BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

TEST SUITE OF THE EAS FRAMEWORK
TESTOVACIA SADA FRAMEWORKU EAS

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **JÚLIA MAZÁKOVÁ**
AUTOR PRÁCE

SUPERVISOR **Ing. ALEŠ SMRČKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

Zadání bakalářské práce

146849

Ústav:	Ústav inteligentních systémů (UITS)
Studentka:	**Mazáková Júlia**
Program:	Informační technologie
Specializace:	Informační technologie
Název:	**Testovací sada frameworku EAS**
Kategorie:	Analýza a testování softwaru
Akademický rok:	2022/23

Zadání:

1. Nastudujte testování softwaru a jeho automatizace. Zaměřte se převážně na nižší úrovně testování dle V-modelu. Seznamte se s frameworkem EAS pro tvorbu webových aplikací.
2. Analyzujte současný stav ověřování kvality frameworku EAS. Navrhněte přístup k testování frameworku EAS s ohledem na pokrytí kritických částí softwaru a dovednosti vývojového týmu.
3. Navrhněte testovací případy. Implementujte automatickou testovací sadu, klaďte důraz na udržitelnost a rozšiřitelnost testovací sady.
4. Vyhodnoťte implementovanou testovací sadu.

Literatura:

- IEEE/ISO/IEC 29119-4-2021 International Standard - Software and systems engineering--Software testing--Part 4: Test techniques

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu:	Hanáček Petr, doc. Dr. Ing.
Datum zadání:	1.11.2022
Termín pro odevzdání:	10.5.2023
Datum schválení:	3.11.2022

Abstract

This work focuses on the software testing of the system with an emphasis on lower layers of testing, which includes unit and integration testing. The primary objective is to showcase the testing process using the EAS (Effective Agenda System) framework, starting with the test plan process followed by the creation of the final test suite that will evaluate the performance of the system's backend microservices. This test suite covers a significant part of the system's functionality. The results are stored and analyzed in the final test report. Performed analysis of the system and the final suite serve as valuable assets in the context of testing.

Abstrakt

Toto dielo sa zameriava na softvérové testovanie systému s dôrazom na nižšie vrstvy testovania, ktoré zahŕňajú testovanie jednotlivých komponent a integráciu. Hlavným cieľom je demonštrovať proces testovania na EAS (Efektívny Agendový Systém), začínajúc plánovaním testov a následne vytvorením konečnej testovacej sady, ktorá vyhodnocuje výkon mikroslužieb systému. Táto testovacia sada pokrýva významnú časť funkcionality systému. Výsledky sú zobrazené a analyzované vo vygenerovanej testovacej správe. Analýza systému a konečný report slúžia ako cenný nástroj v rámci testovania.

Keywords

software testing, automation, EAS framework, Test Driven Development, microservice architecture, V-model in testing, reporting, unit testing, integration testing

Kľúčové slová

testovanie sofvéru, automatizácia, EAS systém, programovanie riadené testami, architektúra mikroslužieb, testovanie podľa V-modelu, správa z testovania, unit testy, integračné testy

Reference

MAZÁKOVÁ, Júlia. *Test Suite of the EAS Framework.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Aleš Smrčka, Ph.D.

Rozšírený abstrakt

Táto práca sa zaoberá softverovým testovaním a jeho automatizáciou, so zameraním na nižšie vrstvy testovania, ako je testovanie komponent a integračné testovanie. Pre účel demonštrácie je proces testovania popísaný na konkrétnom informačnom systéme. Už z názvu práce vyplýva, že sa jedná o EAS alebo Efektívny Agendový Systém od firmy InQool. Tento systém je založený na bázi mikroslužbovej architektúry a slúži pre tvorbu webových aplikácií .

Úvod práce popisuje teóriu potrebnú k otestovaniu tohto systému, so zameraním na automatické testy, programovanie riadené testovaním, testy komponentov a integračné testy. Taktiež sú popísané rozličné metódy testovania a nástroje, ktoré sú použité v samotnej implementácii testovacej sady.

Následujúca časť sa venuje analýze testovaného systému. Vlastnosti tohto systému prinášajú určité riziká a špecifické požiadavky na testovanie. Architektúra tohto systému prináša oproti klasickému monolitickému riešeniu veľa výhod, ale zároveň zvyšuje komplexnosť riešenia, ktoré sa odráža aj na testovaní. Pre testovanie je potrebné poznať všetky aspekty daného systému, odhadnúť najväčšie riziká a na ich základe navrhnúť mitigačné plány. Dôležitý je taktiež aj stav testovania systému v čase analýzy, ktorý bude slúžiť ako základ pre ďalšie testovanie. Tento stav odráža aj skúsenosti a zvyklosti vývojového tímu, potrebné k nastaveniu správneho postupu testovania.

Na základe prevedenej analýzy je následne vytvorený štandardný testovací plán, obsahujúci základné časti, akými sú testovacie položky, kritéria, riziká spojené s testovaním a ďalšie. Automatizácia testov nie je vždy výhodným riešením a pred testovaním je potrebné vykonať analýzu nákladov a prínosov.

Implementačná časť sa skladá z dvoch hlavných častí a popisuje všetky nástroje využité počas testovania, metódy použité pri testovaní už existujúceho kódu a praktiky, či konvencie dodržiavané pri testovaní pre zlepšenie kvality a prehľadnosti testov.

V prvej implementačnej časti sa testuje už existujúci kód. Keďže sa jedná o mikroslužby, objavuje sa v tejto časti problém so závislosťami. Pre správne testovanie komponentov je potrebné čo najviac izolovať kód. Pre tento účel sa používajú techniky ako extrakcia metód či vkladanie závislostí. Dôležitým výstupom tejto časti je testovacia sada testov komponentov, ktorá služi primárne na overenie funkčnosti EAS. Pri vytvaraní nových mikroservís úspešné spustenie tejto sady indikuje, že nový kód nepoškodil existujúci funkčný kód. Táto sada má významnú rolu pri regresnom testovaní. Okrem toho sa vytvárajú aj integračné testy, ktoré simulujú produkčné prostredie.

Druhá čast implementácie sa zaoberá postupom pri vytváraní nových mikroservís. Programovanie riadené testami zaručuje vysoké pokrytie kódu a celkové zlepšenie kvality. Táto časť obsahuje praktickú ukážku tohto procesu, obsahujúcu všetky časti od dizajnu prvotných testov až po plnú funkcionalitu výslednej mikroslužby.

Záver tejto práce skúma získané metriky a reporty z testovacej sady a zvýšenie pokrytia testov mikroslužieb EAS. Táto práca zvýšuje kvalitu testovania tohto systému, no kvôli komplexnosti systému pokrýva len časť testovania. Preto by táto práca mala služiť ako podklad pre ďalšie testovanie EAS pri testovaní vyšších vrstiev V-modelu.

Test Suite of the EAS Framework

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Aleš Smrčka, Ph.D. I have listed all the literary sources, publications, and other sources, which were used during the preparation of this thesis.

. . . .
Júlia Mazáková
May 8, 2023

Acknowledgements

I would like to express my gratitude towards my supervisor Ing. Aleš Smrčka, Ph.D. for his guidance, valuable tips, and expert consultations about this work. I would also like to thank my family for their support and encouragement.

Contents

List of Figures

Chapter 1

Introduction

Microservice architecture is becoming increasingly popular and it is widely used by many organizations. This architecture is beneficial in terms of improved scalability, modularity, isolation of services, and ease of deployment and it also helps reduce overall costs.

To ensure that the system is reliable, secure, and meets all of the business requirements it is necessary to test it properly. It is important to test if all of the services are functioning as expected and that they can interact and communicate with each other. When defects are detected, and subsequently fixed it contributes to the overall quality of the system. The main goal of testing is to capture these defects as early as possible and hence save the organization's time and money.

This can be done both manually and automatically. However, automation testing brings its own benefits like running tests more quickly, often in parallel, more scenarios can be tested, and decreasing the impact of human error. They can be run repeatedly to ensure consistency. In a microservice architecture, automated testing ensures the reliability and scalability of microservices with the help of many automated tools and techniques. It also allows services to be tested in isolation. A very important part of testing is reports which allow us to identify and analyze any potential problems.

An example of a system using microservices is EAS, an internal system of the company InQool. This thesis focuses on developing a testing suite for EAS microservices architecture. The proposed suite enables developers to quickly and accurately assess their microservices for potential errors, bugs, and other issues. The test suite mainly consists of unit tests and integration tests to evaluate the functionality of microservices and their interactions with other components of the system. After the suite is executed, detailed reports of the results are generated so the developers and testers can easily identify and address any issues.

The next chapters provide an in-depth look into the fundamentals of testing terminology with a special emphasis on Test-Driven Development and testing tools used to evaluate the quality of the EAS. In the second half, the current status of the EAS is discussed and a testing suite is introduced to analyze critical components of the system. At the very end, the results of the testing suite are analyzed and discussed. This thesis is a valuable resource for those who are involved in the testing process of the EAS.

Chapter 2

Testing and Development Methodologies for Microservices

In this chapter, the term testing is introduced along with the different testing types, methods, and tools. This part also examines manual and automation testing, the differences between them, and their benefits and disadvantages to the testing process. The concept of test-driven development is also discussed in Section 2.4.

Lastly, the difference between monolithic and microservice architecture is introduced in Section 2.6, as the type of architecture has a significant impact on the way testing is approached.

2.1 Testing

There are many misconceptions about the term testing such as the belief that testing ensures that the applications have no errors or that the program functions properly and it does what it is supposed to do. However, the proper definition of testing is the process of executing a program with the intent of finding errors. [17]

This process consists of many different activities like designing, planning, executing, and analyzing results, as well as other activities such as reporting test progress and results. It is desired to find errors as early in the development process as possible which results in decreasing the costs used for fixing defects. [10]

Another misconception about testing is that testing is only focused on the verification of the product. Even though the process is based on requirements or other specifications, it also involves checking if the final product meets the stakeholder's needs while using the product. This process is called validation. [10]

Testing is not only a technical task but it also leans on a proper knowledge of testing and attitude. Tester's individual aspects, skills and methods used for testing have a strong effect on the results of testing. Nowadays it seems to be less common to execute testing by the thoroughly documented test-case-based process. Increasingly, the experience and skill of the tester come into consideration. [13]

Testing every combination of input and output of the product is impossible and exhausting even with relatively small applications. Therefore, it is important to choose wise testing methods and techniques to achieve an almost bug-free state. [17]

At the same time, to ensure the most efficient approach, it is crucial to properly select the most suitable parts of the application to be tested, also commonly referred to as SUT (System under test). [1]

Testing tools in general and also concepts of specific testing tools used in this work are described in Section 2.5.

2.2 V-model in Software Testing

V-model serves as a general development test model. It divides the system development into several layers each with a corresponding test process. [26] V-model is displayed in Figure 2.1.

V-model was created to combat the notion of testing the product at the end of its life cycle. Defects being found way too late in the process resulted in poor quality of the final product, hence, wasting resources and high costs. [10]

There are many activities to be carried out before the coding phase is over, such as finding defects in the test basis documents. A good practice of involving testers involved as early as possible is what makes V-model a powerful tool for ensuring high-quality products. A common variant of V-model includes four different test levels, which are component testing, integration testing, system testing, and acceptance testing. [10]

Even though V-model is ensuring the testing is executed on several layers and happens as soon as possible in the life cycle, the biggest portion of testing comes after code realization. [10]

This approach is the opposite of the Test-driven development described in Section 2.4.

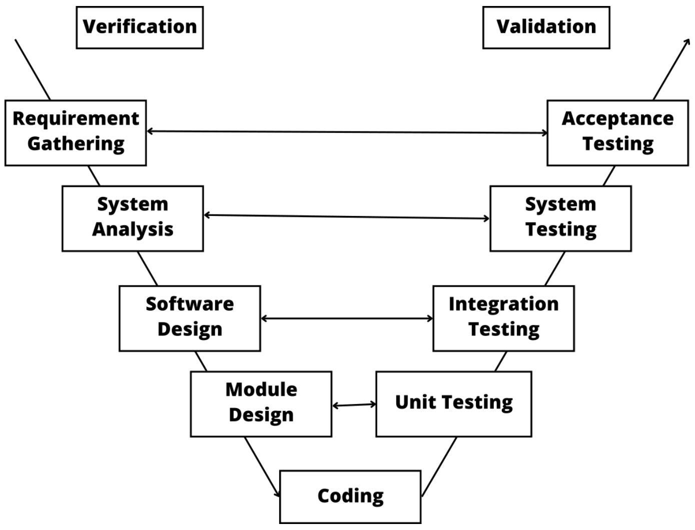


Figure 2.1: V-Model in Software Testing

2.3 Types of Testing

Tests can be divided into different types based on their requirements, levels, design techniques, execution, system structure and accessibility, and many more. [10]

Test levels are describing typical objects, targets of testing, related work products, testers, types of defects, and failures. On the other hand, test types define the targets of testing, since it helps focus on making and communicating decisions against test objectives easier. [10]

Different levels and types of testing are introduced in the next parts.

2.3.1 White Box and Black Box Testing

White-box and Black-box testing types divide tests based on the system's internal structure. With the White-box testing the tester needs to have access to the way the code is built, and the database, but also know what the overall purpose of the product is. [17]

Black-box testing does not require access to the code. [17] The main difference between these two approaches is displayed in Figure 2.2.

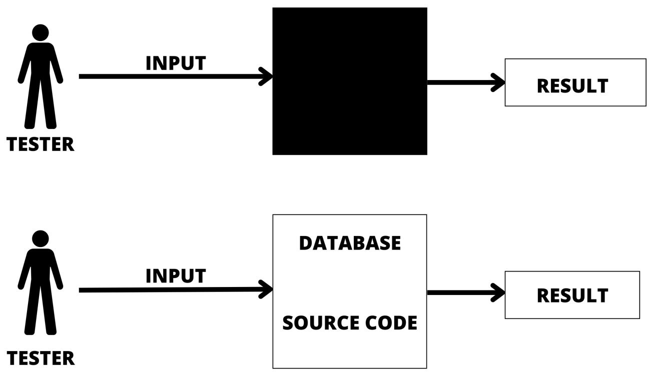


Figure 2.2: Comparison between Black and White Box Testing

2.3.2 Unit Testing

Unit (or module) testing validates the low-level design development phase according to the V-model.

However, the term unit has a different meaning depending on the environment. [4] Unit testing is a process of testing the individual subprograms, subroutines, classes, or procedures in a program. [19]

As the name indicates, the goal of unit testing is to gain confidence that each part functions properly. To ensure that each unit is truly independent, we must simulate the behavior of dependencies, this can be done by mocking. [17]

Mocking

An external dependency is an object interacting with the code under test and cannot be controlled. [19]

Mocking is very useful when dependencies such as databases or web services are present. They may slow down the execution, and be costly to properly set up. External dependencies also require more control. Mocking helps to mitigate inefficient testing. The key challenge with mocking is to decide whether mocking makes the process of testing easier or not. [27]

External dependencies may not be available all the time which could result in test errors. This is where using mocks or stubs comes in useful. Mocking also ensures that when the same test case is run in a new environment, all dependent external services can be called normally. This adds to the overall ability to migrate. [15]

Many microservices are dependent on each other and therefore the problem of dependency in microservice applications is present. To solve this problem controllable mock technologies are used. Mock technologies create virtual objects that simulate other objects needed during testing. Microservice under test can obtain an expected result which helps with isolating the test errors of this microservice due to the errors of other dependent microservices. [15]

Code Coverage

To measure how much of the code is tested there are several metrics that can be calculated. These metrics are also called code coverage. There are several criteria involved, as shown in Table 2.1.

Coverage criteria	Description
Function Coverage	Has each function in the program been executed?
Statement Coverage	Has each statement in the program been executed?
Condition Coverage	Has each Boolean sub-expression been evaluated to both true and false?
Branch Coverage	Has each branch of the control structure been executed?
Class Coverage	Has each class in the package been covered?

Table 2.1: Description of Code Coverage Criteria

2.3.3 Integration Testing

There are many testing approaches in microservice architecture systems. When we make sure that each unit works as designed, higher-order or integration testing takes place. Integration testing is one of the most important types of testing, as it verifies the proper interaction through the entire interface both internally and externally .[31]

On the other hand, analyzing logs from different microservices and writing sufficient test cases can be very difficult tasks.[31]

Moreover, integration tests increase the risk of testing too many things at the same time. This can easily result in not knowing the root cause of the failure. [19]

Integration Test Types

Integration tests can be divided by granularity into narrow and broad integration tests. [31]

Narrow integration tests deal with the code that cooperates with another service while mocking these outside services. They are very similar to unit tests in terms of scope.[8]

Broad integration tests test all of the code, not just the interaction one. It also requires live versions of services and network access. [8]

For a better understanding of the difference between these two types, a visualization of both integration testing types is displayed in Figure 2.3.

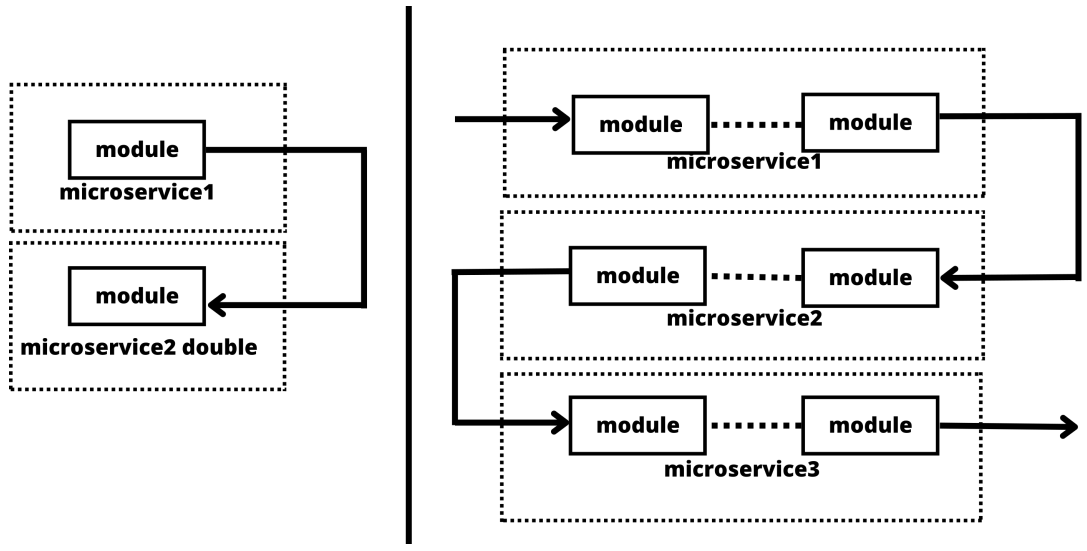


Figure 2.3: Difference Between Narrow and Broad Integration Tests

Another criterion taken into account when dividing tests into subcategories is execution. Based on the execution, testing can be divided into manual and automated testing.

2.3.4 Manual Testing

Manual testing is the basic way of testing the application by hand. Tester takes on the role of a regular user of the final product and simulates the process of using the application. Its reliability is affected by the possibility of human error but at the same time it cannot be completely replaced by automation. [16]

Manually performing testing has the disadvantage of being more time-consuming in terms of setting up and executing the tests. [2]

System-level testing is still very dependent on manual testing since most of the defects are found this way. The automation should serve as a way of removing the repetitiveness of the testing process and allow for more creative types of manual testing such as exploratory testing. [13]

In general, manual tests can be divided into Exploratory and Planned Testing.

Exploratory Testing

Exploratory Testing is an informal and experience-based type of test execution technique performed by simultaneous test design execution and reporting. Tester's experience with testing similar applications is key during this process.[13]

Thanks to the strong subjectivity, some major testing organizations consider this method as auxiliary testing that should only complement basic testing strategies. [35]

Exploratory testing is more common nowadays. It is a form of testing that does not rely on test case documentation. It is an intuitive test performing built on the experience and knowledge of the tester executing the process. [13]

Exploratory thinking can also be used during the designing and execution of test cases. [35]

Planned Testing

Planned Testing is a test-case-based testing focused more on prediction and control. Test cases directed from requirements are created in advance. This method does not cover all the possible defects and tells little about overall user usability.

Construing test cases can be difficult and sometimes quite a useless activity. It barely happens that a user reports particular problems that can be traced back to the lack of structured methods specifically. [12]

This shows that it is valuable to use both test-case-based testing and complement it with experience-based test techniques like error guessing, exploratory testing, and checklist-based testing.

2.3.5 Automation Testing

The goal of automating test script execution is mainly to increase the number of tests being run and also the frequency at which they run while reducing the manual test cycles. On the other hand, automation testing often faces failed projects because of underestimation of the effort required to develop and maintain automated tests. [21]

At first, automation tests setup is more time-consuming because of the initial creation of the scripts but many results show that automated testing is more effective than manual testing in the long run. However, it can also be counterproductive if the initial investment and also continuous maintenance are not handled properly. This happens since any updates and adjustments of the SUT can lead to malfunction in the automation test suite. [7]

Microservice architecture framework requires adding more and more services which usually results in a very complex system. And with every complexity whether inherent or accidental comes a cost. When the complexity of software grows, it takes more time to test everything, find bugs, and retest the system all over again after fixing the defects. This process is repeated after adding every new feature to the product. [1]

There are several ways to automate manual testing. In the case of UI testing, many tools such as Selenium or Cypress come with a recording function that tracks the interactions with web elements and then automates the whole process.

Automated test suites are created to automate lower levels of testing like unit testing and integration testing.

The summary of automation's advantages and disadvantages can be seen in Table 2.2

Advantages	**Disadvantages**
Improves accuracy and quick finding of bugs compared to manual testing.	Choosing the right tool requires considerable effort, time, and an evolution plan.
Saves time and effort by making testing more efficient.	Requires knowledge of the testing tool.
Increases test coverage because multiple testing tools can be used at once allowing for parallel testing of different test scenarios.	Cost of buying the testing tool and, in the case of playback methods, test maintenance is a bit expensive.
The automation test script is repeatable.	Proficiency is required to write the automation test scripts..

Table 2.2: Advantages and Disadvantages of Test Automation [29]

2.3.6 Cost Benefit Analysis Between Manual and Automation Testing

CBA (Cost-benefit analysis) is a process of comparing the costs of an object with the expected benefits to determine if it is worth pursuing it. It consists of the Cost Model and Benefit Model. [14] The example of this analysis can be seen in the Section 4.9.

2.4 Test Driven Development

Test Driven Development consists of the creation of unit tests for the functions to implement. The whole process is displayed in Figure 2.4.

Developers can validate their code by running these tests. After tests succeed, refactoring takes place to increase code readability. It can be done after writing several tests or one by one. These steps are then repeated recursively for each bit of the product's functionality. [33]

This method helps with decreasing defect occurrence and theoretically, it creates a unit test suite with code coverage close to 100 percent. [33]

It has also other benefits like aiding in design and reducing complexity by breaking the problem into small solvable solutions. [19]

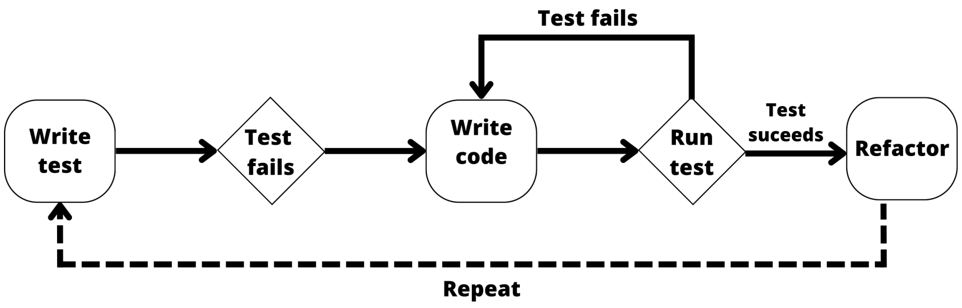


Figure 2.4: Test Driven Development Cycle

2.5 Testing Tools

The growing size and complexity of software systems increase the need for test automation. To easily manage and execute automation, testing tools are a beneficial and popular choice by many companies. [5]

It is very important to choose the correct tools and to properly examine the system under test. Many industrial surveys indicate the lack of the right tools as the main obstacle to test automation.[5] There are both commercial and open-source tools available to choose from. Choosing the right tool can speed up testing and adds to the overall quality of the whole process. [22]

This work mainly focuses on unit and integration testing of backend microservices written in Java language built by Spring framework. That narrows down the circle of possible tools to use. In the next subsections, one of the most popular tools for unit testing, **Junit5**. As for mocking the tested objects, the tool **Mockito** is described. Both of them are integrated with Spring Boot by default.

2.5.1 JUnit 5

JUnit is one of the most popular unit-testing frameworks for Java applications. It utilizes Java classes called test cases, consisting of class components i.e. test methods. By adding multiple test cases together, a testing suite is created. Important methods called fixtures contain **setUp** and **TearDown** methods in order to save the repetitive configuration of each method within one test case. [32]

JUnit is an instance of the xUnit architecture depicted in the UML class diagram 2.5. It provides a variety of assertions for testing expected results. This is very important in automated testing since the system is capable of automatically judging if the test has failed or passed.[20]

The test run consists of two phases, configuration, and test case execution. Another benefit that this framework offers is the distinction between failures and errors. Failures are instances of AssertionFailureError and are created by test case code, anything else that went wrong is considered an error. [30]

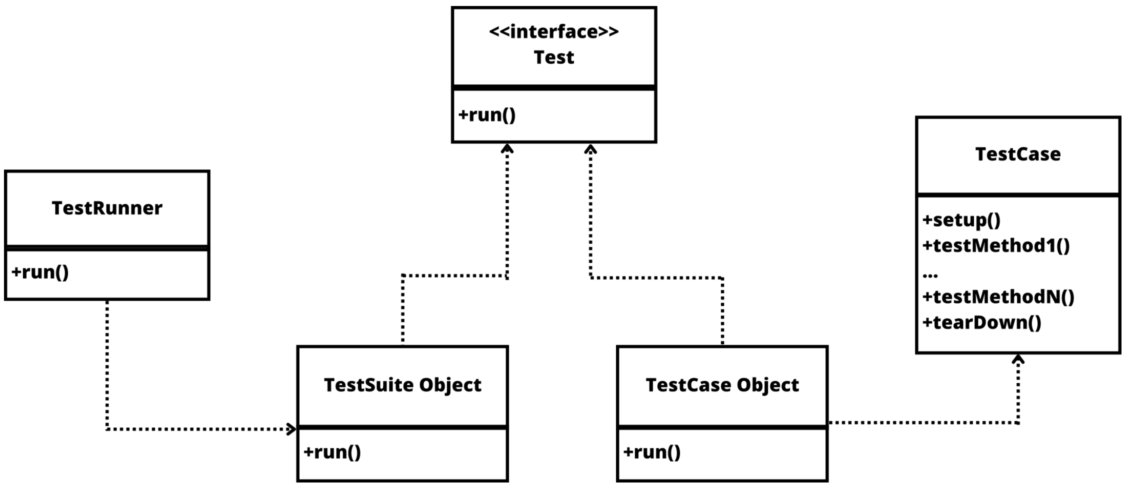


Figure 2.5: UML Diagram of XUnit's Architecture[20]

2.5.2 Mockito

There are many mocking tools for Java language including Mockito, JMock, Mock, or Mocker. Mockito is the most widely used mocking framework. It is derived from EasyMock and one of the main advances is that it is integrated into Spring Testing by default. Mockito dynamically generates proxy objects for each mocked class or object using CGLib handing back pre-designed results. [15]

Mockito architecture is based on a proxy design pattern and uses CGLib to create proxy stubs. [15]

Another reason why using Mockito is a great option is that Mockito uses lenient mocks by default.

The difference between strict and lenient mocks is noticeable when unexpected interaction happens. After this unexpected interaction occurs, strict mocks result in a test failure. On the other hand, while using lenient mocks, tests do not fail and warnings happen. [27]

To demonstrate how Mockito works let's consider this situation. **MethodX** calls **methodY** in itself, therefore methodX is dependent on methodY. Listing 2.1 shows how Mockito verifies that methodY is invoked after calling methodX.

```
// Create a mock object for the Y class
Y mockY = Mockito.mock(Y.class);
// Create an instance of X, passing the mock object Y
X x = new X(mockY);
// Call method X
x.methodX();
// Verify that method Y was called after method X
Mockito.verify(mockY, Mockito.times(1)).methodY();
```

Listing 2.1: General Example of Testing Method's Dependencies with Mockito

2.6 Microservice Architecture

This Section introduces microservice architecture, a new to design software applications, and also the difference relative to the traditional monolithic applications.

2.6.1 Microservices

Microservices can be set up as individual, standalone applications and deployed to either bare-metal or virtualized hardware. [31] Unlike the monolithic style, the microservice architecture system is composed of parts called services. These services can be developed and deployed independently and each of these microservices deals with one task so the business logic can be divided into small maintainable tasks. [3] In a monolithic application, components interact with each other via language-level methods or function calls, whereby in microservice architecture inter-process communication takes place. [28] The difference between these architectures is displayed in Figure 2.6.

Many enterprise companies adopted the microservice architecture - Amazon, Netflix, and Uber to name a few. One of Werner Vogels's arguments, why Amazon switched microservices, is that it gives a level of isolation between each of its pieces of software that allows the company to build individual parts of their software independently and much quicker [3].

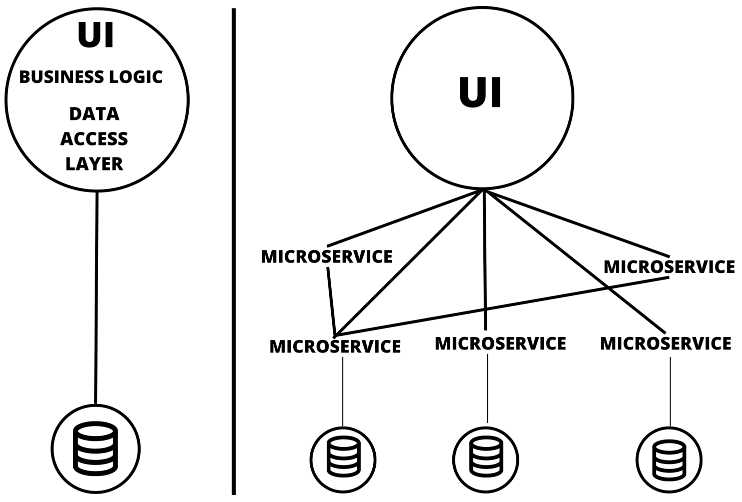


Figure 2.6: Difference Between Monolithic Architecture and Microservices

Microservices offer several advantages over traditional architecture types but they also come with several drawbacks. Table 2.3 demonstrates the benefits and drawbacks of using such architecture.

To pinpoint some other drawbacks of microservices, the main idea of this architecture is the division of the whole system into small units, which can seem tricky in terms of orchestration. To overcome this issue, containerization comes in place. [24]

Benefits	Drawbacks
Better understanding and maintenance.	Increased complexity.
Fast deployment pipeline.	Lack of security.
Parallel development.	Possible services redundancy.
Relationship to the business.	Difficult to move code between microservices.
Team autonomy.	Harder debugging process.

Table 2.3: Benefits and Drawbacks of Microservice Architecture

Chapter 3

Selected System Under Test

For the purpose of testing, it is necessary to select the correct SUT. In order to maximize the value of the solution, the system criteria were carefully selected while taking into account many aspects such as applicability, expandability, business priorities, and testers' capabilities. The final system selected for this purpose has the following characteristics:

- system with microservice architecture—application is broken into small independent services,
- multi-tenant system—allowing clients to use a single application while keeping their data separate,
- real-world system—system in actual use,
- continuous integration system—code changes are frequently integrated, automatically built and tested,
- containerized system—packaging the application and dependencies into containers that can run on different environments.

EAS (Effective Agenda System) framework owned by the company is a system fulfilling all the mentioned characteristics.

The following information about the SUT is acquired from the EAS documentation available for internal employees only.

3.1 About EAS

The company is focusing on creating solutions for its clients by developing new information systems. As many of these systems share several microservices, it has been very beneficial to create a system that would allow sharing them seamlessly and efficiently.

EAS depicted in Figure 3.1, a successor of UAS (Universal Agenda System), is an upgraded business system containing applications with the knowledge acquired from using UAS.

It is an information system of microservice architecture written mainly in Java, TypeScript, and PHP but can be applied to any other language allowing rapid scalability, deployment, and maintenance. Each microservice either works separately or communicates with a limited number of other microservices.

It also contains third-party microservices such as PostgreSQL which is a microservice that provides a secure, easy-to-use web-based interface for managing and accessing a PostgreSQL database.

The second third-party microservice contained as a part of EAS is indexing in Elasticsearch to allow storing and effectively retrieving data from a distributed cloud-based system. It can also be used for log management, data analytics, and other applications.

Microservices in EAS can be divided into frontend, backend-oriented services, or a combination of both. Backend microservices are written in Java and use the shared library **Common**, while frontend microservices are written in Typescript and use the **Common-web** library. It is also possible to have microservices in the EAS that do not use the shared library or are written in other languages, such as PHP.

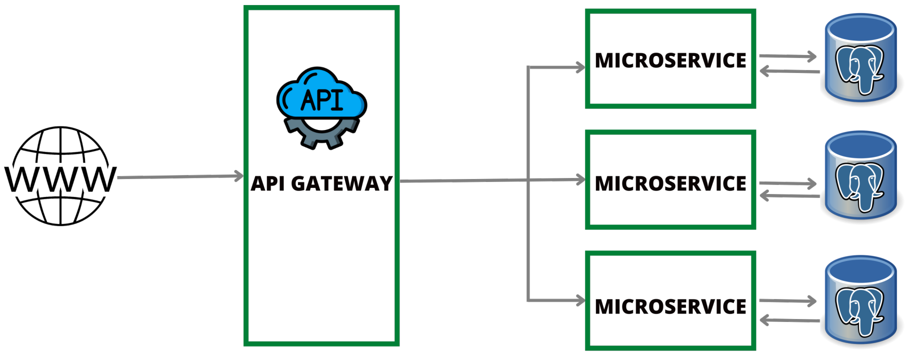


Figure 3.1: EAS Architecture

3.2 Backend Microservices Specifications

In this work, the main focus lies in testing backend (BE) microservices. These modules are built by Gradle script.

An important feature enabled by the script is Layered Jars that create independent layered executable JAR files. By default the following layers are created by the script:

- dependencies,
- spring-boot-loader,
- snapshot-dependencies and
- application.

Built layered JAR file is then copied to Docker which results in a basic container with JAR, ready to run.

When change occurs in the lower-level layer, all the upper-level ones have to be rebuilt too. The architecture is shown in Figure 3.2. Therefore application layer is on the top. When a change is made to the source code, all the dependencies and loader remain cached which results in reduced startup time.

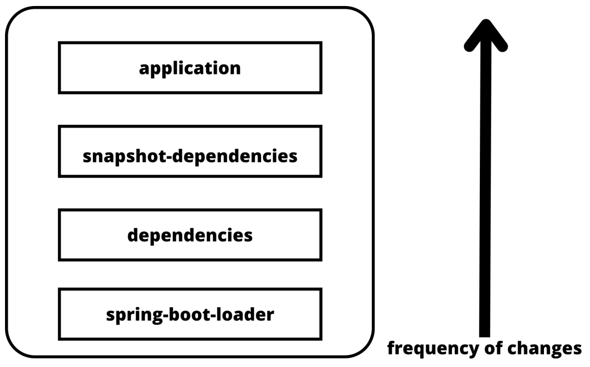


Figure 3.2: Layered Jar Architecture

3.3 Required Tools

EAS applications have basic requirements for deploying standalone applications and subsystems, including Java, NodeJS, PHP, and others. Another important requirement is the ability to specify version numbers in a formal and unified way and provide release management, which is handled by SEMVER (semantic versioning).

The EAS framework enables developers to create applications using modern technologies like Docker, Docker Compose, and Gradle. Docker Compose is used to configure microservices that are hosted in containers, and Gradle is used to build the application and manage dependencies.

3.4 Architecture of The System

EAS is a microservice architecture system consisting of frontend (FE) and BE services. Microservices and their architecture descriptions are stated in Section 2.6. This part focuses on how EAS microservices communicate with each other, and the specific aspects of EAS with an emphasis on critical parts.

3.4.1 Containerization in EAS

EAS uses HyperV and VMware as part of operating service virtualization and hence the deployment of the code is more efficient since it allows various services to be run in units that are resource-independent. Containers are a natural option for microservices-based applications because of their smooth integration with container orchestration platforms. [24] This fact was the main reason for choosing it as a solution for the EAS framework.

3.4.2 Communication Between EAS Microservices

Communication between microservices is based on RESTful API which is a web-based API that uses the HTTP protocol to transmit data between microservices.

This API is described using the OpenAPI Specification, an open-source framework. This gives developers the freedom to define the structure of the API, the types of data to exchange, and the possible operations. The OpenAPI Specification can be used to generate a Swagger UI, which provides a graphical interface for developers to interact with the API.

This allows developers to easily send requests and get responses from the microservices. The communication between microservices is secure and is based on authentication and authorization protocols. The visualization of the communication is visible in figure 3.3 and it also clearly demonstrates the difference between monolithic and microservice architecture systems.

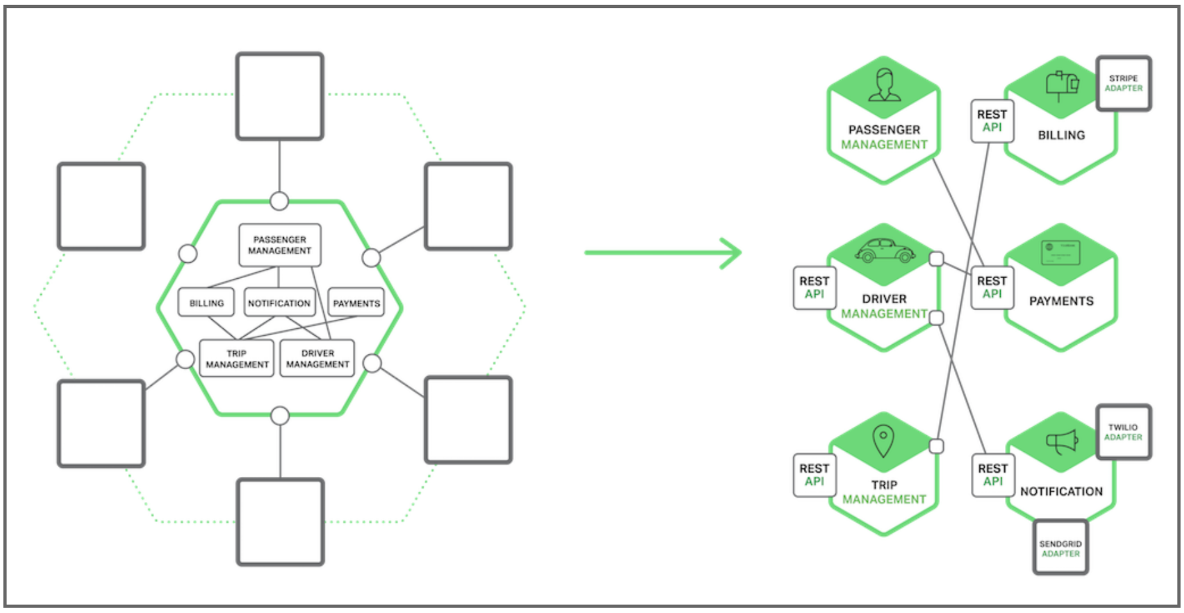


Figure 3.3: Inter-process Communication in Microservice Architecture [28]

3.5 Aspects of The System

The system stores information about the creation, last update, and deletion of objects, as well as information about the author of the objects. The database and indexing support multiple languages, allowing for flexibility in data management.

Evidence of the objects by institute makes the system multi-tenant, meaning it's sharing a single instance of executable software while isolating the data and business process serving each tenant, in this case, the customer.

In Figure 3.4 is displayed how is this architecture different in opposite to single-tenant system.

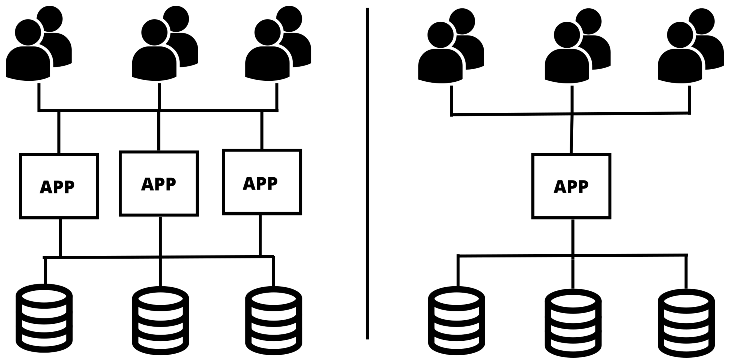


Figure 3.4: Difference Between Single-tenancy and Multi-tenancy

The files are saved on a hard disk drive as an optimized structure created by the UUID file, so that the effect of 'bucketing' of files is achieved and the problem of many files in one folder is solved. This ensures efficient storage and retrieval of files.

The system works on the principle of tabular overview. It supports filtering, multiple sorting, and virtual scrolling, which is used instead of paging. Users can change the order and visibility of columns, save filters and sorting, and share the data. The system also supports full-text searching, allowing users to quickly find the information they need.

The system supports various field types such as text, number, floats, select, autocomplete, checkbox, textarea, and editor with the support of highlighting, date and time without picker, table of dependent objects, and button. This allows for a wide range of data to be captured and managed in the system.

3.6 Future Development

It is planned to create a subsystem of server-side actions in the 'name, code, script' format with the possibility to set permissions for who can execute this action. This should be callable from BE.

For future development, it is also planned to implement a subsystem for workflow established on Activiti BPMN with the support of BPMN 2.0. The graphical definition will likely be done outside of the system in a desktop application.

After that, the system will receive a fully responsive UI, with a menu upon Grid or menu upon detail, as well as an optional floating action menu or hamburger menu. The main menu will also be included.

These are only a few of the possible changes to the system. Each task should be properly planned to prevent any problems especially when it is a complex architecture like this.

3.7 Current State of Testing EAS

The current state of EAS testing is minimal, with only a few of the BE microservices being tested. Even those that have been tested have not been done so thoroughly. The tests are written in Java and include test classes for testing Elasticsearch filters, MultipleFieldsEntity, and its repositories, as well as simple tests for checking if the TestBase is initialized and the database connection is working.

The system also includes tests for Date Utils, multiple filter tests such as AndFilter, ContainsFilter, EndWithFilter, and others, testing for field and index sort, DatedRepository, KeyValue, and Multiple. The Dictionary microservice is also tested.

Out of 45 BE microservices, only 4 of them are covered which makes test coverage of BE microservices pretty low and insufficient as can be seen in Figure 3.5. Overall test coverage of BE microservices is attached in Appendix B.

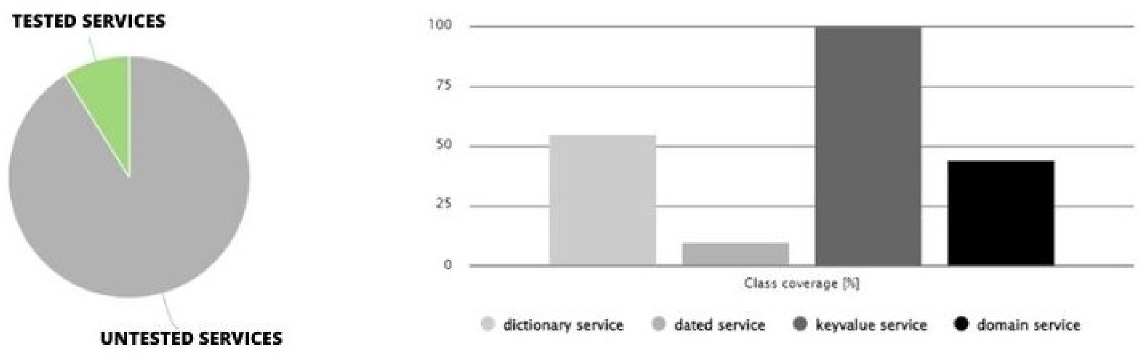
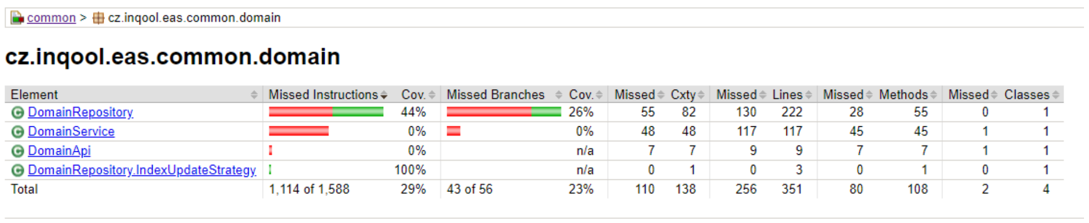


Figure 3.5: Current Testing State of EAS

In order to address this issue, the development team should focus on increasing the test coverage of the BE microservices. This can be done by implementing Test-Driven Development practices demonstrated in section 5.6 and by dedicating more resources to testing.

3.7.1 Test Coverage

This metric can express the level of the system being covered by tests. The coverage can be shown by the popular Java library JaCoCo. It can show the code coverage of each microservice under test. For example, class coverage of the domain service is shown in the figure below 3.6.



common > cz.inqool.eas.common.domain

cz.inqool.eas.common.domain

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
DomainRepository		44%		26%	55	82	130	222	28	55	0	1
DomainService		0%		0%	48	48	117	117	45	45	1	1
DomainApi		0%		n/a	7	7	9	9	7	7	1	1
DomainRepository.IndexUpdateStrategy		100%		n/a	0	1	0	3	0	1	0	1
Total	1,114 of 1,588	29%	43 of 56	23%	110	138	256	351	80	108	2	4

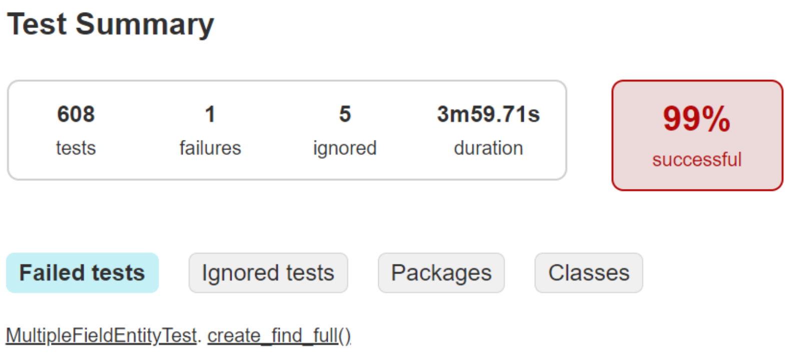
Figure 3.6: Domain Service Class Coverage

The tests are not well-documented and do not contain any comments which is problematic because, in a microservice architecture, it is important to ensure that the services are maintainable, traceable, reusable, and mainly debuggable. These services are often developed by different developers so having fixed naming and commenting conventions for microservices is crucial to ensure that others can understand and navigate the services when changes need to be made. This allows developers to easily edit the tests and understand their purpose, which helps maintain the system's functionality and integrity. Furthermore, well-documented tests can be reused for regression testing and testing different versions of the service, saving time and resources.

Microservices are not implemented by Test-Driven Development which can significantly impact the overall quality and stability of the system in a negative way. Without the use of TDD, there is a higher likelihood that bugs and other issues will go undetected until later stages of development, or even after deployment. It can also be difficult to identify and fix issues that arise when the services are combined. This can lead to more downtime and a less stable system overall.

The current test suite of the system consists of approximately 600 unit tests with the overwhelming majority being filter tests.

The overall test summary is displayed in Figure 3.7



Test Summary

608	1	5	3m59.71s	99%
tests	failures	ignored	duration	successful

MultipleFieldEntityTest. create_find_full()

Figure 3.7: Test Report

3.8 Sensitive Aspects of EAS

EAS has been created by multiple developers and is being used by a variety of employees, and as any other system has its drawbacks and areas for improvement. The following list is depicting the main aspects of the system that are sensitive, meaning it is crucial to take them into consideration when testing the system. Omitting their importance might result in creating faulty solutions. Sensitive aspects are as follows:

- containeriaztion,
- RESTful communication between microservices,
- data consistency,
- security and
- fault tolerance.

3.8.1 Containerization

Given that the EAS has a microservice architecture and is using Docker for containerization, it is important to test the configuration of the containers to ensure they are correctly set. Additionally, communication between the containers must be secure and efficient. Monitoring the containers should also be a priority to quickly identify any issues that may arise and address them properly. Lastly, the deployment process should be tested to ensure its reliability.

3.8.2 RESTful Communication Between Microservices

This system relies on RESTful communication between microservices for efficient, maintainable, and standardized architecture. However, this also poses potential risks such as security vulnerabilities due to inadequate authentication and authorization. Third-party access is a particular area of concern. Additionally, the failure of one service can have a domino effect, causing cascading failures that can lead to a widespread disruption of the system. Finally, the system may become overloaded, leading to a decrease in performance or a complete shutdown.

3.8.3 Data Consistency

Microservice architecture has the potential to create major risks, such as data inconsistency due to its asynchronous structure. With data distributed across many microservices, errors can be difficult to identify and repair. Furthermore, it may take time for updates to be propagated throughout the system. If one of the services malfunctions, data could be lost, resulting in damaged integrity.

3.8.4 Security

In terms of security, injection attacks can be used to gain access to or modify a database within the EAS, as well as exploit weak authentication controls to gain access to its architecture. Therefore, it is important to ensure that appropriate logs and monitoring systems are in place to detect any suspicious activity.

3.8.5 Fault Tolerance

Fault tolerance is critical for any framework to ensure minimal disruption in case of any internal or external failure. To achieve this, concrete measures such as redundancy, fault tolerance modeling, and automated testing must be implemented.

When using a microservice architecture, fault tolerance testing is even more important since each service is a separate unit that is responsible for different tasks. Testing should include both individual services to handle failure scenarios and the overall system to ensure it can handle multiple service failures.

Chapter 4

Testing Strategy for Selected System Under Test

The test plan for the EAS will mainly contain parts that should be in a test plan according to IEEE 29119-3 standards.

A company that owns the EAS has recently hired 3 testers, out of which none have experience in test automation and they do mainly manual testing. Testers have to test 44 BE microservices, with 4 already tested. The goal of testing is to test at least one-third of the BE microservices being covered with unit tests so at least 10 more microservice tests should be created. It is suitable to use an automated test suite of unit and integration tests to ensure that nothing breaks with new changes made to the code.

4.1 Test Items

In this work, the functionality of ten specific BE microservices from the **common** module is tested:

- reporting microservice—used for reporting,
- certificate microservice—used the for creation of certificates,
- sequence microservice—used for generating sequences,
- mail microservice—used as a mail service,
- schedule microservice—used for scheduling,
- template microservice—used for creating templates,
- storage microservice—used for file management,
- pdfa microservice—used for converting files to PDF format,
- intl microservice—used for translation processes and
- multi-string microservice—used for working with multi-strings.

Each microservice needs to be tested in isolation and other dependent microservices are to be replaced by mocks. The EAS testing automation should start from the bottom layers of the V-model described in Section 2.2, so the focus will be mainly on the unit and integration tests.

4.2 Software Risk Issues

Due to the modular nature of the EAS, there is a higher security risk. System exposure to the network causes the possibility of attacks from outside. The dependencies between microservices allow weak spots to appear quickly in the system.

4.3 Features to Be Tested

Reporting Service

- Verify that the Service can list allowed definitions.
- Verify that the Report can be acquired by the definition.
- Verify that the Report is correctly generated.

Certificate Service

- Verify that the Service can create a Certificate object.
- Verify that the Service can update the Certificate object.
- Verify that the Service can delete the Certificate object.
- Verify that Resource can be acquired by code.

Sequence Service

- Verify that the next value of the sequence can be generated by Id.
- Verify that the next value of sequence can be generated by Sequence's code.
- Verify that the sequence can be updated.

Mail Service

- Verify that Mail Service is able to create Mail object.
- Verify that input parameters are assigned to Mail object.
- Verify that Mail object can be retrieved from the Queue.
- Verify that Mail object can be updated with new parameters.

Schedule Service

- Verify that the Service can create scheduled jobs.
- Verify that the Service can update scheduled jobs.
- Verify that the Service can delete scheduled jobs.
- Verify that the Service can change the job into running state.
- Verify that the Service can access times of last and next run of specific job.
- Verify that the Job can start successfully.
- Verify that the Job can stop successfully.

Template Service

- Verify that Template is in Cache.
- Verify that the Template is removed from the cache.
- Verify the computation of fingerprint.
- Verify the creation of a new Template.
- Verify that the new Template is stored.

Storage Service

- Verify that all unsupported and unimplemented methods throw an exception.
- Verify that all File's attributes can be accessed.
- Verify conversion between File object and string.
- Verify correct behavior when the File is null.

Pdfa Service

- Verify that the Service can convert other data types to pdf.
- Verify that the Service can change extensions.
- Verify that the Service can return Content type.
- Verify the correct behavior when the Converter is null.

Intl Service

- Verify that the Service can create different Translations.
- Verify that the Service can update translations.
- Verify that the Service can delete translations.
- Verify that the cache can be successfully evicted.

Multistring Service

- Verify that Service returns correct type.
- Verify that Service can convert String data to entity attribute.

4.4 Features Not to Be Tested

- Creation of basic data objects.
- Filtering of objects.
- Constructors.

These features will not be tested because they are properly tested already and are used as a part of almost every other test case.

4.5 Test Approach

Testing tools that will be used to implement tests in an automated test suite will be JUnit5 as a unit-testing framework and Mockito to inject mocked dependencies in the test class.

JUnit5 should not require special training because Java developers of EAS are used to implementing unit tests using this framework. Mockito is not used by the developers but in general, is closely used with the JUnit5 framework so it should not be difficult to incorporate mocking into the tests. Some of the metrics to be acquired from the test suite:

- total number of test cases,
- number of test cases passed,
- number of test cases failed,
- number of test cases ignored,
- total execution time of the test suite,
- class testing success rate,
- package testing success rate,
- test coverage and
- severity of found defects.

Other metrics can be derived from these. Based on these metrics and mainly on the severity of the found defects, it will be decided how much regression testing should be done. Ideally, regression testing whether manual or automated should be done with every product change. [10]

All of these metrics will be aggregated in the final report generated after the execution of the automated test suite of the EAS.

The main coverage requirements will be class coverage of individual services expressed as a percentage.

4.6 Pass/Fail criteria

- All tests should be executed.
- All requirements from Section 4.1 should be tested.
- No critical defects found by the automation test suite.

4.7 Test Deliverables

Testing deliverables that come with this work:

- test plan,
- test case specification and
- test results reports.

4.8 Risks and Contingencies

Implementation of the test suite should be aware of risks described in Table 4.1 and follow the suggested contingency strategies.

Risk	Contingency plan
Lack of testing staff.	Hiring more personnel.
Multiple dependencies on other microservices.	Mocking of test classes.
The complexity of the system.	Combination of unit and integration tests to ensure overall functionality.
Sensitive data exposure.	Test data do not contain any sensitive information.

Table 4.1: Testing Risks and Contingency Plans

4.9 Cost-Benefit Analysis

To see clear results of analysis all of the benefit factors should be converted into a single unit of comparison. [25] For the purpose of this work we will choose time [h].

Let's say we would like to test 5 microservices each testable by approximately 10 test cases so 50 test cases in total.

Manual approach	
Total number of test cases	**50**
Average time spend designing per test case	**0.4h** [6]
Total time designing test cases	$50 * 0.4 =$ **20h**
Average time spend executing per test case	**0.05h** [6]
Total time executing test cases	$50 * 0.05 =$ **2.5h**

Table 4.2: Manual Testing Time

According to the test results from Figure 3.7 600 tests passed in approximately $\frac{1}{15}$h so 1 test case last $\frac{1}{15} \div 600 = \frac{1}{900}$h.

Automated approach	
Total number of test cases	**50**
Average time spend designing per test case	**1h** [6]
Total time designing test cases	$50 * 1 =$ **50h**
Average time spend executing per test case	$\mathbf{\frac{1}{900}h}$
Total time executing test cases	$50 * \frac{1}{900} = \mathbf{\frac{1}{180}h}$

Table 4.3: Automation Testing Time

The results displayed in Tables 4.2 and 4.3 say that even though designing the automation scripts take a long time in comparison with the manual tests, the execution is much faster.

The initial time and cost investment seem resource-heavy and overwhelming but with automated testing, the return on investment is visible after some time and is also more cost-saving in the long run as displayed in Figure 4.1.

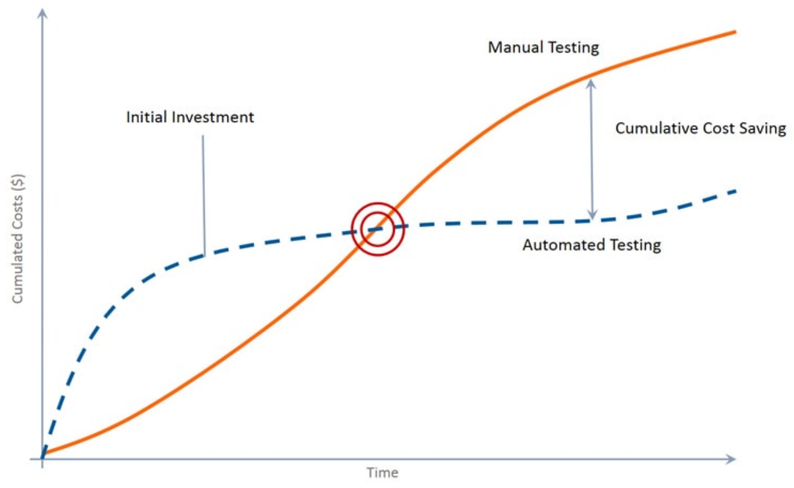


Figure 4.1: Ratio Between Time and Cost of Manual and Automated Testing [11]

With automation comes new testing activities for testers like test scripting, maintenance of test scripts, evaluating test results, and many more according to Figure 4.2. With the current number of EAS testers and their time and cost allocations, it is recommended to increase these values.

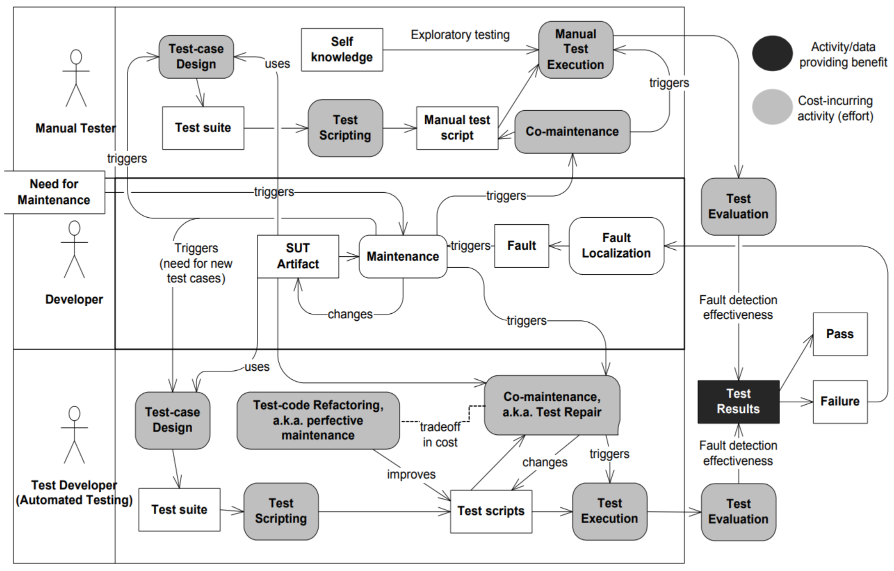


Figure 4.2: SW Testing Reference Process [25]

Chapter 5

Implementation of Tests for Microservices in EAS

This part contains the implementation parts consisting mainly of testing the existing code described in Section 5.4 and Test-Driven Development process outlined in Section 5.6.

5.1 Used Tools

This section of the work describes all the tools that were used while creating unit and integration tests. The most important tools Junit5 and Mockito are described in Section 2.5.1.

5.1.1 PostgreSQL

While executing the integration tests a testing database is created and filled with the test data, so SQL queries and work with the database are properly tested. For viewing the schemes of the database pgAdmin is used, which is a graphical user interface for PostgreSQL management and visualization of the data.

5.1.2 Hibernate

Hibernate is a framework used to map Java objects to relational database tables and then proceed to operate on them. In the SUT, it interacts with the application's database entities through classes' annotations.

5.1.3 Liquibase

Liquibase is another database-related tool that allows the management of the test database schema and provides a way to define changes through the XML configuration file where it is possible to define all the data needed for the testing.

5.1.4 Elasticsearch

To handle large amounts of test objects, methods, and other data it is desirable to be able to quickly search, index, and filter needed information.

As a part of this thesis, Elasticsearch is used for this purpose, since it is a widely used open-source engine. It is relatively fast and accurate. EAS is used mainly for full-text search and indexing of data objects.

The Elasticsearch stack consists of Elasticsearch, Logstash, and Kibana(ELK). In EAS, Kibana is used for data visualization and manipulation via a graphical interface. [9]

5.1.5 Redis

Redis which stands for Remote Directory Server is a data structure storage while also being caching server providing different data structures and data types like strings, hashes, lists, sets, sorted sets, bitmaps, geospatial indexes, and more. This particular storage was chosen to allow the user to run atomic operations on these types, but it also works with in-memory datasets. [23]

Most databases store their data on the database server disk but Redis data stays in memory. That is making this solution profitable in real-time applications and message queuing systems. [18]

5.2 Test Data Management

Test data management in EAS uses all the tools mentioned above when integration tests are executed. The whole process is depicted in Figure 5.1. The test data is primarily stored in PostgreSQL which uses Liquibase to create and initialize the database schema. This database is populated using Hibernate. Redis is used as a caching layer to reduce the number of database queries. The final data can be accessed in the tables via the Kibana interface.

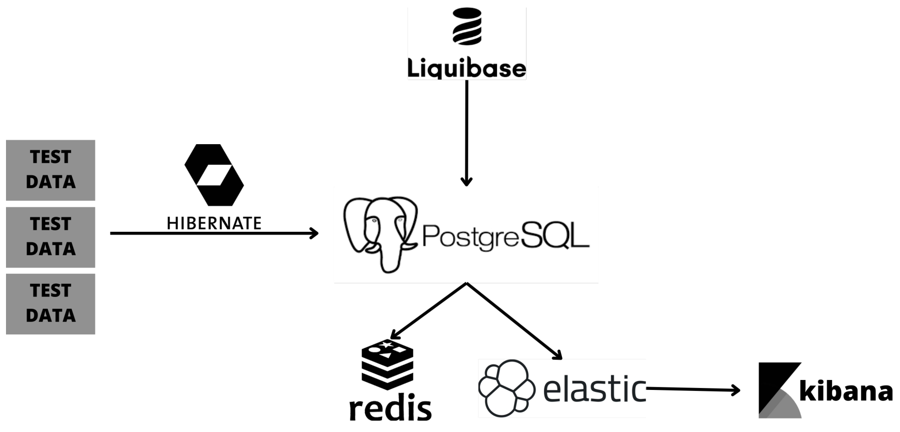


Figure 5.1: EAS Test Management System

5.3 Test Base

To uniformly test EAS, there is a Java class called TestBase designed to test applications that use PostgreSQL, ElasticSearch, and Redis. It sets up Docker containers for each of these services with specific configurations such as the database name, username, and password for the PostgreSQL container.

This class also provides URLs and credentials needed to connect to the Docker containers, however, test containers will automatically shut down after tests.

To ensure that previous test executions do not interfere with other tests, the Elasticsearch indexes are dropped and recreated before each test. It also tests Elasticsearch's data indexing and deletion functionality.

This test base is used as a base for integration tests and it closely resembles the production environment to find also environment-related bugs along with functionality bugs.

5.3.1 Common Test Base

This test class extends the test base in Attachment 5.3 and is primarily used for testing the BE microservices of EAS. It declares that the tests will be Spring Boot tests.

Integration tests should be separated from the unit tests mainly because of higher time consumption and also because they need an actual database to work with.

Common Test Base also contains a test initializer that triggers auto-configuration of the services and component scanning.

5.4 Testing Existing Code

A few problems have occurred while testing existing microservices. Most of the code contains mixed concerns which means that the code is doing more than one task. Another problem is related to the multiple dependencies across the system. In order to solve these issues, extraction methods and mocking are used.

5.4.1 Extraction Method

As stated in the description of unit testing in Section 2.3.2 the purpose of these tests is to test a single unit of code.

To solve this problem a technique called extraction was used while developing unit tests. This method moves code to new methods, classes, or functions. For example, as we can see in the code snippet 5.1, the function **generate** contains the lambda function.

```
public synchronized String generate(@NotNull String sequenceId) {
      TransactionTemplate transactionTemplate = new TransactionTemplate(
         transactionManager);
      transactionTemplate.setPropagationBehavior(TransactionDefinition.
         PROPAGATION_REQUIRES_NEW);

      return transactionTemplate.execute(status -> {
      Sequence sequence = repository.find(sequenceId);
      Long counter = sequence.getCounter();

      sequence.setCounter(counter + 1);
      repository.update(sequence);

      DecimalFormat format = new DecimalFormat(sequence.getFormat());
      return format.format(counter);
      });
      }
```

Listing 5.1: Function generate

To isolate this function an extraction method is used and the newly created method displayed in code 5.2 is now suitable for unit testing.

```java
String getFormat(String sequenceId){
      Sequence sequence = repository.find(sequenceId);
      Long counter = sequence.getCounter();

      sequence.setCounter(counter + 1);
      repository.update(sequence);

      DecimalFormat format = new DecimalFormat(sequence.getFormat());
      return format.format(counter);
}
```

Listing 5.2: Method getFormat

5.4.2 Mocks

Testing code that calls another code can be difficult and it also conflicts with the idea of unit testing as described in Section 2.3.2. To handle this problem unit test creates a mock of the dependency and provides it to the code under test. This process is called dependency injection. For creating mocks or test doubles in EAS, the Mockito tool described in Section 2.5.2 is used, an example of dependency injection in our test suite can be seen in the code 5.3 where classes generator and repository present mocked objects.

```java
@ExtendWith(MockitoExtension.class)
public class SequenceServiceTest {

   private static SequenceService service;

   @Mock
   private static SequenceGenerator generator;
   @Mock
   private static SequenceRepository repository;
}
```

Listing 5.3: SequenceServiceTest class

The disadvantage of mocks is the need to update the mocks when the original dependency has changed with new methods.

5.5 Generating Test Report

The results from the automated test suite are visualized via Gradle. A failing test in the suite creates a test report located in build/reports/tests/test/index.html. By default setting you can see information about the reason test has failed and about packages and classes' success rates.

5.6 Test Driven Development of Microservice

This part of the work serves only as a simple example of Test-Driven Development described in Section 2.4. It covers all parts including designing the tests, implementing the new service, making tests pass, refactoring, and repeating this cycle displayed in figure 5.2 until the whole implementation is done.

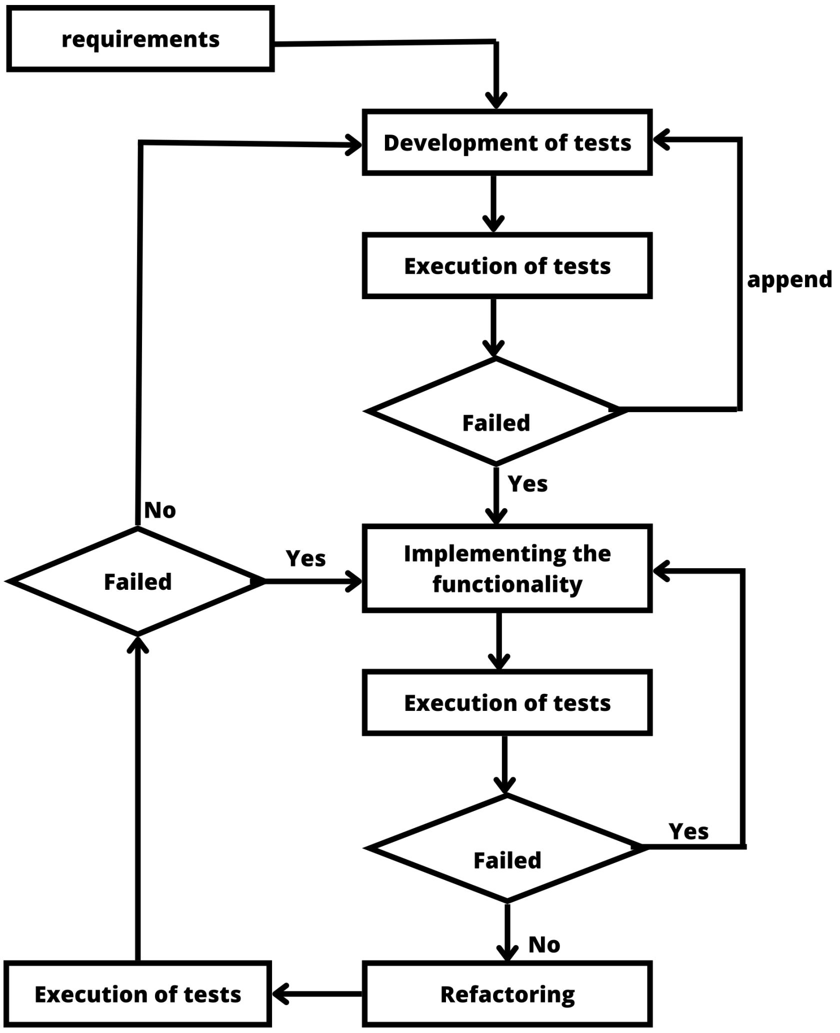


Figure 5.2: Test Driven Development Cycle

5.6.1 Description of Expression Microservice

The developed service is a simple microservice that is able to solve complex math expressions with the help of a stack that converts infix expressions to postfix expressions and returns the correct result.

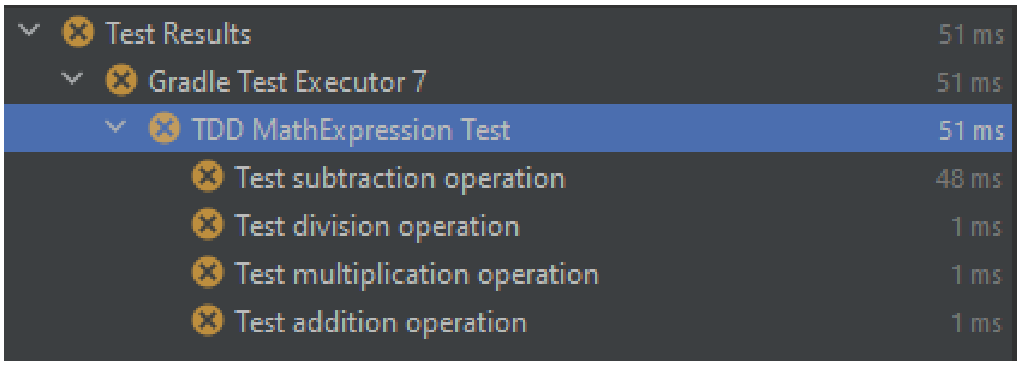
The simplicity of the Service is for the purpose of easier understanding of the development process. This part of the work can serve as a guide for the developers when developing a new microservice into the system.

5.6.2 Design of the Initial Tests

Just as the features to be tested are defined in Section 4.3, it is also required to implement these new features:

- Verify that the Service is able to perform Addition.
- Verify that the Service is able to perform Subtraction.
- Verify that the Service is able to perform Multiplication.
- Verify that the Service is able to perform Division.
- Verify that the Service is able to perform Exponentiation.
- Verify that the Zero Division Operation throws an Exception.
- Verify that invalid operands and operators in input expression throw an Exception.
- Verify that the Service is able to convert an infix expression into a postfix expression.
- Verify that the Service is able to evaluate complex expressions.

Firstly, the focus will be on primary mathematical operations including addition, subtraction, multiplication, and division. It can be seen that the initial tests in Figure 5.3 failed because the basic mathematical operations were not implemented yet.



```
Test Results                         51 ms
  Gradle Test Executor 7             51 ms
    TDD MathExpression Test          51 ms
      Test subtraction operation     48 ms
      Test division operation         1 ms
      Test multiplication operation   1 ms
      Test addition operation         1 ms
```

Figure 5.3: Initial Failed Tests

5.6.3 Implementation of The Code

After seeing the initial tests fail it is time to implement these features and retest the functionality again. In this case, it is sufficient to implement just basic mathematical operations and execute the test suite again. All of the original failed tests in Figure 5.3 have now passed.

5.6.4 Refactoring of The Code

An initial implementation with just the intention to make the initial tests pass may look like the one in Listing 5.4.

```
public double eval(String expression) {
      final String operator;
      final String[] operands;
      if (expression.contains("+")) {
         operator = "+";
         operands = expression.split("\\+");
      } else if (expression.contains("-")) {
         operator = "-";
         operands = expression.split("\\-");
      } else if (expression.contains("*")) {
         operator = "*";
         operands = expression.split("\\*");
      } else if (expression.contains("/")) {
         operator = "/";
         operands = expression.split("\\/");
      } else
         throw new IllegalArgumentException("Invalid expression");

      switch (operator) {
         case "+":
            return Double.parseDouble(operands[0]) + Double.parseDouble(
               operands[1]);
         case "-":
            return Double.parseDouble(operands[0]) - Double.parseDouble(
               operands[1]);
         case "*":
            return Double.parseDouble(operands[0]) * Double.parseDouble(
               operands[1]);
         case "/":
            return Double.parseDouble(operands[0]) / Double.parseDouble(
               operands[1]);
         default:
            throw new IllegalArgumentException("Invalid expression");
      }
   }
}
```

Listing 5.4: First Implementation of Eval Method

After seeing the test pass it is recommended to rethink the effectiveness and logic of the code and write cleaner more understandable code like the one in Listing 5.5 with the help of known techniques.

```
public double eval(String expression) {

    final var expression = Expr.of(expression);

    switch (expression.operator) {
      case ADD:
        return expression.Operand1 + expression.Operand2;
      case SUBTRACT:
        return expression.Operand1 - expression.Operand2;
      case MULTIPLE:
        return expression.Operand1 * expression.Operand2;
      case DIVIDE:
        return expression.Operand1 / expresssion.Operand2;
    }
    throw new IllegalArgumentException("Unsupported operation");
  }
```
Listing 5.5: Refactored Implementation of Eval Method

If the tests pass even after the refactoring, the design of the tests of other features can take place.

5.6.5 Final Microservice

This process of designing tests, implementing, and refactoring is repeated until all of the initial requirements are fulfilled.

Implementation of this service requires handling all possible exceptions, implementing an algorithm for converting infix expressions to postfix expressions with the usage of a stack, and adding other mathematical operations.

The final implementation classes and methods can be seen in Figure 5.4.

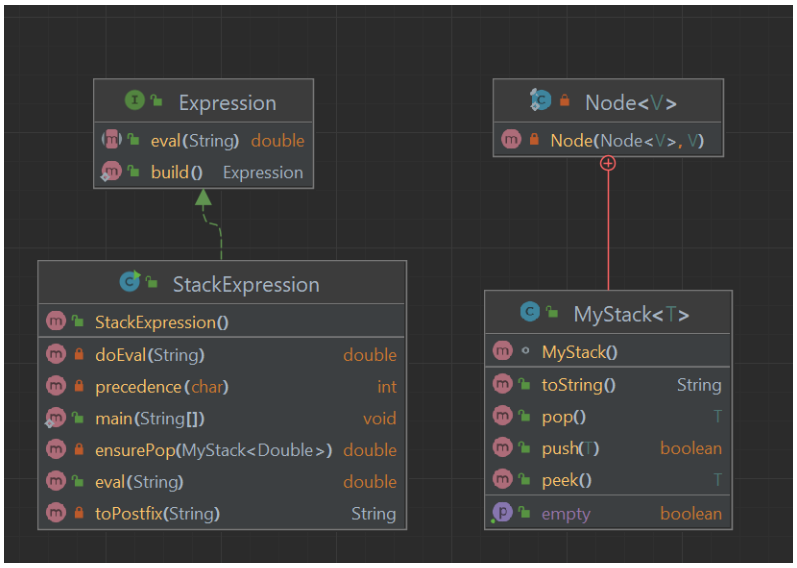
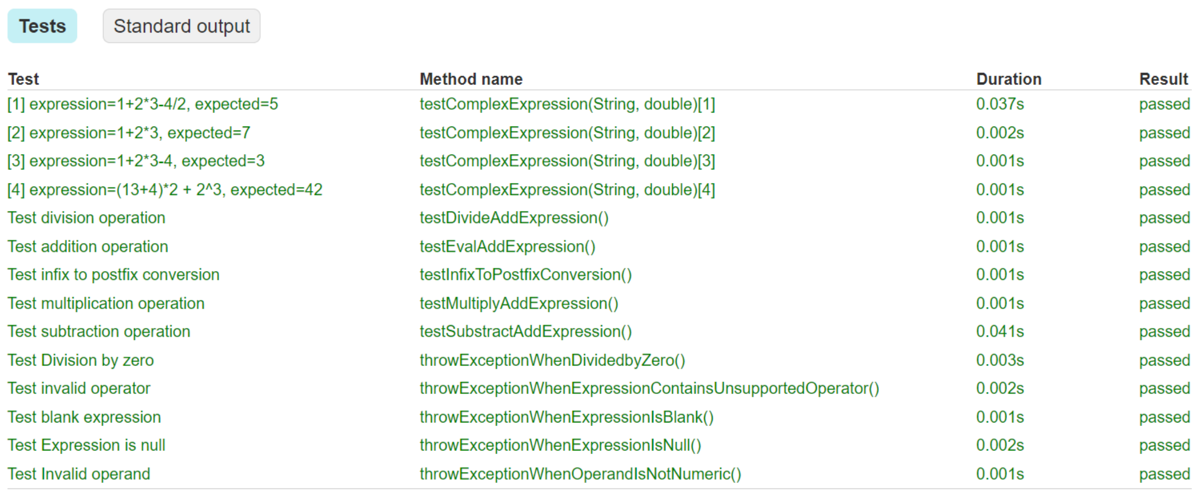


Figure 5.4: Service's Classes and Methods

The whole code of the Expression service can be accessed in the implementation. The Test-Driven Development part can be localized by Appendix A.

After successful Test-Driven Development of the new microservice, all of the tests in Figure 5.5 passed and the service is fully functional.



Test	Method name	Duration	Result
[1] expression=1+2*3-4/2, expected=5	testComplexExpression(String, double)[1]	0.037s	passed
[2] expression=1+2*3, expected=7	testComplexExpression(String, double)[2]	0.002s	passed
[3] expression=1+2*3-4, expected=3	testComplexExpression(String, double)[3]	0.001s	passed
[4] expression=(13+4)*2 + 2^3, expected=42	testComplexExpression(String, double)[4]	0.001s	passed
Test division operation	testDivideAddExpression()	0.001s	passed
Test addition operation	testEvalAddExpression()	0.001s	passed
Test infix to postfix conversion	testInfixToPostfixConversion()	0.001s	passed
Test multiplication operation	testMultiplyAddExpression()	0.001s	passed
Test subtraction operation	testSubstractAddExpression()	0.041s	passed
Test Division by zero	throwExceptionWhenDividedbyZero()	0.003s	passed
Test invalid operator	throwExceptionWhenExpressionContainsUnsupportedOperator()	0.002s	passed
Test blank expression	throwExceptionWhenExpressionIsBlank()	0.001s	passed
Test Expression is null	throwExceptionWhenExpressionIsNull()	0.002s	passed
Test Invalid operand	throwExceptionWhenOperandIsNotNumeric()	0.001s	passed

Figure 5.5: Final Tests Passed

5.7 Final Test Suite

The final test suite contains multiple tests that exercise the functionality of all the microservices that were planned to test and listed in Subsection 4.1.

It is divided into unit and integration parts due to the importance of isolation of integration tests. Having integration tests that require configurations such as the database connection and mixed together with unit tests is generally bad practice in testing. [19]

Having unit tests in separation creates a so-called **the safe green zone** [19]. It should always pass and if some tests don't pass, it is safe to say there is a real problem with functionality and not in the configuration unlike in the case of integration tests. By executing these tests developers can gain partial confidence in the code functionality.

5.8 Automation of The Test Suite

Having the test suite automated is a huge benefit in terms of creating and incorporating new functionalities into the system while ensuring that all the critical parts work properly and new changes do not interfere with the already working code.

Build configurations and build scripts are invoked by a continuous integration (CI) server's build configuration. This whole process of building and executing tests automatically is called CI [19], as displayed in Figure 5.6.

5.8.1 Build Script

This script runs all the unit tests and is meant to gain information in the least amount of time. For this case a tests with the tag **Fast** are mainly used.

The main build to automatically run our final test suite is in Listing 5.6.

```
image: gradle:jdk15

variables:
  GRADLE_OPTS: "-Dorg.gradle.daemon=false"

before_script:
  - export GRADLE_USER_HOME=`pwd`/.gradle

cache:
  paths:
    - .gradle/wrapper
    - .gradle/caches

stages:
  - build
  - test

build:
  stage: build
  script: ./gradlew --build-cache assemble
  cache:
    key: "$CI_COMMIT_REF_NAME"
    policy: push
    paths:
      - build
      - .gradle

test:
  stage: test
  script: ./gradlew check
  artifacts:
    when: always
    reports:
       junit: build/test-results/test/**/TEST-*.xml
  cache:
    key: "$CI_COMMIT_REF_NAME"
    policy: pull
    paths:
      - build
      - .gradle
```

Listing 5.6: Gitlab CI Test Build

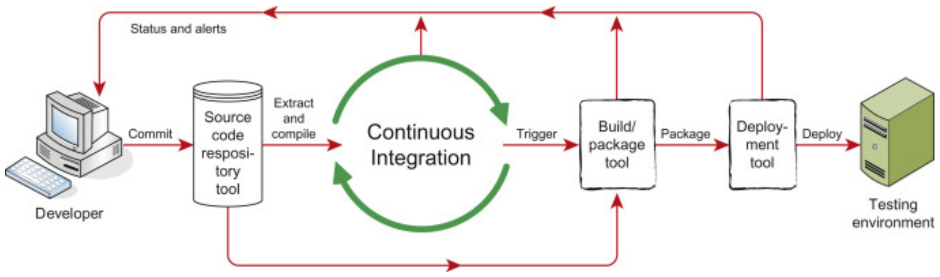


Figure 5.6: Continuous Integration Cycle [34]

5.8.2 Build Triggers

The purpose of triggers is to run the specific build script after special events occur such as deployment, time passing, and so on[19]. Build scripts for the EAS are triggered automatically along with the deployment process. There is also an option for a manual trigger available if needed.

5.9 Testing Best Practices

For better sustainability and extensibility of the test suite, several conventions were used while implementing test cases.

5.9.1 Test Labels

For better orientation in tests and also prioritization it is appropriate to mark tests with some tags. In EAS, there is an interface called **Tags** displayed in Listing 5.7 declaring several markings according to the speed of test execution and severity of functionality being tested.

This is quite useful in manipulating which tests should be run when deploying, cutting down time, and making sure that critical tests are covered. An example of Fast Tag is shown in Listing 5.8.

```java
public interface Tags {
   interface Speed {

      String SLOW = "SLOW";
      String MODERATE = "MODERATE";
      String FAST = "FAST";
   }
   interface Severity {

      String CRITICAL = "CRITICAL";
      String STANDARD = "STANDARD";
      String MINOR = "MINOR";
   }
}
```

Listing 5.7: Tags Instance

```
@FastTest
   void updateSequence() {
      Mockito.when(repository.find(ArgumentMatchers.any())).thenReturn(
         sequence);
      Long counter = generator.updateSequence(sequence.getId());
      Assertions.assertEquals(1L,counter);
   }
}
```

Listing 5.8: Example of Fast Tag Usage

5.9.2 JUnit Annotations

Junit5 offers multiple annotations that help make the code more readable and cut down the number of lines of code.

For less code **@BeforeEach** and **@AfterEach** annotations are used to execute code that repeats before and after every test to remove code duplications.

Annotation **@BeforeAll** serves to initialize the data object that is tested in Listing 5.9 and shows the correct use of these annotations.

```
@BeforeAll
   public static void init(){
      service = new SequenceService();
   }
   @BeforeEach
   void setUp() {
      service.setGenerator(generator);
      service.setRepository(repository);
   }

   @AfterEach
   void tearDown() {
      Mockito.verifyNoMoreInteractions(repository,generator);
   }
```

Listing 5.9: Junit5 Annotations

5.9.3 Naming Conventions

All of the tests in the test suite follow a unified naming system. Every class under the test has the same name ending with the word **Test**.

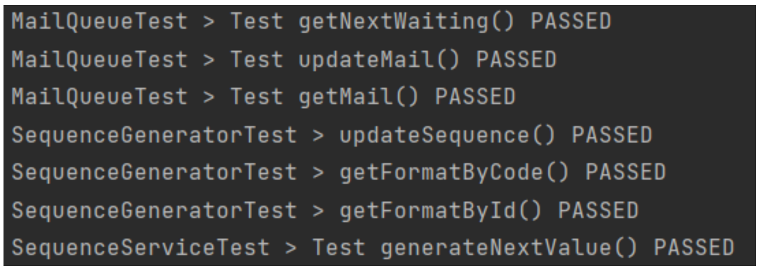
Testing methods follow the naming pattern **UnitOfWork__Scenario__ExpectedResult**.

[19] Example can be seen in Listing 5.10. It also ensures that the test is easy to understand with no need for additional comments.

```
@FastTest
void Process_ProcessingError_ThrowsException() throws IOException{}
```

Listing 5.10: Testing Naming Conventions

The IDE test report shows the class name and method name by default. Adding annotation **@DisplayName** above the test method or class results in a clear summary like the one in Figure 5.7. It also provides a better understanding for non-technical users.



```
MailQueueTest > Test getNextWaiting() PASSED
MailQueueTest > Test updateMail() PASSED
MailQueueTest > Test getMail() PASSED
SequenceGeneratorTest > updateSequence() PASSED
SequenceGeneratorTest > getFormatByCode() PASSED
SequenceGeneratorTest > getFormatById() PASSED
SequenceServiceTest > Test generateNextValue() PASSED
```

Figure 5.7: DisplayName Annotation

5.9.4 Testing Unwanted Interactions

It is important to test both test cases, by happy and unhappy paths. The designed tests check that the methods work as supposed but they also control if other interactions were not executed. Mockito can check if any of the given mocks have any unverified interaction. After each test in **tearDown()** function, this method is called. An example of tearDown method can be also seen in Listing 5.9.

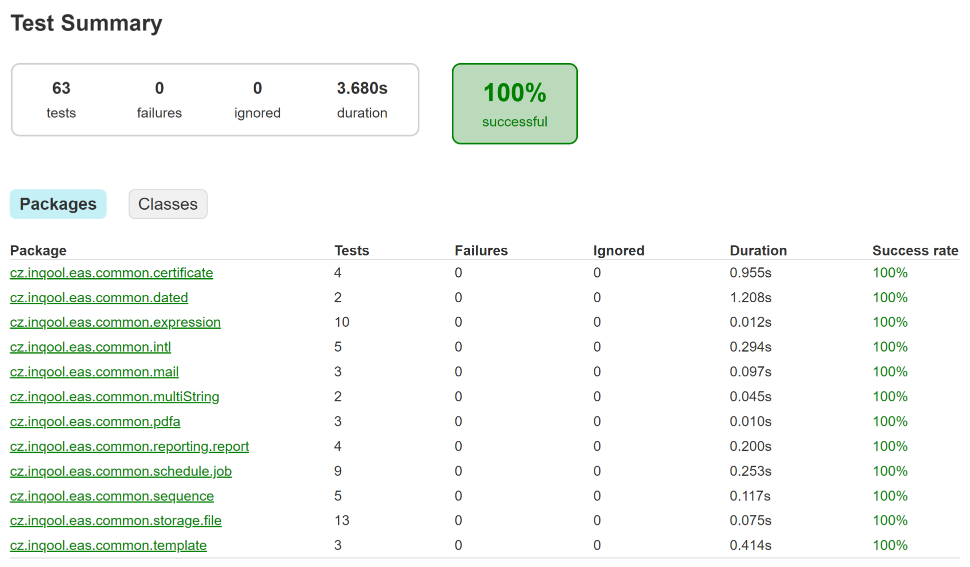
Chapter 6

Test Evaluation and Future Planning

The final testing suite did not reveal any defects and serves for controlling purpose that the tested microservices work well. If there is a failing test in the final suite it is safe to assume that the defect is caused by newly added features. BE of EAS is now more covered.

6.1 Test Metrics

Due to the complexity of the system, the final test suite covers only a part of the BE microservices, more specifically 11 selected microservices. It consists of a total of **63** tests with **100%** success rate and a total execution time of **3.680**s. The tested microservices are listed in Section 4.1 and all of the initial requirements in Section 4.3 are accomplished. All of the metrics can be seen in Gradle generated report in Figure 6.1.



Package	Tests	Failures	Ignored	Duration	Success rate
cz.inqool.eas.common.certificate	4	0	0	0.955s	100%
cz.inqool.eas.common.dated	2	0	0	1.208s	100%
cz.inqool.eas.common.expression	10	0	0	0.012s	100%
cz.inqool.eas.common.intl	5	0	0	0.294s	100%
cz.inqool.eas.common.mail	3	0	0	0.097s	100%
cz.inqool.eas.common.multiString	2	0	0	0.045s	100%
cz.inqool.eas.common.pdfa	3	0	0	0.010s	100%
cz.inqool.eas.common.reporting.report	4	0	0	0.200s	100%
cz.inqool.eas.common.schedule.job	9	0	0	0.253s	100%
cz.inqool.eas.common.sequence	5	0	0	0.117s	100%
cz.inqool.eas.common.storage.file	13	0	0	0.075s	100%
cz.inqool.eas.common.template	3	0	0	0.414s	100%

Figure 6.1: Test Metrics

6.2 Final Test Coverage

Initial test coverage values are displayed in the Jacoco report in Attachment B and have increased with the extended unit test suite and added integration tests. The new values can be seen in the newly generated Jacoco report after each execution of the test suite. Table 6.1 and Figure 6.2 represent the specific test coverage metrics and increases in their values. The overall test coverage of EAS is currently not sufficient but even the small test suite (50 test cases) increased the numbers in a significant manner and set the precedence for future implementation.

Test Coverage metric	**Increase**
Instructions	+912
Code lines	+245
Methods	+137
Classes	+31

Table 6.1: Final Test Coverage Statistics

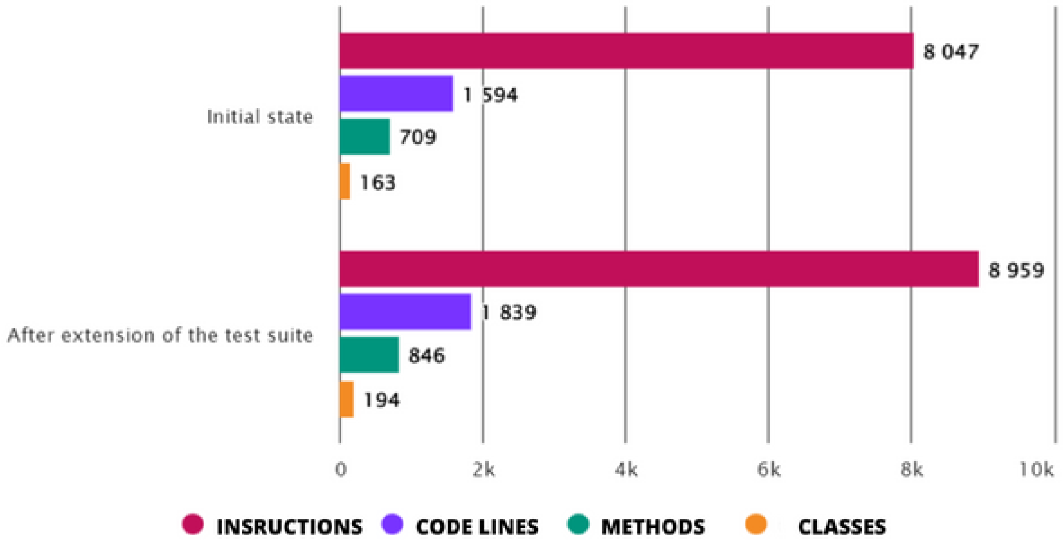
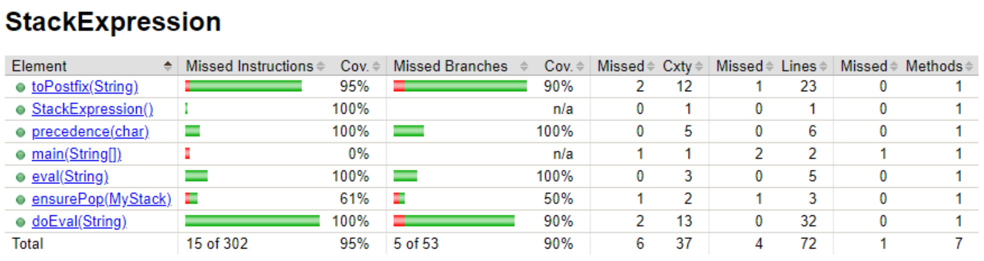


Figure 6.2: Overall Increase in Test Coverage

The Test-Driven Development part of the implementation should aim for 100% test coverage. In figure 6.3 it can be seen that the overall test coverage of the microservice is much higher in the opposite to other services when using this approach.



StackExpression

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
toPostfix(String)		95%		90%	2	12	1	23	0	1
StackExpression()		100%		n/a	0	1	0	1	0	1
precedence(char)		100%		100%	0	5	0	6	0	1
main(String[])		0%		n/a	1	1	2	2	1	1
eval(String)		100%		100%	0	3	0	5	0	1
ensurePop(MyStack)		61%		50%	1	2	1	3	0	1
doEval(String)		100%		90%	2	13	0	32	0	1
Total	15 of 302	95%	5 of 53	90%	6	37	4	72	1	7

Figure 6.3: Test Coverage of Expression microservice

6.3 Future Testing Strategy

The overall testing state of the BE services of EAS has progressed but it is far from acceptable test coverage. With a relatively small suite consisting of approximately 50 tests, we managed to cover 10 more services. In the future, it is recommended to extend the number of unit tests with the attention of more class coverage and aim to at least 80 % test coverage of the system. This thesis is recommended as a basis and a guide for such expansion.

Chapter 7

Conclusion

The objective of this thesis was to conduct an analysis of EAS in the context of microservice architecture and test it accordingly. The primary focus of this analysis was to examine the current state of testing, identify critical components of the system, and evaluate the skills of the development team. The subsequent goal was to devise and implement a comprehensive test suite for the BE microservices of EAS while considering the possibility of automation and potential expansion of test cases in the future.

The final test suite is sustainable and expandable, covering a significant part of BE of the system following proposed contingency strategies to mitigate risks. Consequently, the primary objective of this thesis has been successfully achieved.

To develop the final product of this work, which is the test suite, the initial step involved conducting research and studying the theory behind software testing and automation, with a primary focus on the lower layers of the V-model.

After analyzing the framework, unit, and integration tests were formulated to cover the fundamental functionality of the system. During the implementation process, the best practices for testing were observed to ensure the quality of testing, and established conventions were put in place to guide developers in creating tests independently among other things.

Lastly, final evaluations of the test results from the test suite were made and the final test report contains all the needed metrics for evaluating the confidence in the system's functionality.

At the beginning of this work, existing testing solutions covered only 9% of the BE components of the framework. Moreover, these tests lacked any standardized conventions or best practices, making it challenging for inexperienced developers to test the framework adequately. However, following the creation and implementation of the updated test suite, the coverage of the system has increased, accompanied by a clear guide on how to proceed during testing and potential expansion.

This work is useful as a base for testing EAS and it can be reused for testing the FE part of the application, as well.

Bibliography

[1] Axelrod, A. *Complete guide to test automation techniques, practices, and patterns for building and maintaining effective software projects.* 1st ed. Apress, 2018. ISBN 978-1-4842-3831-8.

[2] Bezbaruah, A., Pratap, B. and Hake, S. B. Automation of Tests and Comparative Analysis between Manual and Automated testing. In: SCES. *2020 IEEE Students Conference on Engineering & Systems (SCES).* 2020, p. 1–5. DOI: 10.1109/SCES50439.2020.9236748. ISBN 978-1-7281-9339-7.

[3] Carneiro, C. and Schmelmer, T. *Microservices from day one: Build robust and scalable software from the start.* 1st ed. Apress, 2016. ISBN 978-1-4842-1936-2.

[4] Castillo, C. and Hamra, M. *Unit Testing of Java EE Web Applications.* Stockholm, Sweden, 2014. Dissertation. KTH Information and Communication Technology.

[5] Chevuturu, A., Mathur, D., Kumar Reddy, B. J. and R, D. A Comparative Survey on Software Testing Tools. *International Journal of Engineering and Advanced Technology.* 1st ed. august 2022, vol. 11, no. 1, p. 32–40. DOI: 10.35940/ijeat.F3664.0811622.

[6] Cui, M. and Wang, C. Cost-benefit evaluation model for automated testing based on Test Case Prioritization. *Journal of Software Engineering.* 1st ed. 2015, vol. 9, no. 4, p. 808–817. DOI: 10.3923/jse.2015.808.817.

[7] Dobles, I., Martínez, A. and Quesada López, C. Comparing the effort and effectiveness of automated and manual tests. In: CISTI. *2019 14th Iberian Conference on Information Systems and Technologies (CISTI).* 2019, p. 1–6. DOI: 10.23919/CISTI.2019.8760848. ISBN 978-9-8998-4349-3.

[8] Fowler, M. *Bliki: Integrationtest.* Jan 2018. Available at: https://martinfowler.com/bliki/IntegrationTest.html.

[9] Georgieva, P. *Elasticsearch explained: Components, usage and benefits.* Sep 2022. Available at: https://flatrocktech.com/elasticsearch/.

[10] Graham, D., Black, R. and Veenendaal, E. v. *Foundations of Software Testing: ISTQB certification.* 3rd ed. Cengage Learning, EMEA, 2020. ISBN 978-1408044056.

[11] Hmelik, I. *You don't need automated testing?* COBE, Mar 2021. Available at: https://www.cobeisfresh.com/blog/you-dont-need-automated-testing.

[12] Itkonen, J., Mantyla, M. V. and Lassenius, C. Defect Detection Efficiency: Test Case Based vs. Exploratory Testing. In: ESEM. *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*. USA: IEEE Computer Society, 2007, p. 61–70. ESEM '07. DOI: 10.1109/ESEM.2007.38. ISBN 0769528864. Available at: https://doi.org/10.1109/ESEM.2007.38.

[13] Itkonen, J., Mantyla, M. V. and Lassenius, C. How do testers do it? An exploratory study on manual testing practices. In: ESEM. *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. 2009, p. 494–497. DOI: 10.1109/ESEM.2009.5314240. ISBN 978-1-4244-4842-5.

[14] Lee, T., Baik, D. and In, H. P. Cost Benefit Analysis of Personal Software Process Training Program. In: CIT. *2008 IEEE 8th International Conference on Computer and Information Technology Workshops*. 2008, p. 631–636. DOI: 10.1109/CIT.2008.Workshops.120. ISBN 978-0-7695-3242-4.

[15] Lin, D., Liping, F., Jiajia, H., Qingzhao, T., Changhua, S. et al. Research on Microservice Application Testing Based on Mock Technology. In: ICVRIS. *2020 International Conference on Virtual Reality and Intelligent Systems (ICVRIS)*. 2020, p. 815–819. DOI: 10.1109/ICVRIS51417.2020.00200. ISBN 978-1-7281-9636-7.

[16] Mateen, A. and Abbas, K. Optimization of model based functional test case generation for android applications. In: ICPCSI. *2017 IEEE International Conference on Power, Control, Signals and Instrumentation Engineering (ICPCSI)*. 2017, p. 90–95. DOI: 10.1109/ICPCSI.2017.8391869. ISBN 978-1-5386-0814-2.

[17] Myers, Sandler, C. and Badgett, T. *The Psychology and Economics of Software Testing*. 3rd ed. John Wiley &; Sons, 2012. ISBN 978-1-118-13315-6.

[18] Nnakwue, A. *A guide to fully understanding redis*. Jan 2020. Available at: https://blog.logrocket.com/guide-to-fully-understanding-redis/.

[19] Osherove, R. and Osherove, R. *The Art of Unit Testing*. Secondth ed. MITP, 2015. ISBN 9781617290893.

[20] Plewnia, C. *A Framework for Regression Test Prioritization and Selection*. Templergraben 55, 52062 Aachen, Germany, 2015. Dissertation. RWTH AACHEN UNIVERSITY.

[21] Ramler, R. and Wolfmaier, K. Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost. *In: Proceedings of 1st International Workshop on Automation of Software Test*. 1st ed. 2006, no. 1, p. 85–91.

[22] Raulamo Jurvanen, P., Hosio, S. and Mäntylä, M. V. Practitioner Evaluations on Software Testing Tools. In: IT University of Copenhagen. *Proceedings of the Evaluation and Assessment on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2019, p. 57–66. EASE '19. DOI: 10.1145/3319008.3319018. ISBN 9781450371452. Available at: https://doi.org/10.1145/3319008.3319018.

[23] *Introduction to redis*. Redis Ltd., 2023. Available at: https://redis.io/docs/about/.

[24] Reile, C., Chadha, M., Hauner, V., Jindal, A., Hofmann, B. et al. *Bunk8s: Enabling Easy Integration Testing of Microservices in Kubernetes*. 2022.

[25] Sahaf, Z., Garousi, V., Pfahl, D., Irving, R. and Amannejad, Y. When to Automate Software Testing? Decision Support Based on System Dynamics: An Industrial Case Study. In: ICSSP. *Proceedings of the 2014 International Conference on Software and System Process*. New York, NY, USA: Association for Computing Machinery, 2014, p. 149–158. ICSSP 2014. DOI: 10.1145/2600821.2600832. ISBN 9781450327541. Available at: https://doi.org/10.1145/2600821.2600832.

[26] Shuping, L. and Ling, P. The Research of V Model in Testing Embedded Software. In: ICCSIT. *2008 International Conference on Computer Science and Information Technology*. 2008, p. 463–466. DOI: 10.1109/ICCSIT.2008.51. ISBN 978-0-7695-3308-7.

[27] Spadini, D., Aniche, M., Bruntink, M. and Bacchelli, A. To Mock or Not to Mock? An Empirical Study on Mocking Practices. In: MSR. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 2017, p. 402–412. ISBN 978-1-5386-1544-7.

[28] Team, W. *Building microservices: Inter-process Communication*. Jan 2023. Available at: https://www.nginx.com/blog/building-microservices-inter-process-communication/.

[29] Umar, M. A. and Chen, Z. A Study of Automated Software Testing: Automation Tools and Frameworks. *International Journal of Computer Science Engineering*. 1st ed. december 2019, vol. 8, no. 1, p. 217–225. DOI: 10.5281/zenodo.3924795.

[30] Wahid, M. and Almalaise, A. JUnit framework: An interactive approach for basic unit testing learning in Software Engineering. In: ICEED. *2011 3rd International Congress on Engineering Education (ICEED)*. 2011, p. 159–164. DOI: 10.1109/ICEED.2011.6235381. ISBN 978-1-4577-1259-3.

[31] Waseem, M., Liang, P., Márquez, G. and Salle, A. D. Testing Microservices Architecture-Based Applications: A Systematic Mapping Study. In: APSEC. *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. 2020, p. 119–128. DOI: 10.1109/APSEC51365.2020.00020. ISBN 978-1-7281-9553-7.

[32] Wick, M. R., Stevenson, D. E. and Wagner, P. J. Using testing and JUnit across the curriculum. In: SIGCSE Technical Symposium on Computer Science Education. *Technical Symposium on Computer Science Education*. 2005. ISBN 10.1145/1047344.1047427.

[33] Wieczorek, S., Stefanescu, A., Fritzsche, M. and Schnitter, J. Enhancing Test Driven Development with Model Based Testing and Performance Analysis. In: TAICPART. *Testing: Academic & Industrial Conference - Practice and Research Techniques (taic part 2008)*. 2008, p. 82–86. DOI: 10.1109/TAIC-PART.2008.17. ISBN 978-0-7695-3383-4.

[34] Williams, L. Agile Software Development Methodologies and Practices. In: Zelkowitz, M. V., ed. *Advances in Computers*. Elsevier, 2010, vol. 80, p. 1–44. Advances in Computers. DOI: https://doi.org/10.1016/S0065-2458(10)80001-4. ISSN 0065-2458. Available at: https://www.sciencedirect.com/science/article/pii/S0065245810800014.

[35] Yu, J., Zhang, J., Pan, L., Chen, Y., Wu, N. et al. Software Exploratory Testing: Present, Problem and Prospect. In: IAECST. *2021 3rd International Academic Exchange Conference on Science and Technology Innovation (IAECST)*. 2021, p. 44–47. DOI: 10.1109/IAECST54258.2021.9695695. ISBN CFP21BJ6-ART.

Appendix A

Contents of the included storage media

- **executable_eas_project/**—EAS project demonstrating usage of microservices.
- **source_code/**—Source code and test code of BE microservices.
- **thesis_tex/**— Source code of the thesis.
- **thesis_pdf/**—PDF version of thesis.
- **user_guide/**—Documentation and Execution Guide.

Appendix B

Overall test coverage before

Test reports on the following pages were generated by the JaCocCo tool. JaCoco is an open-source code coverage plugin for Java applications. It is integrated into IDE Intellij Idea used while developing the implementation part of this work.

JaCoCo was used because of the simplicity of usability and setup. The final reports are informative and provide the exact percentage of code coverage together with these columns in order from left to right:

- Element—the name of the package,
- Missed Instructions—graphical representation of test coverage,
- Cov.—overall instruction coverage,
- Missed Branches—graphical representation of missed branches in the package,
- Cov.—overall branch coverage,
- Missed—missed complexity,
- Cxty—cyclomatic complexity,
- Missed—the number of missed lines,
- Lines—total count of lines,
- Missed—the number of missed methods,
- Methods—total count of methods,
- Missed—the number of missed classes and finally
- Classes—total count of classes.

Sessionscommon

common

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
cz.inqool.eas.common.exception.v2		8%		3%	370	392	233	272	261	280	37	46
cz.inqool.eas.common.signing.request		1%		0%	358	360	663	671	288	290	25	26
cz.inqool.eas.common.reporting.generator.aggregator		0%		0%	389	389	279	279	281	281	31	31
cz.inqool.eas.common.schedule.job		1%		0%	313	315	635	642	263	265	21	22
cz.inqool.eas.common.storage.file		0%		0%	288	288	528	528	233	233	16	16
cz.inqool.eas.common.export.request		2%		0%	271	273	528	536	220	222	15	16
cz.inqool.eas.common.alog.event		4%		0%	258	262	497	506	215	219	16	18
cz.inqool.eas.common.domain.index.dto.filter		49%		32%	422	623	149	488	134	324	12	41
cz.inqool.eas.common.domain.index.dto.aggregation		0%		0%	277	278	290	292	219	220	29	30
cz.inqool.eas.common.mail		2%		0%	240	242	462	470	204	206	17	18
cz.inqool.eas.common.ws.soap.logger.message		0%		0%	209	209	420	420	172	172	14	14
cz.inqool.eas.common.export.template		5%		0%	206	208	406	418	176	178	14	15
cz.inqool.eas.common.utils		14%		9%	270	311	411	486	139	175	7	14
cz.inqool.eas.common.reporting.report		0%		0%	213	213	365	365	168	168	19	19
cz.inqool.eas.common.sequence		0%		0%	179	179	367	367	150	150	16	16
cz.inqool.eas.common.dictionary.store		10%		0%	137	152	278	318	115	130	16	19
cz.inqool.eas.common.exception		0%		0%	120	122	318	320	109	111	18	19
cz.inqool.eas.common.settings.named		0%		0%	177	177	334	334	148	148	14	14
cz.inqool.eas.common.export.batch		3%		0%	158	160	329	336	121	123	13	14
cz.inqool.eas.common.schedule.run		3%		0%	185	187	319	326	154	156	14	15
cz.inqool.eas.common.intl		4%		0%	152	154	338	343	119	121	15	16
cz.inqool.eas.common.history		0%		0%	171	171	326	326	138	138	16	16
cz.inqool.eas.common.action		0%		0%	153	153	312	312	125	125	16	16
cz.inqool.eas.common.certificate		0%		0%	141	141	293	293	111	111	15	15
cz.inqool.eas.common.security.personal		4%		0%	159	161	291	299	128	130	15	16
cz.inqool.eas.common.settings.user		0%		0%	138	138	258	258	108	108	14	14
cz.inqool.eas.common.domain		29%		23%	110	138	256	351	80	108	2	4
cz.inqool.eas.common.state.app		0%		0%	122	122	230	230	93	93	14	14
cz.inqool.eas.common.settings.app		0%		0%	122	122	230	230	93	93	14	14
cz.inqool.eas.common.differ		0%		0%	107	107	242	242	82	82	6	6
cz.inqool.eas.common.domain.store		34%		21%	129	172	193	315	65	103	8	13
cz.inqool.eas.common.exception.v2.rest.processor		0%		0%	95	95	197	197	76	76	11	11
cz.inqool.eas.common.antivirus.scan		3%		0%	120	121	216	220	90	91	11	12
cz.inqool.eas.common.dated.store		20%		5%	80	104	171	205	63	87	9	15
cz.inqool.eas.common.authored		0%		0%	71	72	110	111	54	55	4	5
cz.inqool.eas.common.security.service		21%		4%	98	113	177	219	40	52	0	4
cz.inqool.eas.common.security.saml		0%		0%	59	59	177	177	37	37	3	3
cz.inqool.eas.common.domain.index.reindex.reference		0%		0%	83	83	135	135	45	45	6	6
cz.inqool.eas.common.client.action.dto		0%		0%	113	113	179	179	85	85	7	7
cz.inqool.eas.common.authored.store		23%		15%	78	96	153	193	58	76	9	12
cz.inqool.eas.common.exception.v2.rest		0%		0%	81	81	141	141	54	54	5	5
cz.inqool.eas.common.template		0%		0%	72	72	65	65	52	52	11	11
cz.inqool.eas.common.init		0%		0%	60	60	162	162	43	43	6	6
cz.inqool.eas.common.domain.index.dto.sort		29%		46%	89	115	76	128	77	100	9	13
cz.inqool.eas.common.security.captcha		0%		0%	59	59	102	102	45	45	8	8
cz.inqool.eas.common.security.form.internal		0%		0%	63	63	144	144	44	44	5	5
cz.inqool.eas.common.dictionary		1%		0%	40	42	76	78	34	36	2	3
cz.inqool.eas.common.security		0%		0%	68	69	123	124	56	57	6	7
cz.inqool.eas.common.reporting.generator		0%		0%	54	54	97	97	40	40	4	4
cz.inqool.eas.common.ws.soap.logger.interceptor		0%		0%	39	39	112	112	33	33	4	4
cz.inqool.eas.common.reporting.input		0%		0%	63	63	42	42	37	37	5	5
cz.inqool.eas.common.ws.soap.validator		0%		0%	37	37	98	98	13	13	3	3
cz.inqool.eas.common.domain.index.field		67%		52%	57	138	63	214	16	73	0	7
cz.inqool.eas.common.domain.index.dynamic		1%		0%	27	29	63	65	15	17	1	3
cz.inqool.eas.common.domain.index.reference		0%		0%	65	65	47	47	32	32	4	4
cz.inqool.eas.common.security.form		0%		0%	40	40	101	101	27	27	2	2
cz.inqool.eas.common.domain.index.dto.params		23%		0%	55	80	20	47	14	39	2	5
cz.inqool.eas.common.reporting.dto		0%		0%	52	52	32	32	22	22	1	1
cz.inqool.eas.common.differ.parser		0%		0%	33	33	72	72	14	14	1	1
cz.inqool.eas.common.domain.index		71%		47%	72	148	82	293	19	90	1	6
cz.inqool.eas.common.module		0%		0%	46	46	47	47	31	31	5	5
cz.inqool.eas.common.security.form.twoFactor		0%		0%	44	44	84	84	38	38	6	6
cz.inqool.eas.common.history.operation		0%		0%	42	42	41	41	27	27	4	4
cz.inqool.eas.common.differ.strategy.impl		0%		0%	38	38	69	69	21	21	6	6
cz.inqool.eas.common.client		0%		0%	44	44	80	80	38	38	5	5
cz.inqool.eas.common.script		9%		0%	31	33	81	87	23	25	3	4
cz.inqool.eas.common.differ.model.prop		0%		0%	35	35	65	65	26	26	3	3
cz.inqool.eas.common.exception.dto		0%		0%	43	43	68	68	26	26	4	4
cz.inqool.eas.common.differ.exception		0%		0%	35	35	18	18	25	25	4	4
cz.inqool.eas.common.ws.soap.logger		0%		0%	36	36	79	79	30	30	3	3
cz.inqool.eas.common.client.export		0%		n/a	22	22	72	72	22	22	4	4
cz.inqool.eas.common.client.file		0%		0%	20	20	61	61	18	18	4	4
cz.inqool.eas.common.authored.user		24%		9%	45	57	36	55	24	36	2	7
cz.inqool.eas.common.ws.soap.logger.interceptor.extract		3%		0%	38	41	51	54	30	33	6	8
cz.inqool.eas.common.security.header		0%		0%	22	22	62	62	14	14	2	2
cz.inqool.eas.common.alog.syslog		0%		0%	26	26	68	68	17	17	3	3
cz.inqool.eas.common.domain.index.mapping		0%		0%	39	39	46	46	26	26	4	4

cz.inqool.eas.common.authored.tenant		24%		10%	42	54	31	50	22	34	1	6
cz.inqool.eas.common.differ.model		0%		0%	26	26	51	51	18	18	3	3
cz.inqool.eas.common.exception.handler		8%		0%	27	33	48	56	18	24	1	3
cz.inqool.eas.common.multiString		4%		2%	36	39	14	18	16	19	2	4
cz.inqool.eas.common.security.form.ldap		0%		0%	19	19	61	61	18	18	3	3
cz.inqool.eas.common.security.internal		4%		0%	30	33	63	66	16	19	3	5
cz.inqool.eas.common.export.init		0%		0%	19	19	63	63	9	9	1	1
cz.inqool.eas.common.differ.event		0%		0%	34	34	45	45	24	24	4	4
cz.inqool.eas.common.keyvalue		0%		0%	22	22	47	47	19	19	4	4
cz.inqool.eas.common.exception.parser		10%		0%	26	33	50	57	20	27	1	8
cz.inqool.eas.common.domain.index.reindex.reference.entity		20%		n/a	15	18	30	41	15	18	1	3
cz.inqool.eas.common.domain.index.reindex		0%		0%	21	21	48	48	15	15	3	3
cz.inqool.eas.common.projection		23%		66%	15	23	32	47	13	20	1	4
cz.inqool.eas.common.admin.console.stream		0%		0%	15	15	30	30	9	9	2	2
cz.inqool.eas.common.ws		0%		0%	13	13	44	44	7	7	2	2
cz.inqool.eas.common.security.header.internal		0%		0%	18	18	43	43	17	17	2	2
cz.inqool.eas.common.domain.index.reindex.queue		0%		n/a	18	18	23	23	18	18	5	5
cz.inqool.eas.common.dated.index		21%		0%	18	24	30	44	16	22	1	3
cz.inqool.eas.common.dictionary.index		23%		n/a	27	31	26	39	27	31	2	4
cz.inqool.eas.common.admin.console		0%		n/a	13	13	36	36	13	13	3	3
cz.inqool.eas.common.db		7%		25%	9	12	29	34	5	8	2	3
cz.inqool.eas.common.domain.event.store		0%		n/a	20	20	38	38	20	20	10	10
cz.inqool.eas.common.security.session		7%		0%	21	24	36	40	20	23	1	2
cz.inqool.eas.common.i18n.yml		35%		0%	12	17	30	50	3	8	0	2
cz.inqool.eas.common.client.action		0%		n/a	16	16	30	30	16	16	4	4
cz.inqool.eas.common.domain.index.dto.filter.custom		0%		0%	16	16	19	19	13	13	3	3
cz.inqool.eas.common.dictionary.reference		0%		0%	20	20	15	15	10	10	2	2
cz.inqool.eas.common.differ.util		0%		0%	24	24	24	24	11	11	2	2
cz.inqool.eas.common.authored.index		29%		0%	17	21	13	25	14	18	1	3
cz.inqool.eas.common.client.export.runner		0%		n/a	14	14	27	27	14	14	3	3
cz.inqool.eas.common.security.form.secret		0%		0%	13	13	25	25	10	10	1	1
cz.inqool.eas.common.pdfa		0%		0%	10	10	29	29	9	9	2	2
cz.inqool.eas.common.export.init.dto		0%		n/a	28	28	17	17	28	28	2	2
cz.inqool.eas.common.export		0%		0%	17	17	23	23	16	16	1	1
cz.inqool.eas.common.antivirus		0%		0%	11	11	28	28	10	10	2	2
cz.inqool.eas.common.admin.console.dto		0%		n/a	20	20	20	20	20	20	2	2
cz.inqool.eas.common.xml		0%		0%	23	23	29	29	19	19	6	6
cz.inqool.eas.common		0%		n/a	2	2	14	14	2	2	1	1
cz.inqool.eas.common.schedule		15%		n/a	12	14	21	26	12	14	1	2
cz.inqool.eas.common.antivirus.clamav		0%		0%	12	12	20	20	10	10	1	1
cz.inqool.eas.common.trace		0%		0%	7	7	22	22	4	4	1	1
cz.inqool.eas.common.export.system		0%		n/a	7	7	21	21	7	7	1	1
cz.inqool.eas.common.variable		10%		0%	10	11	21	23	9	10	0	1
cz.inqool.eas.common.stomp		0%		0%	6	6	13	13	5	5	1	1
cz.inqool.eas.common.pdfa.mock		0%		0%	12	12	20	20	9	9	2	2
cz.inqool.eas.common.storage		0%		n/a	6	6	16	16	6	6	1	1
cz.inqool.eas.common.alog		0%		n/a	8	8	15	15	8	8	1	1
cz.inqool.eas.common.reporting		0%		n/a	7	7	12	12	7	7	1	1
cz.inqool.eas.common.reporting.exception		0%		n/a	10	10	20	20	10	10	2	2
cz.inqool.eas.common.domain.index.reindex.reference.event		0%		n/a	8	8	15	15	8	8	3	3
cz.inqool.eas.common.signing		0%		n/a	6	6	11	11	6	6	1	1
cz.inqool.eas.common.exception.v2.rest.dto		0%		n/a	10	10	13	13	10	10	2	2
cz.inqool.eas.common.export.event		0%		n/a	8	8	16	16	8	8	4	4
cz.inqool.eas.common.crypto		0%		0%	5	5	9	9	2	2	1	1
cz.inqool.eas.common.reporting.convert		19%		n/a	3	6	7	10	3	6	0	3
cz.inqool.eas.common.event		16%		n/a	6	8	12	15	6	8	1	2
cz.inqool.eas.common.domain.index.field.java		37%		n/a	4	8	8	14	4	8	1	2
cz.inqool.eas.common.export.request.dto		0%		n/a	8	8	8	8	8	8	2	2
cz.inqool.eas.common.signing.request.event		0%		n/a	6	6	12	12	6	6	3	3
cz.inqool.eas.common.ws.wsdl		9%		n/a	4	5	7	8	4	5	1	2
cz.inqool.eas.common.security.saml.internal		0%		n/a	4	4	10	10	4	4	1	1
cz.inqool.eas.common.exception.v2.dto		0%		n/a	7	7	11	11	7	7	2	2
cz.inqool.eas.common.security.captcha.exception		0%		n/a	6	6	12	12	6	6	2	2
cz.inqool.eas.common.export.access		0%		n/a	5	5	4	4	5	5	1	1
cz.inqool.eas.common.i18n		25%		n/a	3	4	4	6	3	4	0	1
cz.inqool.eas.common.mail.event		0%		n/a	4	4	8	8	4	4	3	3
cz.inqool.eas.common.signing.request.dto		0%		n/a	5	5	4	4	5	5	1	1
cz.inqool.eas.common.ws.soap.interceptor		0%		0%	3	3	6	6	2	2	1	1
cz.inqool.eas.common.domain.event		0%		n/a	3	3	6	6	3	3	3	3
cz.inqool.eas.common.export.provider		90%		n/a	4	8	4	20	4	8	0	2
cz.inqool.eas.common.authored.system		29%		n/a	4	6	5	7	4	6	1	2
cz.inqool.eas.common.differ.rest		0%		n/a	3	3	4	4	3	3	1	1
cz.inqool.eas.common.certificate.event		0%		n/a	2	2	4	4	2	2	1	1
cz.inqool.eas.common.dictionary.event		0%		n/a	2	2	4	4	2	2	2	2
cz.inqool.eas.common.reporting.event		0%		n/a	2	2	4	4	2	2	1	1
cz.inqool.eas.common.ws.soap		0%		n/a	3	3	3	3	3	3	1	1
cz.inqool.eas.common.dated		91%		50%	3	7	2	19	2	6	2	3
cz.inqool.eas.common.differ.strategy		0%		n/a	1	1	1	1	1	1	1	1
cz.inqool.eas.common.domain.index.dto		100%		n/a	0	10	0	7	0	10	0	1
cz.inqool.eas.common.async		100%		n/a	0	2	0	5	0	2	0	1
cz.inqool.eas.common.cache		100%		n/a	0	2	0	2	0	2	0	1
Total	81,026 of 89,073	9%	4,995 of 5,428	7%	10,049	10,818	16,717	18,311	7,367	8,076	852	1,015

Created with JaCoCo 0.8.7.202105040129