Degree Thesis, Åland University of Applied Sciences, Degree Programme in Information Technology

Developing Automated Functional Tests for REST APIs

Oscar Eriksson, Anna Söderlund

2025:24

Date of approval: 24.5.2025
Academic Supervisor: Joakim Isaksson

DEGREE THESIS
Åland University of Applied Sciences

Degree Programme:	Information Technology
Author:	Oscar Eriksson, Anna Söderlund
Title:	Developing Automated Functional Tests for REST APIs
Academic Supervisor:	Joakim Isaksson
Commissioned by:	External employer

Abstract

In our thesis we write about automated functional tests for REST APIs. We also describe software testing in general as well as the technologies used in the project. The project was commissioned by an external employer and the goal was to create automated tests that thoroughly test the APIs for an application in a credit management system. At the end of our thesis we go through how we actually implemented the project.

The purpose of the project was to decrease the amount of manual regression testing over time as well as increase the quality of the application.

In the project we have used frameworks such as Cucumber, to write tests in natural language, and WireMock to mock communication between the application and external services.

The result was a testing environment where it is easy to write new tests, as well as a collection of tests that acts as a proof of concept.

Keywords

Cucumber, Functional Testing, REST API, WireMock

Serial number:	ISSN:	Language:	Number of pages:
2025:24	1458-1531	English	65 pages

Handed in:	Date of presentation:	Approved:
7.5.2025	22.5.2025	24.5.2025

EXAMENSARBETE
Högskolan på Åland

Utbildningsprogram:	Informationsteknik
Författare:	Oscar Eriksson, Anna Söderlund
Arbetets namn:	Utveckling av automatiserade funktionella tester för REST API:er
Akademisk handledare:	Joakim Isaksson
Uppdragsgivare:	Extern uppdragsgivare

Abstrakt

I vårt examensarbete skriver vi om automatiserade, funktionella tester för REST API:er. Vi redogör även för mjukvarutestning i allmänhet samt för de teknologier vi har använt i projektet. Arbetet har gjorts på beställning av en extern uppdragsgivare och målet var att skapa automatiserade tester som utförligt testar API:n för en applikation inom ett kreditansökningssystem. I slutet av uppsatsen går vi igenom hur vi implementerade projektet i praktiken.

Syftet med arbetet var att på sikt minska på mängden manuell regressionstestning samt att höja kvaliteten på applikationen.

I projektet har vi använt oss av ramverk som, bland annat, Cucumber, för att skriva tester i klartext, och WireMock för att isolera applikationen från externa tjänster.

Resultatet blev en testningsmiljö där man lätt kan skriva nya tester, samt en uppsättning tester som demonstrerar funktionaliteten.

Nyckelord (sökord)
Cucumber, Funktionell testning, REST API, WireMock

Högskolans serienummer:	**ISSN:**	**Språk:**	**Sidantal:**
2025:24	1458-1531	Engelska	65 sidor

Inlämningsdatum:	**Presentationsdatum:**	**Datum för godkännande:**
7.5.2025	22.5.2025	24.5.2025

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Purpose

The purpose of this project is to create a proof of concept of automated functional end-to-end testing of REST APIs[1]. The work was commissioned by an external employer and the application tested is part of their credit management system. The long term goal of the project is to minimize the need for manual regression testing[2] whenever changes are made to the system.

1.2 Method

The work with the project consisted of several steps and involved a few key participants apart from ourselves. These key participants were:

- Our scrum master, who helped us structure and plan the work.
- Our technical supervisor, who had technical knowledge of the domain and helped us make decisions on the tech stack to make it align with company standards.
- Our product owner, who helped us define test cases and understand the processes from a business perspective.

Our day-to-day work was conducted in an agile manner. We participated in a daily standup with the rest of the team each morning, where we gave a short update of our progress and had the opportunity to highlight any questions or problems we had experienced. Furthermore, we had a weekly status meeting with our technical supervisor and scrum master where we presented our progress in more detail and discussed how to move forward. When the project had progressed far enough for us to start actually writing the tests, we also involved our

[1] "A REST API is an application programming interface (API) that follows the design principles of the REST architectural style. REST is short for representational state transfer, and is a set of rules and guidelines about how you should build a web API." (RedHat, 2020)
[2] "Regression Testing is a type of testing in the software development cycle that runs after every change to ensure that the change introduces no unintended breaks" (BrowserStack, 2023).

project owner and/or a product specialist in these meetings to get their input on the testing objectives.

The first step of our project was to do the necessary preparations before any actual development could begin. We planned the work and defined the scope together with our scrum master, technical lead and product owner. This was a crucial step to make sure we had a scope that was meaningful and feasible from both a technical and a business point of view. This step also involved us manually testing out and familiarizing ourselves with the APIs that should be tested as well as researching functional testing.

The next step was to start building the tech stack. Each component of the tech stack was thoroughly researched, and based on our findings, we selected the frameworks and technologies we considered most suitable for our purposes. The tech stack will be discussed in more detail in Chapters 4.1 through 4.7.

With the tech stack in place we could start working on the first test suite. The purpose of this test suite was not primarily to test any functionality but to test out the tech stack and serve as a template for the other test suites. This test suite also helped us identify which API calls we needed to mock, to ensure that our tests run independently without relying on the availability of any internal or external services.

With this test suite as a template we could then start writing test suites focusing on testing the functionality according to the specifications provided by our product owner. The process of writing test suites will be covered in detail in Chapter 4.8, *Writing tests*.

The last step was to set up automation using Jenkins. This step also involved a discussion with our scrum master and technical supervisor regarding the test strategy, for example when, and how often, the tests would run. This will be addressed in Chapter 4.9, *Setting up automation in Jenkins*.

1.3 Limitations

During the work with this project, we had three major limitations to take into account, two regarding the development and one regarding the writing of this thesis.

Since developing and maintaining automated tests are a continuous and lasting effort we had to limit the scope of our project so we would have a well defined definition of done to work towards. We decided that our project would be considered completed when we had two complete and working test suites. This limitation is suitable since two test suites are enough to act as a proof of concept and show that every part of the testing procedure (including the tech stack) are working as intended. Adding more tests after the foundation is in place is more a matter of understanding the business logic and creating purposeful test cases than actual test development.

We also decided to limit the scope of the project to only the parts directly relating to functional testing. This means, for example, that the authentication needed to call the API endpoints, though crucial for making the tests work, is considered out of scope.

Regarding the writing of this thesis, we had one really important limitation to always keep in mind. This was to make sure to not include any confidential information about the customer or the employer's systems in general. Therefore the code in this thesis will be anonymized and/or use mock data when necessary to make sure we comply with the company's confidentiality guidelines while still illustrating our work in a comprehensive way.

2. SOFTWARE TESTING

2.1 About Software Testing

Testing is a crucial part of the software development process. Software testing can be defined as "...the process of evaluating and verifying that a software application or system meets its requirements and functions as expected. It involves testing to identify bugs or defects and ensure quality." (BrowserStack, 2025c). In other words, thorough software testing is a way of ensuring that the system does what it is supposed to do, when it is supposed to do it, in a safe and predictable way.

Software testing comes in many different forms and each one serves a different purpose. In this thesis we primarily focus on three categories of testing: unit testing, integration testing and functional testing. These will be described in more detail later, but briefly, they can be summarized as follows:

- Unit testing: testing each unit (component, class, function, etc.) separately to ensure that they behave as expected.
- Integration testing: testing that the separate units fit together.
- Functional testing: testing that the system as a whole delivers the functionality expected. Functional testing is also known as end-to-end testing and the terms are used interchangeably in this thesis.

Unfortunately, there is quite a bit of confusion when it comes to the terminology (Vocke, 2018). In particular, the term *functional testing* can cause confusion since another way of categorizing tests is to divide them into functional tests and non-functional tests. In this categorization, functional tests cover all tests regarding the functionality (such as unit tests and end-to-end tests) while non-functional tests are all other types of tests (such as performance tests and security tests) (Jain, 2025). In this thesis, the term *functional testing* always refers to end-to-end testing according to the definition in the list above.

2.2 The Test Automation Pyramid and Shift-Left Testing

The test automation pyramid (also known as simply the test pyramid), originating from the aircraft industry, was made popular in the software development sphere by Mike Cohn in 2009 (Radziwill & Freeman, 2020). While there are a lot of different interpretations of how to define the test automation pyramid, the underlying concept is always the same. The goal is to have a lot of unit tests, which are small, fast and good for finding bugs in an early stage of the development, while having a smaller amount of integration tests and end-to-end tests, since they are more complex and slower to run (Fowler, 2012). Figure 1 shows one interpretation of the test automation pyramid.

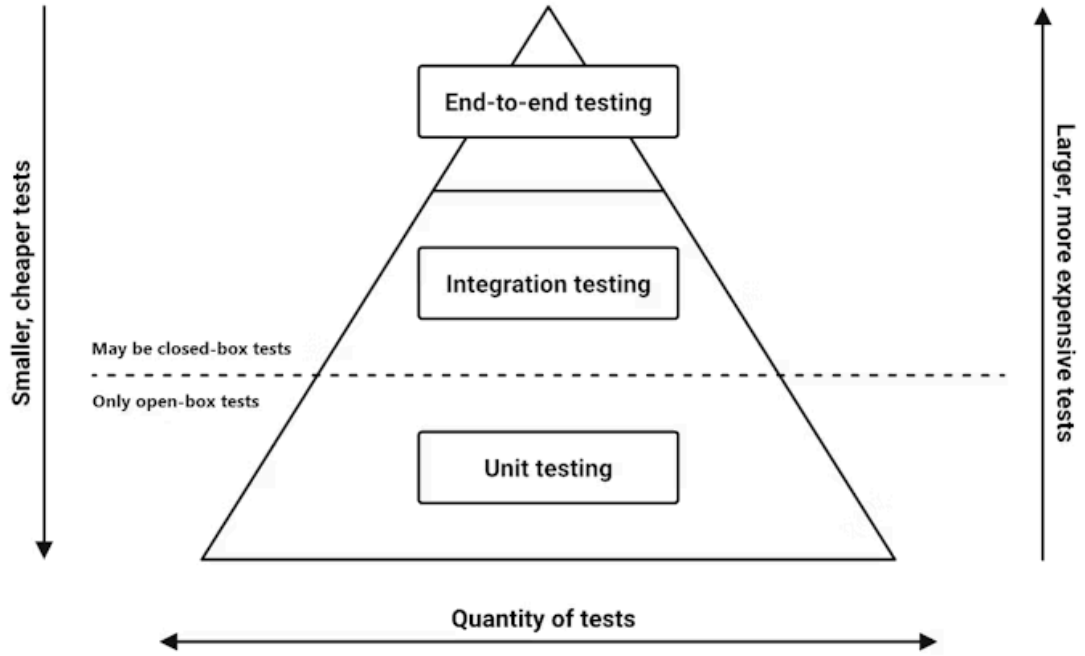

Figure 1. An illustration of the test automation pyramid (Schmitt, 2024).

The testing pyramid is related to a testing concept known as *shifting left*. Shift-left testing can be defined as follows: “Shift-Left Testing is an approach in software development that highlights the importance of incorporating testing activities early in the process to improve product quality, increase test coverage, and expedite time to market.” (Rani et al., 2023). This aligns well with the idea of the test automation pyramid to favor unit tests over more complex

types of tests. The term shift-left testing was originally introduced by Larry Smith in 2001 (Smith, 2001) and Figure 2 gives a graphic explanation of the concept.

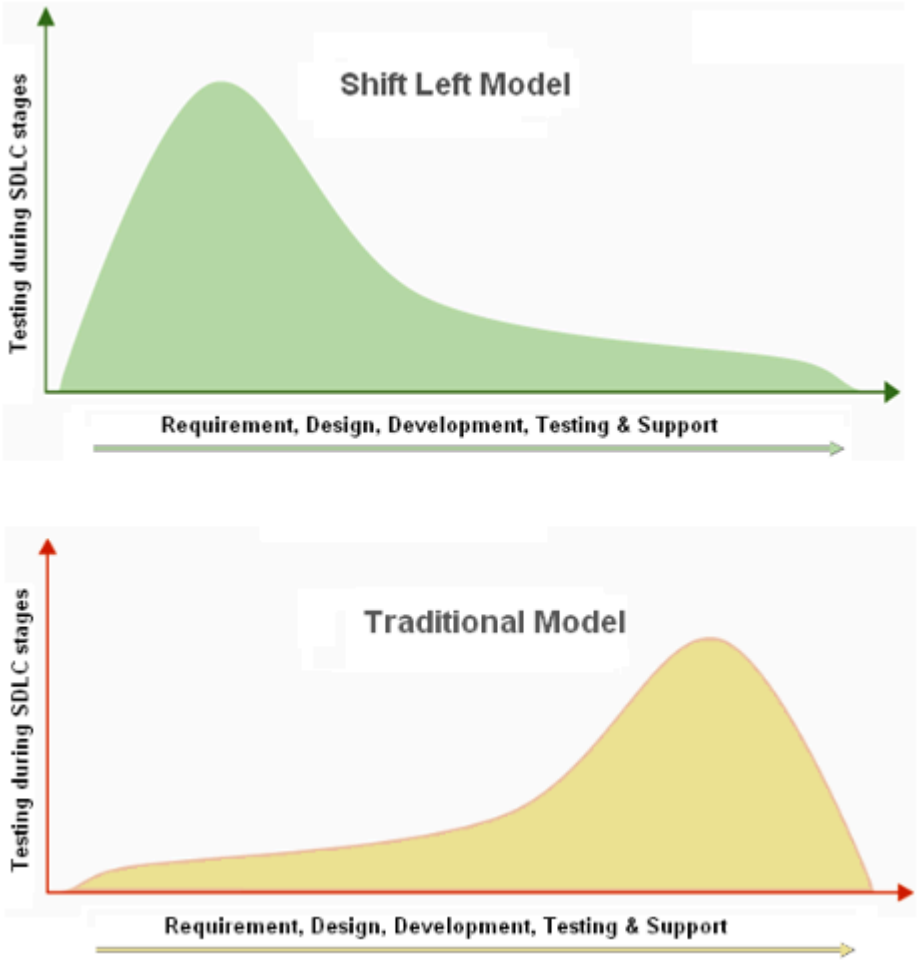

Figure 2. A graphic explanation of the term "shift-left testing" (Magowan, 2024).

Figure 3 illustrates the difference between the more traditional approach to testing and the shift-left testing model. The left side of the figure shows the traditional way of testing where most of the testing is done manually after the development is finished. The process can be described as "code and fix", where the focus is on finding bugs (Brown, 2014). Illustrating this process results in something called a *testing ice-cream cone* which is considered an anti-pattern[3] (Hartikainen, 2020). The right side of the figure illustrates a more agile, left-shifted process resulting in a typical test automation pyramid. This model emphasizes

[3] "The anti-pattern is a commonly-used process, structure or pattern of action that, despite initially appearing to be an appropriate and effective response to a problem, has more bad consequences than good ones." (*Anti-Pattern*, 2025).

identifying and resolving bugs at an early stage, thereby preventing them from developing into larger issues and becoming more costly to fix. The later a bug is discovered and addressed, the more expensive it typically is to eliminate (Cser, 2023). The model also reduces the amount of manual testing, allowing testers and product specialists to focus on testing user experience and so called exploratory testing when more extraordinary use cases are tested (*Exploratory Testing*, 2024).

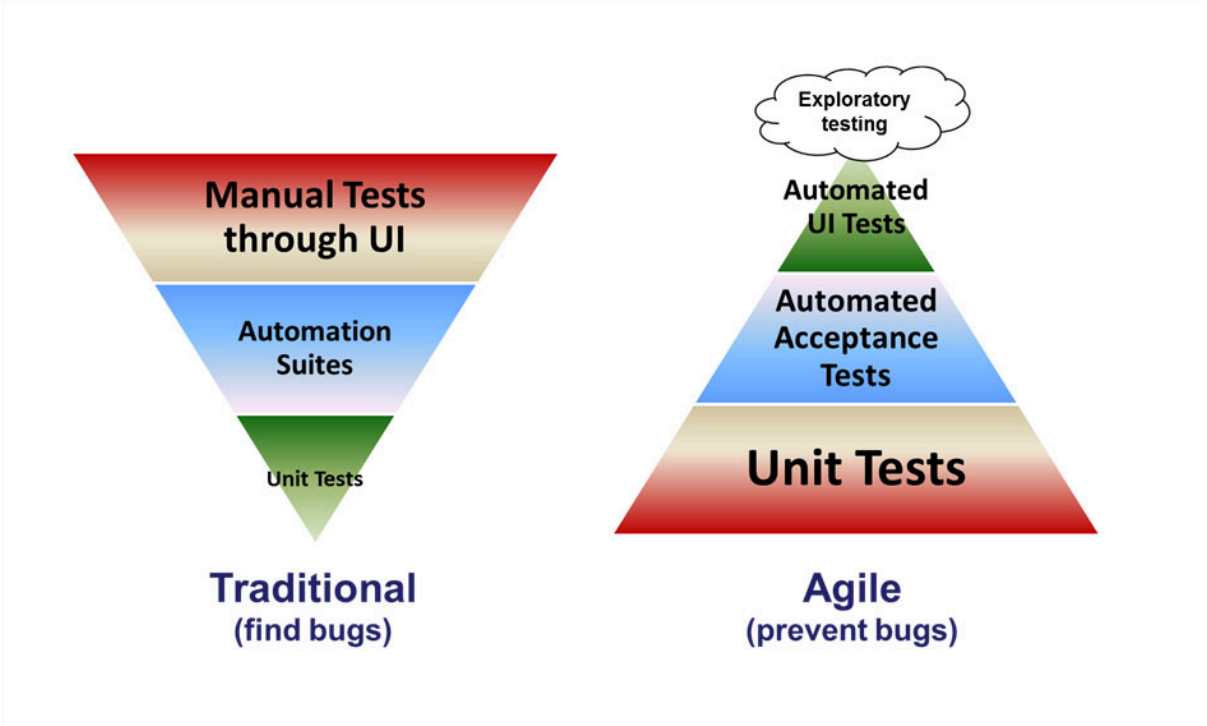

Figure 3. The testing ice-cream cone (left) and the testing pyramid (right) (Brown, 2014).

2.3 What is Unit Testing?

Unit testing is the most fundamental type of software testing. A unit test tests one unit of the code (a unit being, for example, a method or a function) in isolation, verifying that the code in question performs the correct action. Usually, unit tests are testing a method by inserting specific inputs and asserting that the output from the method aligns with the expected output. It is important to test the units with a wide range of inputs to assure expected behavior in both typical scenarios and edge cases. An example of this is illustrated in Figure 4, where the unit tested is the `divide`-method. The first test covers the behavior for various normal inputs and the other test assesses the behavior in an edge case. The tests use JUnit, a popular framework

for writing unit tests in Java, and they utilize the `@ParameterizedTest` and `@CsvSource` annotations to easily run the tests multiple times with different inputs (Dehghani, 2025).

```
@ParameterizedTest
@CsvSource({
        "10, 2, 5",
        "-10, 2, -5",
        "10, -2, -5",
        "-10, -2, 5",
        "0, 1, 0"
})
void testRegularDivision(int dividend, int divisor, int expectedResult) {
    assertEquals(expectedResult, divide(dividend, divisor));
}

@ParameterizedTest
@CsvSource({
        "1, 0",
        "-1, 0",
        "0, 0"
})
void testDivisionByZero(int dividend, int divisor) {
    assertThrows(ArithmeticException.class,
        () -> divide(dividend, divisor));
}

int divide(int dividend, int divisor) {
    return dividend / divisor;
}
```

Figure 4. Example of unit tests using JUnit.

Unit tests are typically small and fast, allowing a project to include many of them without consuming significant resources. They are often written by the developer responsible for the method, usually immediately after its implementation. This helps catch and fix bugs early in the development process. Unit tests are often automated and run everytime new code is added to the codebase, allowing developers to feel confident in refactoring code or adding new features, since they know that if the new code breaks any existing functionality, the unit tests will alert them about it. Writing unit tests can also encourage modular design, given that the more modular the code is, the easier it is to write unit tests for it. If a unit test requires multiple assertions to test the functionality it might be an indication that the tested unit is doing too much and should be divided into smaller, more specialized, units (Berga, 2024).

While unit tests offer many benefits, there are of course also some drawbacks to having a large number of them in the code. The first, and perhaps most tangible, is the extra work required to write and maintain them. Another drawback, especially in more complex projects, is that the unit tests may require mocking in order to test the units in isolation from the rest of the code. This can lead to tests that are more complex and therefore harder to maintain. Using too many mocks can also lead to a situation where so much of the functionality is replaced with mocks that the test no longer replicates real world scenarios enough to add any value or, in the worst case scenario, generate false positives allowing bugs to slip through to production.

One last drawback worth mentioning is the fact that having a codebase thoroughly tested with unit tests might lead to a false sense of security. Even if all unit tests pass and each part of the project works as expected in isolation, there is no guarantee that the system as a whole behaves as intended, since unit tests do not account for how the different components interact. This is why complementary testing, for example integration testing, is important.

2.4 What is Integration Testing?

Integration testing is a form of testing that lies one level above unit testing. After functional units have been thoroughly tested in isolation, there is a need to test the integration between these units to assert that they work properly together. These tests are meant for testing the interactions and data exchange between separate functional modules, ensuring that nothing goes wrong in the process.

Integration testing is a crucial step in making sure that the system that is being built is working as a whole and not only as individual components. There are a couple of important differences between integration tests, unit tests lower in the test stack, and the functional tests that lie higher. Unit tests test that individual functions and methods work as they should, and functional tests that the whole flow of a system functions correctly from beginning to end. Integration tests, on the other hand, tests the integration between larger modules that are made up of lower level programming components, to make sure that the communication between these modules function as they should.

There are a few different ways to go about integration testing, such as Big-Bang, Bottom-Up, Top-Down and Mixed integration testing.

Big-Bang testing is a sort of integration testing where all modules of a system are tested together to see that all integrations, communication and data flow between them function as they should. This sort of testing should only be conducted after all the modules have been developed and all their internal components have been unit tested. One of the benefits of Big Bang testing is that it is relatively simple to conduct; testing of the integrations between modules can start as soon as they are done, to make sure that they work correctly as a single unit. All the interfaces between the modules get checked to ensure they communicate with each other in the correct way, and that they send and get the correct data/parameters in the correct format. Disadvantages of Big Bang testing is that if the system being developed is complex and has many individual modules, there is a need to first develop and unit test each module before they can all be tested together. The time and resource use increases rapidly if there are multiple dependencies between the different modules due to the fact that all their integrations need to be verified against each other. It can also be harder to pinpoint exactly where bugs are coming from, since there is a need to test all the modules and check all their interfaces at the same time (tutorialspoint, n.d.-a). Figure 5 below illustrates the testing of integrations between all modules of a system.

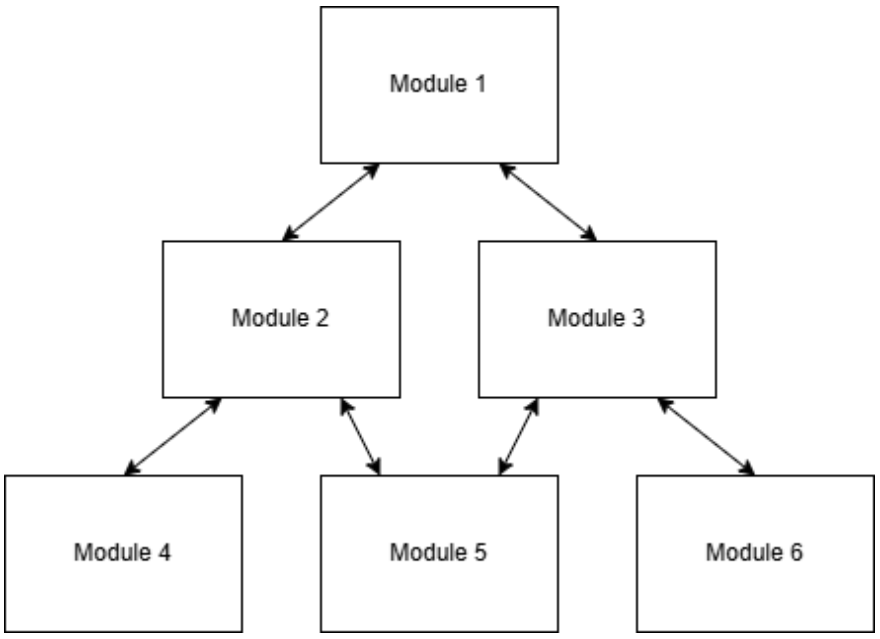

Figure 5. The Big Bang way of integration testing.

Bottom-Up is another way of integration testing that focuses on first testing the lower level components of a system, ensuring that they work correctly together and then moving up the hierarchy of modules until the whole system has been thoroughly tested. The core and utility modules are the first ones being tested, where integrations, communication and data flow are checked to make sure that they are working properly. These lower level modules are then incrementally grouped together and tested with modules higher up. When testing this way it is easier to identify errors between low-level modules early on in the testing process and thus easier to resolve these errors and bugs early to avoid facing them higher up in the hierarchy. Bottom-Up testing is particularly effective when a lot of the important, core functionality exists in the lower-level modules. Some examples of where it is effective are when testing layered architectures, microservices or stable low-level systems. Bottom-Up testing ensures a solid system foundation and enables efficiency and agileness since a team can start testing a lower-level module as soon as it is developed. One part that might be overlooked is the high-level functionality, for example integration between the front-end and core back-end modules (Testlio, 2024a; tutorialspoint, n.d.-b). Figure 6 below illustrates the Bottom-Up approach of integration testing.

Conversely, Top-Down integration testing takes the opposite approach to Bottom-Up integration testing. The higher-level modules get tested first instead of the lower-level modules. For example, the graphical user interface would get tested first so that all components of the UI would work as expected. While the testing of the higher-level modules occurs, the low-level components need to be mocked or implemented as test stubs. The stubs are stand-ins for yet to be developed back-end components, and they mimic the behavior of those. This sort of testing enables faster discovery of errors and bugs in the interfaces that actual users and potential customers will be using. This enables a better user experience, at least for the graphical parts that the user will interact with. This way of integration testing has the opposite issues of Bottom-Up testing, as, in this case, the lower level components get the least amount of attention, and risk getting overlooked during the development of the system. Other disadvantages of Top-Down testing are, for example, that the stubs can be either too simple, as they only provide a simple set of data to the UI and might create false positives that the software functions as it should, or too complex, where too much functionality gets implemented and takes time away from developing the actual real low-level implementations

(Testlio, 2024c; tutorialspoint, n.d.-c). Figure 6 below illustrates how modules get tested in the Top-Down approach.

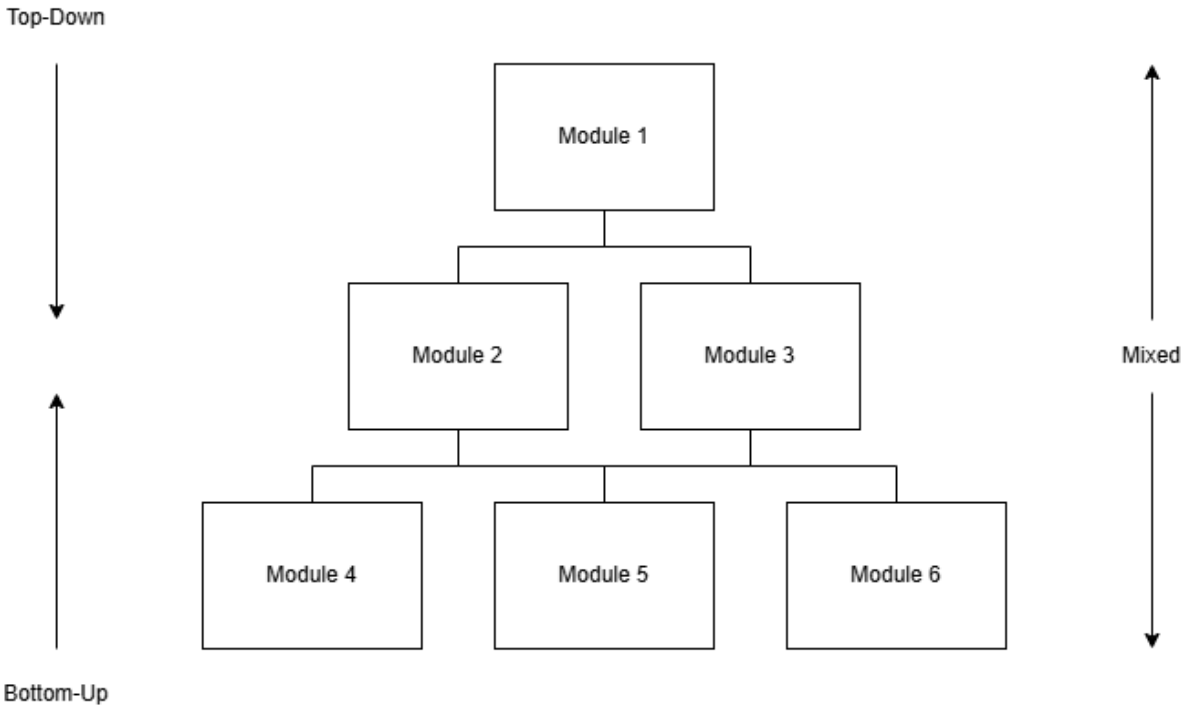

Figure 6. The Top-Down, Bottom-Up and Mixed ways of integration testing.

Mixed integration testing combines the approaches of Top-Down and Bottom-Up integration testing, as illustrated in Figure 6. It also goes by the name of sandwiched integration testing and hybrid integration testing. The system gets split into three different layers: the top, middle and bottom layer. Testing begins at the middle layer, also called the target layer, and moves simultaneously towards the top layer (UI) and the bottom layer (databases and core functionality). This sort of testing enables a team to test integrations from the middle layer to both top and bottom layers in parallel, hence increasing efficiency and facilitating faster delivery rates. This sort of testing fits best for larger projects where whole teams are dedicated to specific parts of the software system, and can develop their own integration tests from the middle layer when they see fit. Depending on how large the software system is, there can be multiple different layers, and the team responsible for a specific layer only has to make sure the integration works to one layer above and one below. In this way mixed integration testing is highly scalable and creates efficient division of labour among the testing/development teams. Coordinating the team members, both on the testing and

development side, with how the processes should work, can be a disadvantage to this testing approach. There has to be sufficient resources, including developers, adequate test environments and server capacity, for mixed integration testing to work seamlessly (Testlio, 2024b).

2.5 What is Functional Testing?

Functional testing is a way of testing that makes sure that an application's or software system's functionality works exactly as expected. As mentioned in Chapter 2.1, *About Software Testing*, the terms *functional testing* and *end-to-end testing* are used interchangeably in this thesis. Functional testing should be conducted after the unit and integration tests, since they are the most time and resource intensive and presuppose that all lower level components and integrations already work as they should. The tests go through all feasible paths and branches of a system, trying to get as complete coverage as possible, and make sure that the system behaves correctly in all scenarios. The method furthermore enhances the quality of software by identifying functional errors, and verifying that user workflows and interactions behave as intended.

When conducting functional testing, the software tested is treated as a black-box, meaning that the tester does not know the internal structure of the system. The tester has functionality requirements and specifications, usually coming from a design document or specification. They conduct their tests either as an independent party with no knowledge of the software's inner workings, or by acting as an external tester who ensures that the software functions properly. Automated functional testing, together with manual testing, is often a part of wider regression testing which makes sure the functionality still follows the requirements after the software has been updated and/or changed. By testing common user workflows through an application in an automated manner, it can also reduce or eliminate the time-intensive manual testing, conducted to ensure usability and intuitive user experience.

The functional testing methodology usually follow these steps in order:

1. Determining what test cases are the most important and valuable to test, since the amount of possible tests in a complex system can be immense.

2. Creating input data that will be sent to the application at different stages in the workflow. These inputs can range from individual mouse clicks or textual input in prompt boxes, to JSON or XML payloads sent to REST API endpoints.
3. Define the acceptable responses and behavior of the application after these inputs have been sent, and verify that the functionality is correct.
4. Execute the tests in test suites which go through a user workflow.
5. Compare the test results to the expected behavior. This shows if the system behaves as expected.

Steps 4 and 5 get repeated after any update or change to the software, to make sure that it still functions properly. Steps 1–3 may change during the development process if the design specifications are updated, thereby altering the expected behavior of the software system (BrowserStack, 2025a).

2.6 What is Behavior-Driven Development?

Behavior-Driven Development (BDD) is a testing methodology that defines the expected behavior of the application. The methodology emphasises frequent collaboration and communication between developers, testers and domain experts. BDD makes use of a DSL (domain-specific language), constructed especially for defining test cases where the correct behavior is specified. The DSL is designed to resemble natural language as much as possible, making it easy to understand by all testing participants. The DSL then gets converted into executable tests by a BDD framework. Common for most BDD frameworks, like Cucumber[4] and JBehave[5], is the framing of behavior in the format of "`Given(initial conditions), When(event occurs), Then(Expected behavior)`". These statements/steps are then mapped to actual test methods that get executed and evaluated.

BDD and functional testing are a natural fit, and using both together make testing of software more efficient and effective. Since functional testing is focused on testing end-to-end workflows from a user perspective it is helpful that the tests are written in an easily understandable format, like, for example, the Gherkin[6] syntax of the Cucumber framework,

[4] https://cucumber.io/
[5] https://jbehave.org/
[6] https://cucumber.io/docs/gherkin/reference

that is close to natural English. This syntax makes it easier for non-developers and developers alike to write all the steps in an end-to-end workflow. In particular, non-developers can iterate and develop test cases on their own. These test cases can then be implemented in actual code by the developers. Since BDD frameworks are very flexible in terms of the code that can be executed from their steps, there is practically no limit to the types of functional testing frameworks that can be run from the step definitions. This combination makes it easy for all stakeholders to contribute to the complete testing of the software system that is being developed, increasing the quality and reliability of the system as a whole (BrowserStack, 2025b).

2.7 Using Mocks in Automated Testing

One crucial aspect of writing automated functional tests is to make sure that the test results only depend on the behavior of the application being tested. Any external or internal services that the application communicates with have to be mocked to make sure that any failing tests really is caused by the application and not by any other service.

Figure 7 shows how a real-life system may be designed. The functional tests are communicating with the tested application which in turn communicates with a number of other services. The tested application then processes the responses from these other services and sends the results back to the functional test for verification. This set-up makes it really hard to determine if a failing test actually indicates a problem with the tested application, or if it is caused by any of the outside services experiencing issues. This may lead to a lot of frustrating and time consuming troubleshooting only to reveal that the failing test was a false negative. In the long run this can make developers lose faith in the test (Parry et al., 2022).

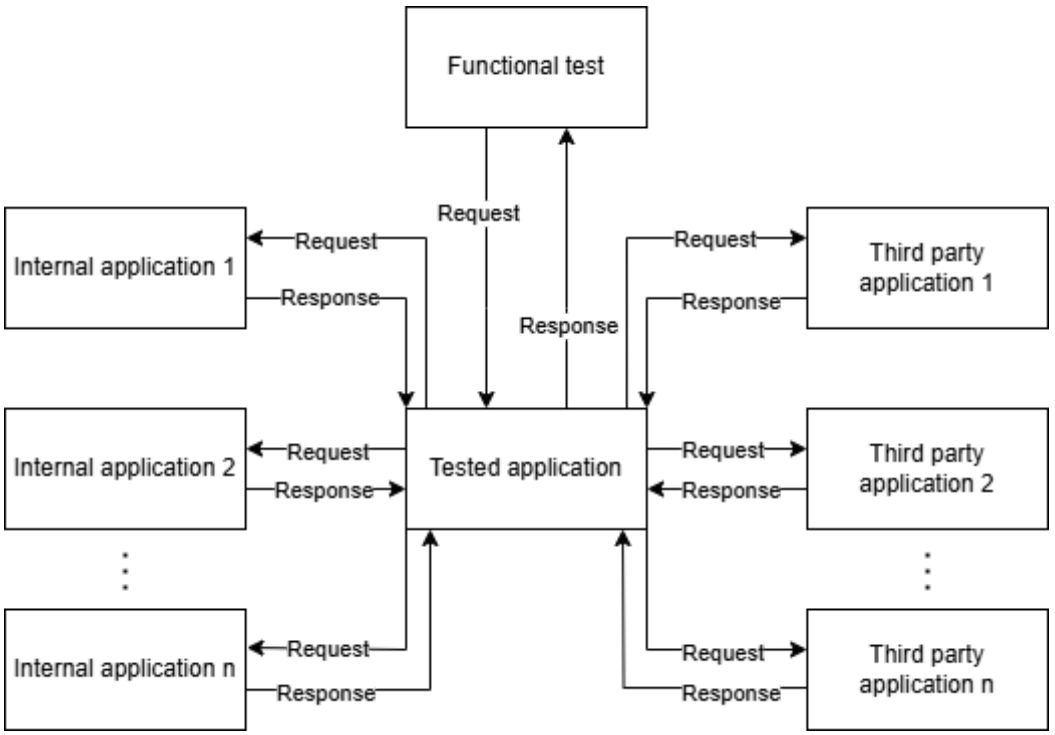

Figure 7. An example of a real-life system.

The set-up for the same system with all outside services mocked is depicted in Figure 8. In the figure all the red components are mocked. The functional tests are still communicating with the tested application which is still sending out the same requests as before. These requests are now, however, interrupted by a mocking tool before they reach the real services and are redirected to mocked services instead. These mocked services then return mocked responses which the tested application uses in its processes.

The set-up using the mocking tool has two major advantages compared to the set-up not using mocks. First of all it helps in avoiding non-deterministic test results. That is, as mentioned above, the test results only depend on the code in the tested application and if there has been no change in the code then the test results should not change. Secondly, it gives the developer control over the responses received by the tested application. This makes it easier to test the behavior of the application in different scenarios. For example, to test what happens if an external service is down might be hard to test in the real world set-up. With the mocked servers it is easy to send a mocked response with a `500` status code (Internal Server Error) and verify that the application handles this in the correct way.

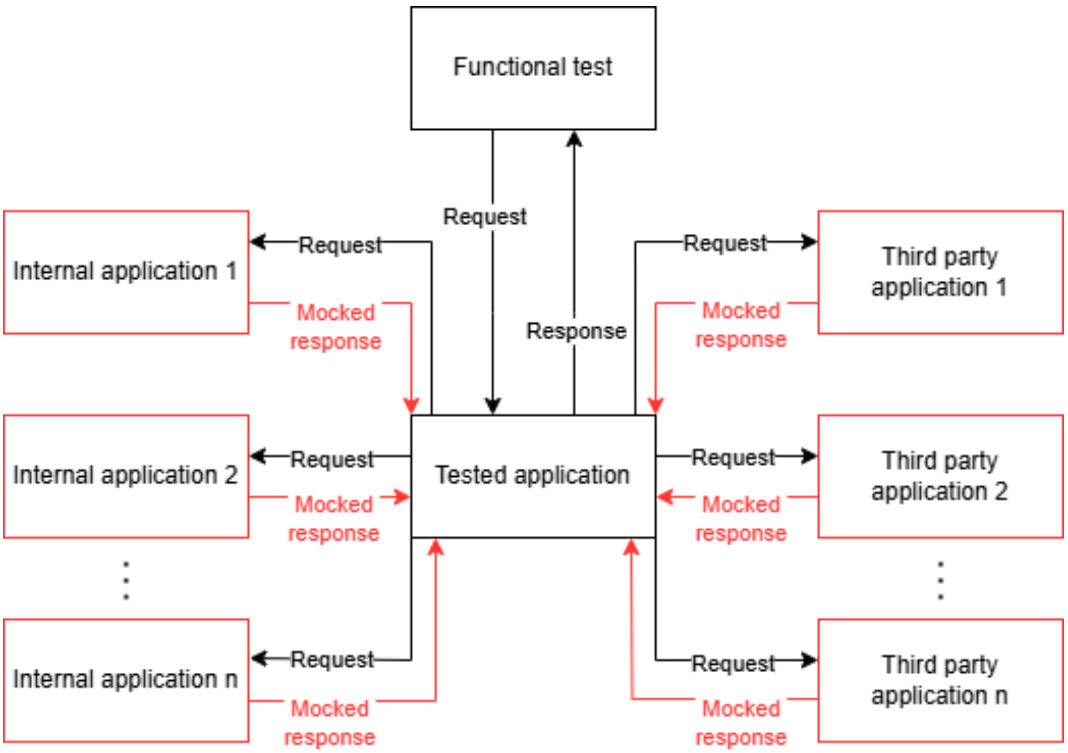

Figure 8. An example of a system with all connections to other applications mocked.

One drawback of using mocked services is that it may create some extra maintenance work since the mocks need to be kept up to date with the real services. An update to an external service can require a manual update of the tested application to keep using the functionality of the service. Failure to also update the mocks may cause some tests to fail since the mocks now do not replicate the real world anymore (Liu, 2020). This should not be too big a problem though since most applications have well defined API contracts that rarely change unless they are still in the development stage.

3. TECHNOLOGIES

3.1 JUnit 5

3.1.1 What is JUnit 5?

Writing tests is one of the crucial ways to ensure that software is functioning correctly. To make testing easier, testing frameworks that can register and run tests in an automated manner have been developed. One of the most widely used test frameworks, often considered to be the de facto standard, is JUnit (*Testing Frameworks & Tools*, n.d.). It is an open-source test framework for Java that is mainly used for unit testing, but can also be used for integration and functional testing. At the time of writing, the latest version of JUnit is version 5. JUnit 5 consists of three modules: JUnit Platform, JUnit Jupiter and JUnit Vintage. The JUnit Platform exposes an API that makes it easy to run different testing frameworks on the platform and enables easy integration with different IDEs and build tools. JUnit Jupiter implements a programming and extension model that allows tests and extensions to be written using JUnit 5's built-in classes, functions, etc. JUnit Vintage provides an API for running JUnit 3 and JUnit 4 tests.

3.1.2 Creating a Test Suite With JUnit 5

When creating functional tests, it is often suitable to only use the JUnit Platform Suite Engine. With this engine, it is possible to declare specific classes that will run as a test suite, and the platform engine enables the usage of any test engine that has been implemented in JUnit 5. The `@Suite` Java annotation is used to declare a class as a test suite, as illustrated in Figure 9.

```
@Suite
public class TestSuite {
}
```

Figure 9. An example of the `@Suite` annotation.

The annotation marks a class as a Test Suite, and the JUnit Platform will be able to discover and execute it when up and running. In addition to the `@Suite` annotation, there are a lot of

different selector and filtering annotations that make it possible to run only specific segments or tests of the test suite.

3.1.3 Specifying an Engine in JUnit 5

A JUnit Test Engine discovers and executes tests that adhere to a specific programming model. The `@IncludeEngines` annotation makes it possible to specify a specific test engine to be used. The annotation takes as a parameter the ID of a specific test engine in string format. Figure 10 illustrates choosing a specific test engine to be used, in this case the Cucumber Test Engine.

```
@Suite
@IncludeEngines("cucumber")
public class CucumberRunner {
}
```

Figure 10. An example of the `@IncludeEngines` annotation.

3.1.4 Integration Between JUnit 5 and Cucumber

Figure 11 below shows the connections between JUnit 5 and Cucumber (see Chapter 3.5.1, *What is Cucumber?*, for more information about Cucumber). First the IDE or build system requests discovery and execution of the tests. For example, the Gradle task `test` in a Gradle subproject can start the process. The request gets sent to the JUnit Platform which, in turn, forwards the request to the Jupiter Test Engine. The Jupiter Test Engine discovers and executes Test Suites, that is, a class annotated with the `@Suite` annotation. Since the class in our case also has an annotation specifying the Cucumber Test Engine as the engine to be used, a request is sent to the Cucumber Test Engine. The test engine then discovers and executes the feature files that includes the features and test scenarios that should be executed.

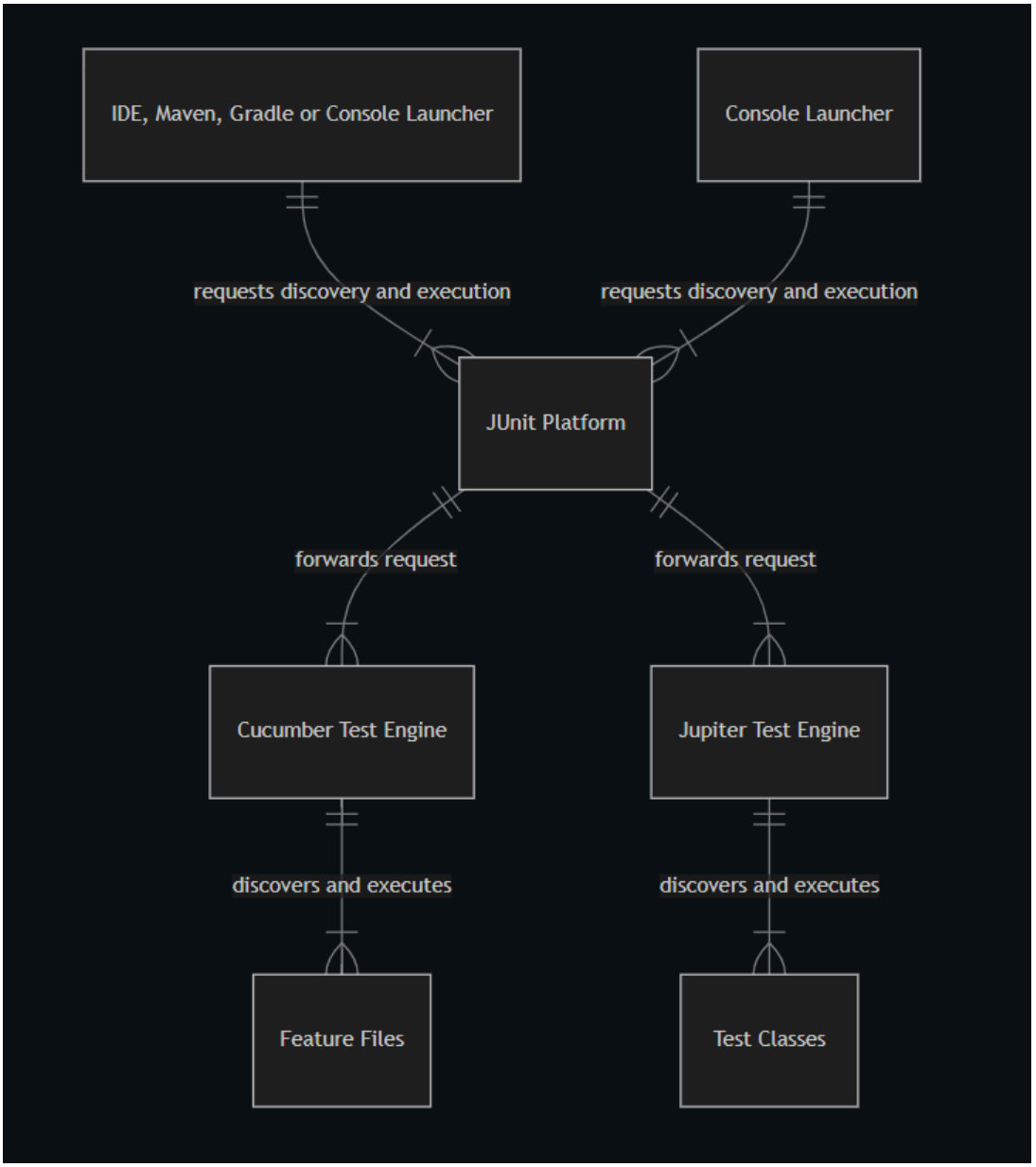

Figure 11. Connections between the Cucumber Test Engine, JUnit Platform, IDE and build system (Cucumber JUnit Platform Engine, 2025).

3.2 Docker

3.2.1 What is Docker?

Docker[7] is a tool that uses OS-level virtualization to create isolated software packages called *containers*. Docker is built to work with the different virtualization features of the Linux

[7] https://www.docker.com/

kernel, but can also run on other operating systems, like Windows and macOS, via virtual machines. By using a single kernel, the containers can reduce the resources used compared to a virtual machine that has to virtualize computer components like the processor, memory and storage. The containers bundle all the needed dependencies for a minimal functioning Linux system, such as libraries and configuration files. Since the containers are entirely self-contained and have all they need for running an arbitrary program, the problem of software working only on a specific computer gets solved. It is easy to transfer containers from one system to another and run them effortlessly with a Docker daemon for the computer's specific architecture (*Docker (software)*, 2025).

Docker was created by the company Docker Inc., which offers both free and premium features. One of the more important services they offer is the Docker Hub. It is an online service where a wide range of software vendors offer *images*[8] of their different software, for example programming language toolchains and databases (GeeksforGeeks, 2025).

3.3 Testcontainers

3.3.1 What is Testcontainers?

When testing applications, it is often necessary to test their connections to, for example, databases and message brokers without requiring the application to connect to the real component. One easy way of achieving this is to use mocked or in-memory services. This may, however, not always be the best solution since such services might not fully replicate the functionality of the services used in production. For example, using the in-memory H2 database in the tests instead of the Oracle database used in production can cause the tests to fail because the H2 database lacks some of the functions of the Oracle database. It can also lead to a situation where the tests all pass, but the functionality breaks in production upon deployment if the new code relies on functions in the H2 database that do not behave the same way in an Oracle database (Testcontainers, n.d.-f). Testcontainers[9] is an open source

[8] "A Docker image is a snapshot or blueprint of the libraries and dependencies required inside a container for an application to run" (Amazon Web Services, n.d.).
[9] https://testcontainers.com/

library available for multiple programming languages that offers functionality to help avoid these situations.

Testcontainers create lightweight, disposable instances of real components inside Docker containers, allowing tests to run locally or in a CI/CD pipeline with minimal configuration of the environment (Testcontainers, n.d.-d). Basically, the only thing required to run the tests is a container runtime environment compatible with Docker Engine API (Testcontainers, n.d.-b).

Figure 12 shows the workflow of a test set-up, using Testcontainers to run throwaway instances of a Postgres database, Kafka message broker and Elasticsearch search engine. Before the tests are run the Testcontainers are configured and started. This step might for example involve defining schemas in a SQL database and populating them with some data needed for the tests. This gives the developer total control over the system's state when the tests start and ensures that the test results are deterministic since the starting conditions are always the same. This step also includes configuring the test application to use the containerized services instead of the real services used in production.

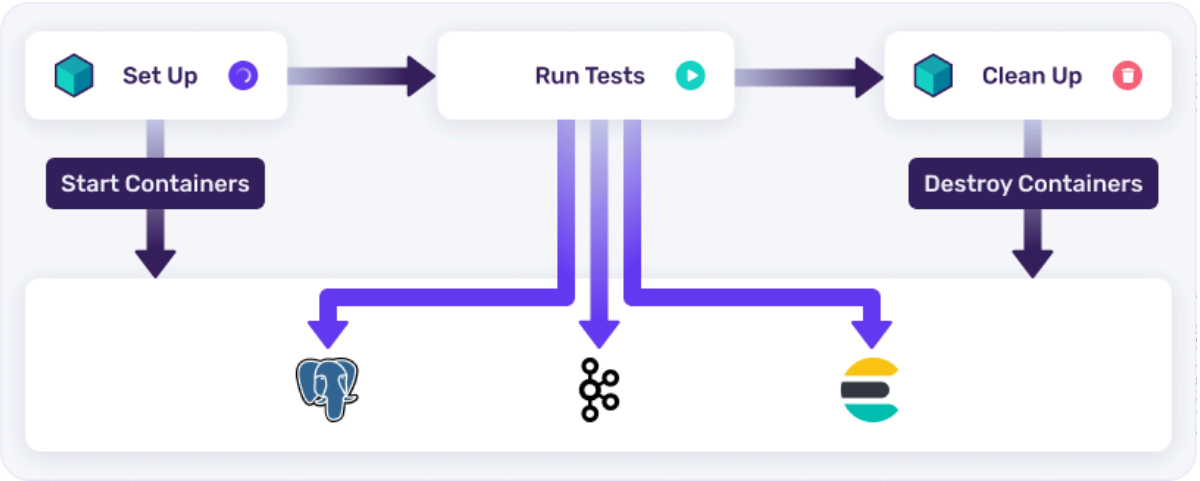

Figure 12. Example of a test setup using Testcontainers (Testcontainers, n.d.-b).

While the tests are running, the test application is communicating with the Testcontainers and modifying their states as needed. When the tests stop, regardless of whether they finish successfully or if they crash, all the containers created prior to the tests are removed automatically, making sure there are no resource leaks due to any unnecessary containers running. The automatic clean-up also ensures that there is no risk for any data leaking from

one test run to the next, causing the results to be different because of modified data at startup (Testcontainers, n.d.-b).

3.3.2 Setting up a Testcontainer

The easiest way to create a Testcontainer is to use one of its numerous modules, which are preconfigured implementations of external services such as databases or message brokers (Testcontainers, n.d.-c). This makes it easy to configure the container to fit the test code while also making sure that the Testcontainer mimics the service used in production as much as possible. Figure 13 illustrates how to create and configure a containerized oracle database using testcontainers.

```
static OracleContainer oracle =
    new OracleContainer("gvenzl/oracle-free:23-slim-faststart")
        .withDatabaseName("testDB")
        .withUsername("testUser")
        .withPassword("testPassword");
```

Figure 13. Example code for creating and configuring a containerized Oracle database using Testcontainers.

Instead of using a specific module to create a Testcontainer, a GenericContainer can be used, allowing a Testcontainer to be created from any container image, and offering greater flexibility to the user (Testcontainers, n.d.-a). Figure 14 depicts an example of how to create a GenericContainer.

```
static GenericContainer redis =
    new GenericContainer(DockerImageName.parse("redis:6-alpine"))
        .withExposedPorts(6379);
```

Figure 14. Example code for creating and configuring a GenericContainer using Testcontainers.

To be able to use a Testcontainer it needs to be started using the `Testcontainers.start()`-method. How this is done depends on the chosen container lifecycle strategy. One way is to use JUnit 5‘s annotations `@BeforeAll` and `@AfterAll` as illustrated in Figure 15. This setup will start the container before the first test and stop it after the last test. Explicitly stopping the containers after the tests are completed is optional, since they will be destroyed by the aforementioned automatic clean-up anyway (Testcontainers, n.d.-e).

```
@BeforeAll
static void beforeAll() {
    oracle.start();
}

@AfterAll
static void afterAll() {
    oracle.stop();
}
```

Figure 15. Example code to start and stop a Testcontainer.

The final step before the Testcontainers are ready for use is configuring the tested application to use the Testcontainers instead of the external services used in production. The way this is done depends on what frameworks the application uses, but Figure 16 illustrates an approach for a Spring Boot application. This example dynamically updates the properties in the properties file with values defined by the Testcontainer.

```
@DynamicPropertySource
static void configureProperties(final DynamicPropertyRegistry registry) {
    registry.add("spring.datasource.url", oracleContainer::getJdbcUrl);
    registry.add("spring.datasource.username",
        oracleContainer::getUsername);
    registry.add("spring.datasource.password",
        oracleContainer::getPassword);
}
```

Figure 16. Example code to configure a Spring Boot application to use a Testcontainer.

3.4 WireMock

3.4.1 What is WireMock?

As discussed in Chapter 2.7, *Using Mocks in Automated Testing*, mock servers are frequently used in functional testing to isolate the system being tested from internal or external services. One widely adopted tool for creating and managing mock servers is WireMock[10]. WireMock was first developed as a Java library in 2011, but can now be used with a variety of programming languages and technologies, such as Python, .NET and Spring Boot (WireMock, n.d.-a).

[10] https://wiremock.org/

WireMock can be configured to intercept HTTP requests to any external service. Instead of allowing the request to reach the real service, WireMock returns a predefined response from the mock server. This response can be defined in, for example, code or JSON files, providing developers with full control over the response content and behavior. For example, the mock server can be configured to return a `500` status code or to have a longer latency than usual in order to test how the tested service behaves in edge cases. Thanks to WireMock's extensive request matching capabilities, intercepted requests can be mapped to specific responses based on nearly any aspect of the request, such as the URL, HTTP method, or values within a JSON body (WireMock, n.d.-b).

WireMock can be used in many different ways depending on the project setup and tech stack. It can run as a standalone server with stub mappings configured via JSON files, or be embedded directly into test code using its Java API. In Spring Boot projects, for example, it can be integrated using annotations and configuration support from testing libraries, such as JUnit (WireMock, n.d.-c, n.d.-d).

3.5 Cucumber and Gherkin

3.5.1 What is Cucumber?

Cucumber is a framework for Behavior-Driven Development (BDD), as discussed earlier in this thesis. While commonly used in testing contexts, it is important to remember that it is not actually a testing framework per se, as it does not perform any assertions directly. What Cucumber does, however, is create a mapping between scenarios and steps written in plain text using the Gherkin syntax (see Chapter 3.5.2, *What is Gherkin?*) and the code that actually performs the operations and assertions described in the scenarios. This code, called step definitions, will then in all likelihood use a testing framework such as JUnit or TestNG to execute the tests. Figure 17 explains how Gherkin, Cucumber and testing frameworks work together when running tests on a system.

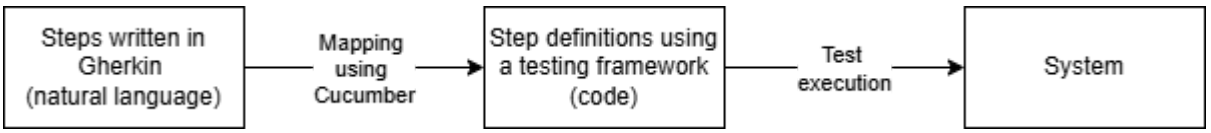

Figure 17. Test workflow using Cucumber. Adapted from the official documentation (Cucumber, 2024b).

3.5.2 What is Gherkin?

Gherkin is the domain-specific language used to write the tests which are then mapped to code by Cucumber. The tests are organized into feature files. These feature files consist of scenarios which in turn consist of steps that are to be executed in the specified order. The feature files are written in plain text using some specific keywords defined by the Gherkin syntax, making them readable for both humans and the Cucumber engine. Some of these keywords are `Scenario`, `Given`, `When`, `Then` and `Background`. A complete list of all keywords and their usage can be found in the official documentation (Cucumber, 2025). All keywords are available in a wide range of natural languages, allowing the feature files to be written in the stakeholders' native language. This makes it particularly easy for them to understand which use cases are being covered by the tests (Cucumber, 2024a).

3.5.3 Example of Writing Tests Using Cucumber and Gherkin

Figures 18 through 20 depict an example of how to write tests using Cucumber and Gherkin. They will serve as an illustration of how the mapping between the steps in the feature file and the step definitions in a Java class works.

Figure 18 shows the feature file. In this case, there is only one scenario consisting of four steps. Each row in the feature file starts with a Gherkin keyword, which Cucumber uses to interpret and parse the feature specification. The scenario is written in a way that makes it obvious what is being tested, making the feature file serve as documentation as well as the definition of the tests, more or less eliminating the need to write explicit test documentation.

```
Feature: Marble sharing
  Scenario: Giving marbles to a friend
    Given I have 53 marbles
    And you have 32 marbles
    When I give you 10 marbles
    Then I will have 43 marbles
```

Figure 18. An example of a feature file using the Gherkin syntax.

Figure 19 shows the step definitions corresponding to the steps in Figure 18. The most interesting parts of the code are the *step definition annotations* that precede each method, the logic in the methods themselves is quite trivial. Each step definition annotation consists of two parts, one Java annotation (`@Given`, `@When` or `@Then`) and one expression stating the pattern used by Cucumber when doing the mapping. First, considering the Java annotations, note that the step "`And you have 32 marbles`" in the feature file is mapped to the method annotated with `@Given("you have {int} marble(s)")`. This works because the keywords used to define steps are actually interchangeable (Cucumber, 2025; Marit, 2017). That is, each step has to start with a keyword (`Given`, `When`, `Then`, `And` or `But`) and each step definition annotation has to start with a Java annotation (`@Given`, `@When`, `@Then`, `@And` or `@But`), but it does not really matter which keyword/annotation is used. It is, however, considered good practice to use the "correct" keyword since it improves readability and aligns with the BDD methodology (England, 2016).

```java
public class MarbleSteps {

    private int myMarbles;
    private int yourMarbles;

    @Given("I have {int} marble(s)")
    public void iHaveMarbles(int count) {
        myMarbles = count;
    }

    @Given("you have {int} marble(s)")
    public void youHaveMarbles(int count) {
        yourMarbles = count;
    }

    @When("I give you {int} marble(s)")
    public void iGiveYouMarbles(int count) {
        myMarbles -= count;
        yourMarbles += count;
    }

    @Then("I will have {int} marble(s)")
    public void iWillHaveMarbles(int expectedCount) {
        assertEquals(expectedCount, myMarbles);
    }
}
```

Figure 19. Examples of step definitions using Cucumber, Java and JUnit.

The step definition's expressions in Figure 19, for example `"you have {int} marble(s)"`, are written using something called *Cucumber expressions*. The expression can also be written as a regular expression, but since Java does not provide a literal syntax for regular expressions, the string in a step definition annotation will automatically be interpreted as a Cucumber expression, unless explicitly declared as a regular expression (*Cucumber Expression - Java Heuristics*, 2019). The expression `"you have {int} marble(s)"` highlights two of the features of Cucumber expressions that make them highly flexible. The first one is the `{int}`-part. This is called an *output parameter* and, using this, instead of a hardcoded integer, will make this step definition reusable for all steps of this type no matter what the integer is. The integer stated in the step will automatically be used as input to the method. Secondly, using `marble(s)` instead of `marbles` will make this pattern match both the plural `marbles` and the singular `marble`. This means that both the step "`Given I have 53 marbles`" and the step "`Given I have 1 marble`" will map to this step definition (*Cucumber Expressions*, 2024).

To execute Cucumber tests, a designated entry point is required. This is commonly referred to as a *Cucumber runner*. Figure 20 illustrates a minimal example of such a runner class. The class itself contains no logic; it simply uses annotations to configure and initiate the Cucumber test execution.

```
@Suite
@IncludeEngines("cucumber")
@SelectClasspathResource("features")
@ConfigurationParameter(key = GLUE_PROPERTY_NAME,
    value = "package.name")
public class CucumberRunner {
}
```

Figure 20. Example of a setup for running Cucumber tests.

3.6 Jenkins

3.6.1 What is Jenkins?

Jenkins is an automation server that helps with building, testing and deploying software that is being developed. It is open source and licensed under the MIT license[11]. Its backend is

[11] https://opensource.org/license/mit

written completely in Java, but it also includes a frontend for browsers written in JavaScript. It supports a multitude of plugins to extend its functionality, and also supports the most popular version control systems, such as Git and Mercurial. Jenkins functions by defining a pipeline with stages and steps that contains various commands, for example commands for building and testing. The pipeline is defined and configured in a text file named `Jenkinsfile`. It can be created and edited through the server's graphical interface or manually added to a software project.

Jenkins, originally known as Hudson, was developed by Kohsuke Kawaguchi while working at Sun Microsystems. After the company was bought by Oracle in 2010, disagreements arose over where to host the source code, and the Hudson developer community decided to fork Hudson and rename it Jenkins. The two projects continued as two independent projects, with Hudson being developed in-house at Oracle until being discontinued in 2016, and Jenkins still being developed as an open source project by the Jenkins community on GitHub (*Jenkins*, n.d.-a, *Jenkins (software)*, 2025; Jenkins, n.d.-b).

Jenkins is an important tool for implementing the processes of continuous integration and continuous delivery, commonly known together as CI/CD. Continuous integration is the process of frequently checking and testing that new development keeps the software in a workable state, and that the new development does not break any existing functionality. All new code gets merged into an integration branch that gets built, tested and run on an automation server, such as Jenkins. This is done to make sure that the software functions as it should, so that further development can proceed. Continuous delivery is the process of frequently shipping and deploying new software features in small incremental steps. This is in contrast to the monolithic way of releasing software, where development of a new version with new features can take many months before getting shipped to the customer. When using CD, features get shipped as soon as they are completed. The features get merged into the integration branch, and after they are built and tested, they get deployed to production immediately (*Continuous Delivery*, 2025, *Continuous Integration*, 2025).

4. IMPLEMENTATION

4.1 Gradle Subproject

First of all, we started with creating a separate subproject `tests` in the larger project that we were working in. The project is structured into many separate Gradle subprojects. A Gradle subproject is a part of a larger Gradle project that contains its own `build.gradle` file with its own tasks, dependencies and configurations. It is beneficial to structure a project in this way for a multitude of reasons, such as facilitating modular and reusable code, separating different architectural parts and optimizing build performance (Gregory, 2022). We structured the subproject very similarly to the Standard Directory Layout (The Apache Software Foundation, 2014), with a `src` folder for all our source code, that contains two separate folders: `main` for application sources and `test` for tests sources. Since we only created tests, we left the main folder empty and it was subsequently removed. The test folder in turn contains the folders `java` and `resources`. The `java` folder contains only Java code, such as code implementations for our Cucumber step definitions and util classes for methods that we use often. The `resources` folder in turn contains such things as our Cucumber feature files, JSON files used in our request bodies, WireMock mappings for our mocked responses from external services, and so on. Below in Figure 21 you can see our subproject `tests` as a tree structure.

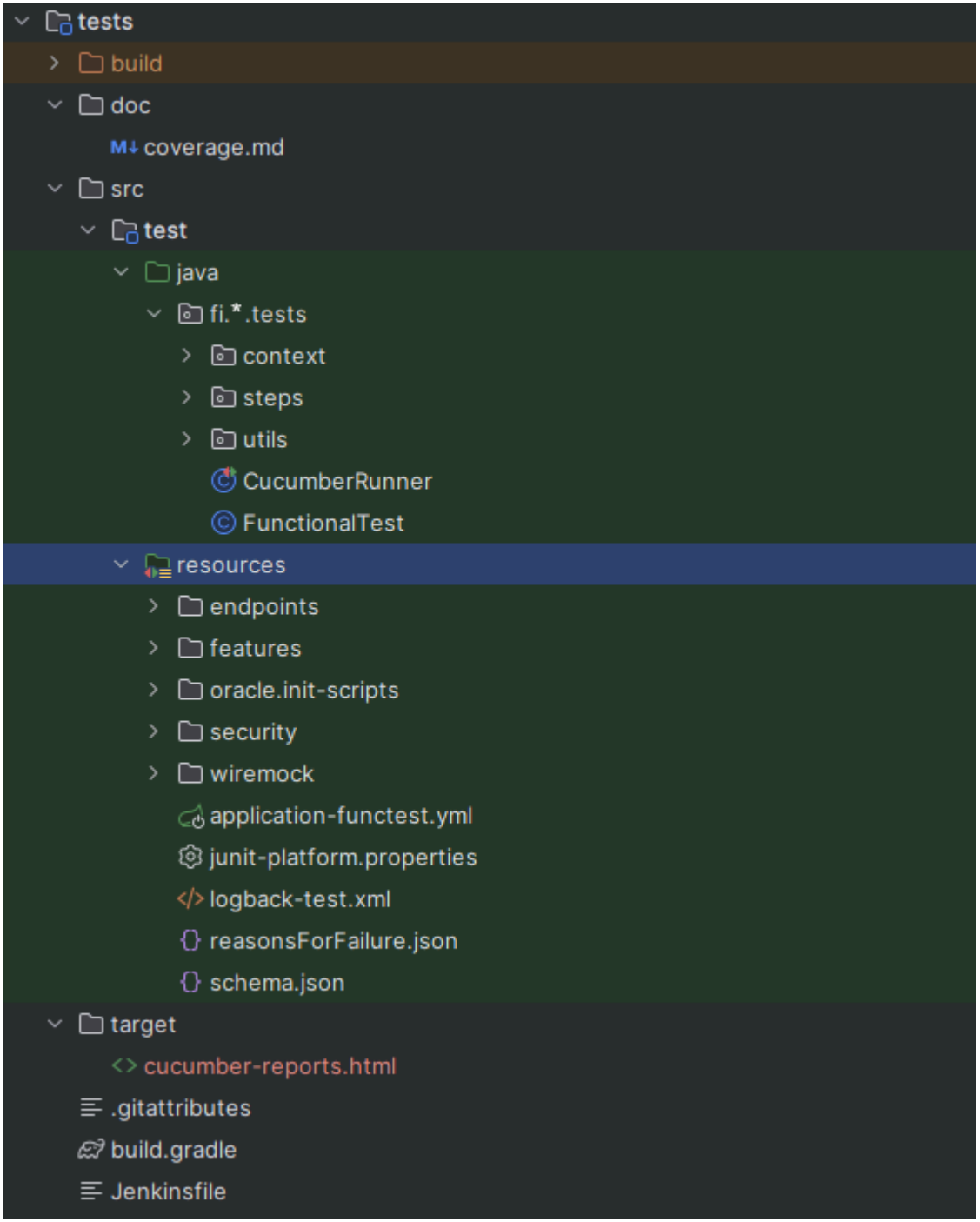

Figure 21. `tests` subproject file structure.

4.2 Dependencies

We import our dependencies via a `build.gradle` file that lies in the root of the `tests` subproject. A `build.gradle` is a file where you can declare various plugins, dependencies and Gradle tasks for your (sub)project. In a multi-module project all subprojects are isolated

from each other and they all have their own configurations. If you for example want to to use code from another subproject you need to import it in the `dependencies` block in the `build.gradle` file as an internal dependency. The dependencies are defined in the file `libs.versions.toml` in the root of the project. This file is called a version catalog, a Gradle feature that makes it easy to centralize all dependencies of a project in a singular location. You can define specific version numbers, dependency groups and artifact names, as well as plugins. You can also bundle together multiple dependencies into a bundle (Gradle, n.d.). Figure 22 below illustrates the contents of our `build.gradle` file.

```
plugins {
    id 'java'
    alias(libs.plugins.lombok)
}

dependencies {
    testImplementation project(':api')
    testImplementation project(':boot')

    testImplementation(
            libs.bundles.cucumber,
            libs.bundles.rest.assured,
            libs.json,
            libs.junit.platform.suite,
            libs.bundles.testcontainers
            libs.bundles.wiremock
    )
}

tasks.test {
    useJUnitPlatform()
}
```

Figure 22. build.gradle file.

We use these dependencies for the following purposes:

- `cucumber` as our BDD framework, to create feature files.
- `rest-assured` to be able to easily send requests to and receive responses from API endpoints.
- `json` for creating JSON payloads that we can send in our requests.
- `junit-platform-suite` for creating runnable test suites.
- `testcontainers` for creating containers for our database.

- `wiremock` as our mock server framework.

We also import the internal dependencies `testImplementation project(':api')` and `testImplementation project(':boot')` to get access to API-related code and the Spring Boot application.

4.3 CucumberRunner

To start the tests we have a class `CucumberRunner` which finds and executes our feature files. It functions simply as a starter to run our tests. The class has the following annotations:

- The `@Suite` annotation marks a class as a test suite on the JUnit Platform, and this suite will in turn discover the tests to be run.
- The `@IncludeEngines` annotation is used for specifying the specific test engine that will be used for executing the tests, in this case the Cucumber Test Engine.
- The `@SelectClasspathResource` annotation selects a specific classpath resource that should be used when executing the tests.

Figure 23 below shows the CucumberRunner class.

```
@Suite
@IncludeEngines("cucumber")
@SelectClasspathResource("features")
public class CucumberRunner {
}
```

Figure 23. CucumberRunner class.

4.4 FunctionalTest annotations

The `FunctionalTest` class is used for configuration and setting up the needed infrastructure before running any test. All step definition classes (see Chapter 4.8.3, *Step Definitions*) extend the `FunctionalTest` class to make sure that when you run a test from any feature file, the test infrastructure will be initialized and ready. Below we will go through all the configurations that are made in this class. We will first start with the annotations to the class that contribute with quite a lot of the configuration and setup needed for running the tests. Below in Figure 24 you can see the annotations in their entirety.

```
@ActiveProfiles({"bank", "functest"})
@CucumberContextConfiguration
@SpringBootTest(classes = {SpringBootApplication.class,
                FunctionalTest.class, MockInternalService.class,
                /* more MockedServices*/, TestContext.class},
                webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@EnableWireMock({
       @ConfigureWireMock(
                registerSpringBean = true,
                name = "internal-service",
                filesUnderClasspath = "wiremock/internal-service",
                baseUrlProperties = "internal-service.url")
       @ConfigureWireMock(
                name = "auth-server",
                filesUnderClasspath = "wiremock/auth-server",
                baseUrlProperties = "auth-server.url"),
              // Several @ConfigureWireMocks follow the above
})
public class FunctionalTest {
```

Figure 24. FunctionalTest annotations.

4.4.1 @ActiveProfiles

The `@ActiveProfiles` annotation is Spring-specific and lets us choose which Spring profile we want to use when running the Spring Boot application. Different profiles have their own configuration files, `application-{insert profile}.yml`,where you can specify different configurations both for the application, for example security and logging configs, and its integrations with other dependencies like, for example, WireMock and the database of choice. We are using the default `bank` profile, which provides a lot of default configurations for running the application locally, as well as our own profile `functest` that we need for specific configurations for our tests, for example URLs to our mocked external servers.

4.4.2 @CucumberContextConfiguration

The `@CucumberContextConfiguration` annotation is used for marking the class as a Cucumber glue class and making the Cucumber Test Engine aware of the test configuration.

4.4.3 @SpringBootTest

The `@SpringBootTest` annotation specifies the classes that should be included in our test context. It starts up the Spring Boot application, includes various mocked services into the

context, and a TestContext class that includes several objects that are reused for the majority of the tests. We also configure the port for the Spring Boot application, which gets assigned a random available port each time it runs. Figure 25 below shows the annotations.

```
@SpringBootTest(classes = {SpringBootApplication.class,
                FunctionalTest.class, MockInternalService.class,
                /* more MockedServices */, TestContext.class},
                webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
```

Figure 25. `@SpringBootTest` annotation.

4.4.4 @EnableWireMock

The `@EnableWireMock` annotation enables the WireMock Spring Boot configuration and allows us to inject WireMock instances and objects into the Spring Context as both properties and beans. The annotation takes as input one or multiple `@ConfigureWireMock` annotations which allows us to create and configure multiple WireMock servers. In Figure 26 below we can see the following configuration parameters:

- `registerSpringBean = true`, which lets us autowire a server into a Spring service class. This allows us to make changes to a server, if needed. See Chapter 4.8.6, *Customizing WireMock Responses*, for an example of when we are changing the mappings for specific tests.
- `name`, which gives the WireMock server a name that is used to refer to the server in question when injecting it.
- `filesUnderClasspath`, specifies the resource folder where the response mappings are stored.
- `baseUrlProperties`, which creates an URL as a Spring property that can be used in `application.yml` files. This is needed to redirect API requests, made by the Spring Boot application, to the WireMock server. Figure 27 shows how to replace an actual external service URL with the mock server URL.

```
@EnableWireMock({
	@ConfigureWireMock(
		registerSpringBean = true,
		name = "internal-service",
		filesUnderClasspath = "wiremock/internal-service",
		baseUrlProperties = "internal-service.url"),
	@ConfigureWireMock(
		name = "auth-server",
		filesUnderClasspath = "wiremock/auth-server",
		baseUrlProperties = "auth-server.url")
})
```

Figure 26. `@EnableWireMock` annotation and `@ConfigureWireMock` annotation.

```
app:
  internal-service:
    baseUrl: ${internal-service.url}
```

Figure 27. Extract from `application-functest.yml`.

4.5 FunctionalTest Implementation

The class `FunctionalTest` itself mostly contains code for starting up the Testcontainers, but also includes a variety of JSON helper methods that we use for fetching specific JSON payloads. The first thing we do is create a Testcontainer that contains the Oracle Database. The `@ServiceConnection` is a Testcontainer-Spring Boot annotation that marks the container as a Spring service that can be connected to. The `@Getter` annotation is a Lombok annotation that automatically creates getters for fields.

We configure the container with some details to make it work as we want. First, we send in a `Bash` script into the container that will be executed when the container starts up. This script creates a database user and password, so that the Spring Boot application later can login to the database and initialize it with Flyway[12]. We have a startup timeout for 2 minutes, intended for situations where the Jenkins node, that the tests will run on, is low on resources and needs time to initialize other tasks before it is ready to start up the container. Figure 28 below illustrates the code for creating and configuring the container.

[12] "Flyway updates a database from one version to the next using migrations" (Baeldung, 2024).

```
@ServiceConnection
@Getter
private static OracleContainer oracleContainer =
    new OracleContainer("gvenzl/oracle-free:23-slim-faststart")
    .withCopyFileToContainer(
        MountableFile.forClasspathResource(
            "oracle/init-scripts/01_create_cred_dat_user.sh"
        ),
        "/docker-entrypoint-initdb.d/01_create_cred_dat_user.sh"
    ).withStartupTimeout(Duration.of(2, ChronoUnit.MINUTES));
```

Figure 28. Creating and configuring the Oracle database Testcontainer.

After the container has been created and configured, we start it up. We set the database usernames and passwords for the two databases as Spring properties. We also run our own method `loadErrorMessages()` to initialize a hashmap, which maps error tags to error messages that get sent when an error occurs after calling an API endpoint (see Chapter 4.8.5, *Tags*, for more info). Figure 29 below shows the container startup and property initialization.

```
static {
    oracleContainer.start();
    System.setProperty("ORACLEDB_USERNAME", "BANK_TEST_DB");
    System.setProperty("ORACLEDB_PASSWORD", /* a password */);
    System.setProperty("ORACLEDB_ARCHIVE_USERNAME", "BANK_TEST_ARCHIVE");
    System.setProperty("ORACLEDB_ARCHIVE_PASSWORD", /* a password */);
    ValidationUtil.loadErrorMessages();
}
```

Figure 29. Starting the Oracle database Testcontainer.

We also use a method called `oracleProperties()` to inject the URLs as Spring properties. Since the port number is always random, we do not know the complete URL beforehand, and therefore use the annotation `@DymamicPropertySource`, a Spring annotation that can fetch a value from a method and initialize it as a Spring property. Figure 30 below shows the injection of URLs as Spring properties.

```
@DynamicPropertySource
static void oracleProperties(final DynamicPropertyRegistry registry) {
    registry.add("spring.datasource.url",
        getOracleContainer()::getJdbcUrl);
    registry.add("app.data.datasource.url",
        getOracleContainer()::getJdbcUrl);
}
```

Figure 30. Adding container URLs as Spring Properties.

4.6 TestContext

The `TestContext` class is a class where we define different objects that are used by all our tests. It is annotated with a `@Component` to make it a Spring-managed bean that we then can inject/autowire into our different step definition classes (see Chapter 4.8.3, *Step Definitions*). The class is also annotated with Lombok's `@Data` annotation that provides, for example, getters, setters and `toString()`-methods to all fields in the class. Below in Figure 31 you can see the `TestContext` class in its entirety.

```
@Component
@Data
public class TestContext {
    public UUID applicationId;
    public UUID applicantId;
    public UUID partyId;
    public Response response;
    public RequestSpecification request;
    public Map<String, String> headers = new HashMap<>();
    public JSONObject jsonObject;
    public final String CREATE_APPLICATION = "create_application";
    public final String ADD_APPLICANT = "add_applicant";
}
```

Figure 31. The `TestContext` class.

The class includes a number of objects that we use in all our tests. It includes a few UUIDs, unique alphanumeric IDs that we use when referring to specific applications or applicants when calling different endpoints. For example, when you want to update a specific application you need to refer to it by its specific UUID. Since everytime we run the tests we get new randomly generated UUIDs, and given that we want to change the UUID values quite frequently for different scenarios, we include them in this common class.

We also have the rest-assured framework objects `Response` and `RequestSpecification`. The objects are continuously used for all our API calls, and gets written over everytime we want to make a new request, and when we receive a new response. We can then assert different values in the response before we move on to the next request. The `Map` object `headers`, in which we put our authentication details as well as `Content-Type` and `Accept` values, is also sent in with every request.

There is also a `JSONObject` which we send in the body of our `RequestSpecifications`. Since we have alot of different JSON payloads that we send in to the various endpoints, we have created the helper methods `loadJson()` and `getJson()`, which we use to load a new JSON payload into the `JSONObject` before we send a request. See Figure 32 below for the methods. These two methods are some of the helper methods located in the `FunctionalTest` class.

```
public static String getJson(String endpoint, String jsonName) {
    try {
        return new String(Files.readAllBytes(Paths.get(
            "src/test/resources/endpoints/" +
            endpoint +
            "/" +
            jsonName +
            ".json")));
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}

public static JSONObject loadJson(String endpoint, String jsonName) {
    return new JSONObject(getJson(endpoint, jsonName));
}
```

Figure 32. The `getJson()` and `loadJson()` helper methods.

Lastly, we have some `String` constants, where we define various folder names. These names refer to folders that contain a lot of different JSON payloads for one specific endpoint. Some endpoints, like the add and update ones, can have a lot of different configurations and that is why we have created folders with multiple payloads for these specific ones. To enable code reuse, we have defined these strings here in the `TestContext` class. This allows us to avoid modifying strings in a lot of different places and ensures that if the folder/endpoint ever changes name, we only need to change the string value in one location.

4.7 WireMock mappings

With the help of WireMock we can mock any response from an external service. When the Spring Boot application calls an external service, instead of actually calling that service, the application instead sends a call to a WireMock server. We covered the annotations that create the server and also how we defined the server URL as a Spring property in Chapter 4.4.4,

@EnableWireMock. We also specified a specific mappings folder where all our responses are stored. This folder is the `wiremock` folder. It contains server folders, each with their own mappings folder that contains mappings for specific URLs.

A WireMock mapping is defined in a JSON file where you first specify the request URL and associated query parameters (if there are any). See Figure 33 for an example of a request to a mocked server.

```
{
  "request": {
    "urlPathTemplate": "/subject",
    "method": "GET"
  },
```

Figure 33. A request mapping.

This request has a `urlPathTemplate` that specifies the path that should be matched if the application makes a call to that specific path. In this case, if the application activates a `GET` to `http://server.url/subject`, WireMock will match the request and return a predefined response. The `server.url` gets mapped to `localhost:{ /* random port */ }`.

After you have defined the request, you define the response that should be returned. See Figure 34 for an example of a response that could be returned for a `GET` request to the `/subject` path.

```
  "response": {
    "status": 201,
    "jsonBody": {
      "sub": "subject",
      "email": "test@test.ax",
      "name": "Test Testsson",
      "idpId": "myIdp",
      "idpSubjectId": "mySubject"
    },
    "headers": {
      "Content-Type": "application/json"
    }
  }
}
```

Figure 34. A response mapping.

Here you define the status code that should be returned, in this case `"status": 201,` and also the JSON that should be returned, in this case `"jsonBody": { /* json */}` . You also specify the headers, in this case `"Content-Type": "application/json"`. The application receives this response after having called a HTTP method, and can then use the JSON values further in the code, exactly the same as a real JSON response.

4.8 Writing Tests

4.8.1 About the Test Strategy

With the tech stack in place we could start writing our tests. As mentioned in the introduction, we started by writing an initial test just to make sure that our tech stack was complete and all configurations were correct. This test was later removed since it did not add any real value and we will therefore not discuss it further.

We decided to create a separate feature file for each endpoint in the system, meaning that all tests related to one specific endpoint are collected into one feature file. This might appear to contradict the point of functional testing where the goal is to test the system from start to finish, following different paths through the system. Our approach will still do this even though it may not be obvious at first glance. Figure 35 shows a simplified overview of our test strategy and illustrates how our approach actually tests the system from end to end.

Feature file 1 is designated to test step 1 of the system (i.e endpoint 1). In this case step 1 is tested using two scenarios, one happy case and one unhappy case. In reality, there may of course be additional cases that need to be tested, but for the sake of readability, we have kept the number of scenarios in this figure to a minimum. Feature file 2 is testing step 2 (i.e endpoint 2), but since step 2 is directly dependent on a successful step 1 we need to pass this step also in feature file 2 (that is, before we can test endpoint 2 we first need to call endpoint 1). Since step 1 has already been tested in feature file 1, we can however assume that step 1 works as expected and instead concentrate solely on testing step 2. This means that the feature file that is testing the last step in the system, in this case feature file 3, will inherently pass through the whole system from end to end.

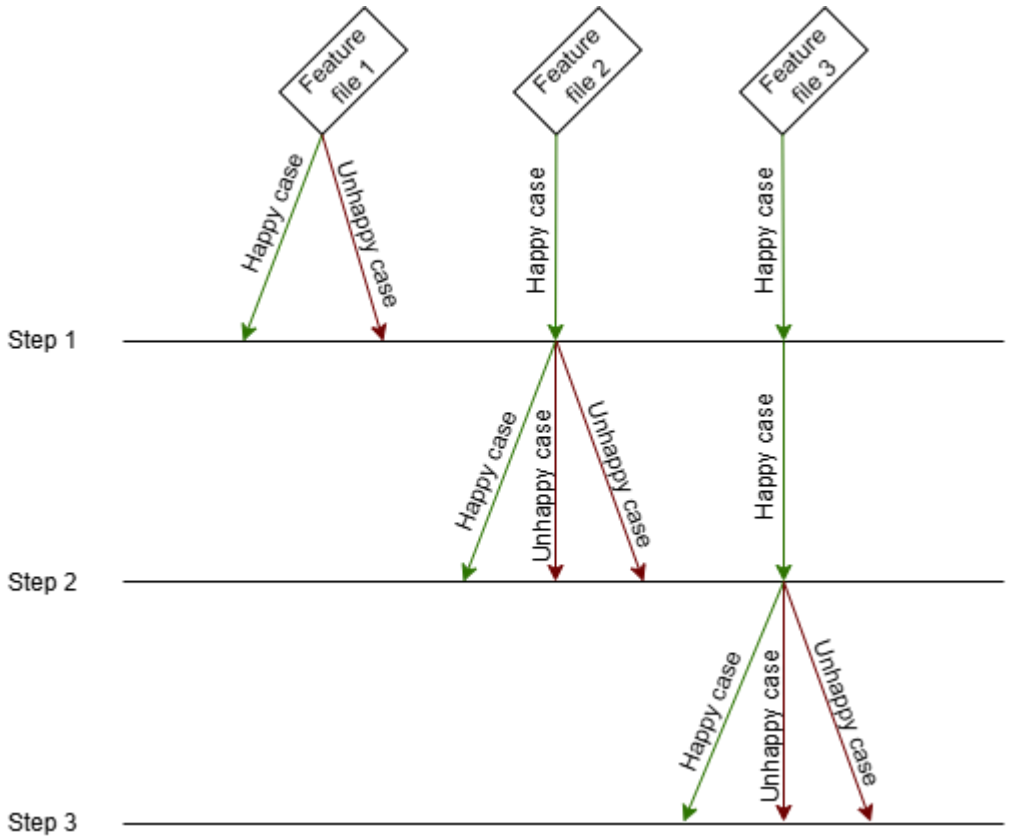

Figure 35. Illustration of the test strategy.

4.8.2 Feature Files

When writing our feature files we strove to maintain a consistent structure and appearance across all of them. To make it easier to keep the style consistent, we drew up a style guide that should be followed. This style guide has been made available to all the members of our team and is included in the appendix of this thesis.

Figure 36 illustrates the structure and style we wanted for our features, exemplified by a code snippet from the beginning of the `addApplicant.feature` file. Each feature file starts with a short description of what is being tested as well as the specific endpoint under test. This is followed by any applicable background steps that are identical for each scenario in the feature. In this case there are two background steps. The first one, "`Given the user is an authenticated user`", is present in every feature since we need a valid token to be able to call any endpoint in the system. How this is done is considered out of scope for this thesis. The second background step calls the endpoint that creates an application and assumes that the call is successful (we can assume this since we already have another feature file that tests

that endpoint). These two steps ensure that every scenario in the feature starts with an authenticated user and a default application, eliminating the risk of inconsistent test results caused by any prior modifications made to the application.

Whenever possible, the first scenario in each feature should be a straightforward happy case. In this feature that means a scenario where the user adds one valid applicant to the application, followed by a check that the applicant really was added.

```
Feature: Add applicant(s)
  The user wants to add applicant(s) to the application
  The endpoint tested in this feature is:
    "/applications/{applicationId}/applicants" with method POST

  Background:
    Given the user is an authenticated user
    And an application has been created

  Scenario: The user adds 1 applicant
    When the user adds a valid main applicant
    Then the applicant is added to the application
```

Figure 36. Example of a part of a feature file from `addApplicant.feature`.

This first happy case scenario is then followed by more scenarios, both happy cases and unhappy cases. The amount of, and content in, the rest of the scenarios vary between features depending on the complexity of the endpoint associated with the feature. A few examples of how to write a bit more elaborate scenarios will be presented later in this chapter, but first we will show how the step definitions behind the aforementioned steps are implemented.

4.8.3 Step Definitions

Each step used in the features maps to a specific step definition where all the logic behind the step is defined. We decided to create a step definition class for each feature file, where all the steps introduced in the feature file are implemented. That is, the `addApplicant.feature` file introduced earlier has a `AddApplicantSteps.java` class associated with it. To improve readability and maintainability, the step definition classes are divided into three sections - a Preconditions-section with all the `Given`-steps, an Actions-section with all the `When`-steps and an Assertions-section with all the `Then`-steps.

Returning to the steps presented in Figure 36, the logic behind the two background steps are located in `AuthenticationSteps.java` and `CreateApplicationSteps.java`, respectively. Figure 37 illustrates how the two steps in the scenario are implemented in the `AddApplicantSteps` class.

```
public class AddApplicantSteps extends FunctionalTest {

    /**
    ******** ACTIONS ********
    */
    @When("the user adds a valid main applicant")
    public void theUserAddsAValidMainApplicant() {
        testContext.partyId = UUID.randomUUID();
        testContext.jsonObject = loadJson(testContext.ADD_APPLICANT,
            "main_applicant");
        addApplicant(testContext.jsonObject,
            testContext.partyId, testContext, port);
    }

    /**
    ******* ASSERTIONS ******
    */
    @Then("the applicant(s) is added to the application")
    public void applicantsWasCreated() {
        assertEquals(201, testContext.response.getStatusCode(),
            "Add applicant should return " + 201);
    }
}
```

Figure 37. Example of step definitions from `AddApplicantSteps.java`.

While writing the step definitions we noticed that the process of adding an applicant had to be repeated in multiple step definitions. In an effort to reduce redundancy, we created a reusable method called `addApplicant()`, responsible for calling the appropriate endpoint and handling the response. To keep the step definition classes clean and maintainable, this method, along with other helper methods, was placed in a utility class. Helper methods were organized into different utility classes based on their functionality, in this case, `addApplicant()` was added to `ApplicantUtil.java`. The `addApplicant()` method is illustrated in Figure 38.

```
public static void addApplicant(JSONObject jsonObject,
                                 final UUID partyId,
                                 TestContext testContext,
                                 int port) {
    jsonObject.put("partyId", partyId);
    testContext.request = given().baseUri(getBaseUrl(port))
       .headers(testContext.headers).body(jsonObject.toString());
    testContext.response = given().spec(testContext.request)
       .post("/applications/" + testContext.applicationId + "/applicants");
}
```

Figure 38. The `addApplicant()` method.

4.8.4 Scenario Outline

We will now present a few more scenarios, highlighting the flexibility that comes with using Cucumber and WireMock, starting with scenario outlines. All examples come from the `addApplicant` feature and all the steps are implemented in `AddApplicantSteps.java`.

Scenario outlines is a compact way of defining multiple scenarios that are almost identical. In our case, we wanted to test that we could add multiple applicants to an application. Instead of writing multiple scenarios, identical except for the number of applicants added, we used the scenario outline depicted in Figure 39. What will happen when the tests run is that the steps in the scenario outline will run multiple times, in this case five times, each time with the values in one specific row of the `Examples`-table as input. In each iteration we are adding the specified number of applicants to the application and asserting that the correct number of applicants have been added. Remember that we still have the background steps, meaning that we start with a default application each iteration of the scenario outline.

```
Scenario Outline: The user adds multiple applicants
  Given a main applicant has been added to the application
  When the user adds <secondary_applicants> secondary applicants
  Then <total_applicants> applicants is added to the application

  Examples:
    | secondary_applicants | total_applicants |
    | 1                    | 2                |
    | 2                    | 3                |
    | 3                    | 4                |
    | 4                    | 5                |
    | 5                    | 6                |
```

Figure 39. An example of a scenario outline.

Additionally, due to the flexible nature of Cucumber expressions mentioned in Chapter 3.5.3, *Example of Writing Tests Using Cucumber and Gherkin*, we only need two step definitions to cover all five scenarios outlined in the scenario outline. These step definitions are illustrated in Figure 40.

```
@When("the user adds {int} secondary applicant(s)")
public void theUserAddsASecondaryApplicant(int max) {
    for (int i = 0; i < max; i++) {
        testContext.partyId = UUID.randomUUID();
        testContext.jsonObject = loadJson(testContext.ADD_APPLICANT,
            "secondary_applicant");
        addApplicant(testContext.jsonObject, testContext.partyId,
            testContext, port);
    }
}

@Then("{int} applicant(s) is added to the application")
public void applicantsWasAddedToTheApplication(int applicants) {
    getApplication(testContext, port);
    assertEquals(applicants,
        testContext.response.getBody().jsonPath().getList("applicants")
        .size());
}
```

Figure 40. The step definitions associated with the scenario outline.

So, to summarize, when utilizing the scenario outline functionality in Cucumber and the flexibility of Cucumber expressions, we can test five different scenarios using just one scenario outline and two step definitions.

4.8.5 Tags

When we started writing scenarios covering unhappy cases for adding an applicant we noticed that there are many reasons why an applicant can not be added to the application. Since we wanted to check that the call to the `addApplicant`-endpoint failed for the correct reason, while also not wanting to define a specific step for each scenario, we decided to use tags. Figure 41 illustrates two scenarios where we expect a failure. The scenarios each have a tag describing the expected reason for failure, `@sameApplicantTwice` and `@noMainApplicant`.

```
@sameApplicantTwice
Scenario: The uses attempts to add the same applicant twice
  Given a main applicant has been added to the application
  When the user attempts to add the same applicant again
  Then the applicant is not added

@noMainApplicant
Scenario: The user attempts to add a secondary applicant without first
          adding a main applicant
  When the user attempts to add a secondary applicant
  Then the applicant is not added
```

Figure 41. Two scenarios with tags describing unhappy cases.

Both scenarios use the same `Then`-step, "`Then the applicant is not added`". Figure 42 shows the step definition for this step and Figure 43 the implementation of the `validateFail()` method used.

```
private Scenario scenario;

@Before
public void beforeScenario(Scenario scenario) {
    this.scenario = scenario;
}

@Then("the applicant is not added")
public void anApplicantShouldNotBeAdded() {
    validateFail(400, scenario, testContext);
}
```

Figure 42. The step definition for the "`Then the applicant is not added`" step.

The step definition is simple, we just call the `validateFail()` method. However, we need to send the scenario being executed as an input to this method to later be able to extract the correct tag.

The `validateFail()` method in Figure 43 starts by extracting the tag that contains the reason for failure from the correct scenario. It then gets the expected error message for that specific tag from the local variable `errorMessages`. The variable is a hash map consisting of key-value pairs which are read from a JSON file when the tests are initialized.

```
public static void validateFail(int errorCode,
                                Scenario scenario,
                                TestContext testContext) {
    String reasonTag = getReasonTag(scenario);
    String expectedMessage = getMessage(reasonTag);
    String actualMessage = extractFailureMessage(testContext.response);
    assertEquals(errorCode, testContext.response.getStatusCode(),
        "The error code should be: " + errorCode);
    assertEquals(expectedMessage, actualMessage,
        "Validation message mismatch for reason: " + reasonTag);
}

private static String getReasonTag(Scenario scenario) {
    return scenario.getSourceTagNames()
           .stream().filter(tag -> tag.startsWith("@"))
           .map(tag -> tag.substring(1)).findFirst()
           .orElseThrow(() -> new RuntimeException("No reason tag
                found for scenario"));
}

public static String getMessage(String reasonTag) {
    return errorMessages.get(reasonTag);
}

private static String extractFailureMessage(Response response) {
    String message = response.getBody().jsonPath()
        .getString("validationProblems[0].message");
    message = (message == null || message.isEmpty())
           ? response.getBody().jsonPath().getString("detail")
           : message;
    return message;
}
```

Figure 43. The `validateFail()` method and associated helper methods.

After that it extracts the actual error message from the response JSON using the `extractFailureMessage()` method. If the response has a `validationProblems` field, it will fetch this message, since it usually describes the error in detail. Some responses do not have a `validationProblems` field and the actual error message is then instead fetched from the `detail` field. Lastly, the `validateFail()` method asserts that both the error code and the error message in the response correspond with the expected values.

This might seem like a lot of work just to assert that something fails, but since the method is versatile we can reuse it whenever we want to check that an API call fails for the correct reason. We just have to call the method in the step definition, add a tag to the scenario and add the tag and the expected error message to the JSON file. A snippet from this JSON file is depicted in Figure 44.

```
{
  "noMainApplicant": "At least one primary applicant is required",
  "sameApplicantTwice": "Applicant with same party ID already exists",
  "noSuchCustomer": "Customer not found"
}
```

Figure 44. Snippet from the JSON file with expected error messages.

4.8.6 Customizing WireMock Responses

We also wanted to test scenarios for adding an applicant, which should fail due to the response from one of the underlying services that we had mocked. For example, we created a scenario where we attempt to add an applicant that is not already a customer. This operation should fail, as illustrated in Figure 45.

```
@noSuchCustomer
Scenario: The user attempts to add an applicant that is not a customer
  When the user attempts to add an applicant that is not a customer
  Then the applicant is not added
```

Figure 45. The scenario for attempting to add an applicant that is not a customer.

When attempting to add an applicant, the system we are testing will call an external service to check if the `partyId` provided corresponds with the `partyId` for any existing customer. We have already mocked this external service using WireMock and configured it to send a `200` response, meaning that the applicant is an existing customer. To be able to test the scenario in

Figure 45 we need to override this default response and instead send a `404` response, meaning that there is no existing customer with the `partyId` provided. Figures 46 and 47 illustrate how this is done.

The step definition, depicted in Figure 46, is similar to the step definition for adding a valid main applicant, presented in Figure 37 above, except for the `MockInternalService` object used to customize the WireMock response before the `addApplicant()` method is called. After the method call, the `MockInternalService` object is used again to reset the WireMock response to default, making sure that the customized response is not used in any other scenarios.

```
@Autowired
MockInternalService internalService;

@When("the user attempts to add an applicant that is not a customer")
public void theUserAttemptsToAddAnApplicantThatIsNotACustomer() {
    testContext.partyId = UUID.randomUUID();
    internalService.stubCustomerNotFound(testContext.partyId);
    testContext.jsonObject = loadJson(testContext.ADD_APPLICANT,
        "main_applicant");
    addApplicant(testContext.jsonObject, testContext.partyId, testContext,
        port);
    internalService.resetToDefault();
}
```

Figure 46. The step definition for attempting to add an applicant that is not a customer.

Figure 47 shows the implementation of the `MockInternalService` class. Without going into too much detail about the underlying Spring Boot mechanics, the class injects the existing `internal-service` bean into the `WireMockServer` field using the `@Autowired` and `@Qualifier` annotations. For this to work we needed to add `registerSpringBean = true` to the original `internal-service` configuration, as discussed in Chapter 4.4.4, *@EnableWireMock*.

The class has two methods. The first one, `stubCustomerNotFound()`, resets the existing mappings attached to the `internal-service` bean and replaces it with the `404` response we define in the method. The second method, `resetToDefault()`, then resets the mapping to the default values, effectively undoing any custom stubs that were set up.

```
@Service
public class MockInternalService {

    @Autowired
    @Qualifier("internal-service")
    private WireMockServer internalService;

    public void stubCustomerNotFound(final UUID partyId){
        ObjectMapper mapper = new ObjectMapper();
        ObjectNode jsonNode = mapper.createObjectNode();
        jsonNode.put("title", "Entity not found.");
        jsonNode.put("detail", "Party not found: PartyId ["+partyId+"]");

        internalService.resetMappings();
        internalService.stubFor(get(urlPathTemplate("/parties/"+partyId))
            .willReturn(aResponse()
                .withStatus(404)
                .withHeader("Content-Type", "application/problem+json")
                .withJsonBody(jsonNode)));
    }

    public void resetToDefault() {
        internalService.resetToDefaultMappings();
    }
}
```

Figure 47. The `MockInternalService` class.

Using this technique, we can theoretically mock any possible response from any of the services we have mocked, giving us the opportunity to test almost any use case.

4.9 Setting up Automation in Jenkins

After we had written a good amount of test suites, we were ready to start thinking about deploying our tests to a CI/CD environment. There was already an available Jenkins environment for us to use, so we started writing a Jenkinsfile. We used the existing Jenkinsfile for the project as our base, and added a stage that runs all the existing test suites. Below in Figure 48 you can see the additional stage that we added to the pipeline.

There is one additional stage `'Functional Test'` and also a `post` step. The `'Functional Test'` stage runs all the tests in our subproject `tests`. The `post` step publishes the HTML test report, which gets generated when running the feature files, to the Jenkins server. This additional pipeline step is not yet integrated into the production pipeline, but we are planning

to integrate it in the future. The plan is to run the tests once everyday in the morning with a `cron` job, to spot any errors that may have occurred. We have also planned to configure the functional tests to be run after every project build, making sure that any new development does not change the expected behavior of the software. See Chapter 5.2, *Future Work* for more future plans.

```
stages {
    stage('Functional Test') {
        steps {
            sh ./gradlew :tests:test
        }
    }
}
post {
    always {
        publishHTML(target: [allowMissing: true,
            alwaysLinkToLastBuild: false,
            keepAll: false,
            reportDir: 'tests/build/reports/tests/test/',
            reportFiles: 'cucumber-reports.html',
            reportName: 'Test Report',
            reportTitles: 'Test Report'])
    }
}
```

Figure 48. The `'Functional Test'` stage.

5. CONCLUSION

5.1 Result

The goal of this project was to create a proof of concept for automated functional testing of REST APIs in an effective and stable way. We feel like we can say that we have both reached and surpassed this goal. At the time of writing we have a total of 95 test scenarios spanning 21 endpoints, already covering a lot of the basic functionality of the application we are testing. The tests are easy to understand and modify, and they run way faster than we initially expected them to.

The choice to use Cucumber feels like an especially good decision since we have gotten exclusively positive feedback from the product specialist involved in the project, emphasizing how easy it is for him to understand the tests and how this will help improve the collaboration between product specialists and developers. The structured and reusable nature of the Cucumber steps and step definitions also opens up the interesting possibility of leveraging AI, for example IntelliJ's built-in AI assistant, to generate new test scenarios, thereby reducing the time and effort required from developers to write tests.

5.2 Future work

As stated in the introduction, maintaining automated functional tests is a continuous effort. If no one acts on the test results then the tests are more or less useless. The next step will therefore be to set up a process to make sure that the test results are checked and acted on in an appropriate way. This step will also involve us giving demos to the team, both the developers as well as the product specialists and testers. For the developers the focus will be on explaining how the tests work and how to write new tests, thus making sure there will be no personal dependence on us writing all the tests. For our non-technical colleagues the focus will be on showcasing how our work could reduce the need for mundane regression testing and to get their help deciding which use cases could, and should, be automated.

With the process in place our tests will be merged to the main branch and will thereby be integrated into the daily workflow for the developers. A trial period will commence after

which the tests and the process will be evaluated and modified as needed. If the trial period yields positive results and the tests can be seamlessly integrated without reducing the team's efficiency, similar tests can also be developed for other applications developed by the team in the future.

5.3 Reflections

The work with this project has been quite extensive and time consuming, maybe more than we would initially have thought. We have, however, been allowed and encouraged to take the time required to do all the research and experimenting needed for us to reach a good result. We have learnt a lot of new technologies and frameworks as well as gained a better understanding of software testing and its importance.

In conclusion, we are really proud of what we have accomplished and look forward to seeing how it will be received by the rest of the team and the value it may create in the future.

REFERENCE LIST

Amazon Web Services. (n.d.). *The Difference Between Docker Images and Containers*. Retrieved April 29, 2025, from https://aws.amazon.com/compare/the-difference-between-docker-images-and-containers

Anti-pattern. (2025, January 14). Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Anti-pattern

Baeldung. (2024, January 8). *Database Migrations with Flyway*. https://www.baeldung.com/database-migrations-with-flyway

Berga, K. (2024, September 18). The Ultimate Guide to Unit Testing: Benefits, Challenges, and Best Practices. *TestDevLab Blog*. https://www.testdevlab.com/blog/the-ultimate-guide-to-unit-testing

Brown, R. (2014, January 28). *The Agile Testing Pyramid*. Agile Coach Journal. https://www.agilecoachjournal.com/2014-01-28/the-agile-testing-pyramid

BrowserStack. (2023, June 24). *Regression Testing: A Detailed Guide*. https://www.browserstack.com/guide/regression-testing

BrowserStack. (2025a, January 14). *Functional Testing : A Detailed Guide*. https://www.browserstack.com/guide/functional-testing

BrowserStack. (2025b, March 13). *End to End (E2E) Testing in Cucumber*. https://www.browserstack.com/guide/end-to-end-testing-in-cucumber

BrowserStack. (2025c, April 4). *What is Software Testing: Definition, Types and Best Practices*. https://www.browserstack.com/guide/what-is-software-testing

Continuous delivery. (2025, January 26). Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Continuous_delivery

Continuous integration. (2025, February 21). Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Continuous_integration

Cser, T. (2023, January 5). The Cost of Finding Bugs Later in the SDLC. *Functionize*. https://www.functionize.com/blog/the-cost-of-finding-bugs-later-in-the-sdlc

Cucumber. (2024a, April 5). *Localisation*. https://cucumber.io/docs/gherkin/languages/

Cucumber. (2024b, December 17). *Introduction*. https://cucumber.io/docs/

Cucumber. (2025, January 26). *Reference*. https://cucumber.io/docs/gherkin/reference

Cucumber Expression - Java Heuristics. (2019, March 16). GitHub. https://github.com/cucumber/cucumber-expressions/blob/main/java/heuristics.adoc

Cucumber Expressions. (2024, March 21). GitHub. https://github.com/cucumber/cucumber-expressions

Cucumber JUnit Platform Engine. (2025, March 27). GitHub. https://github.com/cucumber/cucumber-jvm/tree/main/cucumber-junit-platform-engine

Dehghani, A. (2025, March 7). *Guide to JUnit 5 Parameterized Tests*. Baeldung. https://www.baeldung.com/parameterized-tests-junit-5

Docker (software). (2025, March 20). Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Docker_(software)

England, T. (2016, August 31). Cucumber anti-patterns (part #2). *Cucumber*. https://cucumber.io/blog/bdd/cucumber-anti-patterns-part-two/

Exploratory testing. (2024, June 3). Wikipedia, The Free Encyclopedia; Wikimedia Foundation, Inc. https://en.wikipedia.org/wiki/Exploratory_testing

Fowler, M. (2012, May 1). *Test Pyramid*. Martinfowler.com. https://martinfowler.com/bliki/TestPyramid.html

GeeksforGeeks. (2025, January 4). *What is Docker Hub?*

https://www.geeksforgeeks.org/what-is-docker-hub/

Gradle. (n.d.). *Version Catalogs*. Retrieved April 10, 2025, from https://docs.gradle.org/current/userguide/version_catalogs.html

Gregory, T. (2022, March 1). 4 Benefits Of Using Gradle Multi-Project Builds. *Tomgregory*. https://tomgregory.com/gradle/gradle-multi-project-build-benefits/

Hartikainen, V. (2020). *Defining suitable testing levels, methods and practices for an agile web application project* [Master's thesis, Lappeenranta-Lahti University of Technology LUT]. https://lutpub.lut.fi/bitstream/handle/10024/161159/mastersthesis_hartikainen_ville.pdf?sequence=1

Jain, V. (2025, February 13). *Functional vs. Non-Functional Testing*. Baeldung. https://www.baeldung.com/java-functional-vs-non-functional-testing

Jenkins. (n.d.-a). GitHub. Retrieved April 27, 2025, from https://github.com/jenkinsci/jenkins

Jenkins. (n.d.-b). *Using a Jenkinsfile*. Retrieved April 27, 2025, from https://www.jenkins.io/doc/book/pipeline/jenkinsfile/

Jenkins (software). (2025, March 11). Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Jenkins_(software)

Liu, A. (2020, March 3). *Mock APIs vs. Real Backends – Getting the Best of Both Worlds*. Confluent. https://www.confluent.io/blog/choosing-between-mock-api-and-real-backend/

Magowan, K. (2024, October 30). *Shift Left Testing in Software Development*. BMC Blogs. https://www.bmc.com/blogs/what-is-shift-left-shift-left-testing-explained/

Marit. (2017, November 4). *Two Step Definitions that are the same but one is for Given and one is for Then*. Stack Overflow. https://stackoverflow.com/a/47108050

Parry, O., Kapfhammer, G. M., Hilton, M., & McMinn, P. (2022). Surveying the developer experience of flaky tests. *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, *1*. https://doi.org/10.1145/3510457.3513037

Radziwill, N., & Freeman, G. (2020). Reframing the test pyramid for digitally transformed organizations. In *arXiv [cs.SE]*. arXiv. http://arxiv.org/abs/2011.00655

Rani, V. S., Babu, D. A. R., Deepthi, K., & Reddy, V. R. (2023). Shift-left testing in DevOps: A study of benefits, challenges, and best practices. *2023 2nd International Conference on Automation, Computing and Renewable Systems (ICACRS)*, 1675–1680.

RedHat. (2020, May 8). *What is a REST API?* https://www.redhat.com/en/topics/api/what-is-a-rest-api

Schmitt, J. (2024, December 19). *The testing pyramid: Strategic software testing for Agile teams*. CircleCI. https://circleci.com/blog/testing-pyramid/

Smith, L. (2001, September 1). *Shift-Left Testing*. Dr. Dobb's. https://www.drdobbs.com/shift-left-testing/184404768

Testcontainers. (n.d.-a). *Creating a container*. Retrieved March 19, 2025, from https://java.testcontainers.org/features/creating_container/

Testcontainers. (n.d.-b). *Getting Started*. Retrieved March 11, 2025, from https://testcontainers.com/getting-started/

Testcontainers. (n.d.-c). *Modules*. Retrieved March 19, 2025, from https://testcontainers.com/modules/

Testcontainers. (n.d.-d). *Testcontainers*. Retrieved March 11, 2025, from https://testcontainers.com/

Testcontainers. (n.d.-e). *Testcontainers container lifecycle management using JUnit 5*.

Retrieved March 19, 2025, from https://testcontainers.com/guides/testcontainers-container-lifecycle/

Testcontainers. (n.d.-f). *What is Testcontainers, and why should you use it?* Retrieved March 18, 2025, from https://testcontainers.com/guides/introducing-testcontainers/

Testing Frameworks & Tools. (n.d.). Maven Repository. Retrieved March 20, 2025, from https://mvnrepository.com/open-source/testing-frameworks

Testlio. (2024a, December 6). *Bottom-Up Integration Testing: Steps, Challenges & Examples*. https://testlio.com/blog/bottom-up-integration-testing/

Testlio. (2024b, December 13). *The Ultimate Guide to Sandwich Integration Testing*. https://testlio.com/blog/sandwich-integration-testing/

Testlio. (2024c, December 13). *Top-Down Integration Testing: What It Is, Benefits and Best Practices*. https://testlio.com/blog/top-down-integration-testing/

The Apache Software Foundation. (2014, March 9). *Introduction to the Standard Directory Layout*. Apache Maven. https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

tutorialspoint. (n.d.-a). *Software Testing - Big-Bang Testing*. TutorialsPoint. Retrieved April 2, 2025, from https://www.tutorialspoint.com/software_testing_dictionary/big_bang_testing.htm

tutorialspoint. (n.d.-b). *Software Testing - Bottom Up Testing*. TutorialsPoint. Retrieved April 10, 2025, from https://www.tutorialspoint.com/software_testing_dictionary/bottom_up_testing.htm

tutorialspoint. (n.d.-c). *Top Down Integration Testing*. TutorialsPoint. Retrieved April 10, 2025, from

https://www.tutorialspoint.com/software_testing_dictionary/top_down_integration_testing.htm

Vocke, H. (2018, February 26). *The Practical Test Pyramid*. Martinfowler.com. https://martinfowler.com/articles/practical-test-pyramid.html

WireMock. (n.d.-a). *Overview*. Retrieved March 21, 2025, from https://wiremock.org/docs/overview/

WireMock. (n.d.-b). *Request Matching*. Retrieved April 30, 2025, from https://wiremock.org/docs/request-matching/

WireMock. (n.d.-c). *WireMock Spring Boot Integration*. Retrieved April 30, 2025, from https://wiremock.org/docs/spring-boot/

WireMock. (n.d.-d). *WireMock - flexible, open source API mocking*. Retrieved April 30, 2025, from https://wiremock.org/

APPENDIX

Cucumber style guide

Guide to Writing Cucumber Features

General Rules:

- Use third person (for example "the user" or "a bank employee"), NOT first person.

YES!	NO!
Given the user is an authenticated user *When* the user does this *Then* that happens	*Given* I am an authenticated user *When* I do this *Then* that happens

- Use correct spelling and grammar.
- Don't use any punctuation.

YES!	NO!
Given the user is an authenticated user *When* the user does this *Then* that happens	*Given* the user is an authenticated user, *When* the user does this. *Then* that happens!

- Capitalize keywords (Given, When, Then, Scenario etc.).

YES!	NO!
Given the user is an authenticated user *When* the user does this *Then* that happens	*given* the user is an authenticated user, *when* the user does this. *then* that happens!

- Use lower case in steps.

YES!	NO!
Given the user is an authenticated user *When* the user does this *Then* that happens	*Given* The user is an authenticated user, *When* The user does this. *Then* That happens!

- Use camelCase in tags.

YES!	NO!
@noSuchCustomer	*@nosuchcustomer* *@no_such_customer*

- Use *Background* whenever possible to keep the scenarios clean.

Features:

- Focus on testing only one function/endpoint per feature.
- If one function/endpoint have many scenarios (for example many possible inputs), consider splitting it up in multiple features.

Scenarios:

- Test exactly one rule per scenario.

YES!	NO!
Scenario: User adds 1 applicant *When* the user adds a main applicant *Then* the applicant was created *Scenario*: User deletes 1 applicant *Given* there is one applicant *When* the user deletes the applicant *Then* the applicant was deleted	*Scenario*: User adds and deletes 1 applicant *When* the user adds a main applicant *Then* the applicant was created *When* the user deletes the applicant *Then* the applicant was deleted

- Avoid unnecessary details. For example: if the scenario needs an authenticated user, use "Given the user is an authenticated user" instead of describing the authentication process.
- Focus on functionality, not technical details. Avoid using CURL-operations, explicit endpoints and HTTP status codes.

YES!	NO!
Scenario: User adds 1 applicant *When* the user adds a main applicant *Then* the applicant was created	*Scenario*: User adds 1 applicant *When* the user calls the endpoint /applications/ + applicationId + /applicants *Then* the response status code is 201

- When writing scenarios that are supposed to fail, use words such as "attempts" or "tries" to symbolize this.

YES!	NO!
Scenario: User tries to add an applicant that isn't a customer *When* the user attempts to add an applicant that isn't a customer *Then* an applicant shouldn't be added	*Scenario*: User adds an applicant that isn't a customer *When* the user adds an applicant that isn't a customer *Then* an applicant shouldn't be added

- Use as descriptive scenario titles as possible.
- Try to write scenarios in a way that doubles as documentation of the functionality. Someone not overly familiar with the system should be able to understand the functionality from the scenarios.
- Use keywords in the correct way:

- *Given*: Preconditions, something that has happened prior to the start of the test.
- *When*: The action, something that happens right now. Usually just one *When* per scenario.
- *Then*: The thing that's supposed to happen, something that happens in the future.
- *And*: Used when there are multiple *Given* or *Then*. Try not to use with *When*.

YES!	NO!
Given precondition 1 *And* precondition 2	*Given* precondition 1 *Given* precondition 2

More Information:

- Reference | Cucumber
- Best practices for scenario writing | CucumberStudio Documentation
- Cucumber anti-patterns (part #1) | Cucumber
- Cucumber anti-patterns (part #2) | Cucumber