POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Politecnico
di Torino

Master's Degree Thesis

Integration Testing for Enterprise Web Applications

In collaboration with
RCS Etm Sicurezza Spa

Supervisors
Prof. Giovanni MALNATI

Candidate
Takla TRAD

October 2023

Abstract

In today's fast changing software ecosystem, efficient microservice integration has emerged as a vital problem for software development teams. To guarantee that these scattered components work together seamlessly, the integration process necessitates extensive testing. This thesis aims to provide a thorough examination of automated integration testing within the realm of microservices, with a focus on the use of a ubiquitous language to promote larger test engagement.

Throughout this research endeavor, in collaboration with RCS lab, a company looking to improve their integration testing practice, the goal was to create an automated integration testing flow that not only empowers developers but also extends testing capabilities to Quality Assurance team members. The encircling goal is to provide a test suite exemplar that encompasses best practices and concepts allowing successful testing operations.

A multifaceted strategy was used to overcome this challenge. Initially, an exploration was conducted on the applications under test, laying the groundwork for microservices integration testing efforts. Subsequently, the focus shifted to the implementation approaches. These techniques include configuring the test environment to accurately reflect the real-world conditions, managing databases for seamless data interactions, and strategically using Testcontainers to ensure smooth test execution while maintaining a high degree of fidelity to the production environment. Testcontainers is a framework that provides throwaway, lightweight instances of test-dependent services wrapped in Docker containers. Emphasis is placed on the value of adopting Cucumber as the selected testing framework. Cucumber is a tool used to test applications written in a Behavior-Driven Development style, which supports a team-centric and cross-functional workflows. Cucumber provides a language abstraction that encourages developer and Quality Assurance collaboration. This collaboration approach fostered a culture of shared testing accountability which increases overall testing involvement. Furthermore, Jenkins, an open-source automa-

tion tool equipped with plugins built for continuous integration, was selected to orchestrate the automated execution of the integration tests, providing frequent and timely validation on microservice integrations. Alongside the achievements, one can find discussions revolving the difficulties experienced during the implementation process and the tactics used to overcome them. To review and debate the results of the testing efforts, successful scenarios are outlined, using Cucumber reports and Jenkins pipeline performance metrics, ensuring alignment with the specified objectives. Finally, a comprehensive overview of the study's achievements is presented, stressing the significance of the chosen technique, and highlighting possible future improvements.

This thesis provided a complete grasp of the problems, tactics, and outcomes connected with automated integration testing in the microservices domain. Collaborating with RCS lab produced a helpful resource for future testing attempts, allowing widespread participation to perform tests that ensure the integrity of microservice communications.

Contents

List of Figures

List of Tables

List of Abbreviations

API Application Programming Interface

BDD Behavior-Driven Development

CD Continuous Deployment

CEO Chief Executive Officer

CI Continuous Integration

CI/CD Continuous Integration and Continuous Delivery/Deployment

DDD Domain-Driven Design

DDoS Distributed Denial-of-Service

DSL Domain Specific Language

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

IP Internet Protocol

JSON JavaScript Object Notation

JWT JSON WEB TOKEN

KPI key performance indicator

QA Quality Assurance

REST Representational State Transfer

ROI Return on Investment

SVN Subversion

TDD Test-Driven Design

UI User Interface

URL Uniform Resource Locator

UX User Experience

VM Virtual Machine

Chapter 1

Introduction

The pursuit of excellence has become synonymous with the success of modern businesses. The necessity for frequent releases, the need to manage an endless number of device combinations, and the push for standardization have all created challenges for development and Quality Assurance teams. Among these challenges, the iron triangle of time, cost, and quality looms big, asking a basic question: which of these aspects consumes the most time and is the most important?

With each passing week, new technologies, trends, and approaches arise in the software development sector. Automation, continuous testing, continuous integration, and DevOps have all become essential components of this transformative path. The software industry is pushed by a persistent goal of achieving more with less as expectations for speedy delivery rise. While time and quality remain top concerns, market trends have opened the path for a more holistic strategy that balances all three important elements.The Return on Investment (ROI) paradigm places enormous pressure on firms to offer the most recent features quickly and flawlessly, all while minimizing hassles and lowering costs. Companies are increasingly experimenting with new work practices, with ROI serving as a litmus test for the feasibility of change. Automated testing emerges as a keystone in this pursuit for efficiency and efficacy.

Continuous Integration (CI) testing, helping organizations shift left, has become a cornerstone of software development lifecycles. Shifting left has a simple philosophy: "Shift left is an approach that moves testing to earlier in the software development

lifecycle (hence, “shifting left”)", as stated by GitLab B.V. (2023) [3]. This method acknowledges that as software proceeds from left to right throughout the development cycle, its complexity grows exponentially, as does the expense of correcting faults. Hence, considering the measurable advantages of detecting a defect early in the development cycle, according to Arvinder Saini (Jan 11, 2017) [4], "The Systems Sciences Institute at IBM reported that it cost 6x more to fix a bug found during implementation than to fix one identified during design. Furthermore, according to IBM, the cost to fix bugs found during the testing phase could be 15x more than the cost of fixing those found during design.". Early bug discovery saves developers from the downstream ripple effects, allowing them to fix a single problem rather than dealing with a slew of problems. Furthermore, it guarantees that developers address the fault while it is still fresh in their brains, maintaining contextual details that help speed up resolution.

The shift left method is beneficial not just to developers, but it also coincides with larger company objectives. As mentioned by GitLab B.V. (2023) [5], "Organizations that adopt continuous integration have a competitive advantage because they can deploy faster. Organizations that have implemented CI are making revenue on the features they deploy, not waiting for manual code checks.". Improved software delivery performance and increased availability are concrete benefits of testing more frequently and earlier. These advantages are appealing not only to engineers, but to CEOs and stakeholders, fostering a culture of shared success.

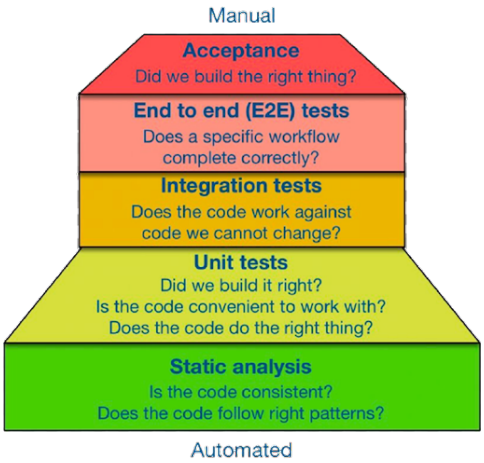


Fig. 1.1 The Testing Pyramid. From *Lecture Slides on Modularità (Organizzare il codice sorgente) - Le domande del test* by G. Malnati (2021-23) [1]

The testing pyramid, as illustrated in Figure 1.1 (G. Malnati, 2021-23) [1], is an iconic guidance in the world of automated testing. At its core are a collection of quick unit tests capable of quickly identifying problems without incurring significant expenditures. Integration tests, at a higher level, examine the behavior of coupled components, revealing how they interact smoothly. End-to-end testing, at its peak, orchestrates full user journeys across several encounters. Though end-to-end testing is beyond the scope of this thesis, the techniques we describe apply to all levels of the pyramid.
The idealized testing pyramid, on the other hand, frequently deviates from reality. In practice, a familiar pattern emerges: a scattering of unit tests, a visible absence of integration tests, and end-to-end tests that shatters repeatedly, eventually being marginalized and forgotten. Leaving aside judgment, this reality check highlights the complexity of testing, particularly in the context of microservices.

1.1 Importance of Integration Testing in Microservices Environment

The isolation concept is glamourised by microservices architecture, which allows autonomous teams to operate independently and assume control of their services. However, this autonomy comes with a catch: no single body is responsible for guaranteeing the flowless integration of all services. Each team pushes their service directly to production, hoping that these disparate efforts would come together effortlessly. Hope, on the other hand, is not a strategy. This leads us to the crux of the issue: Integration Testing.

In a microservices environment where autonomous teams go ahead independently, the dependence on contracts between teams becomes critical. These contracts, represented through semantic versioning and service interfaces, are critical for ensuring that the different microservices interact amicably.
A fascinating alternate viewpoint arises, promoting the concept of “Testing in Production”. This trend supports approaching production deployments as intrinsically unpredictable and supporting the practice of quality testing in live settings. The reasoning is straightforward: regardless of one’s faith in the code, production settings remain unique from non-production contexts, and unanticipated complications

inevitably arise. Production testing involves both scheduled testing and proactive error triggering with advanced technologies. When rigorously performed, it provides a disciplined structure for responding to unavoidable faults, as opposed to the useless quest of averting every prospective failure. However, pushing for production testing does not diminish the need for pre-production testing. It instead stresses a risk-based strategy. It involves careful consideration of prospective difficulties and their repercussions, allowing for educated risk tolerance decisions. As a result, it necessitates a constant process of improving integration testing.

Fidelity, isolation, speed, scalability, ephemerality, and cost are the six important dimensions being optimized:

- *Fidelity:* How similar is the testing environment to production. The closer it is, the more diagnostic it is, but also more expensive and complicated it is likely to be. It is also likely that you may not actually be able to match it exactly. Possible, but not likely.
- *Isolation:* Tests should not interfere with each other, or interfere with production systems. Everything should have a known environment that holds throughout the duration of the test. It is also not best to test against external resources or production resources when running a local test. It is not an idea, as even if only a read operation is done against an **API**, it is possible to **DDoS** the servers, as well as it is possible to send a malformed request that totally break things. Hence, every test should have its own environment that no one else can touch.
- *Speed:* Reducing the time from commit to deploy is correlated with positive business outcomes. Fast **CI/CD** results in fast deployments.
- *Scalability:* No matter how fast individual builds are, situations where more than one build running at a time will occur. And if there are more builds than workers, queuing will have to take place and that makes developers less productive.
- *Ephemerality:* How long does the infrastructure last? Ideally, as long as it is needed. If it is not being used, we do not want to pay for it. This would mean that the test infrastructure should disappear as soon as the tests are over, or

maybe not. Generally, if everything works okay and tests have passed, then the infrastructure is not needed anymore. However, if a failing test takes place, the infrastructure might be needed to figure out what went wrong, only for a limited period of time.

- *Cost:* No one would want to spend more money on their testing infrastructure than they spend on their production infrastructure. It depends on two factors: the first is rational factor which is how much money are you spending? and the other is a motive factor which is am I getting my money's worth. Cost is correlated with usage, so even if you are spending a lot, at least you know you are getting some insights out of it.

As we dive deeper into the world of microservices integration testing, we will dissect each optimization dimension, uncovering its profound implications and demonstrate the critical role these factors play in ensuring the harmony, reliability, and performance of microservices-based applications. QA and automated testing take center stage in an era defined by microservices, paving the way to a future where time, cost, and quality achieve balance, reshaping the landscape of software development and innovation.

1.2 Thesis Objectives

The prevailing question is no longer whether integration testing is necessary, but rather, how to make it efficient, comprehensive, and cost-effective in an environment defined by microservices architecture. This thesis embarks on a comprehensive exploration of automated integration testing in the realm of microservices, with a central focus on fostering greater engagement of various teams in RCS lab.

1.2.1 Identify and Evaluate Existing Integration Testing Methodologies and Tools

The first objective is to investigate and evaluate the various integration testing approaches and tools available in the context of microservices architecture. This requires assessing the current environment, including the strengths, shortcomings, and applicability of various testing approaches. By achieving the above, this thesis aims to provide a strong foundation for making informed decisions regarding the chosen integration testing solutions and their deployment.

1.2.2 Select the Most Suitable Tools for The Company and Experiment Their Effectiveness

The second goal is to practice implementing integration testing technologies within a specific organization setting. It involves selecting the best appropriate technologies that are compatible with RCS lab's particular microservices infrastructure. Following that, experiments will be carried out to assess the usefulness of these technologies in real-world circumstances. This part of the study aims to bridge the theoretical knowledge and practical application gap, ensuring that the chosen tools improve the testing process and deliver executable discoveries.

1.2.3 Inspire the Employees and Improve Software Development Process

Recognizing that technology transformations frequently involve changes in organizational culture and procedures, thus the third goal is to empower employees to embrace the newly selected integration testing methodology and tools. This includes training programs designed specifically for development and Quality Assurance teams, with the purpose of equipping them with the knowledge and skills needed for smooth adoption. Furthermore, the goal is to show how these improvements may greatly enhance the software development process by speeding up the testing process, improving cooperation among different teams, and eventually providing higher-quality software.

1.3 Thesis Structure

The efforts of this thesis resulted in the development of a Spring Boot application, implementing integration tests for specific endpoints of a couple of microservices, Clientsadmin and Activitylist. In particular, the analyzed testing tools are: Testcontainers for builidng a containerised testing environment suitable to performing integration testing, Cucumber for the ubiquitous language, and a sample of REST Assured for testing RESTful APIs.
The ultimate goal was to build a CI pipeline, which automatically performs compilation and testing of the developped application. The thesis will be structured as follows:

- *Chapter 2, Related Work:* In this chapter, we dive into existing literature on microservices, integration testing, and related methodologies, establishing a solid theoretical foundation for our research.
- *Chapter 3, Proposed Approach:* We provide an in-depth description of our research methodology, including the selection of microservices for testing and the tools and techniques employed.
- *Chapter 4, Results and Discussion:* This chapter presents our test scenarios, their results, and in-depth discussions about the implications of our findings.
- *Chapter 5, Conclusion and Future Work:* We conclude the study by summarizing our key findings, contributions, and insights. Additionally, we identify the limitations of our research and suggest potential directions for future studies.

Chapter 2

Related Work

2.1 Overview of Microservices Architecture

Monolithic applications were once considered as the norm, however, they are not well-suited for applications that require scaling, according to Alex Soto Bueno, Andy Gumbrecht, and Jason Porter (2018, 3) [6]. Further Alex Soto Bueno, Andy Gumbrecht, and Jason Porter (2018, 3) explain further that "Microservices aren't here to tell you that everything else is bad; rather, they offer an architecture that is far more resilient than a monolith to changes in the future", from which we can conclude that microservices are more resilient to change and can be scaled more easily. This makes them a better choice for applications that need to be able to adapt to changing requirements.

2.1.1 Defining the Scope and Size of Microservices

In the context of microservices, granularity or scope of each individual application is guided by the design and decision-making process. As stated by Alex Soto Bueno, Andy Gumbercht, and Jason Porter (2018, 4), "Regarding application size, there are no rules, other than a rule of thumb" [6]. In this context, the rule of thumb could be translated into a "bounded context" from Domain-Driven Development (DDD), it suggests that each microservice should align with a specific domain context or business capability. Another informal guideline is that each should be manageable by a small team and this team should have a single responsibility. However, the right

size can be subjective and varies based on factors like the complexity of business logic, the organization's structure. Thus this statement emphasizes that there's no one-size-fits-all or definitive metric for the size of a microservice. Instead, developers should rely on general guidelines, understand the specific needs of their domain, and iterative refinement to determine the optimal service granularity.

2.1.2 Advantages of Adopting Microservices Architecture

Microservices architecture allows you to break down an application into smaller, self-contained units that can be scaled independently. This can be beneficial for applications with fluctuating traffic, as you can scale the microservices that are experiencing the most load. It can also make it easier to develop and maintain applications, as each microservice can be maintained by a dedicated team of developers. Additionally, they can be rolled out independently of each other, which can reduce the risk of deployments.

2.2 Development Methodologies for Microservices Testing

Testing in any software architecture, including microsevices, is difficult due to its disperse nature. Various testing strategies may be adopted to guarantee that each piece of code performs as expected. there exists many different methodologies to test microservices architecture that will vary depending on the specific application, and the most common ones include:

2.2.1 Test-Driven Development

Test-Driven Development guarantees that code is developed to pass tests which helps to improve code quality. As dictated by Benjamin J. Evans, Jason Clark, and Martijn Verburg (2022) [7], "Its basic premise is that you write tests during your implementation rather than afterward, and those tests influence the design of your code".

TDD requires tests to be developed for each unit of code, these tests are then used to

guide code development which takes a little longer than other techniques. They can also assist in improving code quality and minimize the amount of problems, however, it does not guaranteed project fulfillment, nor prove robustness of the written tests, nor maintainability. It is easy to get wrapped up in the TDD approach and forget about how to properly write unit tests that answer the following questions: *Does the test aim on testing the correct part of the system? Is it legible? Is it maintainable?* The implementation consists of three phases that must be repeated for each test case:

1. *Write a failing test:* Create a failing test to show which feature is missing and how it should behave.
2. *Make the test pass:* Write only enough code to pass the test. At this point, the code does not need to be beautiful or tidy.
3. *Refactor your code:* The code should be safely tidied up to make it more legible and manageable while it is protected by the passing test.

Refactoring steps should be tiny and incremental, and all tests should be performed after each little step to ensure that nothing broke during the modifications. Refactoring can be done after numerous tests have been written or after each test. It is a crucial technique since it guarantees that the code will become simpler to understand and maintainance can be performed while still passing all previously defined tests.

2.2.2 Domain-Driven Development

Domain-Driven Development (DDD) is a software development methodology that focuses on the domain of the problem that the product is attempting to solve. DDD established the domain's basic concepts and then models the software around those principles. As a result, it is well suited to microservices architecture, in which each microservice is responsible for a single domain notion.

DDD tests are created to ensure that the program models the domain ideas appropriately and are frequently conducted in a domain-specific manner, which might make them difficult to comprehend for non-domain specialists. They can be very useful at ensuring that the program appropriately represents the domain. Further, DDD extends TDD by formalizing the top TDD practitioners' excellent behavior.

The finest TDD practitioners approach the problem from the outside in, beginning with a failed customer acceptance test that explains the system's behavior from the user's perspective. We take care as BDD practitioners to build acceptance tests as examples that everyone on the team may understand. The process of developing those examples is focused on collecting input from business stakeholders about whether the previously mentioned tests reflect the correct concept before the development even starts. This is done after a coordinated attempt to create a common, ubiquitous language describing the system.

2.2.3 Behavior-Driven Development

Behavior-Driven Development (BDD) is a software development methodology that combines the best practices of TDD with DDD which centers its attention on the behavior of the software rather than its implementation. As a result, it is well suited to microservices architecture, in which each microservice is responsible for a certain function.

BDD tests are developed in a way that represents the expected program behavior and are then used to guide software development. Adding to that its frequent use in conjunction with pipeline for continuous integration and continuous delivery (CI/CD), which automate software testing and development. This development approach aims to bring together members of business and technical teams. Gayathri Mohan (2022, 71) provides an example of a BDD framework by highlighting its benefits and structure: "For example, BDD frameworks like Cucumber provide facilities to write tests in natural language, resembling a typical user story with the Given, When, Then structure." [8]. This allows the business people to supply requirements as failing tests, while the technical people may begin implementing features by correcting the failed tests.

Summary

All three strategies attempt to create high-quality software, but they concentrate on distinct features and methodologies.

- *BDD:* is preferable for projects requiring clear communication between technical and non-technical team members.
- *DDD:* thrives when modeling a complicated domain that necessitates a thorough understanding.
- *TDD:* is used to ensure that the software is clean and well-tested, providing solid support for future updates and refactoring.

The approaches are not mutually exclusive and may be blended efficiently in a single project to capitalize on their individual strengths.

2.3 Testing Approaches for Microservices Architecture

The evolution in development approaches has made it a necessity to adopt robust testing paradigms, especially for microservices architecture as asserting that microservices are dependable, reliable, and robust is a challenging task due to their diffused nature.
The focus in the following section will target the different aspects of testing.

2.3.1 Manual vs. Automation Testing

Software testing has evolved significantly over the years, strating from a transition from manual testing to automated testing, in addition to a transition from a development-centric activity to one that is more collaborative and may be conducted, independently of the development team.

Manual Testing

Testing was primarily a manual procedure carried out by QA teams, developers, or even end-users. This method requires QA testers to engage with the program in the same way that end users would, exploring the application to detect bugs, inconsistencies, and usability concerns.

Previously, test cases were explicitly specified, and testers would go through the program step by step to check its functionality and behavior. The manual testing approach had advantages; it was simple and intuitive, making it ideal for exploratory, usability, and ad-hoc testing. Furthermore, no specialist programming abilities were required. It did, however, have certain limitations, such as being time-consuming and less accurate owing to potential human mistakes. Furthermore, the manual technique proved inefficient when dealing with recurring and large-scale experiments.

Transition to Automation

The limits of manual testing became more apparent as software programs got more complicated and the rate of development accelerated. This resulted in the growth of automated testing in the software industry, which provided a faster and more reliable method of identifying software faults.

Automated tests, which are designed to perform actions and validations autonomously, previously needed specific programming abilities, therefore they were often created by developers or highly technical QA team members. While automated testing had the benefits of speed and dependability, it was also cost-effective for regression testing and suitable for load and performance testing. However, these advantages were accompanied by certain drawbacks, such as the initial expense of establishing the automated environment and a higher learning curve. Furthermore, the ever-changing nature of software features necessitated continuous updating of test scripts, and the early phases of automation typically need programming abilities.

The Rise of User-Friendly Automation Tools

As the industry grew, it began to shift toward framework and technologies that enabled QA teams to write automated tests with little or no programming knowledge. Selenium and Appium are two tools that have changed the sector by providing user-friendly interfaces and domain-specific languages, making test script writing more accessible. Some cutting-edge platforms have gone so far as to provide codeless automation, allowing tests to be built visually.

Because of the democratization of automated testing, QA teams became even more self-sufficient, reducing their reliance on development teams for both test generation and maintenance. While this provided benefits such as removing the need for programming skills, allowing the QA team to take entire ownership of the testing cycle, and allowing for faster turnaround times for test preparation and execution, it also had certain drawbacks. There may be some initial setup and learning required, and codeless solutions may provide limited customization choices. As Mark Winteringham (2022, 95) pointed out "Automation tools are very good at giving us rapid feedback in a consistent manner based on explicit instructions we set. But we get out what we put in, nothing more." [9].

Summary

Overall, as stated by Gayathri Mohan (2022, XIII) "Manual testing has evolved into manual exploratory testing, and remains a fundamental part of the testing discipline today." [8]. Over and above that, the transition from manual to automated testing, as well as the following creation of user-friendly test automation tools, was a reaction to the rising complexity and speed of modern software development. This has resulted in more agile, efficient, and effective testing methods, allowing the QA team to work independently of the development team, allowing for faster and more dependable releases.

2.3.2 Types of Software Testing

Software testing is an exceptionally critical in a microservices setup since they are loosely coupled. Testing them needs to be done in a way that can oversee the communication between services and making sure, beyond any doubt, data is consistent and reliable. This requires a different approach than the one usually used in conventional testing strategies.

There are several types of testing that may be applied on any architecture including the microservices architecture. The best way to do it will highly depend on what part of the system is chosen to be under test, but some of the foremost well-known ways to do it are:

Static analysis

In an article, Xiao Pu (2021) defines static analysis as "the process of analysing computer programme without executing the code. This practice is often used to ensure that codes follow certain structures or standards (e.g coding standards)." [10]. Static analysis, unlike other types of testing, analyses the source code without running it, focused on discovering possible flaws and assuring compliance with coding standards. These evaluations are performed automatically by specialized tools, which examine the codebase for code complexity, possible flaws, security vulnerabilities, and breaches of coding style requirements.

Moreover, they assist developers to discover and correct flaws early in the development process by offering insights into code quality and maintainability, eventually leading to greater code consistency and adherence to coding standards. They also contribute to the general robustness and stability of software projects, making them a vital addition to any development team. This approach also includes continuous integration and monitoring to ensure that code quality is maintained throughout time.

Unit testing

Unit testing is when we check if small parts of the code work properly. This can be a prominent method to target glitches at the starting stages of development, but it does not assess how diverse microservices interact together. In a microservice environment, they ought to center on how well each individual service works. Tools like JUnit for Java or PyTest for Python can help with creating thorough unit tests. "A dependency is something we don't have full control over during a unit test.", said Roy Osherove (2022, 11) [11]. Unit testing can be done in two different ways. The first way is to test only the main thing we want to test without considering any other dependencies. The second way is to include everything that the code under test collaborates with as part of the test.

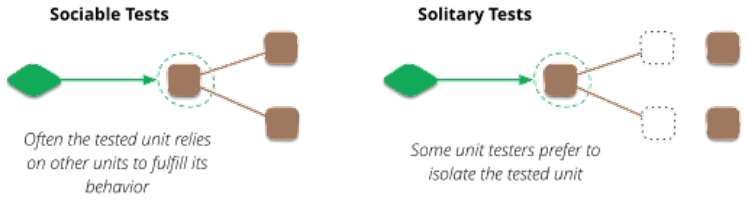


Fig. 2.1 Sociable vs. Solitary Unit Tests

Jay Fields (2015) [12], contributes by giving different names for the above-mentioned ways to do unit testing. A test that interacts with its associated dependencies in the system is called a sociable unit test. On the other hand, a solitary unit test is the one that is specifically designed to test only one part of the system, which is isolated.

Sociable Unit Tests are all about how a class works with other classes it depends on, and how it carries on with changes in its state. This strategy is a great choice for testing business-domain logic in a system. Knowing that solitary unit testing, excludes the dependencies of the class under test, the main difference between sociable and solitary unit testing is the use of test doubles to imitate the state of dependencies that the class undergoing unit testing depends on.

Test Doubles are used in Sociable unit testing where objects tend to be replaced with some fictitious dependencies. Acronyms, such as dummies, fakes, mocks, spies and stubs, are usually used concurrently but they are not synonyms. Each one of them could substitute a genuine item in the testing environment, although their behavior varies greatly.

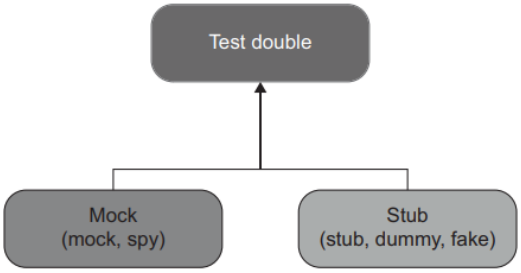


Fig. 2.2 Variations of test doubles. From *Exploring Mocks in Unit Testing* Chapters chosen by Vladimir Khorikov (2020), Manning Publications Co. Copyright 2020 Manning Publications. [2]

The distinction between these categories is as follows:

- *Mocks:* aid in the simulation and analysis of future encounters. These interactions are requests made by the system under test to its dependents aiming on modify their status.
- *Stubs:* aid in simulating incoming interactions. These interactions are calls made by the system under test to its dependents in order to get input data. Used when you want to replace a real implementation with an object that will return the same response every time.
- *Dummies:* are the easiest of the four test double types to use. It is designed to help fill parameter lists or fulfill some mandatory field requirements where you know the object will never get used. In many cases, you can even pass in an empty object.
- *Fakes:* can be seen as an enhanced stub that almost does the same work as your production code, but that takes a few shortcuts to fulfill your testing requirements. Fakes are especially useful when you'd like your code to run against something that is very close to the real third-party subsystem or dependency that you'll use in the live implementation.
- *spies:* are stubs that record information during the calls to it. For example, an email gateway stub might be able to return all the messages that it "sent".

Unit Test Coverage should be at a remarkable degree, achieved by the hard work of the development teams. The fundamental goal of unit testing is to guarantee the impeccable performance of individual components or isolated code units. This comprehensive analysis of individual units not only assists in the identification and correction of problems, but it also plays an important role in the early discovery of bugs, hence, improving the general resilience of the system.

Unit Test Limitations incorporate the fact that they are fundamentally restricted in their capacity to ensure that these components work flawlessly when integrated into a larger system. Further, another notable weakness is their inability to detect problems caused by complex interactions between numerous components. These interactions can result in complicated situations and subtle faults that are hidden beyond the scope of unit testing.

Hence, **Integration Testing Need** arises. Integration testing is considered critical for ensuring that diverse components of the system work in collaboration. This criterion stems from the fact that unit tests alone are unable to guarantee the overall operation of the system and integration tests help in bridging the gap between isolated unit testing and complete software capability evaluation.

Integration testing

Integration testing is a critical stage in the software testing lifecycle that focuses on assessing the interactions and collaborations between various components or modules inside a software system. Integration testing, as opposed to unit testing, which tests individual code units in isolation, evaluates how these components perform together when integrated into a coherent system. This level of testing is fundamental because it tackles the critical question of whether the microservices operate together coherently to offer the required functionality.
Several critical components and elements are required for successful integration testing:

- *Components to be Integrated:* Integration testing entails identifying and integrating certain software components, such as modules, microservices, or APIs, to ensure that they perform well together

- *Test Environment:* A controlled testing environment is required for reproducing the production configuration. It consists of hardware, software, databases, and network settings that are designed to simulate the actual deployment environment
- *Test Data:* To properly conduct integration tests, realistic and diversified test data sets are necessary. These data sets mimic numerous scenarios and inputs to thoroughly examine component interactions
- *Test Cases:* Integration test cases are intended to analyze the interfaces and interactions of integrated components. These instances define the inputs, anticipated results, and test circumstances.

Transitioning from integration testing to end-to-end testing is a significant stage that guarantees the system's validity. Integration tests are generally concerned with the interactions of individual components or microservices. They frequently fall short of offering an aggregate understanding of how these components work together to achieve user scenarios and whole business processes. End-to-end tests, on the other hand, take a larger view, determining if certain user workflows perform properly when all components are integrated.

End-to-end testing

End-to-end tests are critical for determining if certain operations inside a software system are completed appropriately. They are intended to address the question: *Does a particular user journey, which often spans multiple components or microservices, function as expected?*. These tests are often run automatically using scripted scenarios, simulating user interactions and transactions throughout the system.

End-to-end tests are distinguished by their ability to check while business processes, including many components, databases, and external dependencies, while accurately simulating actual user behavior and interactions, such as data entry and retrieval. They give a comprehensive perspective of system behavior, making them especially useful for evaluating key procedures that span numerous components. Selenium, Puppeteer, and Cypress are some common tools for running end-to-end tests. These tests are necessary to ensure that a program runs smoothly and consistently,

providing the desired user experience and functionality. However, as communicated by Daniel Irvine (2022, 519), "End-to-end tests are costly to build and maintain. Fortunately, they can be ntroduced gradually, so you can start small and prove their value before increasing their scope" [13].

Acceptance testing

Acceptance tests are used to ensure that software development efforts are in line with the planned business objectives and user requirements. The tests, which are often performed manually by testers or stakeholders acting as end users, assess whether the software meets high-level user expectations and serves its intended function. They are distinguished by scenario-driven evaluations that include real-world use cases and concentrate on the journeys, user interface (UI) interactions, and overall user experience (UX). While acceptance tests are performed less often than automated tests, they provide a comprehensive view of the application's functioning, which is important in the evaluation of the final product's value to end users.

2.3.3 Continuous Testing

Continuous testing is a broad technique that incorporates a variety of practices aimed at incorporating automated testing into every stage of the software development and delivery process. The purpose is to guarantee that the software fulfills quality criteria and functions as intended at each stage, from development through deployment.

Continuous Integration

Continuous Integration, in the context of Continuous Testing, is the stage at which code changes are routinely merged and the build is automatically initiated. This is followed by unit tests, component tests, and other forms of automated testing to detect bugs as early as possible.
Tool for Continuous Integration: *Jenkins*
"Jenkins is a continuous integration tool using Java language and is configurable via both GUI interface and console commands", affirms Katalon (2019) [14]. Jenkins can develop, deploy, and automate any project, making it extremely versatile.

Continuous Delivery

Katalon (2019) adds that "Continuous Delivery (CD) can often be confused with continuous deployment. A great way to think about the difference is continuous delivery is having any code version ready to deploy to production." [14]. Automated tests are run against the codebase and infrastructure, frequently increasing test coverage to include integration, functional, and end-to-end testing.
Tool for Continuous Delivery: *GitLab CI/CD*
GitLab CI/CD platform serves as a single platform for managing both CI and CD as it enables teams to automate testing at each level of the pipeline, from coding to deployment.

Continuous Deployment

Continuous Deployment takes it a step further by guaranteeing that any update that passes through all stages of your production pipeline is automatically delivered to your client, with no human involvement. It encompasses the idea of completely automated end-to-end testing, which frequently includes extra levels of testing such as performance, security, and smoke tests.
Tool for Continuous Deployment: *Spinnaker*
By the definition on its official website, Spinnaker (2023) is "an open-source, multi-cloud continuous delivery platform that combines a powerful and flexible pipeline management system with integrations to the major cloud providers." [15]. It integrates with Jenkins and other CI tools for automated triggering of pipelines based on code changes and passing tests.

2.4 Frameworks and Tools

Docker

Docker is a containerization technology that has transformed application packaging and deployment as defined on the official website by Docker Inc. (2023), "Docker is a platform designed to help developers build, share, and run container applications. We handle the tedious setup, so you can focus on the code." [16]. It enables you to package a program and its dependencies into a container image that can subsequently operate reliably across several environments. This containerization simplifies testing by giving your program with a repeatable and isolated environment, guaranteeing that it functions consistently regardless of where it runs.

Testcontainers

"Testcontainers for Java is a Java library that supports JUnit tests, providing lightweight, throwaway instances of common databases, Selenium web browsers, or anything else that can run in a Docker container.", as documentated by Richard North and other authors (2015-2021) [17]. It allows you to design, maintain, and deconstruct Docker containers as part of your unit and integration tests. This is especially useful if your program makes use of other services like databases, message queues, or web services. Testcontainers enables you to run these services in containers for testing, ensuring that your tests are self-contained and reproducible.

Jenkins

Jenkins is a flexible automation server that is essential for accomplishing CI/CD. It enables you to automate many steps of the software development process, such as code compilation and testing, as well as deployment and delivery. Jenkins pipelines assist teams in streamlining development operations, allowing for the timely and dependable delivery of software updates. It supports a wide range of plugins, making it suitable for a variety of development environments and tools.

Cucumber

"It's simple. Whether open source or commercial, our collaboration tools will boost your engineering team's performance by employing Behavior-Driven Development (BDD).", arrogantly introduced SmartBear Software (2023) their Cucumber tool [18]. Further, Cucumber is a tool that encourages cooperation among developers, testers, and non-technical stakeholders. It defines test cases in an organized manner using Gherkin, a human-readable plain-test format, and enables teams to specify software behavior in the form of executable specs. This method promotes a better knowledge of requirements and makes automated testing easier, guaranteeing that software matches defined criteria.

JUnit

JUnit is a Java testing framework that is used to write and perform unit tests. It includes a collection of annotations and assertions that make testing specific components of a Java program easier. Junit promotes test-driven development (TDD) approaches, in which tests are developed before real code, guaranteeing that each unit of code (such as methods or classes) works correctly in isolation.

REST Assured

In the documentation, Johan Haleby (2023) presented that "REST Assured is a Java DSL for simplifying testing of REST based services built on top of HTTP Builder. It supports POST, GET, PUT, DELETE, OPTIONS, PATCH and HEAD requests and can be used to validate and verify the response of these requests." [19], which basically contains the key features of the Java-based package REST Assured.

Awaitility

Johan Haleby additionally have presented Awaitility, which is a Java library for testing asynchronous code, which is common in modern applications, particularly those that use distributed networks and microservices. Johan Haleby (2023) defines it as "a DSL that allows you to express expectations of an asynchronous system in a concise and easy to read manner." [20]. Thus it provides a short and clear syntax for

expressing assertions that await the occurrence of certain circumstances or events. This tool is required in circumstances when your tests must support asynchronous processes, such as waiting for a message or a resource to become available.

Wiremock

"WireMock: Mock the APIs You Depend On" [21], Wiremock (2023) published on their official website, hence, it can be defined as an effective tool for simulating HTTP servers. Wiremock is especially handy when your program relies on external APIs or services that are not always available during testing. You may imitate these external services with Wiremock by describing the anticipated behavior of HTTP requests and replies. This allows you to thoroughly test your application's interactions with external services without making any actual network calls.

Chapter 3

Proposed Approach

3.1 Applications Under Test

As discussed in previous chapters, this thesis carries the analysis of finding a way to facilitate the testing phase and automate it.

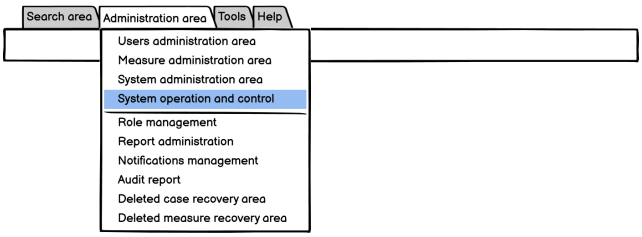


Fig. 1: Clients, printers and BS configuration entry of the Tools menu in the upper navigation bar.

Fig. 3.1 Clients, printers and burning stations configuration entry of the Tools menu in upper navigation bar.

As shown in the previous Figure 3.1, the first feature to be tested is “System Operation and Control”, that can be found in the Administration Area menu of *MITOCube*. This feature consists of three tables:

- *Clients:* are provided with an IP address, mandatory to access *MITOCube*.
- *Printers:* can then be enabled for certain clients, where they will be installed as local printers and be used in their operating system and applications.

- *Burning station:* configuration can be created to allow their use from the clients and set specific authorizations on Legal Archiving for each client.

Printers, clients and burning stations can be configured with a dedicated functionality to be registered in the system, as mentioned in the functional requirements of the module *clientsadmin* [22].
Figure 3.2 represents the clients' configuration page after "System Operation and Control" is selected, in addition to the tabs of the other two pages in which we may configure printers and burning stations.

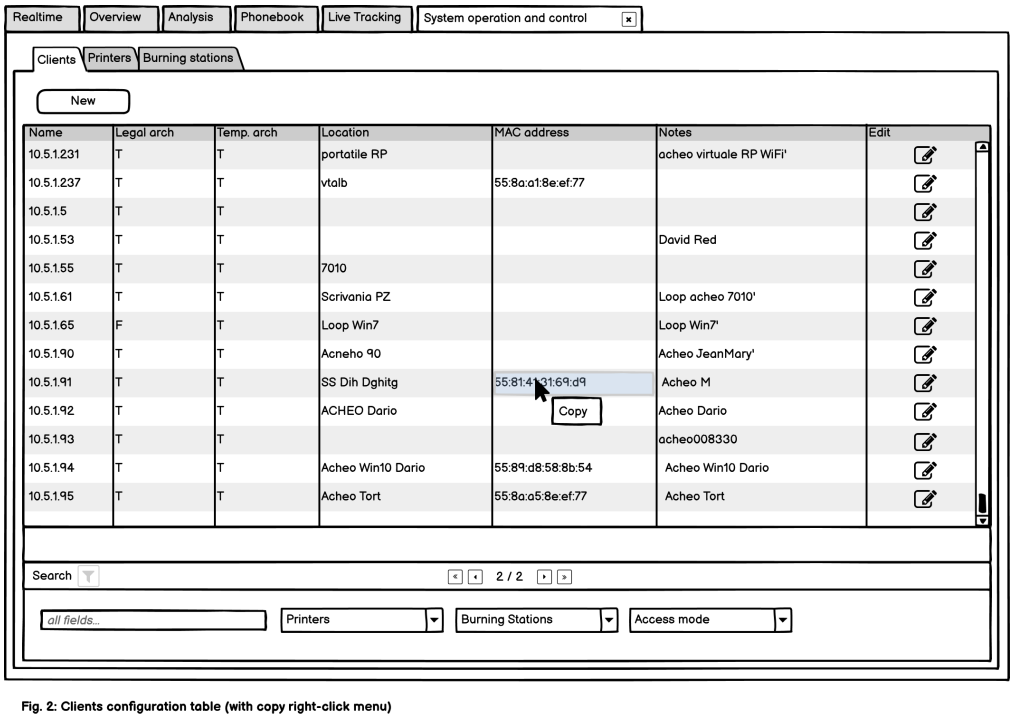


Fig. 3.2 Clients configuration table (with copy right-click menu)

In the microservice called *clientsadmin*, endpoints handling the implementation of the functionalities of the above feature may be found:

- `/printer/model`
- `/printer`
- `/client`

- `/burningstation`

As per the second feature to be tested, we focus on the following endpoints located in *activitylist* microservice:

- `/search`
- `/search/after`

These endpoints constitute a dynamic search system in this search implementation, where it notably provides two search options:
Basic Search Mode: The ordinary search feature where `/search` focuses on returning the correct list of "Liu"s based on the search terms, query, received as input.
Contextual Search Mode: `/search/after` provides contextual searching in addition to the ordinary search mode. In this mode, an additional input of type “Liu” is accepted and considered as a point of reference in the content. This extra “Liu” ensures that the subsequent search results, from the content, are fetched.
Activitylist, in its turn, enriches the query it receives with a series of additional data, that will be communicated to *Xplora/Elasticsearch*. Knowing that the accuracy of the result depends on other services (*Xplora/Elasticsearch*), we will try to focus on testing *activitylist* by assuring that it is not compromising the scope of the original query it received as input.

3.2 Implementation Techniques

3.2.1 Test Environment

To evaluate the aforementioned features, not only *clientsadmin* and *activitylist* microservices should be up and running, but all of the services involved in the flow of the architecture in Figure 3.3.

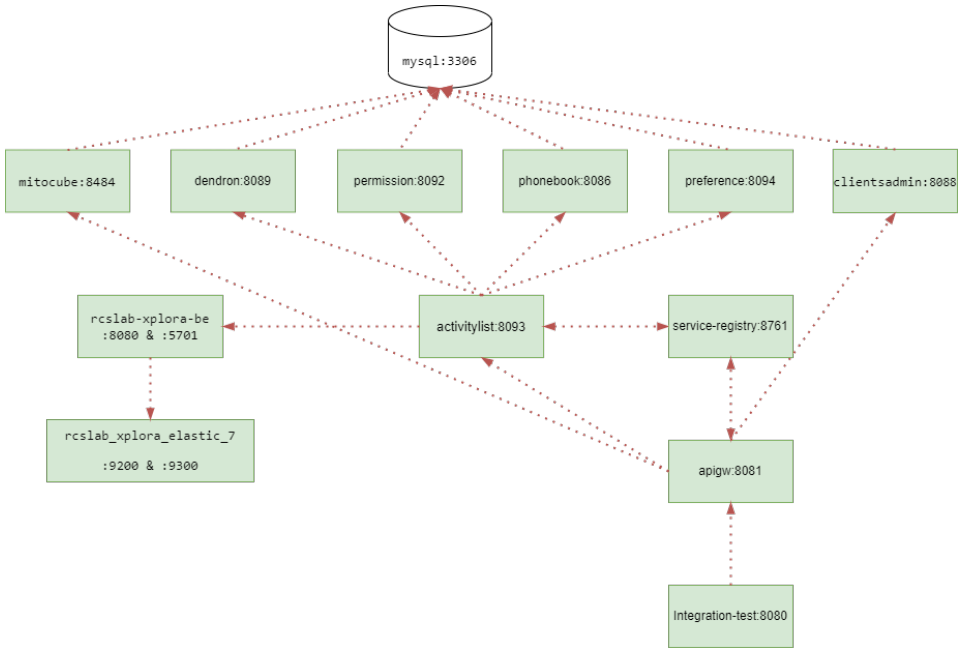


Fig. 3.3 Services involved in the test environment represented along with their dependencies

As illustrated in figure 3.3, the testing environment should consist on having all of the mentioned microservices running and able to communicate among themselves, where each one of them has an exclusive role:

- *integration-test*: The service that contains the implemented tests.
- *service-registry*: Eureka server used for other microservices discovery and registration; Hence, it is intended to be ready and functioning before all other microservices.
- *apigw*: API Gateway and acts as a single point of contact for client requests, delivering a variety of key features that improve the security, scalability, performance, and management of microservices-based systems, resembling the

system under test. It encapsulates the complicated structure of the underlying architecture, allowing clients to engage with the services more easily while enforcing critical restrictions and optimizations.

- *clientsadmin*, *activitylist*: The applications under test.
- *mysql*: MySQL database service.
- *dendron*, *phonebook*, *permission*, *preference*, *mitocube*: *activitylist* depends on all these microservices to conclude the testing procedure.
- *Xplora*, *ElasticSearch*: Xplora acts as an intermediary between *ElasticSearch* and the microservice requesting the queries, *activitylist*. It handles transforming the raw data fetched from *ElasticSearch* and guarantees that *activtylist* receives them in a well-structured data.

3.2.2 Databases

In order to assess the microservices, it is essential to connect them to their databases. However, the content of the databases needed depends on the testing objectives and the precise scenarios covered in the testing approach.
For instance, when performing tests on the *clientsadmin*'s endpoints, it is not recommended to have the database filled with any data. This is because we are looking to validate whether the application can successfully initialize, set up, and interact with the database; this procedure is better done on an empty database.
On the other hand, testing the search APIs in *activitylist* would require the database to be pre-populated, which is essential for thorough and realistic testing when recreating real-world settings. The relational database referred to is called ‘mysql‘ and it contains the users registered and needed for authentication, as well as it contains the tables essential to all the microservices that *activitylist* is dependent on.
Furthermore, to rigorously analyze the performance of the search endpoints, ElasticSearch must be filled with a replica of real-world data. This methodology enables us to undertake extensive testing that mimics the complexity and delicacies of actual usage scenarios.

3.2.3 Docker Compose Configuration

A docker compose file has been created in the following path:
`src/test/resources/docker-compose/docker-compose.yml`,
which defines how docker containers of all the services this project depends on should be built and run.
Below is stated an example of how *apigw* microservice is defined, in the YAML file, to be used later in the test environment:

```
version: '3.3'
services:
  apigw:
    image: nexus.rcslab.it:5000/rcs/apigw:2.0.R04-SNAPSHOT_integ-test
    environment:
      - server.port=8081
      - APIGW_LOGS=/app/apigw/logs
      - hazelcast.serviceUrl.default=http://service-registry:8761/eureka/
#     ...
    volumes:
      - /c/Users/Public/logs/apigw/:/app/apigw/logs:rw
    depends_on:
      - "service-registry"
      - "clientsadmin"
      - "activitylist"
      - "mitocube"
#   ...
```

Here is a quick explanation of each section:

- *'version'*: Specifies the version of the docker compose file format. Here, it is set to "3.3".
- *'services'*: Contains all the services (containers) that should be run when docker compose is up.
- *'apigw'*: States the name of the service.
- *'image'*: Contains the docker image to be used.
- *'environment'*: Environment variables to be passed to the service.
- *'volumes'*: Mounts paths for volumes, allowing data to persist.
- *'depends on'*: Specifies the services this service depends on. Docker compose will start the services considering the dependency order.

3.2.4 Testcontainers Configuration

For the configuration of Testcontainers, *AbstractIntegrationTest* class is created and contains, mainly, the docker compose file location, the services which their ports are exposed, and a *WaitStrategy* for each service.

```
@Testcontainers
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public abstract class AbstractIntegrationTest {
    @Container
    public static final DockerComposeContainer dockerComposeContainer =
        initializeDockerCompose();

    private static DockerComposeContainer initializeDockerCompose(){

        File file = new File(Objects.requireNonNull(ContextHolder.getProperty(
            TESTCONTAINER_DOCKER_COMPOSE_YML)));
        DockerComposeContainer dockerComposeContainer = new DockerComposeContainer
            (file)
            .withLocalCompose(true)
            .withOptions("--compatibility");
        // set WaitStrategy
        // expose selected services
        return dockerComposeContainer;
    }
}
```

It is worth mentioning that *.withLocalCompose(true)* is added for the sole purpose that on my local machine, i am unable to get the required permissions to modify any file, in the case where a volume is created to mount the log files for each service to a local file on my machine. Moreover, *.withOptions("–compatibility")* is used to tell Docker Compose to use compatibility mode.
Using the above configuration alone, we are unable to acknowledge at what point the container is going to be ready to receive some traffic. Hence, we have added *WaitStrategy* for each service selected to be exposed for later use. This inspection has a timeout of 10 minutes and the types of *WaitStrategy* used are:

- *Wait.forLogMessage():* For *mysql*, we are looking in the container's log output for `ready for connections.`, which indicates that the database is ready to be used.

- *Wait.forHttp():* This strategy is used for all the other exposed services; By default, `/actuator/info` is checked and should return an HTTP 200 OK success status response code. However,
  - For *mitocube*, we check `/mitocube/actuator/health`. Checking if this URL is up is not enough; we need to make sure that *mitocube* is registered to the Eureka server, *service-registry*. We do this by checking if the response contains **"MITOCUBE:1"**.
  - For *apigw*, we check `/actuator/health`. Similarly to *mitocube*, we should check if both services, *mitocube* and *activitylist*, are registered to the *service-registry* by checking if the response contains **"MITOCUBE:1"** and **"ACTIVITYLIST:1"**.
  - For *ElsticSearch*, we check the health of `/_cluster/health`.
  - For *Xplora*, we check the health of `/XploraWS/es/getClusterStatus`.

In Table 3.1, exposed and unexposed microservices are listed along the assigned port(s) to each one of them.

Exposed		**Unexposed**	
Service	**Port**	**Service**	**Port**
MYSQL	:3306	SERVICE_REGISTRY	:8761
APIGW	:8081	KAFKA	:9293
MITOCUBE	:8484	KAFKA_REP	:9294
DENDRON	:8089	ELASTIC_SEARCH	:9200&:9300
PHONEBOOK	:8086	XPLORA	:8080&:5701
PERMISSION	:8092	ZOOKEEPER	:2181
PREFERENCE	:8094		
ACTIVITY_LIST	:8093		
CLIENTS_ADMIN	:8088		

Table 3.1 Exposed and unexposed services in the test environment

When all the wait strategies are satisfied and the needed services' ports are exposed, we can affirm, with confidence, that all the containers are up and running.
AbstractIntegrationTest also defines a static class called *Initializer*, which implements the *ApplicationContextInitializer* interface. The latest interface provides a way to customize the spring boot application context before it is created. Hence, *Initializer*

class starts the docker containers and sets the correct port of the *mysql* container to MYSQL_PORT environment variable, as shown in code snippet.

```
public static class Initializer implements
        ApplicationContextInitializer<ConfigurableApplicationContext> {

    @Override
    public void initialize(@NotNull ConfigurableApplicationContext context) {
        ContextHolder.setContext(context);
        dockerComposeContainer.start();
        List<Services> selectedServices = getSelectedServices();
        int mysqlPort = 3306;
        if (selectedServices.contains(Services.MYSQL)) {
            mysqlPort = dockerComposeContainer.getServicePort(Services.MYSQL.
                getName(), Services.MYSQL.getPort());
        }
        TestPropertyValues.of("MYSQL_PORT=" + mysqlPort).applyTo(context.
            getEnvironment());
    }
}
```

Further, the above mentioned class, *AbstractIntegrationTest*, can be extended by other classes that will implement integration tests. These classes, subsequently, possesses the capability of using *dockerComposeContainer* to access the docker containers already initialized in *initializeDockerCompose()* method.

3.2.5 Cucumber Configuration

Cucumber's configuration may be found in RunTest class. Initially, using *@RunWith(Cucumber.class)*, JUnit is informed to run the class with cucumber's test runner. Further, we specify various options for running cucumber tests, using *@CucumberOptions*, as follows:

- *features = "src/test/resources/features"*: Specifies the path where the '.feature' files may be found and in the project we chose to put them in `src/test/resources/features`.
- *extraGlue = { "it.rcslab.cucumber.context" }*: Specifies the additional packages where hooks can be found. By default, Cucumber will only look in the package where the runner is located. For this project, the glued package is `it.rcslab.cucumber.context` and, basically, it contains Authentication-

Holder and ResponseHolder classes that will hold on to the authentication information and the response entity for the requests done by each test.

- *plugin = { "pretty", "html:target/cucumber-report.html", "json:target/cucumber-report.json" }* : Specifies the output format of the test report. In the case of this project, a pretty report, an HTML report file, and a JSON report file will be generated.

An extra option may be added to help organize the test runs or run only a subset of the tests at a time. For instance, to specify that only scenarios or features tagged with *@ActivityList-Search* should run, we add *tags = "@ActivityList-Search"* to *CucumberOptions*.

In the Table 3.2, the association of each tag used in the project, with its respective location, and the corresponding endpoint used to accomplish its tests, is stated.

Tag	Feature file	Endpoint
@ClientsAdmin		
@ClientsAdmin-PrinterModel	ClientsadminPrinterModel	`/printer/model`
@ClientsAdmin-Printer	ClientsadminPrinter	`/printer`
@ClientsAdmin-Client	ClientsadminClient	`/client`
@ClientsAdmin-BurningStation	ClientsadminBurningStation	`/burningstation`
@ActivityList		
@ActivityList-Search	ActivityListSearch	`/search`
@ActivityList-SearchAfter	ActivityListSearchAfter	`/search/after`
@ActivityList-SearchDiffUsers	ActivityListSearchDiffUsers	`/search`
@ActivityList-SearchEnrichLius	ActivityListSearchEnrichLius	`/search`
@ActivityList-SearchFails	ActivityListSearchFails	`/search`
@ActivityList-SearchWithSort	ActivityListSearchWithSort	`/search`

Table 3.2 Cucumber tags, its respective location, and the corresponding endpoint they cover

Feature file

In the test automation process, the capabilities of cucumber feature files are used to create and execute tests with clarity and precision. Starting from the feature files, let

us take a simple example of a scenario and follow the implementation step-by-step. The selected scenario belongs, precisely, to *@ClientsAdmin-PrinterModel*, hence, its implementation may be found in `ClientsadminPrinterModel.feature` and it targets `/printer/model` endpoint that belongs to *clientsadmin* microservice.

```
@ClientsAdmin @ClientsAdmin-PrinterModel
Feature: ClientsAdmin PrinterModel API works properly

  Background: apigw login
    When I submit username: "flavio" and password: "12345678"
    Then I receive a JWT

  Scenario: The client retrieves the list of printer models and receives status
      code OK and one printer model
    Given PrinterModel contains the following records
      | name        | driver       |
      | pm_model01  | pm_driver01  |
    When The client calls GET /printer/model
    Then The client receives status code of 200
    And The client receives content-type application/json
    And The client receives a List of 1 PrinterModel
    And The client receives one PrinterModel having name "pm_model01"
```

Background

In this scenario, we are replicating an API test that includes authentication and fetching a list of printer models. The scenario starts with a background section in which an API gateway login is performed and an authentication JSON Web Token (JWT) is received. This JWT is required to authenticate subsequent calls. The use of the cucumber background section is chosen with regards to its definition provided by Matt Wynne and Aslak Hellesøy [23], "A background section in a feature file allows you to specify a set of steps that are common to every scenario in the file. Instead of having to repeat those steps over and over for each scenario, you move them up into a Background element".
The JWT is safely stored in an ‘extraGlue’ class, AuthenticationHolder, allowing its access across the tests’ stages while maintaining their independence. This technique improves maintainability, reusability, and consistency while managing authentication data effectively and keeping scenarios clean and compact.

Scenario

Each scenario begins after the user authenticates successfully. In the example scenario provided above, the client is supposed to obtain a list of printer models from the API in this simulation. In the scenario, it is described the behaviors and expectations using Given, When, and Then phases, where:

- *Given:* step creates a predefined state by indicating the record that PrinterModel table should contain, named "pm_model01" and a driver named "pm_driver01".
- *When:* step defines the client's intention to perform an HTTP GET request to the endpoint `/printer/model`.
- *Then:* step describes the expected results of the request. A successful response should result in an HTTP 200 OK success status response code for the client. It should also get a response with the content-type of application/json, which confirms that the data is in JSON format. Furthermore, the client should expect to get a list of printer models, containing one printer model, that should be named "pm_model01".

In summary, this scenario ensures that the API is running properly by determining whether it delivers the necessary data in response to a client request and that the authentication phase is performing as anticipated. This is known as BDD, and it involves writing tests in a human-readable style to check that the software acts as expected from the user's perspective.

Step Definition

Moving to the implementation of the steps mentioned in the feature file, the relevant actions for Given, When, and Then steps are implemented in the feature file's step definitions.
In the step definition classes, the login action, done in the background, is handled by the following definition:

```
@When("I submit username: string and password: string")
@Then("I receive a JWT")
```

In the first line, a *When* annotation is used from cucumber to define a step for the login and represent a user submitting a username and password. As a next step, using *Then* annotation, we ensure that the JWT, saved in the AuthenticationHolder, was received correctly and that it is not null.
The main scenario holds the implementations of:

1. @Given("PrinterModel contains the following record(s)")

   This function annotation is in charge of storing the printer model records stated in the feature file and passed to the step definition class as a function parameter.

2. @When("The client calls GET (/printer|/client|/burningstation)([/a-zA-Z0-9]*)$")

   This is a versatile step that handles GET requests to several endpoints (`/client`, `/printer`, `/printer/model`, and `/burningstation`). It created the URL based on the route supplied and sends the GET request using the service client.

These step definitions bridge the gap between the high-level scenario described in the feature file and the actions and assertions executed throughout the test automation process. They provide explicit and exact communication among test scenarios and the code behind it, thus, guaranteeing the correctness and the effectiveness of the tests being conducted.

Database Cleanup

To enhance the dependability and predictability of out tests, we prioritize test scenario isolation. Following each test execution, an approach to clean and reset the database is involved, which returns it to its predefined baseline state. This method has various advantages. For starters, it ensures the predictability of the test results as, initially, the database was in a known state. Second, it encourages test independence, reducing the possibility of incidences where tests affect each other. Third, by preventing data contamination or obsolete records from other tests, we ensure data integrity. This technique simplifies the debugging efforts because errors are less likely to occur as a result of test interactions.
As our microservices ecosystem grows, this technique scales effectively, giving us a solid basis for preserving our system's stability. This technique is consistent and

considered as best practice for testing as it increases the confidence in the quality and performance of services as they interact inside a distributed system.

Scenario Outline

Cucumber's scenario outline is a handy tool that allows you to run the same scenario several times with various data sources. It is especially beneficial when you want to test a scenario with multiple input combinations or under different conditions. A scenario outline is a scenario template which specifies a scenario that includes placeholders for inputs or data that will change each time the scenario is run.
For this project, this approach is selected to test the search endpoints with some tests, especially the ones that are data driven. Data-driven testing is well suited to search endpoints, which frequently contain a broad variety of potential input combinations and expected outputs.

```
@ActivityList @ActivityList-SearchWithSort
Feature: ActivityList Search API works properly with sorting enabled

  Scenario Outline: Search in <order> order, receive first: <firstLiuId> and last: <
      lastLiuId> LiuId
    Given I create a query having the category "events"
    And I open a group with condition "AND"
    And I add to the group a rule where "liid" "equal" "20080312044460"
    And I close the current group
    And I set query size to <querySize>
    And Enable sorting "<field>" in a "<order>" order
    When I send a POST search request
    Then The client receives status code of 200
    And The client receives content-type application/json
    And I receive <returnedLius> items with liid equals "20080312044460"
    And I receive the "first" item having liuId equals to <firstLiuId>
    And I receive the "last" item having liuId equals to <lastLiuId>

    Examples:
      | querySize | field | order | returnedLius | firstLiuId | lastLiuId |
      | 607       | 1     | asc   | 607          | 61665      | 83275     |
      | 607       | 1     | desc  | 607          | 83275      | 61665     |
      | 620       | 1     | desc  | 607          | 83275      | 61665     |
```

The template in the code snippet mentioned above, the scenario outline approach is selected for testing the search endpoint with some extra parameters, in particular, by adding a sort measure on the field parameter. These tests are present particularly under *@ActivityList-SearchWithSort*.

In the implementation, the template offers placeholders different input variables (query size, field, order, returned Lius, first LidId, and last LiuId) and during execution, these placeholders will be replaced with the actual values from the Examples table.

Cucumber Reports

As mentioned earlier, Cucumber was utilized in the test automation process to execute and report on our integration tests. We were able to gain extensive insights on the behavior of our microservices as a result of this integration. The following are paths to the Cucumber test reports:

- HTML Report: `/target/cucumber-report.html`:
  The HTML report displays test results in a user-friendly format, making it simple to browse through scenarios and stages.
- JSON Report: `/target/cucumber-report.json`:
  The structured data regarding the test outcomes is contained in the JSON report. Although it cannot be viewed directly, it can be utilized for sophisticated analysis or interaction with other technologies.

These reports are critical in determining the accuracy and robustness of our microservices integration.

3.2.6 REST Assured

REST Assured with cucumber is a combination for creating BDD-styled tests for RESTful APIs in feature files, using natural language, while utilizing REST Assured's capabilities for performing HTTP requests and assertions from within the step definition implementations.
For the sake of completeness, our target is to illustrate an example of how REST Assured implementation should look like in case it gets adopted for later use in the test application. The example will target the developed tests for both search APIs and below is outlined the demonstration of how REST Assured is set up and used.
As a first step, *@Before* annotation is needed to cover the setup of *RequestSpecification*'s configuration with the base URI, content-type header... Other settings needed

for the tests should be configured at this point in the process.

```
@Before
public void beforeScenario() {
    // ...
    requestSpecification = new RequestSpecBuilder()
        .setBaseUri(String.format("http://%s:%d",
            dockerComposeContainer.getServiceHost(Services.APIGW.getName(),
                Services.APIGW.getPort()),
            dockerComposeContainer.getServicePort(Services.APIGW.getName(),
                Services.APIGW.getPort())))
        .addHeader(
            HttpHeaders.CONTENT_TYPE,
            MediaType.APPLICATION_JSON_VALUE
        )
        .build();
    // ...
}
```

The next step would be handling the login. Normally, a POST request is dispatched to the authentication path containing the username and password.
As we already know, this response holds the JWT that should be used in the following API call, which means that we should keep it stored, this is done using the glued *AuthenticationHolder* class.

```
Response response = RestAssured.given(requestSpecification)
        .when()
        .body(postBody)
        .post(AUTHENTICATE_PATH);
authenticationHolder.setAuthenticationInfo(
        response.getBody().as(AuthenticationInfo.class));
```

Following that, the natural next step would be to perform a POST request to the search API endpoint. This request is sent with the defined request headers and body that contains the search parameters and criteria. Once the request is sent and the response is extracted and parsed for further analysis.

```
response = RestAssured.given(requestSpecification)
        .when()
        .headers(requestHeaders())
        .body(queryBuilder.build())
        .post(ACTIVITYLIST_SEARCH_PATH);
activityListResult = getActivityListResultFromResponseBody(
        response.getBody().asString());
```

Eventually, we make use of REST Assured built-in assertions to evaluate the response's status code and content type.

```
response.then().statusCode(expectedStatusCode);
response.then().contentType(contentType);
```

It is worth mentioning that REST Assured is a robust library for testing RESTful APIs that offers a variety of functionalities for making requests, processing responses, and verifying claims. However, the decision between REST Assured with cucumber or RestTemplate on the other hand, is determined by the project requirements and team preferences. RestTemplate is more straightforward and may be a better solution when BDD-style tests are not required. REST Assured with cucumber shines when it comes to BDD, improving collaboration, and creating exceptionally legible API tests.

3.2.7 Jenkins

Creating a jenkins pipeline as part of this project is an important step in automating and enhancing both the development and testing processes. Jenkins pipelines simplify the whole software development lifecycle by automatically creating, testing, and possibly deploying an application whenever changes in the code are made. This automation provides consistency and fast reproducibility in the development process. Overall, CI automation improves the quality of the code and fasten up the delivery of trustworthy software to production. Accordingly, in what follows an explanation of the Jenkins pipeline implemented for this project may be found. A declarative pipeline script which serves the automation and the testing process of the project. It is written in Groovy, a scripting language, and has a well-defined structure:

- Starting from a "pipeline" block that signifies the start of a Jenkins pipeline and encapsulates the whole pipeline.
- "agent any" indicates that the pipeline is not tied to a specific agent label and may run on any node that is free when the execution kicks off.
- The first stage, “Checkout SVN repository” is in charge of downloading the source code of the project from a Subversion (SVN) repository. The checkout step is used from within a script block along required parameters.

- The second stage, “Clean and Test”, oversees cleaning the project and executing the tests using maven. A simple bat step is responsible for running the maven command "mvn clean test" (Windows host machine).
- Finally, a "post" block defines an "always" section in the scenario, and it will be executed whether the precedent step is successful or not. This block contains a cucumber step expected to handle a cucumber report for the tests using the project generated file `target/cucumber-report.json`. Jenkins analyzes this JSON report and delivers it in a user-friendly style. This presentation covers numerous important aspects: methodically deconstructed test scenarios, their separate phases, and a demonstration of the results of each test.

This pipeline can serve as a part of CI, however, whether it is a full CI depends on the context and requirements of the software development project.
The corresponding pipeline script is presented below:

```
pipeline {
    agent any
    stages {
        stage('Checkout␣SVN␣repo') {
            steps {
                echo "Checking␣out␣SVN␣repo..."
                script {
                    currentRevision = checkout changelog: false, poll: false, scm:
                    [
                        $class: 'SubversionSCM',
                        additionalCredentials: [],
                        excludedCommitMessages: '',
                        excludedRegions: '',
                        excludedRevprop: '',
                        excludedUsers: '',
                        filterChangelog: false,
                        ignoreDirPropChanges: false,
                        includedRegions: '',
                        locations:
                        [
                            [
                            cancelProcessOnExternalsFail: true,
                            credentialsId: 'ab31b425-82c7-4977-b8c9-9d4c3a56cb05',
                            depthOption: 'infinity',
                            ignoreExternalsOption: true,
                            local: '.',
                            remote: 'https://svn.rcslab.it/svn/Mito2/Dev/trunk/
                                applications/Mito/branches/MitoCube.2.0.R04/
                                Integrazione/integration-test'
                            ]
                        ],
                        quietOperation: false,
                        workspaceUpdater: [$class: 'UpdateUpdater']
                    ]
                }
                echo "Done␣checking␣out␣SVN␣repo..."
            }
        }
        stage('Clean␣and␣Test') {
            steps {
                echo "mvn␣clean␣test..."
                bat 'mvn␣clean␣test'
                echo "Done␣mvn␣clean␣test..."
            }
            post {
                always {
                    cucumber 'target/cucumber-report.json'
                }
            }
        }
    }
}
```

3.3 Challenges

3.3.1 Dependencies and Integration Testing Complexity

The effective handling of dependencies between distinct microservices, guaranteeing their harmonic interaction, was a major problem in the microservices setup. This included coordinating versions, settings, and communication protocols across various services, such as testing the API under test /search in Activitylist microservice, which has a dependent implementation of logic on several microservices, such as Xplora, Dendron, Preference, and Permission.

A planned strategy was used to effectively solve this difficulty. A realistic method was implemented rather of relying simply on abstract coordination. Docker images with exposed ports were developed for each of the dependent microservices. This assured that the API under test, /search, could call the relevant APIs from the dependent microservices and obtain the requisite replies. This method provides a tangible and practical manner of managing dependencies, improving compatibility, and allowing for thorough testing of interactions across microservices. Furthermore, it aided in the creation of a more streamlined and efficient testing environment, fostering the dependable coordination of interdependent services.

3.3.2 Data Setup

Setting up the necessary data for testing may be difficult, especially when various situations demand different database setups. It is critical to manage data consistency and integrity throughout testing.

Addressing this challenge entailed the adoption of a thorough data management approach. This multimodal strategy included the use of database seeding scripts and fixtures to create customized data scenarios for testing. Notably, this technique included the inclusion of pre-filled databases holding data closely approximating real-world circumstances, allowing for the testing of specific data scenarios. Docker containers were fundamental in managing database instances, allowing for the quick creation of isolated environments to handle various test situations.

3.3.3 Resource Management and Time Constraints

Managing resource constraints, such as memory and CPU limits, was a significant difficulty, particularly in a resource-intensive testing environment. Maintaining the consistency and dependability of testing findings while dealing with these constraints necessitated close attention.

Several tactics were used to monitor resource management and timing constraints, such as the charts provided by Docker Desktop. Reusing the containers during test was a significant time savior, adding to that deleting containers created for testing following test completion was a critical part of resource management. Despite these resource improvements, the testing environment required a significant number of resources to be operational, often requiring 6-8 minutes to attain readiness using a maximum of 9GB of the allocated memory (24.27GB) and a maximum of 400% of the CPU (8 cores allocated equivalent to 800%), as displayed in the Figures 3.4 and 3.5.

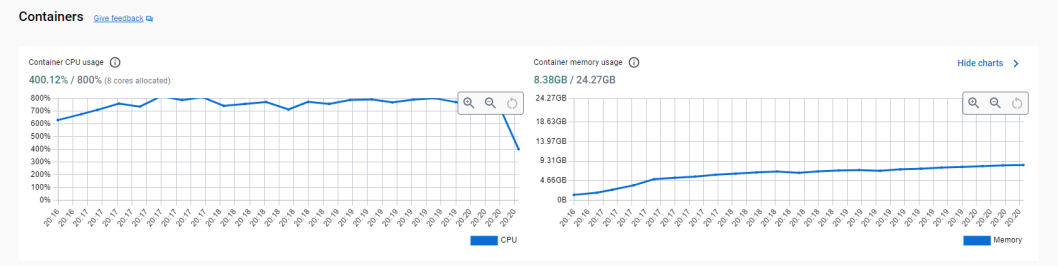


Fig. 3.4 Docker Desktop Charts (Screenshot 1) - 4 minutes into the testing phase: Containers are going up.

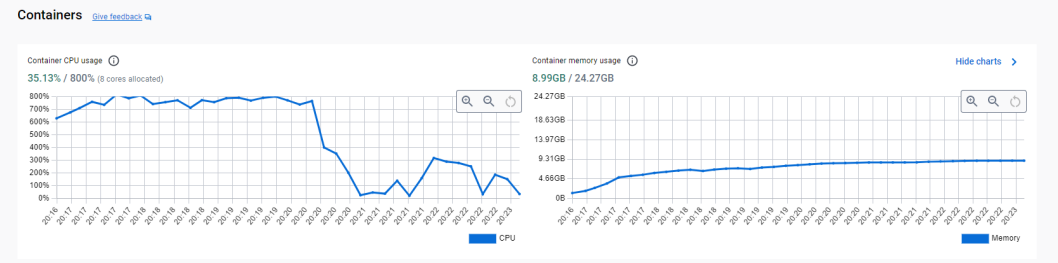


Fig. 3.5 Docker Desktop Charts (Screenshot 2) - 7 minutes into the testing phase: Containers are up and running and tests are almost done.

3.3.4 WireMock vs. Docker Images

A big difficulty arose during the preparation of our integration testing strategy while picking between two unique ways for replicating the behavior and interactions of microservices. The main decision was whether to use the WireMock framework, a lightweight, in-memory mock server, or Docker images to represent each of the needed microservices. Both choices had advantages and drawbacks.

On the one hand, WireMock provided the benefit of rapid and simple API response simulation. It allowed us to design and manage stubs for individual API endpoints, giving us fine-grained control over the simulated answers.

Using Docker images of actual microservices, on the other hand, offered a more accurate depiction of our production environment in our testing process. This method allowed us to examine real-world interactions between microservices, including dependencies and communication paths. Docker containers were capable of emulating real-world circumstances, exposing integration flaws that would have been hidden behind the simplicity of mocks.

After careful research, we determined that using Docker images of real microservices was the best strategy for our integration testing needs. Several things influenced this decision. While requiring more initial setup and resource management, Docker images enabled a faithful simulation of microservice interactions, mimicking the intricacies of the real-world production environment. This method would allow for thorough testing of end-to-end scenarios in the future, including data persistence, communication paths, and possible bottlenecks. This move helped to a more reliable integration testing process by lowering the likelihood of inconsistencies between the two environments and boosting the overall robustness of our testing efforts by aligning our testing environment more closely with the production environment.

Chapter 4

Results and Discussion

4.1 Covered Scenarios

Comprehensive testing is critical in the context of software development and quality assurance to assure the functionality, stability, and resilience of application programming interfaces (APIs). The focus has been centered around examining the functionality, performance, and reliability of the backend applications. Integration tests were rigorously created to guarantee that these backend components work fluidly and respond appropriately to varied input conditions. The tests entail the creation of HTTP requests as well as forceful validations of return responses. It is crucial to note that, while these integration tests thoroughly cover backend operations, they do not cover frontend logic or interactions that a user may perform. Instead, the emphasis has been on testing the essential backend functionalities, assuring data integrity and the proper operation of the microservices.

4.1.1 Clientsadmin Scenarios

A collection of precisely developed test cases for Clientsadmin API are included, which is a vital microservice in the ecosystem. These test scenarios cover a wide range of use cases, investigating Clientsadmin API's functionality under various settings and user interactions.
The goal of putting the API through these stringent tests is to evaluate its capacity and to administrate multiple entities (PrinterModel, Printer, Client, and Burningstation)

while ensuring data integrity, security, and user experience. These scenarios not only provide quality assurance but also information about the API's performance, robustness, and conformance to functional requirements. The purpose of this systematic examination is to verify that the Clientsadmin API functions as an essential and dependable components of the infrastructure.

PrinterModel

Starting with PrinterModel endpoint, it is put through tests: collecting printer models, executing POST and PUT requests, and dealing with deletions. Maintaining the integrity of printer models is critical for overall system consistency.

Table 4.1 Conducted tests under the tag @ClientsAdmin-PrinterModel

Target	Description
GET /printer/model	• Fetch a non existent printer model • Fetch a single printer model • Fetch more than one printer model
POST /printer/model	• Create without sending data • Create a single printer model • Create a not unique printer model (Name)
PUT /printer/model	• Update without sending data • Update a single printer model • Update non existent printer model
DELETE /printer/model	• Delete without sending an id • Delete a single printer model • Delete using an invalid id • Delete using non existent id
GET /printer/model by id	• Fetch a single printer model by id • Fetch using a non existent id • Fetch using an invalid id

The above tests are considered as standard or "baseline" tests that focus on the basic functionality and behavior of the /printer/model API, with no specific tests related to business logic, as the API works independently. The conducted tests on this endpoint can be shown in the Table above 4.1.

Client and Printer

Similarly, for the /client and /printer endpoints, tests encompass retrieving client and printer lists, POST and PUT requests, handling various error conditions, and verifying the relationships between printer models, printers, and clients.

These tests include both baseline scenarios for printer functionality and extra scenarios focusing on printer model dependencies. Tests connected to POST requests on /printer endpoint, in particular, emphasize the significance of supplying appropriate printer models when creating new printers, stressing the dependent relationship between these two entities.

Table 4.2 Conducted tests under the tag @ClientsAdmin-Printer

Target	Description
GET /printer	• Fetch a non existent printer • Fetch a single printer • Fetch more than one printer
POST /printer	• Create without sending data • Create printer with non existent printer model • Create without assigning to it a printer model • Create a single printer • Create a not unique printer (Name) • Create a not unique printer (MAC Address)
PUT /printer	• Update without sending data • Update non existent printer
DELETE /printer	• Delete without sending an id • Delete a single printer • Delete using an invalid id • Delete using non existent id
GET /printer by id	• Fetch a single printer by id • Fetch using a non existent id • Fetch using an invalid id

Some of the tests are especially designed to address the following requirements: unique printer names, unique printer MAC addresses, and printer model selection. The conducted tests on this endpoint can be shown in the Table above 4.2.

Baseline tests are implemented for /client, similarly to previous tests, additional tests are especially designed to address the following requirements: unique client names, unique client MAC addresses, and activation of the link established between clients and printers. The conducted tests on this endpoint can be shown in the Table below 4.3.

Table 4.3 Conducted tests under the tag @ClientsAdmin-Client

Target	Description
GET /client	• Fetch a non existent client • Fetch a single client • Fetch more than one client
POST /client	• Create without sending data • Create client with non existent printer • Create without assigning to it a printer • Create a single client • Create a not unique client (Name) • Create a not unique client (MAC Address)
PUT /client	• Update without sending data • Update non existent client
DELETE /client	• Delete without sending an id • Delete a single client • Delete using an invalid id • Delete using non existent id
GET /client by id	• Fetch a single client by id • Fetch using a non existent id • Fetch using an invalid id
GET /printer/getclients	• Fetch non existent client • Fetch client linked to printer • Fetch client linked to printer(2 diff clients) • Fetch clients linked to printer

Burningstation

For the /burningstation endpoint, tests include checking the retrieval of burning stations, making POST requests, handling invalid input, and verifying the behavior when interacting with burning stations, printers, and clients by validating that both successful and unsuccessful scenarios are appropriately handled.

Table 4.4 Conducted tests under the tag @ClientsAdmin-BurningStation

Target	Description
GET /burningstation	• Fetch a non existent burning station • Fetch a single burning station • Fetch more than one burning station
POST /burningstation	• Create a single burning station
PUT /burningstation	• Update non existent burning station
DELETE /burningstation	• Delete without sending an id • Delete a single burning station • Delete using an invalid id • Delete using non existent id
GET /burningstation by id	• Fetch a single burning station by id • Fetch using a non existent id • Fetch using an invalid id
GET /burningstation/getclients	• Fetch non existent client • Fetch client using an invalid id • Fetch client using a non existent id
PUT /burningstation/connectclients	• Connect client to a burning station • Connect clients to a burning station • Connect client to a burning station • Connect client to a non existent id • Connect client to invalid id
PUT /burningstation/disconnectclients	• Disconnect client from station • Disconnect clients from station • Disconnect non existent client • Disconnect client from non existent id • Disconnect client from invalid id

Additionally, tests for connecting and disconnecting clients to burning stations have been covered. These tests collectively ensure that /burningstation endpoint is robust and reliable, addressing diverse scenarios to provide a significant level of confidence in its functionality. The conducted tests on this endpoint can be shown in the Table above 4.4.

By implementing this extensive set of tests, not only a validation on the core functionality of the covered APIs, but also addressing various edge cases and potential error scenarios. This approach helps in building robust and reliable APIs that can handle a wide range of real-world situations, contributing to the overall quality and stability of your software system.

4.1.2 Activitylist Scenarios

The Activitylist Search API is an important component of the software system. A thorough suite of test scenarios has been created to ensure its smooth operation. These examples include a wide range of functionality and usage patterns, from simple searches to more complex interactions. The API's replies to requests of differing complexity are examined, and assessments are carried out in various user settings. The examples address query size, sorting, user-specific searches, and error handling, targeting the API's resilience, correctness, and capacity to smoothly handle a wide range of real-world circumstances.
Additionally, they also evaluate the API's performance in terms of offering enriched data insights based on query parameters.The goal of these defined tests is to guarantee that the Activitylist search APIs not only meet its intended purpose, but also function as a dependable and responsive component without the software's ecosystem.

Search

Starting from the test scenarios written under the tag named **@ActivityList-Search**, the search API capabilities are introduced by searching for some specific values using various pre-defined criteria. It also evaluates the API's flexibility by verifying that the pre-defined criteria on different parameters, in the request query, are respected and reflected in the received response.
As dictated in Table 4.5, the API is evaluated in the first couple of scenarios for its

ability to search for certain “liid” values, with varied query sizes. The third and fourth example have a more complicated query with an alternation of the operator between the condition: non-empty URI (AND/OR) ending in “.com”.

Query Size	Liid	More Rules	Returned Items
100	20080312044460		100
50	20080312044460		50
50	17110818000063	Size > 10000 AND (URI not empty AND URI ends with .com)	50
50	17110818000063	Size > 10000 AND (URI not empty OR URI ends with .com)	50

Table 4.5 Conducted tests under the tag @ActivityList-Search

@ActivityList-DiffUsers covers the performance of the API under various user credentials. It includes a variety of test scenarios, each characterized by a distinct combination of "liid", "username", "password", and the anticipated number of Lius (items) with a matching "liid". The scenario outline begins with the login, which results in the acquisition of a JWT. Following that, a query is built with a specific "liid" value. The final step is to ensure that the number of recovered items from the received response matches the predicted number for the provided "liid". The example table is parameterized with various "liid" values, users, passwords, and expected counts, these values are noted in the below Table 4.6.

Liid	Username	Password	Total Returned Lius
20080312044460	bgemmi	12345678	50
20080312044460	flavio	12345678	50
18092814114376	bgemmi	12345678	50
18092814114376	flavio	12345678	0
17110818082492	flavio	12345678	0
17110818082492	bgemmi	12345678	0

Table 4.6 Conducted tests under the tag @ActivityList-DiffUsers

@ActivityList-SearchFails is intended to assess how the system handles invalid input parameters in its search functionality. It is a parameterized approach, with test cases covering different "category" and "querysize" value combinations. Specifically, when an invalid input is submitted, it ensures that the client receives the expected error code (e.g., 400 for bad request). As stated in Table 4.7, the conducted tests includes examples of null "category" values, empty "category" values, and negative "querysize" values, enabling comprehensive error handling for many forms of erroneous inputs during search queries.

Category	Query Size	Expected Error Code
null	0	400
	0	400
events	-1	400

Table 4.7 Conducted tests under the tag @ActivityList-SearchFails

@ActivityList-SearchWithSort dives into the API's behavior after enabling the sort functionality. This parameterized scenario evaluates the system's responsiveness when looking for data based on certain sorting criteria. It starts by making a query for the "events" category, specifying the query size, and enabling sorting based on a given field and order (ascending or descending). These tests verify whether the API correctly returns the results based on provided search criteria and "liid" values, with a focus on different combinations of "querysize" and "liid" values. The above mentioned tests can be seen in the Table below 4.8.

Query Size	Field	Order	Returned Lius	First LiuId	Last LiuId
607	l	asc	607	61665	61665
607	l	desc	607	83275	61665
620	l	desc	607	83275	61665

Table 4.8 Conducted tests under the tag @ActivityList-SearchWithSort

In **@ActivityList-SearchEnrichLius**, the emphasis is on determining the API's ability to produce accurate results depending on certain query parameters. These scenarios cover situations such as looking for specified "liid" values with given query

sizes and predicted total Lius counts and ensuring the returned data matches particular criteria such as "caseName", "caseId", "leaId", "voiceTargetId", and "liidStatus", and that the comments connected with the findings are proper. These characteristics are part of the data enrichment process that occurs during query execution. This step entails getting extra information from other microservices in order to improve the query results. Hence, not just the core query criteria is checked, but also the integrity of additional data obtained from external sources, to verify the overall quality and completeness of the ActivityList Search API's results. The above mentioned tests can be seen in the Table below 4.9.

Liid	Query Size	Total Returned Lius
20080312044460	10	607
18092814114376	10	0
19040309210322	10	0
20092216251673	10	327
17110818000063	10	1500
17112409114079	10	15

Table 4.9 Conducted tests under the tag @ActivityList-SearchEnrichLius

Search After

@ActivityList-SearchAfter focuses on ensuring that the search feature with the search after capability functions properly. Various test cases are run in this set of scenarios to verify the API's performance and correctness. Each scenario involves creating an event query, selecting the query size, and providing a reference to a specified "liuid".

These scenarios cover a variety of query sizes and "liuId" values, with the goal of ensuring that the API delivers the required number of items with the proper "liid" values. Additionally, the "liuId" values of the first and last items in the returned array of results are checked to verify correct ordering. It's worth noting that some cases contain negative "liuId" values, which means that the liuid is not provided and the API should function as a normal search, or queries with no anticipated results as they intend to test border cases.

Overall, these scenarios thoroughly evaluate the validity and effectiveness of the

ActivityList Search After API's functionality. There are no obvious flaws in the given table, but the ordering of received "lius" in contrast to "liuId" values should be thoroughly reviewed to guarantee correctness. The above mentioned tests can be seen in the Table below 4.10.

Query Size	LiuId	Returned Lius	First LiuId	Last LiuId
10	-1	10	83275	83235
10	83265	10	83258	83191
20	83162	20	83149	83079
607	-1	607	83275	61665
10	83275	10	83273	83223
10	61665	10	83273	83235
700	-1	607	83275	61665
800	61665	0	-1	-1
800	83275	606	83273	61665
607	61665	0	-1	-1
607	83275	606	83273	61665

Table 4.10 Conducted tests under the tag @ActivityList-SearchAfter

In summary, these extensive test cases provide in-depth coverage of the ActivityList Search and Search After APIs. They analyze its essential functionality, error handling, and how it responds to different user scenarios. These tests are critical in confirming the APIs' dependability and accuracy in a variety of real-world circumstances, hence, improving the overall quality of the software system it serves.

4.2 Results

As part of our extensive integration testing approach, 118 tests were conducted. Within the RCS lab's microservices-based application ecosystem, the 118 test cases demonstrate the wide coverage of scenarios and interactions performed to examine the functionality, performance, and reliability of the tested endpoints. The results of these tests demonstrate insights about the small piece of the system's overall health and preparedness, serving as a core indication of its robustness and quality.

4.2.1 Cucumber Report

Html Report

The chosen testing framework's HTML report is a useful tool for visualizing the outcomes of the conducted integration tests, as shown in Figure 4.1. It has a user-friendly UI that makes interpreting test results easier. Users may quickly identify both succeeded and failed cases within this report, offering a comprehensive summary of the testing process. Furthermore, the HTML report include unique insights and crucial results, such as identifying essential test cases that require immediate attention or emphasizing regions of the program that have shown stability.

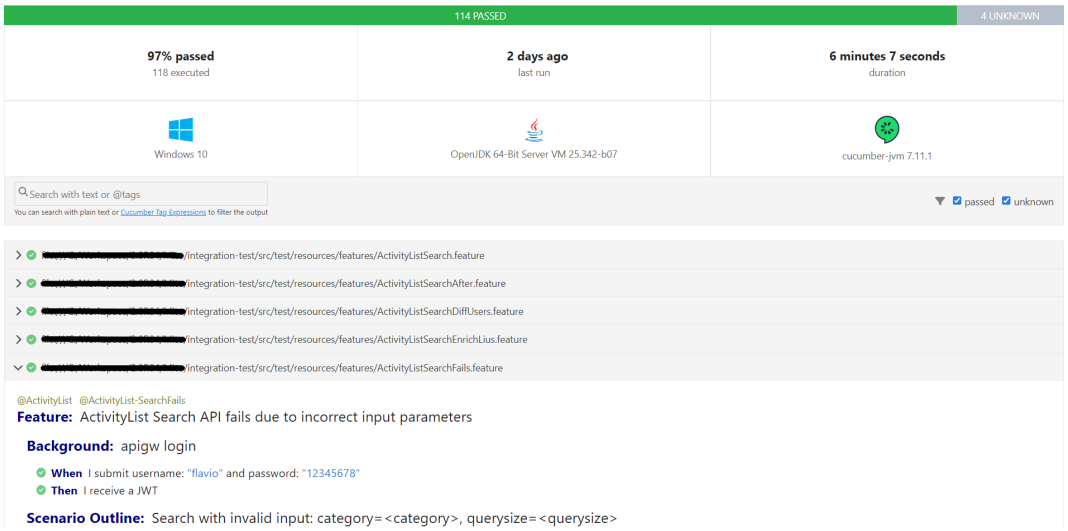


Fig. 4.1 Cucumber HTML Report

JSON Report

Futhermore, we create a JSON report that is particularly designed for interaction with our Jenkins workflow. This JSON report serves as a structured data source for Jenkins, allowing it to provide summaries of test results within the Jenkins environment. The transformation can be shown from Figure 4.2 to Figure 4.3.

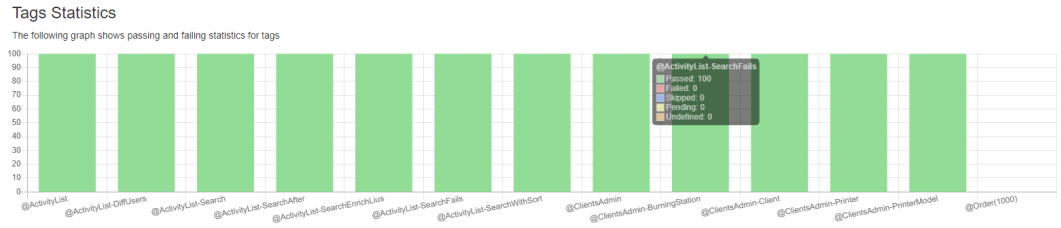
```
[
  {
    "line": 2,
    "elements": [
      {
        "line": 4,
        "name": "apigw login",
        "description": "",
        "type": "background",
        "keyword": "Background",
        "steps": [
          {
            "result": {
              "duration": 3143000000,
              "status": "passed"
            },
            "line": 5,
            "name": "I submit username: \"flavio\" and password: \"12345678\"",
            "match": {
              "arguments": [
                {
                  "val": "\"flavio\"",
                  "offset": 19
                },
                {
                  "val": "\"12345678\"",
                  "offset": 42
                }
              ],
              "location": "it.rcslab.cucumber.activitylist.ActivityListStepDefinition.loginNEW(java.lang.String,java.lang.String)"
            },
            "keyword": "When "
          },
          {
            "result": {
              "duration": 18000000,
              "status": "passed"
            },
            "line": 6,
            "name": "I receive a JWT",
```


Fig. 4.2 Cucumber JSON Report

4.2.2 Jenkins Pipeline Reports

Cucumber Report

Adding to the abovementioned, among the Jenkins plugin exists a Cucumber plugin that provides enhanced reporting capabilities for Cucumber test results within Jenkins based on the previously generated JSON report.
The choice about whether to use the Cucumber-generated HTML report within your Java application or rely on Jenkins for reporting is determined by the requirements. Both techniques offer advantages, and the choice should be based on the team's workflow. The Jenkins Cucumber Report Plugin is an excellent solution if centralized reporting, historical data analysis, and easy interaction with Jenkins are the target, otherwise, the Cucumber-generated HTML report may be preferable. The detailed Jenkins Cucumber Report can be seen in the below Figure 4.3.



	Steps						Scenarios			Features	
Tag	Passed	Failed	Skipped	Pending	Undefined	Total	Passed	Failed	Total	Duration	Status
@ActivityList	400	0	0	0	0	400	33	0	33	20.843	Passed
@ActivityList-DiffUsers	66	0	0	0	0	66	6	0	6	2.490	Passed
@ActivityList-Search	51	0	0	0	0	51	4	0	4	6.557	Passed
@ActivityList-SearchAfter	132	0	0	0	0	132	11	0	11	8.524	Passed
@ActivityList-SearchEnrichLius	106	0	0	0	0	106	6	0	6	1.285	Passed
@ActivityList-SearchFails	9	0	0	0	0	9	3	0	3	0.150	Passed
@ActivityList-SearchWithSort	36	0	0	0	0	36	3	0	3	1.835	Passed
@ClientsAdmin	350	0	0	0	0	350	85	0	85	10.888	Passed
@ClientsAdmin-BurningStation	124	0	0	0	0	124	26	0	26	2.854	Passed
@ClientsAdmin-Client	108	0	0	0	0	108	23	0	23	5.219	Passed
@ClientsAdmin-Printer	63	0	0	0	0	63	19	0	19	1.999	Passed
@ClientsAdmin-PrinterModel	55	0	0	0	0	55	17	0	17	0.815	Passed
@Order(1000)	0	0	0	0	0	0	4	0	4	0.000	Passed
	1500	0	0	0	0	1500	240	0	240	1:3.464	13
	100.00%	0.00%	0.00%	0.00%	0.00%		100.00%	0.00%			100.00%

Fig. 4.3 Cucumber Report Pluggin in Jenkins

Stage View

The Stage View in Jenkins pipeline, as displayed in Figure 4.4, serves as a visual dashboard and presents a simple, high-level overview of the whole testing process. It provides a visual depiction of each stage of the pipeline, making it simple to determine which stages have passed successfully and which have failed. This high-level overview is crucial for immediately spotting any bottlenecks or difficulties in the pipeline.

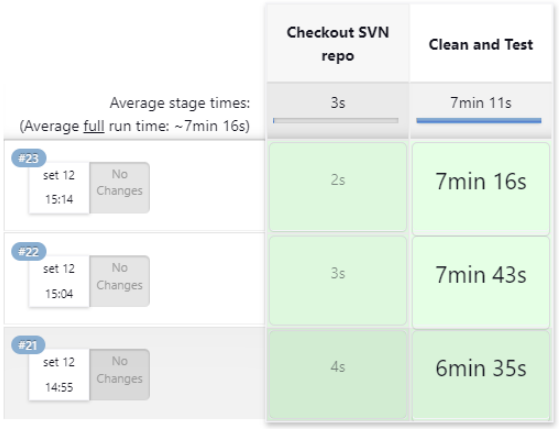


Fig. 4.4 Jenkins Stage View

Chapter 5

Conclusion and Future Work

5.1 Summary of the Study

In this research, we embarked on a comprehensive exploration of software integration testing, focusing on the critical aspect of API testing. We aimed to assess the functionality, performance, and reliability of some endpoints of a couple of microservices within the RCS lab's application ecosystem. This study encompassed 118 integration tests, which rigorously examined various scenarios and interactions to ensure the robustness and quality of what is tested. The key findings include:

- The effective validation of the covered APIs' main functionality, establishing confidence in their dependability.
- Identifying opportunities for improvement in error handling and boundary conditions that might improve the overall stability of the software system.
- Insights on the system's readiness to deal with a variety of real-world events, which contributes to its general health and resilience.

5.2 Contributions and Implications

This research made several significant contributions to the field of software quality assurance and microservices testing at RCS lab. These contributions include:

- A comprehensive set of 118 integration tests that can serve as a benchmark for assessing the quality of similar microservices-based applications.
- Valuable insights into the performance, functionality, and reliability of microservices, aiding in the development of robust software systems.
- Practical implications for software developers and quality assurance teams, emphasizing the importance of thorough testing to ensure the dependability of microservices.

The implications of this thesis work extend to the development and maintenance of microservices-based applications, where these findings can guide best practices in testing and quality assurance processes. Additionally, this research highlights the need for ongoing monitoring and improvement of microservices to enhance their reliability and resilience.

5.3 Future Research Directions

While the study has yielded useful results, it is not without limitations. These limitations include:

- The focus on a narrow collection of microservices within RCS lab's ecosystem, which limits the generalizability of the findings to other domains.
- The lack of real-time testing scenarios, which might enable a more complete assessment of the performance of microservices under dynamic situations.
- The reliance on predefined test cases, which may not cover all real-world scenarios.

Future research in this field can expand on our findings in the following ways:

- Broadening the scope of testing to encompass a broader range of microservices and domains, allowing for a more thorough study of microservices-based applications.
- Investigate the feasibility and advantages of adding end-to-end testing to the existing integration testing framework, with an emphasis on improving overall quality assurance and identifying possible system-level concerns.
- Training and Familiarization: Begin training sessions and workshops to acquaint the QA team with the testing frameworks and tools discussed in this thesis. This will allow team members to become proficient in the use of these resources.
- Quality Improvement Metrics: Use metrics and key performance indicators (KPIs) to measure the efficacy and efficiency of the QA team's testing operations. These indicators can help you track your success and discover areas for improvement.

References

[1] G. Malnati. Slides on Modularità (Organizzare il codice sorgente) - Le domande del test. Lecture Slides, Politecnico di Torino, Department of Control and Computer Engineering, 2021-23.

[2] Vladimir Khorikov. *Exploring Mocks in Unit Testing*. Manning Publications Co., 2020.

[3] GitLab B.V. How to shift left with continuous integration. *https://about.gitlab.com/topics/ci-cd/shift-left-devops/*, 2023.

[4] Arvinder Saini. How much do bugs cost to fix during each phase of the sdlc? Jan 11, 2017.

[5] GitLab B.V. What is continuous integration (ci)? *https://about.gitlab.com/topics/ci-cd/benefits-continuous-integration/*, 2023.

[6] Andy Gumbercht Alex Soto Bueno and Jason Porter. *Testing Java Microservices*. Manning Publications Co., 2018.

[7] Jason Clark Benjamin J. Evans and Martijn Verburg. *The Well-Grounded Java Developer*. Manning Publications Co., 2022.

[8] Gayathri Mohan. *Full Stack Testing*. O'Reilly Media, Inc, 2022.

[9] MARK WINTERINGHAM. *Testing Web APIs*. Manning Publications Co., 2022.

[10] Xiao Pu. Introduction to static analysis. *https://se-education.org/learningresources/contents/staticAnalysis/intro.html*, 2021.

[11] Roy Osherove. *The Art of Unit Testing with examples in JavaScript*. Manning Publications Co., 2022.

[12] Jay Fields. *Working Effectively with Unit Tests*. CreateSpace Independent Publishing Platform, 2018.

[13] Daniel Irvine. *Mastering React Test-Driven Development, Second Edition*. Packt Publishing Ltd., 2022.

[14] Katalon. Introduction to continuous testing | continuous testing 101. *https://medium.com/katalon-studio/introduction-to-continuous-testing-continuous-testing-101-5cb5bbfb4623*, 2019.

[15] Spinnaker. *https://spinnaker.io/*, 2023.

[16] Inc. Docker. *https://www.docker.com/*, 2023.

[17] Richard North and other authors. *https://java.testcontainers.org/*, 2015-2021.

[18] SmartBear Software. *https://cucumber.io/*, 2023.

[19] Johan Haleby. *https://github.com/rest-assured/rest-assured/wiki/Usage*, 2023.

[20] Johan Haleby. *http://www.awaitility.org/*, 2023.

[21] WireMock. *https://wiremock.org/*, 2023.

[22] *System Operation and Control*.

[23] Matt Wynne and with Steve Tooke Aslak Hellesøy. *The Cucumber Book Second Edition*. The Pragmatic Bookshelf, 2017.

Acknowledgements

to my family and friends