Duy Anh Vu

HARMONIZATION OF STRATEGIES FOR CONTRACT TESTING IN MICROSERVICES UI

Faculty of Information Technology and Communication Sciences (ITC)
Bachelor's thesis
April 2022

Abstract

Duy Anh Vu: HARMONIZATION OF STRATEGIES FOR CONTRACT TESTING IN MICROSERVICES UI
Bachelor's thesis
Tampere University
Bachelor's Degree Programme in Science and Engineering
April 2022

In microservices world, a reliable continuous integration (CI) and continuous deployment (CD) contributes significantly to the delivery speed and the success of the product. One major contributor to the result of CI-CD pipeline is testing. Microservices consists of a number of small services communicating with each other through a defined interface. Different services might be managed by different teams and people, and thus, the agreed interfaces of the communication between services are potentially violated. A reliable way of testing is needed to prevent such situation.

Consumer-driven contract testing (CDC), a fast and reliable test method, is introduced to test the interface of the interaction between two services. The case study project is lacking of the interface testing, which usually leads to mismatch between the expectations of the two services in an interaction. There exists already some API route testing, but these tests do not help catch potential problems as mentioned. The thesis implementation replaces such test with CDC which cover the API route testing, communication interface, and even more.

As a new and immature test method, CDC needs to be written in a systematic and robust way to ensure a good outcome. The thesis proposes a set of practices or guideline for the case study project to help bring a systematic way of writing CDC. Some workflows for managing CDC are also introduced. In a big project, a common guideline is highly important to avoid the divergence in the way of working, which would lead to potential errors and bad quality product. It is important to note that there is no panacea, so the guideline is adapted and suitable for the case project only.

Keywords: contract testing, consumer-driven, practices, Pact, guideline, best practices, microservices

Contents

1 Introduction

The world of software development is evolving continuously with the emergence of multiple architectures. Software architectures play an important role in the development, scaling, and success of software. However, to ensure quality during the aforementioned processes, testing is a must. With microservices architecture becoming more popular across the software community, consumer-driven contract testing (CDC) is introduced as a lightweight and reliable testing method [40]. Together with new ways of testing, new practices and guidelines for writing tests are also needed.

Testing is an important and costly step in software development [3]. There are various types of testing with different purposes to help keep the software at a certain quality level. Unit tests are applied to all services to test the functionalities of the services themselves. End-to-end (E2E) tests and integration tests are used widely to test the integration and communication of a service in the whole net of services. However, with a big software, running E2E and integration tests would require spinning up some other services, or even all services in the case of E2E. When it comes to business context, running tests in the continuous integration (CI) pipeline means consuming resources and consuming money, so companies would prefer to minimize the number of these long and heavy tests while still achieving a good quality for the software. Consumer-driven contract testing can be used to test integration-heavy systems, especially microservices, more efficiently [11].

Contract testing is a state-of-the-art, yet immature, testing method, and it is not yet accustomed by a large number of developers. The number of researches on contract testing is so small compared to other ways of testing. There is also no officially published guideline on the good practices of writing and maintaining contract tests. This is a quality obstacle for companies who desire to integrate contract testing into their system. Inappropriate testing requires more resources to debug and fix. More dangerously, the tests might not test anything [14]. In the case project, there are existing contract test cases written with low quality or without any efficiency.

As a result, a guideline or a set of practices is needed to avoid inappropriate mistakes and to make the code understandable to all people in the project. Systematic practices are crucial for all organizations, especially the large-scale ones, as they are the experiences that are proved to help the organizations grow well. It is important to note that practices depend heavily on teams, organization cultures, and the infrastructure setup. Therefore, there is no best set of practices for all companies and projects.

The thesis aims to first re-introduce contract testing to the user interface (UI) services, and then study and propose a common set of practices to write and maintain contract tests using Pact tool in the company's project so that developers can follow to achieve a common workflow, and thus ensuring quality and saving time debugging. The practices are not limited to what to test, how to write the test efficiently, and what to do when breaking changes are introduced. These practices are not coming directly from any study or blog, but rather they are adapted to the case project. The guideline can also be used to help developers who are not familiar with contract testing to easily start with it and acquire a systematic way of working. The thesis targets the contract tests from a UI to its services via REST interface, but the practices should be similar across any pair of services in the system. Moreover, the scope is limited to only the consumer side of consumer-driven contract testing.

The implementation of the thesis will replace all current API router testing. Every UI service consists of two parts: a frontend and an Express server. The Express server, or the router, does the proxy for the API routes and handles the authorization. There are existing test cases for the API routes, and contract testing will replace those tests for better robustness and reliability.

The research is done by writing the tests in the project and examining the good and the bad. The practices come from the experience of the researcher, related literature, and the nature of contract testing. The experience is gained by examining the existing contract test cases, which are written in a bad way, with the recommendations on the contract testing tool's website. Breaking changes are temporarily introduced to observe the best way to catch and avoid errors. Some of the practices come from related online blogs and are adjusted to fit the case project. Seniors' opinions also contribute to the practices. Moreover, issues and limitations are discovered by searching the tickets toward the tools and also by looking at the source code.

Following the Introduction are four more chapters. Chapter 2 presents more detailed but preliminary background knowledge on microservices architecture and contract testing. Chapter 3 discusses the company's project, tools used in the research, and the research process. Chapter 4 presents the proposed set of practices for writing and maintaining the tests. Finally, chapter 5 summarizes the results.

2 Background

This chapter explains commonly used terms in the thesis. After this chapter, readers should be able to understand the context where consumer-driven contract testing happens. Microservices will be discussed first as it is the main architecture where contract testing is used. After microservices, consumer-driven contract testing will be introduced.

2.1 Microservices

Dividing the code into small parts has been accommodated for a long. Microservices architecture shows up, following a similar concept of diving into small pieces. Microservices is composed of ”micro” services, which follows the Single Responsibility Principle. Each service will be developed, tested, and deployed independently [38]. An example of microservices could be a system for registering for courses at the university. There can be one service to handle the student’s role and information, one to handle the registration, one to manage the courses, and one to do all the document export.

The characteristics and advantages of microservices architecture can be seen by taking monoliths architecture for comparison. Monolith means everything in one place. This is the traditional way of writing applications [12]. The following subsections will present the characteristics and benefits of microservices with monoliths as a competitor.

2.1.1 Modularity

As mentioned in the name, a system following microservices architecture is ”componentised” into multiple ”micro” services [29]. This modular division helps enhance the system’s scalability and maintainability. Despite having the ”micro” in the name, there is no strict agreement on how large a service in the microservices world should be. Services do not necessarily need to be small in terms of size. Amazon has the Two-Pizza Team rule, which means that a team of a dozen of people can manage a service [12]. At the same time, a service can be maintained by only a team of around six people. Moreover, a service should be small enough so that a requirement change would change only a single service. If it requires the changes to multiple services, cross-team collaboration is inevitable, and this process is costly. Therefore, the goal is to minimize the dependencies between services [23].

The modularity helps microservices to have long-term advantages compared to monoliths. In monolith, components are coupled together and communicate internally, usually through direct function calls. As monolith is a single unit, changing one part leads to the testing and the deployment of the whole system. The change costs a substantial amount of resources and hence prolongs the release cycles. A change in one service of microservices usually only requires testing and deployment of that single service, and thus the cost is much lower. For a large-scale system, using monolith could be an obstacle for scaling.

2.1.2 Organized around business capability

Traditionally, a large application is organized around functional areas [29]. This means the teams are silos where each will be responsible for one functional part of the system such as the UI or the server-side logic. When the requirements change, cross-team communication will inevitably take place. Such cross-team collaboration is time-consuming and might cause budget loss to the company.

Microservices, on the other hand, is organized around business capability [12]. Business capability is what a business uses to generate values [26]. As a result, teams following microservices are cross-functional and autonomous. This arrangement would increase the efficiency not only when developing new features but also when adapting to revised requirements. Additionally, decomposing services based on business capability makes the team's boundaries and responsibilities more apparent.

2.1.3 Smart endpoints and dumb pipes

Monolith is a single big unit, so components communicate with each other internally, usually through direct function calls. Microservices, however, decouples the components. In order to ensure the encapsulation and the loose coupling between services, they have smart endpoints. Services communicate with each other through either synchronous ways like REST interface or asynchronous ways like message queue [20]. The latter method is usually referred to as dumbness. Microservices decouples the messaging mechanism from the services and helps improve the longevity of the services [29]. However, communications in microservices depend on the networking performance, so more attention will be put into ensuring the communications happen smoothly.

2.1.4 Decentralized governance and data

A traditional solution like monoliths tends to have single and standardized technology stacks. However, each programming language, framework, and technology is designed to tackle different problems. Therefore, there is nothing that can solve everything perfectly. Decentralized governance allows services to have their development paths, programming languages, and technologies. Moreover, this way of governance brings all responsibilities to the development teams. Amazon has a famous notion ”you build, you run it” [22], which implies the devolvement of responsibilities toward the teams. When the teams are responsible for what they build and run, they are motivated to develop things with good quality.

In the traditional monoliths, the data is centralized with a single database. Microservices, on the other hand, prefers having services managing their own databases. In this way, different database technologies can be used to flexibly optimize the tasks that the service is designed for. For instance, some services can use MariaDB while some others take data from PostgreSQL. However, out-of-sync data across databases can be observed quite regularly in real life.

2.2 Consumer-driven contract testing

Consumer-driven contract testing is introduced recently as a fast and reliable testing method. It is used mainly in microservices architecture. Contract testing will be discussed more in detail in the following subsections.

2.2.1 Software testing in general

Software testing was introduced along with the software development [5]. It aims to protect the software from defects and ensures that the software functions properly. Usually, software testing is done by executing a program with a finite of test cases to detect errors [6]. The number of test cases can vary a lot based on the inputs, preconditions, or post-conditions. Traditionally, there are two main software testing techniques according to [6] and [13]:

1. Functional testing (or black-box testing): The developers do not have knowledge of the internal logic of the being tested software [8, 15, 6, 1]. Test cases are formed based on the requirements or the specification of the design of the tested software [21]. The test cases observe the differences between the output of the executed program and compare that output with the predefined expectations.

2. Structural testing (or white box testing): The developers have a clear knowledge of the code of the being tested software [21, 6]. Test cases are formed by inspecting the codebase to find ways to make the test cases execute some particular locations in the software. The locations can be a statement or condition branches. A combination of coverage criteria is used to evaluate the outcome of this testing method [21].

Recently, another testing technique is introduced called gray box testing [6]. This is a combination of black-box and white-box testing techniques [28] where the developers have some knowledge of the logic in the being tested software, i.e. the level of codebase knowledge is more than black-box technique but less than white-box technique [7].

Software testing plays an important role in the quality of the software [13]. Although testing costs resources, it helps catch defects before delivering to the customer. This can result in better customer satisfaction. When a software meets or even surpasses the customer expectations, it can gain more sales and market share [5].

Throughout the evolution of the software industry, multiple testing methods have emerged. Some popular testing methods nowadays are acceptance testing, end-to-end testing, integration testing, performance testing, and unit testing. The methods have their distinguishing characteristics and different objectives [13]. Consumer-driven contract testing is one of the new testing methods.

2.2.2 Concept

Design by contract has been a famous paradigm for years, especially in Object-oriented Programming (OOP). This approach uses a contract with assertions and constraints to ensure the benefits for both sides, client and contractor. The obligation for one side is usually beneficial for the other [16]. However, design by contract is not a way of testing, but rather a way of checking. Consumer-driven contract testing is built based on the idea of design by contracts, and it is a method for testing.

Consumer-driven contract testing is mainly used in microservices architecture to test the interactions between two services. Contract testing is not component testing, and it focuses only on the input and the output interfaces of service calls instead of on how the outputs are generated. However, contract testing does not help catch bugs if a value in a field does not match the real exact value expectation. Rather, it catches bugs when the format of the contents is incorrect. Catching the exact value mismatches is the mission of other testing methods such as integration or E2E.

CDC is more similar to grey-box testing rather than black-box testing. Sometimes, the consumer needs to know about the state or the preconditions of the provider to generate the response. The state is usually represented as a string, and the consumer needs to define exactly the state from the provider to correctly do CDC. Otherwise, CDC will fail. The topic of the provider's state will be discussed more in subsection 2.2.4.B. Knowing this string violates the definition of black-box testing from [8], [15], [6], and [1] that in black-box testing, there is no knowledge of the codebase of the provider. According to [6], [8], and the characteristics of CDC, the testing method falls into the grey box testing category. Again, CDC aims to guard against the interface mismatch between the expectations of the consumer and the provider, but not the functionality generating that interface.

A contract test involves two parties, a consumer and a provider. The consumer is the service that receives data, and the provider is the service that serves the data that the consumer wants. A service in microservices world can be either consumer, provider, or both. A provider might have many consumers, and vice versa, a consumer can consume data from multiple providers.

A contract in CDC is a document that defines the expectations regarding the interface of the interaction between the consumer and the provider. Traditionally, the contracts are written down, and the consuming and the providing teams will have to follow such specifications. There is no automated way to ensure that both teams follow the interface definitions. In modern software development, organizations are adopting agile practices, and they want the process to be more efficient and less wasteful [41]. Modern contract testing offers an automated method to transfer generated contracts from the consumer to the provider and identify if either the consumer or the provider does not comply with those contracts. The contract is usually in JSON format nowadays.

Consumer-driven contract testing is a subcategory in contract testing. In CDC, the consumer is the leader by defining the contract, i.e. defining what the consumer expects from the interface. This approach has a major advantage in that it can test the interface partially. ”Partially” means only parts of the interaction that the consumer uses are tested, and the rest are free for changes [31]. For example, the provider response looks like the following:

```
{
    "name": "Andy",
    "age": 21
}
```

There are two consumers, service A only needs the ”name” field to work, while service B needs ”age” fields for its operation. In the contract defined by service A, only the ”name” field is asserted. In this way, if service B and the provider agree to change the ”age” to ”ages”, the contract between service A and the provider will not fail. The isolation is achieved in this testing.

2.2.3 Why contract testing?

As the services in microservices usually communicate with each other using HTTP resource APIs or lightweight message bus [12], interface mismatch is a potential problem. Therefore, testing is required to test the interactions between the services.

There are multiple methods for testing, each with different purposes, complexity, and load. The differences are usually depicted using the testing pyramid (Figure 2.1).

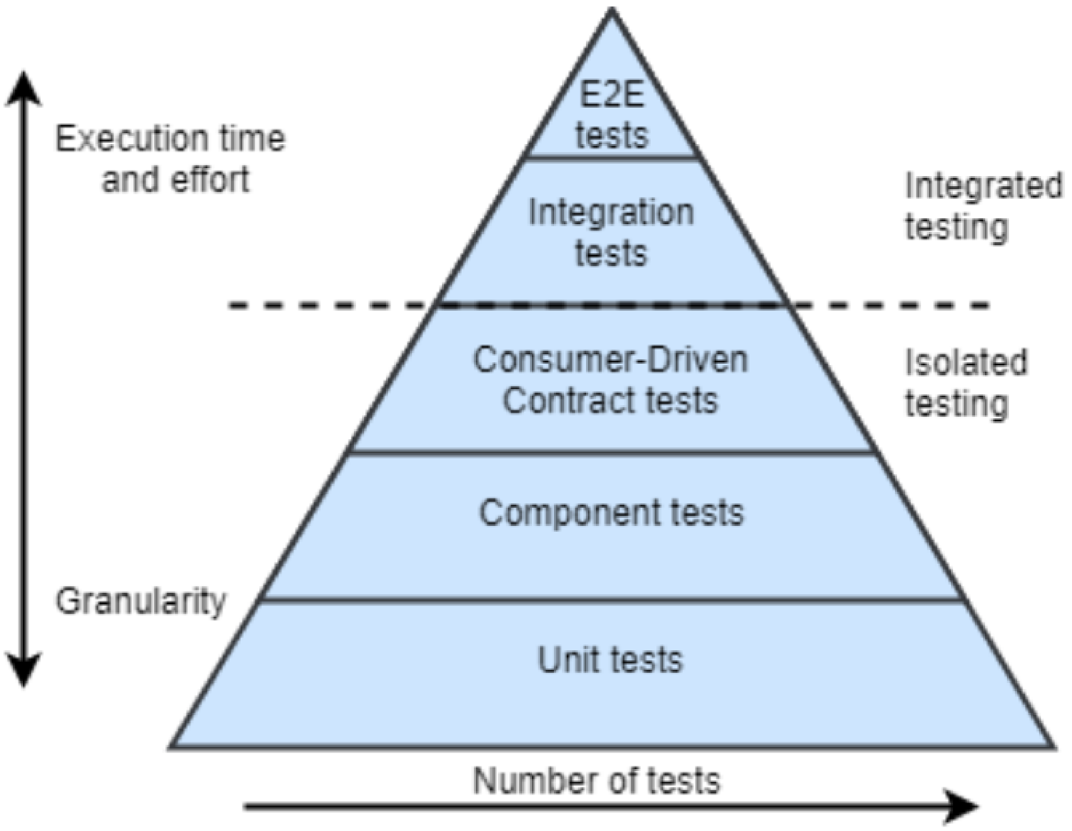

Figure 2.1 *Testing pyramid with consumer-driven contract tests [11]*

Going from the bottom to the top of the pyramid, the complexity, execution time, and cost increase. Therefore, companies try to move the tests to the lower level of the pyramid while keeping the reliability unchanged. When there are a tremendous number of communications between services, there inevitably exists a misunderstanding or accidental changes to the agreed interfaces. Therefore, E2E and integration tests exist to catch those defects.

In microservices, E2E and integration tests are the most relied approach and are the closest to what the customer experiences. However, these methods have some particular drawbacks. Firstly, they are slow and resource-consuming. Running an E2E or integration test would require spinning up several services, or the whole system. The bigger the software, the more resource is consumed, and the more money is spent. Moreover, when the system grows, more tests are added, and the complexity will rise exponentially. Secondly, they are hard to maintain when they fail. E2E and integration depend on the configuration, the environment where they run, and the state of other dependencies before they run. For a large number of services, ensuring a correct versioning and data status is not an easy and cheap task. Lastly, they are hard to fix. There are a large number of services that can cause problems. Moreover, usually, E2E and integration tests are executed after the code changes have been merged, so getting the correct changes merged would require even more time. If the CI pipeline of the problematic service is unstable, getting the bug fix in the testing environment might take more than a week.

CDC offers a faster and more reliable method. It does not require the whole system to be involved in testing, but only two services at a time. When including only the interactions between two services, it does not require the developers to have a wide understanding of the system to write the tests efficiently. CDC helps isolate the problem detection to only one component that the developer is working on in an earlier stage of development. It is somewhat similar to unit testing, so exact problems can be identified before the changes are merged. Most importantly, writing CDC correctly can replace the integration tests [11] and save a huge amount of cost. However, E2E is still needed as there are some spots that CDC cannot cover. For example, testing the interactions between two UIs might not be possible with CDC.

2.2.4 Mechanism

In CDC, the consumer and the provider do not interact directly with each other like in the integration test. Instead, there is an intermediate platform that allows the consumer to publish the expectations and lets the provider fetch those expectations to verify against its implementation. In case the tests fail, communication between teams is needed to resolve the problem. An overview of the testing process is illustrated in Figure 2.2.

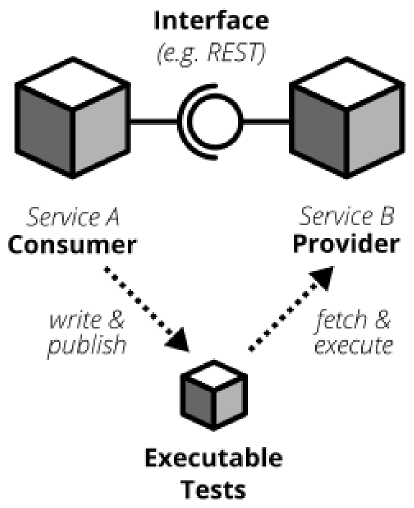

Figure 2.2 *Automated process in CDC [41]*

From Figure 2.2, there are two steps in CDC: consumer testing and provider verification. The service which stores the contracts, or the expectations, from the consumers, can be anything depending on the projects. One of the best tools nowadays for this purpose is Pact Broker, which will be mentioned in later sections.

A. Consumer testing

Consumer testing is the first step in CDC and happens in the CI pipeline of the consumer. This part acts as a unit test for the API function in the consumer to assert if the function generates the correct payload and handles the response or error appropriately. This is also the phase where the list of expectations, which is also called a contract, from the consumer is generated. The flow is illustrated in Figure 2.3.

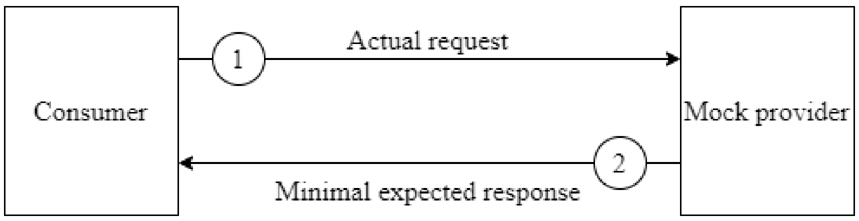

Figure 2.3 *Consumer testing phase in CDC*

The consumer defines its expectations and registers a mock response to the mock provider. The mock response is ”minimal” because it will contain only part of the real response that the consumer wants. In this stage, the consumer runs the tests, usually in form of unit tests, against a mock server. After the tests pass, the testing tool generates a contract and publishes it to a remote inventory during the CI build. The contract (usually a JSON file) can be transferred to the providing team to implement the interface according to the consumer’s expectations, but a remote artifact repository would promote better automation. The consumer testing phase reinforces the idea of being consumer-driven as the provider has to follow what its consumers want.

B. Provider verification

After the consumer pushes the expectations, the provider can pull these and run the verification with its implementation. This part verifies the contract generated from the previous phase. The verification is important to see if either the consumer or the provider can be merged or deployed to production. The process is displayed in Figure 2.4.

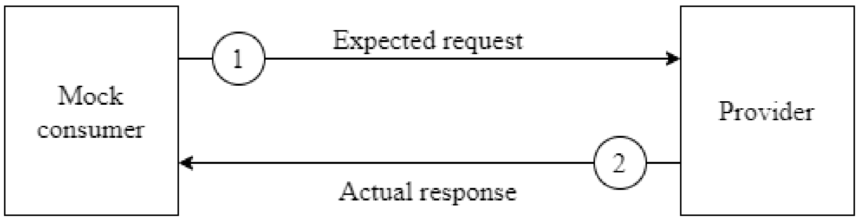

Figure 2.4 *Provider verification phase in CDC*

The mock consumer fires a request toward the provider and the provider will produce the actual response. The response will be checked with the consumer's expectations, which are defined in the contract file. The actual response might contain more information than what is expected by the consumer, and the verification passes when the actual request includes at least the data defined in the minimally expected response [32].

In reality, usually, the expectations from the consumer rely on some specific states of the provider. A state is the precondition to generate the desired response. Therefore, the provider would need to do a setup so that it has the correct states. Examples of states could be log-in status or particular database data. When using state in CDC, the consumer should use the correct identifier for the state as defined in the provider so that the tests are run with the correct preconditions. If the provider depends on other 3rd services to produce the response, which happens frequently in microservices, it has to mock the responses from those 3rd services to produce the response for the consumer. Another way is not mocking the responses but spinning up dependent services through docker-compose. This might not be feasible if the number of dependent services is huge or those services are resource-consuming. However, the latter way is not recommended as it loses the lightweight characteristic of CDC. If someone wants to follow the latter way, the integration tests should be conducted instead to cover more aspects in the same run.

The flow of CDC might be more complicated than what has been described above, especially when either side introduces a breaking change. More details on the flow will be discussed in later chapters.

3 Case study

The chapter describes the background of this study, including the company's project, used tools, and situation analysis.

3.1 The company's project

The company's project is a network management software using microservices architecture. The system used in this thesis is a small slice of the whole big system. Deeper communications and components have been omitted for simplicity and confidentiality. From now on, the word "system" is referring to the sliced system. Figure 3.1 illustrates the system in this study.

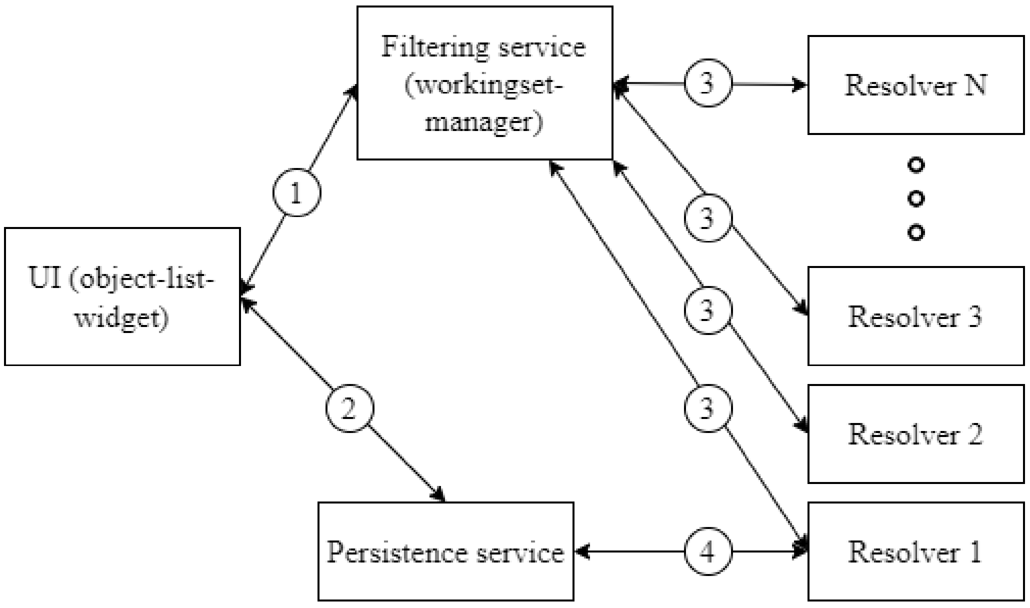

Figure 3.1 *A part of the company's project*

Communication explanations:

1. UI sends the filtering formula to the filtering service and receives a filtered set of simplified data

2. UI queries for detailed data from persistence service

3. Filtering service separates the filtering formula into criteria and sends each of those to corresponding resolvers. The resolver will return the filtered data for a criterion. For example, the formula is A & B, A will go to a resolver, while B goes to the other. The filtering service will analyse the formula conditions

with results from resolvers to produce the last filtered output. In the example, the result from resolver A will intersect with the result from resolver B.

4. A resolver communicates with some other services such as persistence service or database to do the filtering on the persisted data.

The system is a part of a big microservices architecture. In the study, the focus is put on a UI and two of its direct backend services. The UI displays the information on network components with a grid and is able to trigger some operations on the network components. Filtering service and resolvers are responsible for filtering the data related to network objects. Persistence service provides different types of queries on persisted network data. Communications between components in the whole system are both synchronous via REST interfaces and asynchronous via message queue (Kafka). However, in the scope of this thesis, only the synchronous way is discussed.

3.2 Tool used

Pact is a tool supporting CDC. Pact supports various popular programming languages nowadays, e.g. JavaScript, Java, and Go. Writing tests with Pact is similar to writing unit tests, so it is easy to learn and adapt. The biggest selling point of Pact is that it provides some useful ready-made tools for a smoother testing process. Pact offers Pact Broker which is a platform to store and share the contracts as well as verification results [17]. For CI pipeline and automated testing, a ready-made platform like this saves a lot of time developing storage and fixing bugs. Pact Broker provides the matrices of different versions of consumer and provider and the test result between those particular versions. The company project is using only Pact for the purpose of contract testing. However, one big downside is that Pact is not suitable for testing the pass-through services, which is a common case in the microservices world [34].

Can-i-deploy and webhook [35, 33] are also essential tools in the CI pipeline of the services. These two tools are offered by Pact to help test automation become easier. The case project uses both tools in all consumers' CI pipelines. Can-i-deploy tool, by its name, protects the developers from deploying falsy changes. Originally, deploying means bringing the merged changes into production, so the tool's name indicates that it should be used only in the master build to see if the product can go to production. In the case project, however, can-i-deploy tool is used also in the development pipeline, before the change goes to master, to prevent a breaking change from being merged. Therefore, the tool's name can be understood as ”can-i-merge” for more clarity. In CDC, can-i-deploy tool is used to check if the currently built implementation is compatible with the version in production of all of its providers.

This tool is called after the consumer testing part passes and the consumer publishes new contracts to Pact Broker. It helps detect the breaking changes from an early stage of development and prevents bad code inside the master branch.

Webhook exists for each pair of services in CDC. It is used to trigger the provider verification pipeline when the contract changes. Webhook is usually used together with can-i-deploy tool to ensure the compatibility of the consumer's implementation before getting merged. It should be configured so that the correct version of the provider or the consumer is used in the verification step by using tags assigned to builds. More details on how tagging should work will be discussed in chapter 4. It is important to note that the webhooks in the company project have been configured, but not all of them are working as expected.

The research is conducted on a Windows machine with NodeJS version 14.19.0 and NPM version 6.14.16. The list of involved Node packages and their version is presented below:

```
{
    "jest-pact": "0.9.0",
    "jest": "23.6.0",
    "@pact-foundation/pact": "9.16.0",
    "@pact-foundation/pact-node": "10.12.2"
}
```

It is important to note that the different versions listed above might cause the packages to behave differently, and some limitations might be fixed in the later versions.

3.3 Situation analysis

This section presents the current situation regarding contract testing in the project, particularly from UI, so that the reader can have a better understanding of the problems that this thesis is attempting to solve.

An interview was conducted in form of a questionnaire to gather answers for the current situation of the project. Interviewees are selected based on their work responsibilities and job level. Most of the interviewees have long years of experience and hold high positions in the project. This is because they see a big picture of the project's situation and future development. Not all interviewees specialize in UI development, but all have experience with CDC. The table 3.1 below represents the details of the interviewees:

The following sections will be the summary and discussion regarding the answers.

Table 3.1 *Interviewees*

Index	**Job title/Main responsibility**
1	Software Engineer - Lead UI developer
2	Agile coach - UI development coach
3	Software Engineer - Scrum Master
4	Technical Specialist

3.3.1 Current situation

Experts' answers all indicate that there is a lack of contract testing in the project in terms of both quantity and quality. Some backend components already have sufficient contract testing coverage, but this is not the case with the UI components. For UI components, either there is no contract testing or the existing test cases do not catch necessary errors. However, all interviewees believe that the intentions and behaviors of CDC between UI - backend services and between backend services - backend services are similar. Therefore, as UI is also a component inside the microservices system, it should have CDC at a certain level.

CDC is a relatively new concept in software development, and thus not many people understand it and know how to write the tests. From the point of view of the interviewees, there exists only a small number of people have good knowledge of CDC. Many developers find implementing and maintaining proper contract tests a real struggle. During the planning period, many people estimate CDC part as a big effort, so there is certainly a gap in know-how here. A possible reason for this situation is that CDC tends to be used more frequently in microservices, but not all companies in the industry use microservices.

3.3.2 Expectations

Based on the current situation, it is evident that contract testing should be introduced to the UI components. There is a high need to avoid any bug related to malformatted requests or responses by detecting breaking changes in the API. Examples of bugs given by interviewees can root in incorrect payload formation or deprecated endpoints. CDC could help prevent the breaking API changes from breaking the communication between services, and API versioning could be used to upgrade the API without causing immediate breaking changes.

Currently, some backend services have both CDC and integration tests. Whether writing CDC properly sufficiently could replace integration tests is a controversial topic among interviewees. Some believe that having CDC implemented with high quality could replace all integration tests, but the rest asserts that integration tests are still required in certain scenarios where CDC does not provide enough coverage.

For the latter group, integration tests still help test the functionalities rather than just interface like CDC. Nevertheless, all agree that CDC is needed as the project scales due to its lightweight, cost-saving, reliability, and independence.

Finally, the biggest concern of all interviewees is having a common guideline on writing CDC. In a small project, an individual can guarantee that all CDCs are written with good practices, but this is not the case with a large project. In a large project, people have various levels of skills and points of view, and hence it is easy for everyone to have divergent opinions on how CDC should be implemented correctly. In this project size, team effort, which requires good communication, is mandatory for success. A set of shared practices is a way of such communication to ensure sufficient trust in the contract testing layer.

4 Practices

This chapter is the main part of the thesis where the results are presented. First, an important tool Matcher is introduced as well as an example of a good CDC test case with the explanation. Then, a set of practices to write good CDC is proposed. Finally, the workflow when facing breaking changes is presented.

The practices are based on the nature of CDC and some guidelines on the Pact's official website. Some literature act as the foundation for the practices. Experiences of the thesis writer when writing CDC in the case project also contribute to the result. This experience might come from the writing of the test cases and from the comparison of the existing flaky test cases with the correct ways. All the practices focus on the consumer part of CDC due to the scope of the thesis. It is important for all project members to follow the common, approved guideline in order to achieve a quality result.

The chapter concerns only testing with JavaScript as it is targeted for the UIs which are all written in JavaScript. However, the idea and practices should be applicable to any backend service with different programming languages. Pact developed for different languages behaves differently in some special cases as well.

The study uses Pact specification version 2 as versions 3 and 4 are in the Beta phase. The later version of the Pact verification comes with more Matchers and more flexible matching rules.

4.1 Writing contract tests with Pact

This section first explains the structure of a consumer test (described in subsection 2.2.4.A) file in CDC through example code extractions. It then introduces the Matchers, an important tool in CDC consumer test, which can be used to define rules for the interface.

4.1.1 Structure through example

The section presents the structure of the consumer test by walking through the writing process of a test file using existing tools/services in the case project. The example is about communication from the UI (object-list-widget) to a direct backend service: workingset-manager. Working set is a JSON file representing the user's preferences on what they want to see in the table of the UI. The API fetch function is listPrivateWorkingset(). The tool supporting CDC is Pact and jest-pact, so the example is specific to those tools.

The consumer test aims to test the API fetch function [37] and generates the contract which can be used to validate against the provider during the provider verification phase (described in subsection 2.2.4.B). In order to achieve such goals, a CDC test case should include sufficient information. Therefore, a test case will contain two steps to the final goals: a step that inserts the expectations into the contract file and a step that tests the API fetch function.

Firstly, the consumer and the provider IDs are defined using pactWith provided by jest-pact.

```
pactWith(
    {
        consumer: "object-list-widget-rest",
        provider: "workingset-manager-rest"
    },
    provider => {
        // test case definition goes here
    }
);
```

These IDs will be written into the contract and later on used by the Pact Broker to bring the contracts from the consumer to the provider appropriately.

Secondly, the test case is defined in the second callback of pactWith. Before any consumer test case, a mock server is spun up. This mock server will respond to the request sent from the consumer application. All UI repositories in the case project contain a fake backend service that is used for development purposes. This fake backend represents closely the response of the real services, so it can be used for CDC.

```
provider => {
    let server;
    beforeAll(async () => {
        process.env.WS_MANAGER_URL = provider.mockService.baseUrl;
        const { app, config } = require("../server/src/app");
        await new Promise(resolve => {
            server = app.listen(config.port, () => {
                resolve();
            });
        });
    });
}
```

Then, the definition of a test case is written. The way of defining the test case for CDC is similar to unit testing using Jest. Inside each test case, the first job is to define the interaction details.

```
provider => {
    ...
    beforeAll(...);
```

```
        it("Fetch list of saved private working sets", async () => {
            await provider.addInteraction({
                state: "Resolve Working Set 'normalWS', which has
                    objects",
                uponReceiving: "GET a list saved private working sets",
                withRequest: {
                    method: "GET",
                    path: "/v1/workingsets",
                    headers: {
                        "Content-Type": "application/json"
                    },
                    query: {
                        private: "true"
                    }
                },
                willRespondWith: {
                    status: 200,
                    headers: {
                        "Content-Type": "application/json"
                    },
                    body: eachLike({
                        admin: false,
                        id: "ws-1",
                        name: "My working set",
                        formula: "c",
                        criteria: eachLike({
                            id: "c",
                            type: "some-type",
                            descriptor: like({})
                        }),
                        labels: {
                            dataType: "integratedId"
                        }
                    })
                }
            });
        });
    }
```

An interaction description includes at least the request details and the response details. The state of the provider is optional. This interaction definition will generate the expectations into the contract file in JSON format. By defining the interaction, one step to the goals has been achieved. Furthermore, the willRespondWith field defines the mock response for the mock provider.

Thirdly, the last step to the goals will be executed. The API fetch function, which in this case is listPrivateWorkingset(), is triggered.

```
provider => {
    ...
    beforeAll(...);

    it("Fetch list of saved private working sets", async () => {
        await provider.addInteraction(...);

        await listPrivateWorkingset().then(res => {
            expect(res).toBeInstanceOf(Array);

            res.forEach(r => {
                expect(r).toMatchObject({
                    admin: expect.any(Boolean),
                    id: expect.any(String),
                    name: expect.any(String),
                    formula: expect.any(String),
                    criteria: expect.arrayContaining([
                        expect.objectContaining({
                        id: expect.any(String),
                        type: expect.any(String),
                        descriptor: expect.any(Object)
                    })
                ])
            });
        });
    });
}
```

A then() statement is attached after the API call to examine if the response is handled properly and satisfies the conditions. This concludes a test case and completes the last step toward the goals.

Finally, the mock server needs to be turned off after all test cases run. This is done inside the afterAll block.

```
provider => {
    beforeAll(...);

    it(...);
    ...

    afterAll(async () => {
        await new Promise(resolve => {
            server.close(() => {
                resolve();
            });
        });
    });
}
```

This is the last statement in a CDC test file. Shutting down the mock server is to avoid potential interference with test cases that run later on.

Putting all the abovementioned parts together results in a complete CDC test file. The full example test file is attached in the Appendix A. The generated contract is presented in the Program 4.1 below:

```
{
    "consumer": {
        "name": "object-list-widget-rest"
    },
    "provider": {
        "name": "workingset-manager-rest"
    },
    "interactions": [
        {
            "description": "GET a list saved private working sets",
            "providerState": "Resolve Working Set 'normalWS', which has objects",
            "request": {
                "method": "GET",
                "path": "/v1/workingsets",
                "query": "private=true",
                "headers": {
                    "Content-Type": "application/json"
                }
            },
            "response": {
                "status": 200,
                "headers": {
                    "Content-Type": "application/json"
                },
                "body": [
                    {
                        "admin": false,
                        "id": "ws-1",
                        "name": "My working set",
                        "formula": "c",
                        "criteria": [
                            {
                                "id": "c",
                                "type": "some-type",
                                "descriptor": {
                                }
                            }
                        ],
                        "labels": {
                            "dataType": "integratedId"
                        }
                    }
                ],
                "matchingRules": {
                    "$.body": {
                        "min": 1
                    },
                    "$.body[*].*": {
                        "match": "type"
                    },
                    "$.body[*].criteria": {
                        "min": 1
                    },
                    "$.body[*].criteria[*].*": {
                        "match": "type"
                    },
                    "$.body[*].criteria[*].descriptor": {
                        "match": "type"
                    }
                }
            }
        }
    ],
    "metadata": {
        "pactSpecification": {
            "version": "2.0.0"
        }
    }
}
```

Program 4.1 *Example generated contract*

All necessary information has been included in the contract after running the written test case. Most of the fields are quite similar to what has been defined in the test case. However, there is a field ”matchingRules” which defines the rules for the values in the payload fields. ”matchingRules” is determined by the Matchers which is discussed in the next subsection.

4.1.2 Matchers

Matchers are crucial tools offered by Pact to work with CDC. Matchers exist to assert if the shapes of the values are as expected. This is because CDC concerns with the shape of the payload including the fields’ names and the contents in those fields. The contents can be asserted by the exact match or by some other rules such as type, number of elements, and even format. However, usually, the contents should be asserted with the most flexibility as possible, and Matchers serve this purpose well. The detailed reasons are explained in subsection 4.2.4. Furthermore, Matchers are usually used in the response part of the interaction as the request part needs to be strict (details in subsection 4.2.2).

There are three common Matchers in Pact specification V2: like(), eachLike(), and term(). term() is the Matcher to match a regular expression; however, it is not widely applicable in the JavaScript repositories of the project. Therefore, the focus of this section would be like() and eachLike(). like() and eachLike() are the most generic Matchers. eachLike() is basically similar to like() but is applied to an array and asserts all elements inside the array with like(). For primitive type arguments, like() generates a matching rule so that it matches the type of that argument. For object type arguments, like() asserts two things: if the fields exist and if the values inside the corresponding fields are in the correct type. This Matcher works with nested object structure, so it is unnecessary to rewrite the Matcher everywhere. An example of the code that uses Matchers and the corresponding generated matching rules from Program 4.1.1 are illustrated in the Program 4.2 and Program 4.3 below.

```
eachLike({
    admin: false,
    id: "ws-1",
    name: "My working set",
    formula: "c",
    criteria: eachLike({
        id: "c",
        type: "some-type",
        descriptor: like({})
    }),
    labels: {
        dataType: "integratedId"
    }
})
```

Program 4.2 *Example use of Matchers*

```
{
    "$.body": {
        "min": 1
    },
    "$.body[*].*": {
        "match": "type"
    },
    "$.body[*].criteria": {
        "min": 1
    },
    "$.body[*].criteria[*].*": {
        "match": "type"
    },
    "$.body[*].criteria[*].
       descriptor": {
        "match": "type"
    }
}
```

Program 4.3 *Generated matching rules*

The top-level eachLike() generates the matching rule `"$.body[*].*"` which indicates everything in the path will be matched based on value type. At the same time, the path has a wildcard at the end, which means `labels.dataType` being a string is included as well. Therefore, a single like()/eachLike() defined on the top level would be sufficient for generating rules. Despite having a wildcard at the end of the JSON path, more specific rules can be established. For instance, eachLike() appearing in line 6 would impose more detailed rules for field ”criteria.” This means ”criteria” holds an array with at least 1 element (rule `"$.body[*].criteria"`), and each element has the fields and corresponding value types as defined in the argument of eachLike() (rule `"$.body[*].criteria[*].*"`).

The main reason to have eachLike() here instead of just using a single top-level like()/eachLike() is that top-level like()/eachLike() imposes too strict rules on ”criteria” which is not true in the real response. The generated matching rules look like below when no eachLike() is used in line 6:

```
{
    "$.body": {
        "min": 1
    },
    "$.body[*].*": {
        "match": "type"
    },
    "$.body[*].criteria[0].descriptor": {
        "match": "type"
    }
}
```

and when the contract is tested against the provider, the differences between the actual and the expected response are:

```
{
    "differences": [
        {
            "description": "Actual array is too long and should not
                contain a Hash at $[0].criteria[1]"
        }
    ]
}
```

Without eachLike(), the generated matching rules expect that the array in the ”criteria” field contains only one element with the defined interface. The actual response has more than 1 element in the array.

Finally, like() with an empty object as an argument in line 9 indicates that the type of value in ”descriptor” is an object regardless of the fields inside that object. This ”hack” follows Postel’s Law (more details in subsection 4.2.2). If an empty object is set to be the minimum expectation, any non-empty object can easily fulfill this minimum expectation.

4.2 Testing strategies

The section presents the objectives of writing contract testing as well as things to avoid. Testing coverage of CDC is also discussed.

4.2.1 Use CDC to cover API fetch functions

Using Pact, the consumer testing part can be used to unit test the API call components [37]. Test cases on the consumer side help test the internal functionality to create the request on the consumer side already, so there is no need for redundant unit tests for API call functions in the consumer service. Redundant tests waste resources and make them very difficult to maintain in the long run.

In the case project, testing is crucial. Unit tests are written for all functionalities including the API fetch function. SonarQube is used to prevent a change that reduces overall test coverage from being merged. However, for the API fetch functions before CDC is introduced to the project, there are existing unit test cases to check if the responses are handled properly, but there is no test to assert the correct request creation. With CDC, a single consumer test case can cover both the request creation and the response handling. The missing gap in the request creation would be filled, and all response handling unit tests can be safely integrated into CDC. A single place for testing both the request and the response would help the project to scale better with higher maintainability.

4.2.2 Apply Postel's Law

The request payload should be treated with the most strict rules, while the response should be treated with the most lenient rules [18]. This follows Postel's Law which is also known as the robustness principle: ”be conservative in what you send, be liberal in what you accept” [9]. This means, for the request or what is sent, unexpected fields, query parameters, and values are not allowed. Matchers (refer to subsection 4.1.2) such as like() or eachLike() are less likely used in request verification. Moreover, as Pact is doing unit testing for the API fetch function, being lenient with the actual request breaks the purpose of unit testing. Therefore, while writing the expectations for the request, it is recommended not to use any Matchers so that the exact value in each field is checked to ensure the consumer does not send any uncontrolled data. There is one exception, however, for the strictness of the request. If the consumer sends something that is unpredictable but does not affect the functionalities of both the consumer and the provider such as a random ID, a Matcher is needed to allow flexibility to that ID field. However, if that ID field is controlled and affects the functionalities, it should be asserted as strictly as possible. The question of whether a Matcher is needed depends on the impact level of the value. Developers should consider this question carefully when writing CDC.

The response or what is received, on the other hand, allows more freedom on the structure and values. Freedom comes from two aspects:

1. The consumer only needs to consider the fields/data it needs so that its functionalities do not break. The consumer might need to partially use the real response payload. The test should check if the needed fields are available while ignoring the unused fields.

2. The content in a response field is not necessarily equal a particular value. Rather, CDC might consider only the type or format of the content. However, if the content from an endpoint is always expected to be constant, it should be asserted with exact value matching.

Point 1. allows the provider to serve multiple consumers with different expectations. For example, if the full response from the provider is:

```
[
  {
    "name": "Jyri",
    "age": 23,
    "address": [
      {
        "street": "Orivedenkatu",
        "number": "8D"
      }
    ]
  }
]
```

and a consumer A is only interested in the field ”name” and ”age”, it can and should ignore the field ”address.” Another consumer B might be interested in the field ”address,” so the provider still has to return that field. In this example, the provider can serve both consumers A and B without having to customize the API for each consumer.

Point 2. means the response values should be treated as loosely as possible regardless of the request. This means if in the contract the request to fetch data for a person named Sam is registered, the response should not expect exact returned data for Sam. Rather, the contract should care about the type of values of the response fields in general which can fit any entry that the request is asking for. Checking that the provider correctly responds with a particular value based on the requested data is not the mission of CDC. More details on the reason for this point will be discussed at the end of subsection 4.2.4. An important note is that random data should be avoided as Pact Broker will compare the hash of the previous contract with the current contract to re-verify the contract with the provider. Random data will cause the contract to be re-verified unnecessarily [30].

Some exceptions to Postel's Law in CDC still exist. Headers should be asserted strictly both in the request, and the response as this might remarkably affect the functionalities of the APIs. Moreover, sometimes a field's value is expected to be a constant even when it is a response field. In this case, there is no other way than to assert an exact value in the response of the contract.

4.2.3 Ensure the scope of testing

Coverage is a useful metric to determine the quality of testing in a project [4]. However, there is no official definition of the coverage for contract testing. As a consumer-driven method, CDC aims for testing whatever potentially breaks the consumer's functionalities. In subsection 4.2.1, it is discussed that CDC test cases should cover the functionality of API fetch functions similar to doing that with the unit testing framework [36]. Therefore, CDC first should fulfill the coverage of the fetch functions in the consumer, i.e. how the request is correctly created and if the response is handled normally. Furthermore, as CDC aims to test the interactions between services, it is important to have test cases for all the direct provider services as well as all the used endpoints. The UI in the case project has 12 direct providers and a total of 22 used endpoints. Therefore, at the time of writing, a starting criterion for good coverage is all of those 22 endpoints are covered by CDC.

Response status code, however, is a complicated topic and there is no strict guideline on whether all status codes should be tested. Whether a status code case should be tested depends on the convention used in the service or the project. Some APIs in the case project return code 200 with an error message to indicate there are some problems, while the normal REST convention is to use 4xx or 5xx codes [2]. Generally, all the status codes that the provider may respond to during the normal operation should be tested. If an error code (4xx or 5xx) is returned and the underlying system is working incorrectly, a contract does not help alleviate the problem. For instance, if code 401 Unauthorized is returned, the consumer does not have enough permission to get the data, so CDC does not help too much in this case. If the underlying system is malfunctioning or the credentials expire, the consumer should be unusable until further fixes or re-authorization. This is particularly applicable to the case project as the user is kicked out if the credentials are no longer valid, and the operational APIs do not return 401. Therefore, it is not meaningful to test code 401 in the case project. However, if a service responds with an error code due to bad input from the consumer, a contract test is required to guard against such a scenario. For example, a service in the case project expects to receive a formula with criteria and logical operators, e.g. intersection, union, and except. This service would return 500 if the formula is malformatted, such as ”criterion1 &” (& = intersection). A contract test for this situation is beneficial. In summary, a good endpoint coverage for CDC is when all endpoints and all meaningful status codes of the endpoints are tested.

It is important not to go too far away from the purpose of CDC when testing the status code. CDC targets the interface of the interaction or the status code in this discussion. Taking the service which requires a formula in the previous paragraph as an example, there might be various reasons for the formula to be malformatted. They can be redundant criteria, redundant logic operators, or bad criterion identifiers. There is no need to write CDC for all the mentioned error cases as the provider’s functional tests should be responsible for covering such cases. The provider would respond with 500 anyway, and that is all the consumer concerns. The Pact documentation states that the appropriate way to go is to write test cases for how the errors occur, not why the errors happen [19].

4.2.4 Test only the interface of interactions

CDC means to test only the interface of the interactions [37]. The interface is limited to not only the payload interface but also the query parameters and headers. Possible criteria that CDC should assert with the response:

1. The headers are exactly matched.
2. The fields’ names are correct. This applies to the nested objects as well.
3. The contents inside each field have the correct value, type, or format.
4. If the content inside a field is an array, the number of elements (none or some) should be considered. The point 2. and 3. are also applicable to each element in the array.

The structure of the payload and the shape of the value in each field are the top concern. Usually, CDC should not expect a value to be exactly similar to the expectation unless that value is known to be constant and does not vary with different requests. If the value in a response field varies depending on the request information, asserting that value with the exact expected value is not a good idea. Being strict with the response might lead to some possible problems. Firstly, the test is very quite fragile during the provider test phase. A slight change in the mock data of the provider when verifying the contract can cause the test to fail. This makes the maintenance of the tests cumbersome while there is no value gained in real life. Secondly, the test focuses too much on the implementation of the provider, i.e. how it generates the exact payload. This is out of the scope of CDC. Furthermore, it also makes the provider harder to evolve in the future [10]. Lastly, developers might be tempted to write redundant tests with exact responses to cover more cases. This makes CDC heavier and harder to maintain in the long run as there are more expectations in the consumer and more states in the provider. No interface mismatch occurs in real life if the provider responds with different data in the same payload shape. Using many strict comparisons in the contract might allow spaces for false-negative errors in the product [10].

In the case project, there is an existing CDC test that violates the discussed practice. The response expectation is registered to Pact like this:

```
willRespondWith: {
    status: 200,
    headers: {
        "Content-Type": "application/json"
    },
    body: {
        result: [
            {
                moId: "MRBTS-1",
                moClass: {
                    id: "MRBTS",
                    version: "<some version>"
                }
            }
        ]
    }
}
```

There is no Matcher used, which indicates that the provider must respond with both the correct field name and the value. If the provider responds with `moId: "not-MRBTS-1"`, the test would fail. This is not usually the expected outcome when doing CDC. In this case, the provider can respond with anything depending on the data currently persisted in the database. Writing the expectations like this test limits the benefits of CDC and might violate the testing scope of other testing methods.

The last thing to note is that it is not always the case that every element in the array shares exactly the same structure or some common parts. If the payload structure cannot be controlled, Pact might not be a suitable tool. Moreover, it might not be a good API design if the payload is always unexpected.

4.2.5 Keep the test case minimal

As the consumer test is backed by the unit test framework of the consumer service [32], it should be minimal for better resilience in the future [39]. Usually, one test case covers one interaction between the consumer and the provider. One test case should also contain only one API function call. An interaction contains at least the expectation for the request (except for GET) and the response, and a description of the interaction. The provider state is optional as some backend services do not define any state. However, if the backend service contains a setup with different states, it is important to communicate with the backend teammates to know about the defined states, e.g. state identifier and mock data.

4.2.6 Write the test before API related changes are merged

The most ideal time to write the CDC is right after the features have been developed but not yet merged into the master branch. Of course, CDC should be written only when the new features or changes are related to the APIs that CDC aims to test. The APIs here can be the introduction of new fetch endpoints or the changes in the existing API fetch functions. Writing the test at this phase encourages developers to have sufficient testing to catch and fix errors early [25]. It is completely possible if the team wants to write contract tests after everything from the consumer and the provider has been merged, but some hidden interface mismatching problems might exist, and things can crash before the contract is written. Writing the contract as soon as possible helps prevent such a situation.

All consumer pipelines which contain CDC in the project have to pass the provider verification (subsection 2.2.4.B). This stage is done by Pact's can-i-deploy tool. This means the consumer needs to wait for the provider to merge new changes before the consumer can merge its change with CDC. At this phase, communication across people and teams is crucial to ensure that can-i-deploy tool passes. Agreed interaction interface needed to be communicated well between the consumer team and the provider team. Otherwise, contract testing will fail, and efforts are required to ensure the consumer's pipeline turns green. Waiting for the provider's changes to come to production is a blocker for the consumer, but this is still better than accidentally merging problematic changes.

Contract testing in the project is not strict about "consumer's driven." It is defined by the consumer and contains the expectations of the consumer, but the process of testing should involve both the consumer and the provider. Merging a consumer with breaking changes into microservices system would inevitably cause severe crashes. The provider has at least the API versioning to prevent breaking changes in the consumer. Therefore, even though the testing is called "consumer's driven," when new features are developed, and the contracts are written for the first time, the provider should merge first and provide sufficient setup, such as the states, for the consumer to pass CDC and merge the changes. After the consumer merges the changes, the contracts are in a good state.

4.2.7 Maintain a good communication

Communication is key in software development for a successful product regardless of how much testing is used [24, 27]. Once the project's size grows up, communication is not just about discussing within a team, but rather across teams. No tools can help achieve good results without collaboration between people developing the product. For instance, the scenario in the first paragraph of subsection 4.2.6 could not be completed smoothly without good communication. Therefore, either CDC or any other activity, communication should always be kept at the highest priority in order to achieve remarkable things [11].

4.3 Updating the contracts

The section concerns different cases when the contract needs to change. The practices in the section relate to both the consumer and the provider. Although the thesis focuses on the consumer side, collaborating with the provider people contributes greatly to the success of CDC in the consumer.

In the project, a change first stays in the development branch and then gets merged. The change exists in the master branch after the merge, but it is not yet in the production environment. A change can go to the production once the system E2E test passes. The section is written based on an assumption that can-i-deploy tool for a change in the development branch or the master branch (but not yet in production) is tested with the counterpart service in production. This should be the correct way of testing.

4.3.1 Removing a field in both the consumer and the provider

When both the consumer and the provider want to remove a field from the payload, the consumer should be available in the production before the provider. If the field is removed from the contract, the consumer's can-i-deploy tool should still pass with the provider's production version because of Postel's Law. However, when the provider makes the same change to the response payload, its pipeline fails. This is because the pipeline runs a provider test with all consumers in production, and those consumers still expect the field to exist. Therefore, the provider's changes should stay in the branch until the consumer's changes go to production. After that, the provider's changes can be merged and go to production. One downside of this workflow is that the consumer is blocking the provider from progressing as the way from master to production can be very long on ”rainy” days. However, this could be the best way for the project's way of work. Communication would be the key to make everything exists in production as soon as possible.

4.3.2 Adding a field in both the consumer and the provider

Unlike the previous case, in this scenario, the provider should go to production before the consumer. Postel's Law indicates that when a new field is added to the provider response payload, the consumer's contract should still be fulfilled. The consumer's pipeline is blocked as the provider in production does not have the new field yet, and thus can-i-deploy tool fails. The consumer's changes should stay in the branch until the new provider exists in production. Again, communication would play a crucial part in the successful development and testing.

4.4 Tooling limitation and future development

CDC with Pact has a lot of room for improvement in the project. This is not limited to the tool configurations or the tool choices. One major dilemma with Pact is that it does not suitable for testing the pass-through services. In the project or microservices world, pass-through services are used for encapsulating the logic and for easier future refactoring. It is not an easy task to use Pact to test pass-through services [34] as a test involves only two parties: one consumer and one provider. If a pass-through service just merely forwards part of the consumer's payload to other services, there exists a testing gap here. CDC cannot catch the mismatch between the consumer's payload interface with the end service expectations. This is because the consumer does not know anything about the end services, but only the pass-through service. This topic is in the discussion at the moment of writing, but there is not yet a concrete solution to tackle the problem. There are testing gaps in the case project where Pact cannot help catch errors related to interface mismatch after going through a pass-through service. Perhaps one proposal is to design the pass-through service so that it transforms the payload from the consumer somehow to match the end services' expectations. By doing this, the consumer no longer needs to consider the interface expectation from the end services. The responsibilities now go to the pass-through service to ensure the data flow from the consumer until the end services as smoothly as possible by doing the payload transformation correctly.

There are some limitations regarding the Pact specification version 2. One of the limitations is that no dynamic key or key matching exists. In real cases, the metadata service of the project returns the response in this way (the data in the example has been modified to follow the regulation):

```
{
    "com.org.class.MRBTS": ["ADAPTATION_1"],
    "com.network.class.NRCELL": ["ADAPTATION_2", "ADAPTATION_3"]
}
```

The UI wants to use both the keys and the values for its operation. The keys are actual data from the database. They are unexpected and should not be included in the test. Pact specification version 3 introduces new ways to solve this situation.

Another limitation is that Matchers for Pact JS generate incorrect JSON expression paths for the matching rules. If the key of an object contains some invalid identifier such as a slash, the JSON path would look like this:

```
"$.body.MY-SITE/MRBTS-1[*]"
```

The provider written in Java does not allow a slash to exist in the path. The correct behavior would be the square bracket expression `["MY-SITE/MRBTS-1"]` instead. However, if the key contains words separated by dots, Pact generates the path correctly.

Tool configuration in the project hinders some problems as well. Can-i-deploy and webhook configurations for some services are not in an ideal state. Firstly, if a new pipeline job runs without any change in the contract, can-i-deploy tool fails as it cannot find the last verified contract. Secondly, the tags for the providers are not in harmony. The best way is to run can-i-deploy tool against the provider in production when the consumer's pipeline runs. This means the provider with the tag ”production” should be used instead of ”dev” or ”candidate.” This topic does not relate to the focus of the thesis, but it can be an obstacle for someone new to CDC. The tool configuration should be in place before any further testing activity continues.

5 Conclusions

In this thesis, two results regarding the consumer side of the contract testing are presented. The results consist of (i) a simple test writing walkthrough with the structure explanation and important objectives of writing the tests, and (ii) the workflow when the contract needs to change. The results are based on three sources: existing flaky tests in the product, related literature, and the nature of CDC. The set of practices and common workflow is crucial for a rapidly growing project to have a systematic way of working, which then helps ensure a steady quality throughout the development of the product.

Consumer-driven contract testing should be utilized correctly to have an efficient guard against breaking interface changes. The test should focus only on the interface of the interaction between two services and should be written right after the changes are done in the development branch. There are also rules for asserting the request and response payload interface which help the system to scale more efficiently. Lastly, it is never enough to emphasize the importance of communication in any software development activity, and testing is not an exception. No tool could completely replace human interaction for a great outcome.

After the thesis, part of the API router tests in the router server will be replaced by the contract tests. The request creation and response handling will be harmonized into a single place which is the contract test. The implementation is conducted in only one UI service, and it will be done for the other UI services later on. The replacement moves the project into a state-of-the-art way of testing microservices harmonically. Although some limitations still exist somewhere in the project, they should be mitigated to ensure a complete harmony in the way of doing Consumer-driven contract testing. Tooling like can-i-deploy, webhook, and tagging activities require a systematic refactor to make things more stable and reliable. A newer version of Pact should also be used once it is available to provide developers more options to test more efficiently and flexibly. Lastly, more studies need to be organized to expand the practices to end-to-end, i.e. the practices for the provider as well.

References

[1] Harsh Bhasin, Esha Khanna, and Sudha. “Black Box Testing based on Requirement Analysis and Design Specifications”. In: *International Journal of Computer Applications* 87.18 (2014), pp. 36–40. ISSN: 0975 – 8887. URL: https://www.researchgate.net/profile/Esha-Khanna/publication/262992890_Black_Box_Testing_based_on_Requirement_Analysis_and_Design_Specifications/links/5821dde708ae5385869ff13a/Black-Box-Testing-based-on-Requirement-Analysis-and-Design-Specifications.pdf.

[2] MDN contributors. *HTTP response status codes.* Feb. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status (visited on 03/31/2022).

[3] Vahid Garousi and Junji Zhi. “A survey of software testing practices in Canada”. In: *Journal of Systems and Software* 86.5 (2013), pp. 1354–1376. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2012.12.051. URL: https://www.sciencedirect.com/science/article/pii/S0164121212003561.

[4] J.R. Horgan, S. London, and M.R. Lyu. “Achieving software quality with testing coverage measures”. In: *Computer* 27.9 (1994), pp. 60–69. DOI: 10.1109/2.312032.

[5] IBM. *What is software testing?* URL: https://www.ibm.com/topics/software-testing (visited on 04/25/2022).

[6] Irena Jovanović. “Software testing methods and techniques”. In: *The IPSI BgD Transactions on Internet Research* 30 (2006).

[7] Mohd Ehmer Khan. “Different forms of software testing techniques for finding errors”. In: *International Journal of Computer Science Issues (IJCSI)* 7.3 (2010), p. 24.

[8] Mohd. Ehmer Khan and Farmeena Khan. “A Comparative Study of White Box, Black Box and Grey Box Testing Techniques”. In: *(IJACSA) International Journal of Advanced Computer Science and Applications* 3.6 (2012), pp. 12–15. ISSN: 2156-5570. DOI: 10.14569/issn.2156-5570. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.685.1887&rep=rep1&type=pdf#page=22.

[9] Charles M. Kozierok. *The TCP/IP-Guide: A comprehensive, illustrated internet protocols reference.* No Starch Press, 2009.

[10] Jyri Lehvä. *Testing Integrations with Consumer-Driven Contract Tests.* eng. 2019. URL: URN:NBN:fi:hulib-201908133204;%20http://hdl.handle.net/10138/304680.

[11] Jyri Lehvä, Niko Mäkitalo, and Tommi Mikkonen. "Consumer-Driven Contract Tests for Microservices: A Case Study". In: *Product-Focused Software Process Improvement.* Ed. by Xavier Franch, Tomi Männistö, and Silverio Martínez-Fernández. Cham: Springer International Publishing, 2019. ISBN: 978-3-030-35333-9.

[12] James Lewis and Martin Fowler. *Microservices a definition of this new architectural term.* Mar. 2014. URL: `https://martinfowler.com/articles/microservices.html` (visited on 09/29/2021).

[13] Lu Luo. "Software testing techniques". In: *Institute for software research international Carnegie mellon university Pittsburgh, PA* 15232.1-19 (2001), p. 19.

[14] Qingzhou Luo et al. "An Empirical Analysis of Flaky Tests". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014. ISBN: 9781450330565. DOI: `10.1145/2635868.2635920`. URL: `https://doi.org/10.1145/2635868.2635920`.

[15] Y.K. Malaiya. "Antirandom testing: getting the most out of black-box testing". In: *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95.* 1995, pp. 86–95. DOI: `10.1109/ISSRE.1995.497647`.

[16] B. Meyer. "Applying 'design by contract'". In: *Computer* 25.10 (1992), pp. 40–51. DOI: `10.1109/2.161279`.

[17] Elliot Murray. *Dockerized Pact Broker.* Aug. 2020. URL: `https://docs.pact.io/pact_broker/docker_images` (visited on 10/19/2021).

[18] Elliot Murray. *Gotchas.* Aug. 2020. URL: `https://docs.pact.io/getting_started/matching/gotchas` (visited on 03/29/2022).

[19] Elliot Murray and joklek. *Contract Tests vs Functional Tests.* Jan. 2021. URL: `https://docs.pact.io/consumer/contract_tests_not_functional_tests` (visited on 03/31/2022).

[20] Florian Nagel and Martin Leucker. "Analysis of Consumer-driven contract tests with asynchronous communication between microservices". In: (Dec. 2019). URL: `https://www.isp.uni-luebeck.de/sites/default/files/01_thesis_NagelFlorian.pdf`.

[21] Srinivas Nidhra and Jagruthi Dondeti. "Black box and white box testing techniques-a literature review". In: *International Journal of Embedded Systems and Applications (IJESA)* 2.2 (2012), pp. 29–50.

[22] Charlene O'Hanlon. "A Conversation with Werner Vogels: Learning from the Amazon Technology Platform: Many Think of Amazon as 'That Hugely Successful Online Bookstore.' You Would Expect Amazon CTO Werner Vogels to Embrace This Distinction, but in Fact It Causes Him Some Concern." In: *Queue* 4.4 (May 2006), pp. 14–22. ISSN: 1542-7730. DOI: 10.1145/1142055.1142065. URL: https://doi.org/10.1145/1142055.1142065.

[23] C. Pautasso et al. "Microservices in Practice, Part 1: Reality Check and Service Design". In: *IEEE Software* 34.01 (Jan. 2017), pp. 91–98. ISSN: 1937-4194. DOI: 10.1109/MS.2017.24.

[24] G. Purna Sudhakar. "A model of critical success factors for software projects". In: *Journal of Enterprise Information Management* 25.6 (2012), pp. 537–558. ISSN: 1741-0398. DOI: https://doi.org/10.1108/17410391211272829. URL: https://www.sciencedirect.com/science/article/pii/S0164121212003561.

[25] SMK Quadri and Sheikh Umar Farooq. "Software testing–goals, principles, and limitations". In: *International Journal of Computer Applications* 6.9 (2010), p. 1.

[26] Chris Richardson. *Pattern: Decompose by business capability.* URL: https://microservices.io/patterns/decomposition/decompose-by-business-capability.html (visited on 10/02/2021).

[27] Ahmad Salman et al. "An Empirical Investigation of the Impact of the Communication and Employee Motivation on the Project Success Using Agile Framework and Its Effect on the Software Development Business". In: *Business Perspectives and Research* 9.1 (2021), pp. 46–61. DOI: 10.1177/2278533720902915. eprint: https://doi.org/10.1177/2278533720902915. URL: https://doi.org/10.1177/2278533720902915.

[28] Abhijit A Sawant, Pranit H Bari, and PM Chawan. "Software testing techniques and strategies". In: *International Journal of Engineering Research and Applications (IJERA)* 2.3 (2012), pp. 980–986.

[29] Dharmendra Shadija, Mo Rezai, and Richard Hill. "Towards an understanding of microservices". In: *2017 23rd International Conference on Automation and Computing (ICAC)*. 2017, pp. 1–6. DOI: 10.23919/IConAC.2017.8082018.

[30] Beth Skurrie. *Matching.* Aug. 2020. URL: https://docs.pact.io/getting_started/matching (visited on 04/01/2022).

[31] Beth Skurrie, Matt Fellows, and Elliot Murray. *Getting started.* Mar. 2021. URL: https://docs.pact.io/ (visited on 10/13/2021).

[32] Beth Skurrie, Matt Fellows, and Elliot Murray. *How Pact works*. May 2021. URL: https://docs.pact.io/getting_started/how_pact_works (visited on 10/14/2021).

[33] Beth Skurrie, Matt Fellows, and Elliot Murray. *Webhooks*. Oct. 2021. URL: https://docs.pact.io/pact_broker/webhooks (visited on 03/24/2022).

[34] Beth Skurrie and Elliot Murray. *When to use Pact*. July 2020. URL: https://docs.pact.io/getting_started/what_is_pact_good_for (visited on 02/16/2022).

[35] Beth Skurrie, Elliot Murray, and obinna240. *Can I Deploy*. Nov. 2021. URL: https://docs.pact.io/pact_broker/can_i_deploy (visited on 03/24/2022).

[36] Beth Skurrie et al. *FAQ*. Mar. 2022. URL: https://docs.pact.io/faq (visited on 03/31/2022).

[37] Beth Skurrie et al. *Writing Consumer tests*. Jan. 2021. URL: https://docs.pact.io/consumer (visited on 03/31/2022).

[38] Johannes Thönes. "Microservices". In: *IEEE Software* 32.1 (2015), pp. 113–116. DOI: 10.1109/MS.2015.11.

[39] *Unit testing best practices with .NET Core and .NET Standard*. Nov. 2021. URL: https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices (visited on 03/31/2022).

[40] Markos Viggiato et al. *Microservices in Practice: A Survey Study*. 2018. arXiv: 1808.04836 [cs.SE].

[41] Ham Vocke. *The Practical Test Pyramid*. Feb. 2018. URL: https://martinfowler.com/articles/practical-test-pyramid.html (visited on 10/13/2021).

A Example CDC

```
pactWith(
  {
    consumer: "object-list-widget-rest",
    provider: "workingset-manager-rest"
  },
  provider => {
    let server;
    beforeAll(async () => {
      process.env.WS_MANAGER_URL = provider.mockService.baseUrl;
      const { app, config } = require("../server/src/app");
      await new Promise(resolve => {
        server = app.listen(config.port, () => {
          resolve();
        });
      });
    });

    it("Fetch list of saved private working sets", async () => {
      await provider.addInteraction({
        state: "Resolve Working Set 'normalWS', which has objects",
        uponReceiving: "GET a list saved private working sets",
        withRequest: {
          method: "GET",
          path: "/v1/workingsets",
          headers: {
            "Content-Type": "application/json"
          },
          query: {
            private: "true"
          }
        },
        willRespondWith: {
          status: 200,
          headers: {
            "Content-Type": "application/json"
          },
          body: eachLike({
            admin: false,
            id: "ws-1",
            name: "My working set",
            formula: "c",
            criteria: eachLike({
                id: "c",
                type: "some-type",
                descriptor: like({})
            }),
            labels: {
                dataType: "integratedId"
            }
          })
        }
      });

      await listPrivateWorkingset().then(res => {
        expect(res).toBeInstanceOf(Array);

        res.forEach(r => {
          expect(r).toMatchObject({
            admin: expect.any(Boolean),
            id: expect.any(String),
            name: expect.any(String),
            formula: expect.any(String),
            criteria: expect.arrayContaining([
                expect.objectContaining({
                id: expect.any(String),
                type: expect.any(String),
                descriptor: expect.any(Object)
                })
            ])
          });
        });
      });
    });

    afterAll(async () => {
      await new Promise(resolve => {
        server.close(() => {
          resolve();
        });
      });
    });
  }
);
```

Program A.1 *Example test case*