper (see section 2.6, page 17).

The efforts outlined in the “Solution” chapter of this paper (see chapter 3, page 19) culminated in the development of a comprehensive and highly configurable automated E2E testing solution. Leveraging resource sharing through the GitLab container registry, the author successfully integrated the E2E testing project into the CI/CD pipeline of the Charon plugin project repository. Notably, these projects can be maintained separately and are located in distinct GitLab repositories.

The E2E tests were structured in alignment with the solution requirements, with distinct projects outlined for both production and development environment testing.

The preceding details delineating the workflow for constructing the E2E testing solution are intended to showcase the author's complete adherence to the solution requirements established prior to the initiation of the solution development process.

4.2 Summary of the final solution

In this section, the final solution is presented through two distinct components: firstly, the E2E testing project itself; and secondly, the integration of this project into the Charon repository’s CI/CD pipeline.

4.2.1 E2E testing project

The E2E testing project incorporates nine tests for development and two for production environments. Additionally, it includes seven modules that act as abstractions of complex and reusable code segments. The project boasts high configurability, facilitated by the auto-generated `playwright.config.js` and the custom-made `.env` files. The `.env` file accommodates fourteen environment variables, added by the author to align with solution requirements (see appendix 2).

The tests defined within the testing projects emulate user interactions with both the Moodle website and the features of the Charon plugin. To simulate interactions for both guests and logged-in users, cookies stored within two JSON files are employed.

Specifically, an empty `guestState.json` file serves to represent guest cookies, while the `userState.json` file encapsulates cookies for logged-in users, which are updated by the setup project prior to test execution.

Furthermore, alongside the JSON files storing cookies, another JSON file was introduced to map solutions to various coding assignments. The solutions themselves were stored as text files and utilized within tests conducted in the production environment.

The E2E testing project also includes a Dockerfile, which serves as a set of instructions within the `niogon/E2E-tests` repository's CI/CD pipeline. These instructions are utilized to build the E2E testing project image, after which it is pushed to the GitLab container registry by the runner container for future utilization (see appendix 3).

4.2.2 Integration of the E2E testing project into the CI/CD pipeline

The creation of the E2E testing environment within the CI/CD pipeline, followed by subsequent testing, was accomplished using custom shell scripts and docker-compose files. A total of three new shell scripts and three new docker-compose files were added for this purpose. Additionally, modifications were made to the `.gitlab-ci.yml` file of the `ained/charon` repository. These modifications were made to provide the GitLab runner with instructions on how the aforementioned tasks should be executed.

As part of the CI/CD pipeline, an E2E testing environment is created. Subsequently, the E2E testing project image is pulled from the GitLab container registry. With the E2E testing project image, a container is constructed and used for executing E2E tests designed for the development environment. Upon completion of the tests, the results, along with videos and traces of the test execution, are saved as artifacts (see appendix 4).

Developers are also provided with the option of manually executing E2E tests for the production environment. Within the scope of the `test_prod_moodle` job, an E2E testing environment is not required. Instead, the only prerequisite for running tests for the production environment is to appropriately configure the E2E testing project. However, the process of storing test results, videos, and traces remains the same as for the `test_moodle` job (see appendix 5).

4.3 Solution testing

To validate the adequacy of the solution presented by the author, each development environment test was executed five times across WebKit, Chromium, and Firefox browsers. These executions were conducted within the `ained/charon` repository's CI/CD pipeline. Out of a total of 135 executions for the development environment tests, only one test failed. Specifically, the logout test for the Firefox browser encountered a failure in executing a click action on the "Log out" button. However, upon retrying the same test, it successfully passed. Consequently, a total of 136 test executions occurred, exceeding the initially planned 135 executions.

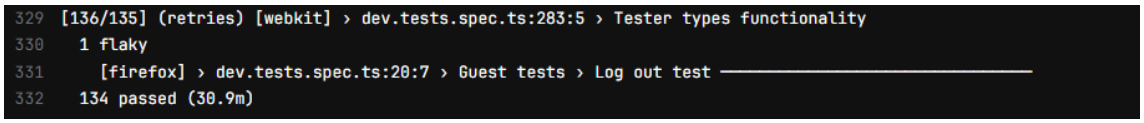


Figure 12. Results for development environment testing within the CI/CD pipeline.

It is noteworthy, that despite not all tests succeeding, the failure of individual tests did not result in the failure of the job responsible for test execution. This is attributed to the test execution configuration, which permits a test to fail and be retried.

Production environment tests were executed locally as the author lacked the necessary Moodle login credentials for execution within the Charon plugin repository's CI/CD pipeline. Two tests were repeated five times across WebKit, Chromium, and Firefox browsers. It is worth noting that code submission via the editor was omitted for the WebKit browser due to technical constraints, resulting in a total of 25 fully completed tests. No tests exhibited signs of flakiness, meaning that each test was successfully completed on the first attempt.

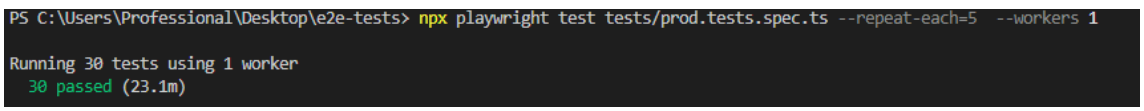


Figure 13. Results for production environment testing on local machine.

4.4 Encountered problems

During the thesis-writing process, the author faced numerous challenges and encountered a variety of issues.

Bureaucratic procedures significantly slowed down progress and hindered the author's ability to steer the development process. Simple requests, such as changing the runner's executor and adding runner tags, took weeks to complete. Even after a month of waiting, requests for login credentials for the Moodle website for an account without two-factor authentication enabled were still unresolved, preventing the testing of production environment tests in the CI/CD pipeline.

The lack of access to the runner machine substantially delayed the debugging process when making modifications to the CI/CD pipeline. Throughout the configuration process, the author had to meticulously log information related to pipeline job execution to understand the root causes of certain issues. This process, along with subsequent log analysis, proved to be challenging and time-consuming, as determining the impact of changes made to the CI/CD pipeline required the completion of multiple-step jobs by the runner.

Additionally, the author encountered browser-specific issues during test creation. For instance, the author was unable to emulate clipboard functionality for WebKit browsers, rendering it impossible to execute tests that required copying and pasting emulation. Furthermore, the Firefox browser exhibited behavioural inconsistencies regarding click actions on buttons. Occasionally, instead of pressing the button, it would be highlighted with a blue border, leading to noticeable flakiness in the “Log out functionality” test.

4.5 Areas of potential improvement

While the author's solution is robust and has shown effectiveness during rigorous testing, there remains room for improvement.

Firstly, there is an opportunity to enhance test coverage. The tests created by the author could serve as a guide for future developers on how to create tests for other website components. Increasing test coverage would improve the ability to detect various bugs, thereby enhancing the end-user experience.

Additionally, while the E2E testing solution has undergone comprehensive testing, there is potential to test modules used for certain tests in isolation. Integration and unit tests could be developed specifically for the `dbContext.ts` and `gitContext.ts` modules. Implementing these tests would help to further reduce test flakiness and could be integrated into the CI/CD pipeline of the E2E testing project and executed before deployment to the GitLab container registry.

Addressing test flakiness is another area for improvement. Currently, one out of eleven E2E tests has shown significant signs of flakiness. It is crucial to conduct a thorough analysis and research into the root causes of test flakiness. Taking a proactive approach will help prevent significant levels of flakiness in future tests.

At the time of writing, the implementation of production environment tests differs from that of the development environment. This discrepancy arises because bypassing Moodle's two-factor authentication (2FA) during test execution using the author's Moodle account is not possible. It requires human intervention to input a one-time password (OTP) generated by the authenticator application. Once it becomes possible to utilize an account without 2FA enabled during test execution, the logic responsible for logging into Moodle in the production environment could be modified. This adjustment would facilitate test initialization without human intervention, enabling production environment tests to run alongside development environment tests in the Charon repository’s CI/CD pipeline.

The author encountered difficulty in replicating clipboard command emulation for WebKit browsers, hindering the execution of production environment tests reliant on this functionality. An extensive investigation is warranted to explore potential solutions to this challenge, enabling cross-browser test coverage for cases dependent on clipboard emulation.

5 Summary

Moodle plugins serve as valuable tools for enhancing the Moodle Learning Management System (LMS), offering users numerous customization opportunities. However, custom Moodle plugins are often susceptible to containing numerous bugs and demand substantial testing, especially as plugin functionality expands. Charon plugin is no exception, given its extensive range of functionalities. Consequently, the author of this thesis took on the responsibility of automating the testing process for the Charon plugin.

E2E tests were selected to be written and executed as part of the Charon plugin repository's CI/CD pipeline to replicate manual human testing. For test creation, the author opted to employ the Playwright framework, a modern tool developed by Microsoft for testing websites.

The author successfully established and automated the creation of an E2E testing environment within the Charon plugin GitLab repository. This environment is subsequently leveraged within a separate pipeline job to execute development environment tests. Additionally, developers are provided with the option to execute production environment tests through a separate job within the CI/CD pipeline, though manual initiation is required.

The E2E testing project was developed with a focus on maintainability and scalability. The author aimed to minimize the use of hard-coded values and provide end users with the ability to configure various aspects of test execution. Additionally, numerous modules were defined to effectively organize the codebase and facilitate code reuse and abstraction. These practices lay the foundation for future improvements to the E2E testing project, including the addition of more tests to expand current test coverage.

References

[1] Moodle, “About Moodle,” [Online]. Available: https://docs.moodle.org/403/en/About_Moodle. [Accessed 4 May 2024].

[2] Moodle, “Statistics,” [Online]. Available: https://stats.moodle.org/. [Accessed 4 May 2024].

[3] Bitrock Inc., “Bitnami LMS powered by Moodle™ LMS,” [Online]. Available: https://bitnami.com/stack/moodle. [Accessed 4 May 2024].

[4] GitLab Inc., “Get started with GitLab CI/CD,” [Online]. Available: https://docs.gitlab.com/ee/ci/. [Accessed 4 May 2024].

[5] GitLab Inc., “GitLab Runner,” [Online]. Available: https://docs.gitlab.com/runner/. [Accessed 4 May 2024].

[6] Docker Inc., “Use containers to Build, Share and Run your applications,” [Online]. Available: https://www.docker.com/resources/what-container/. [Accessed 4 May 2024].

[7] D. Adetunji, “How Docker Containers Work – Explained for Beginners,” 23 October 2023. [Online]. Available: https://www.freecodecamp.org/news/how-docker-containers-work/. [Accessed 4 May 2024].

[8] Docker Inc., “Networking overview,” [Online]. Available: https://docs.docker.com/network/. [Accessed 4 May 2024].

[9] Microsoft, “Playwright enables reliable end-to-end testing for modern web apps.,” [Online]. Available: https://playwright.dev/. [Accessed 4 May 2024].

[10] J. Lennon, “Type Checking In TypeScript: A Beginners Guide,” [Online]. Available: https://zerotomastery.io/blog/typescript-type-checking/. [Accessed 4 May 2024].

[11] Microsoft, “Playwright,” [Online]. Available: https://www.npmjs.com/package/playwright. [Accessed 4 May 2024].

[12] A. Romano, Z. Song, S. Grandhi, W. Yang and W. Wang, “An Empirical Analysis of UI-based Flaky Tests,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, vol. 43, Madrid, Institute of Electrical and Electronics Engineers, 2021, pp. 1585-1590.

[13] Software Freedom Conservancy, “Tips on working with locators,” 10 February 2022. [Online]. Available: https://www.selenium.dev/documentation/test_practices/encouraged/locators/. [Accessed 4 May 2024].

[14] Mozilla Corporation, “JavaScript modules,” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules. [Accessed 4 May 2024].

[15] MariaDB, “Using Connection Pools with MariaDB Connector/Python,” [Online]. Available: https://mariadb.com/docs/server/connect/programming-languages/python/connection-pools/. [Accessed 4 May 2024].

[16] J. Dalrymple, “@gitbeaker/rest,” [Online]. Available: https://www.npmjs.com/package/@gitbeaker/rest. [Accessed 4 May 2024].

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

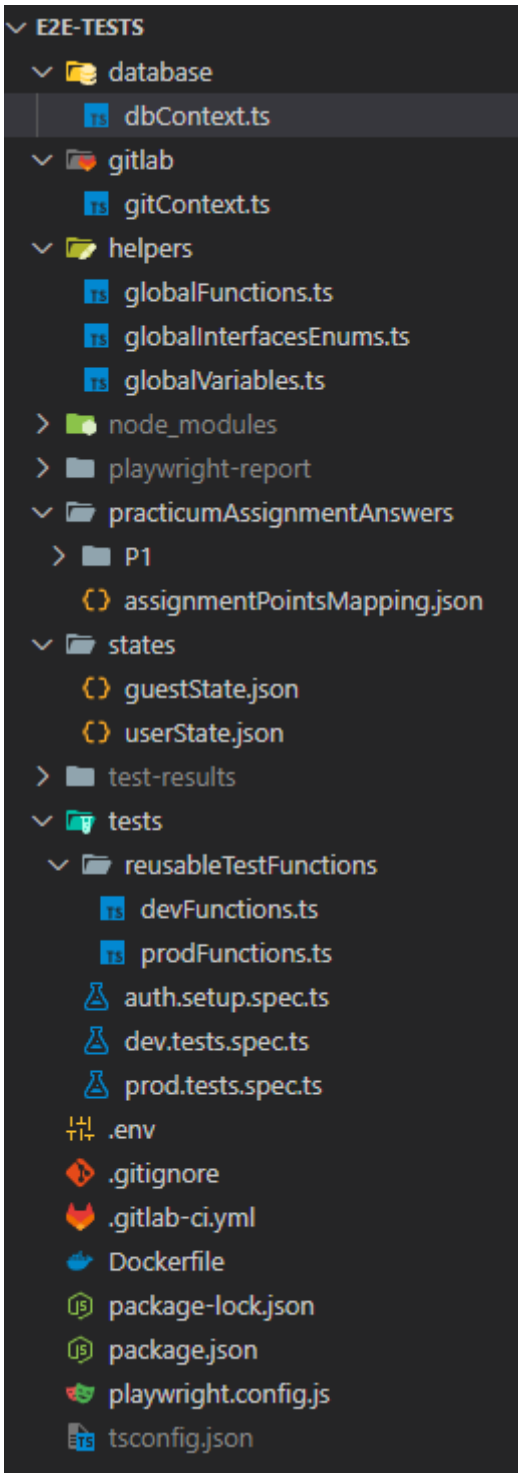
I Nikolai Ogonkov

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Charon Moodle plugin end-to-end test automation”, supervised by Bahdan Yanovich
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

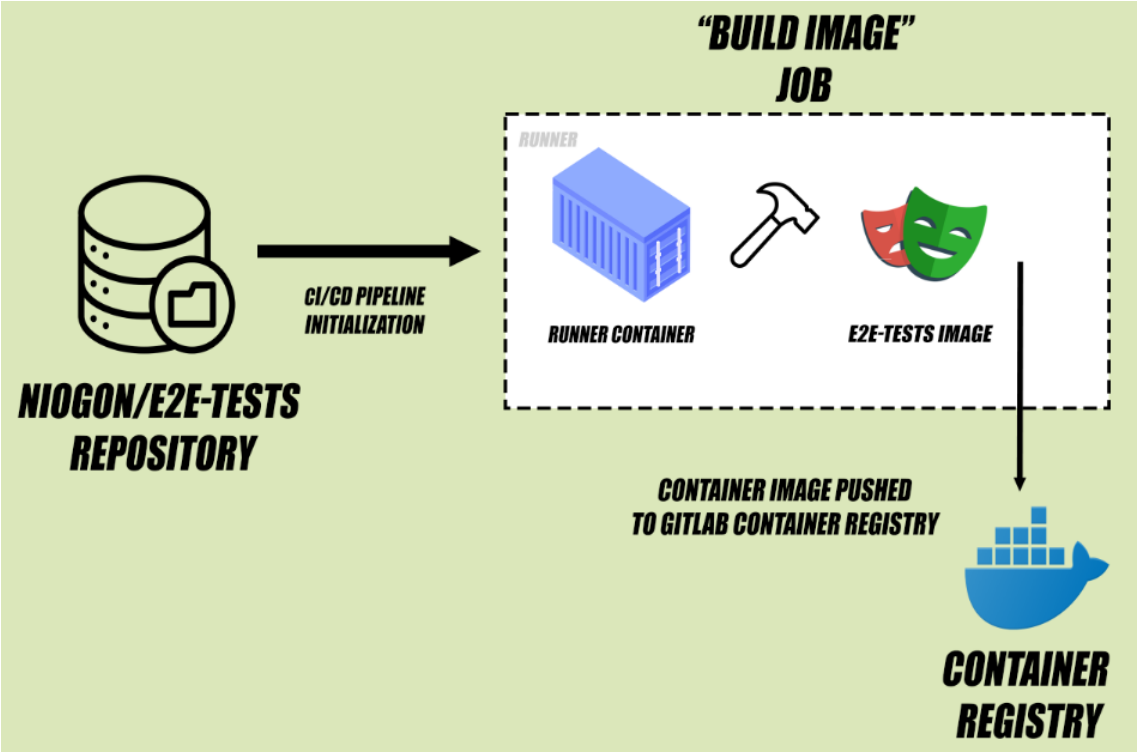
20.05.2024

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

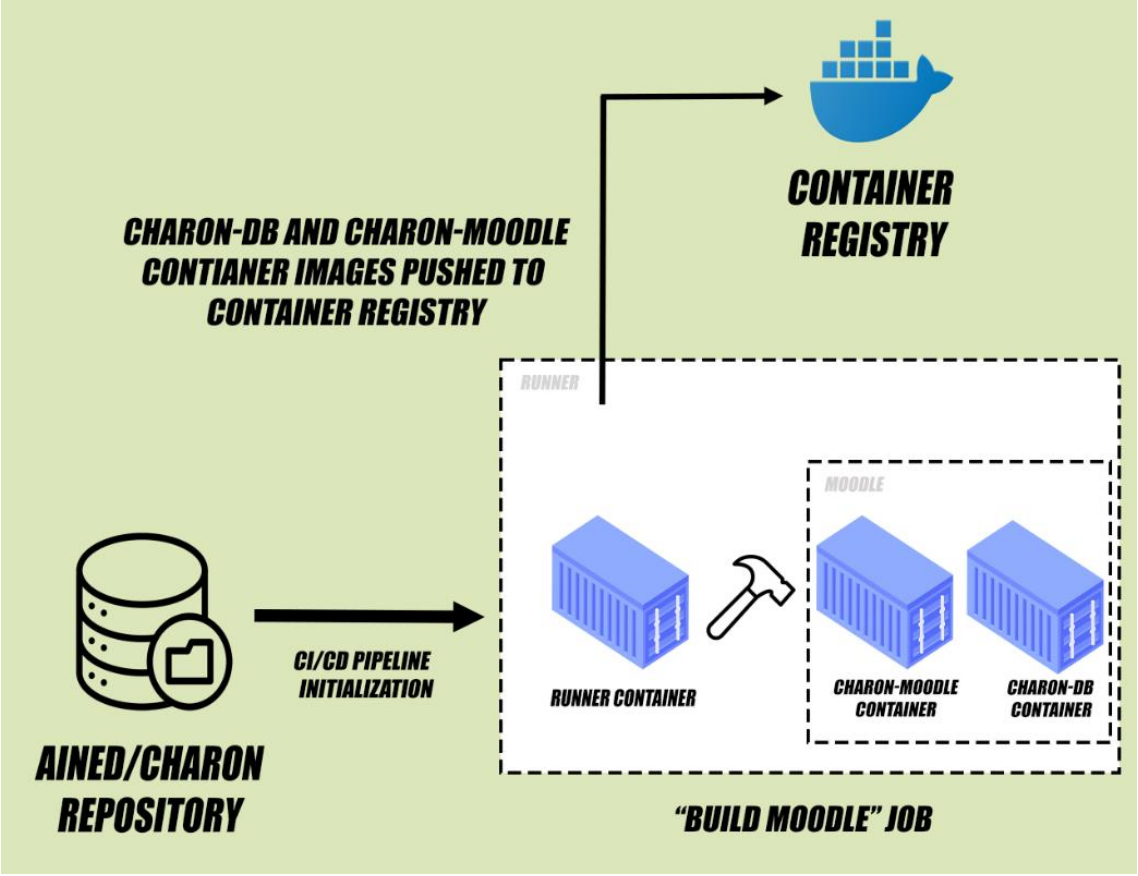
Appendix 2 – E2E testing project structure

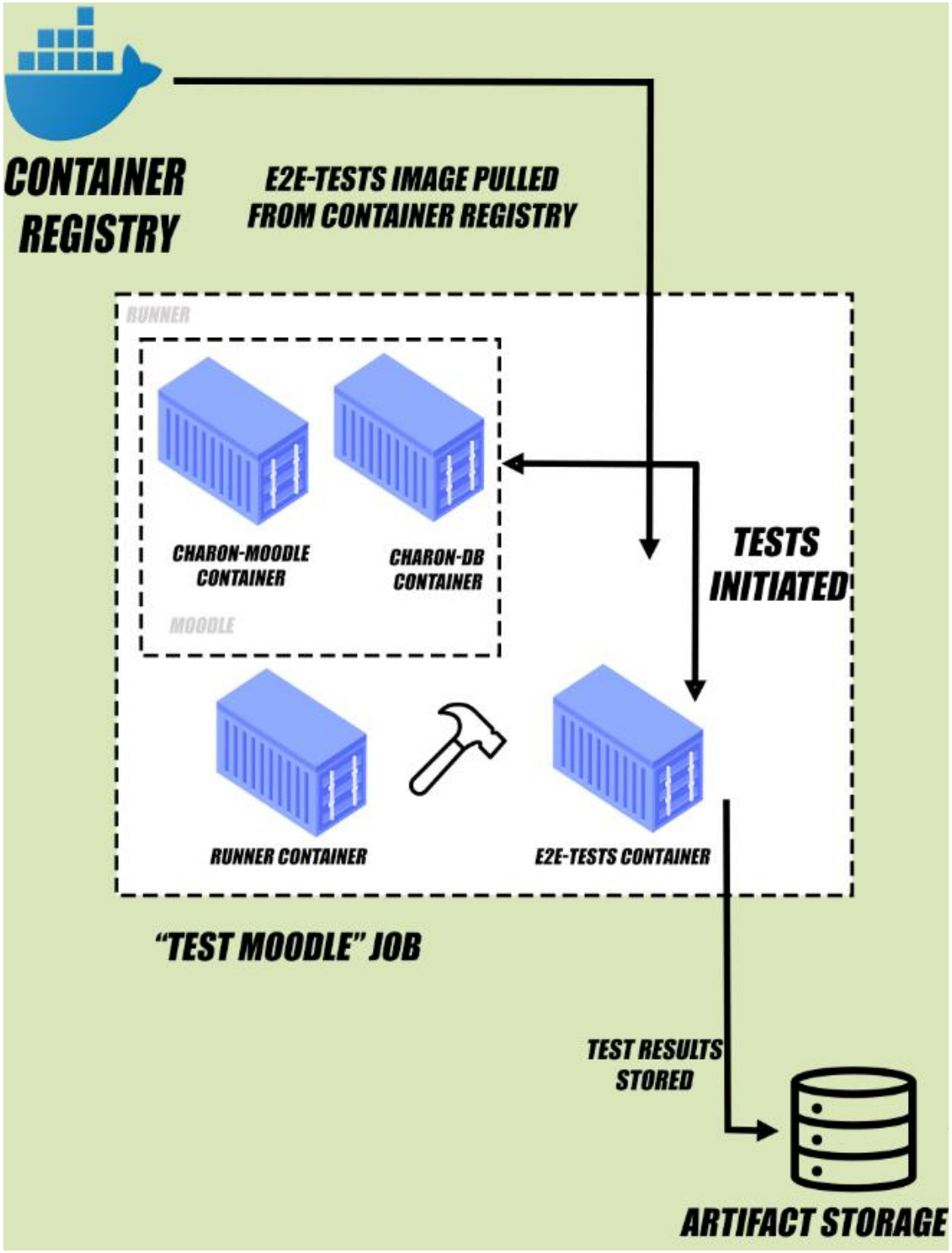


Appendix 3 – “niogon/E2E-tests” CI/CD pipeline’s “build_image” job



Appendix 4 – “ained/charon” CI/CD pipeline’s “build_moodle”, “test_moodle” jobs




CONTAINER REGISTRY
E2E-TESTS IMAGE PULLED FROM CONTAINER REGISTRY
RUNNER
CHARON-MOODLE CONTAINER
CHARON-DB CONTAINER
MOODLE
TESTS INITIATED
RUNNER CONTAINER
E2E-TESTS CONTAINER
"TEST MOODLE" JOB
TEST RESULTS STORED
ARTIFACT STORAGE

Appendix 5 – “ained/charon” CI/CD pipeline’s “test_prod _moodle” job