UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR SOFTWARETECHNIK
UND PROGRAMMIERSPRACHEN

Analysis of Consumer-driven contract tests with asynchronous communication between microservices

Analyse von Verbraucher-gesteuertem Vertragstesten mit asynchroner Kommunikation zwischen Microservices

Bachelorarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Florian Nagel

ausgegeben und betreut von
Prof. Dr. Martin Leucker

mit Unterstützung von
Kathrin Potzahr, Dr. rer. nat. Annette Stümpel

Die Arbeit ist mit Unterstützung der Firma Capgemini Deutschland GmbH entstanden.

Lübeck, den 04.12.2019

Deklaration

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Quellen und Hilfsmittel angefertigt zu haben.

(Florian Nagel)
Lübeck, den 04.12.2019

Abstract In the context of microservice architecture the speed of the delivery of microservices is one of the most important parts in the release cycle. To be able to deliver microservices fast and reliably the continuous integration (CI) and continuous deployment (CD) pipeline have to be efficient, of good quality and optimized. This way we can ensure the maximum speed of reliable updates. To get the maximum out of the most beneficial aspect of microservices, the fast release cycle, we need to have fast testing. With the introduction of microservices in an architecture the amount of interfaces that can be reached is significantly higher. Most, if not all, of the communication between microservices rely on these interfaces. This proves to be a new problem in testing. These interfaces have to be tested reliably and quick.

Consumer-driven contract (CDC) tests apply exactly at these interfaces between microservices without needing as much time as end-to-end tests. One of the reasons consumer-driven contract testing is deployed in testing pipelines in the context of microservice structure is their fast execution with reliable enough results to catch many errors. In the best case, before the end-to-end tests are even run. This allows the testing department to run less end-to-end tests and reduce the intensity and necessity of end-to-end tests. CDC testing is well worked out in synchronous messaging architectures even if it still lacks popularity. It is however not as seemingly integrated in asynchronous communication as it is in its synchronous counterpart.

There are certain difficulties when it comes to asynchronous CDC testing. One is the lack of tools. This does not propose a big problem because one of the currently used tools for these tasks (Pact) also supports asynchronous messaging queues. As mentioned earlier it does lack a fluent integration in those. The different advantages and drawbacks of using Pact over other CDC tools are examined.

This thesis explores aforementioned tool and the possibilities of integrating this tool into the testing realm of asynchronously communicating microservices.

Kurzfassung Im Kontext einer Microservice-Architektur stellt die Geschwindigkeit der Auslieferung des Service' mit den wichtigsten Aspekt dar. Um dies bestmöglich zu gewährleisten, müssen die „continuous integration“ und „continuous deployment“ Abläufe so schnell und optimiert wie möglich sein. Die Phase, die am meisten Zeit benötigt, ist die Phase des Testens. Um also das Maximum aus einem der größten Vorteile der Microservice-Architektur, dem schnellen Ausliefern eines Service, zu erreichen, muss schnelles Testen etabliert werden.

Eine der Gründe warum „consumer-driven contract“ (CDC) Testen in vielen Entwicklungsabläufen integriert ist, ist die schnelle Ausführung dieser Tests und deren, zumindest zu einem bestimmten Grad, verlässlichen Ergebnisse. So können sie viele Fehler finden, bevor die teuren end-to-end Tests ausgeführt werden müssen. CDC Testen funktioniert sehr gut in einem synchron kommunizierenden Netzwerk von Microservices, auch wenn es dem CDC Testing immer noch an Popularität mangelt. Allerdings sind diese im asynchronen Kontext nicht so gut implementiert wie in dem synchronen Gegenstück.

Es gibt einige Schwierigkeiten bei asynchronen CDC Tests. Eine ist der Mangel an Hilfsmitteln. Dies ist jedoch vernachlässigbar, da eines der am meisten genutzten Programme (Pact) auch asynchrone Kommunikationsprotokolle unterstützt. Es fehlt allerdings an einer nahtlosen Integration. Wie dieses Tool im Vergleich zu anderen arbeitet, wird in der Arbeit untersucht.

Diese Arbeit erforscht zuvor benanntes Tool und die Möglichkeiten, die sich bei der Nutzung dieses erschließen, wenn man es in dem Kontext der asynchronen Kommunikation nutzen möchte.

Contents

1 Introduction

With the industry shifting more and more from monolithic architectures to a microservice architecture [CDT18], it gets more important to ensure that the change in architecture benefits the software and in regards to a microservice architecture the whole release cycle. The speed of this release cycle as well as a reliable deployment is very important. For this we need to use the correct tests. One testing method that has come up with the use of microservices is consumer-driven contract (CDC) testing. To use this testing method we also need the correct testing tools. Microservice deployment is all about continuous integration (CI). CI stresses the factor of time even more as in classical software deployment. The goal is to provide fast and reliable testing pipelines to be able to release software often and regularly.

Most of the time the services act as a provider, *providing* information, or as a consumer, processing data that was provided. The central point of communication between consumer and provider can either be a REST interface or asynchronous communication through message queues like Kafka. The two ways to handle this while doing integration testing was to either do no integration testing at all or to fire up the whole system. I.e. running provider and consumer, to make sure that they communicate correctly with each other. This is a very reliable way of testing, but takes a lot of time.

This is where consumer-driven contract (CDC) testing comes into play. It provides a quick and reliable way of checking whether or not a provider service fulfils all of the specifications for the communication, set by the consumer services. And that even before running end-to-end tests. On consumer side it gives you the ability to tell the provider which features are being using. This gives the provider the ability to check if it is introducing breaking changes to the communication without starting up every dependent system. With this methodology it takes less time to run integration tests while still keeping most of the reliability and confidence of the classical tests.

While this is mostly implemented for synchronous REST communication between microservices, there is still a lot to do to get asynchronous CDCs to the same level. The current state of CDCs with asynchronous communication between microservices is analysed in this thesis. Whilst providing theoretical insight to the testing mechanics itself, it is also investigated what the current tools provide to make CDC testing possible for asynchronous communication. Further there are suggestions made for the further development of these tools to enable software engineers to use

them as easily for asynchronous communication as they can be used for synchronous communication.

The importance of these tests and how they perform in a case study can be found in [LMM19] as a very recent example taken from the international conference on Product-Focused Software Process Improvement. Further work on consumer-driven contract tests include the creation of a framework for consumer-driven contract testing of Java APIs [Sel18] by a Swedish student.

After reading this thesis the following questions should be answered and it should be how these conclusions were reached.

This thesis aims to answer five main questions:

1. What is CDC testing and why do we need it in the context of microservice architecture?
2. What are the differences between synchronous and asynchronous CDC testing?
3. What tools are there to provide CDC testing capabilities for asynchronous communication between microservices?
4. Why do we use the Pact tool to provide CDC testing?
5. How can we use this tool for CDC testing in an asynchronous environment and how does it differ from being used with synchronous communication between microservices?

The thesis is structured in the following way: After this introduction there is the preliminaries chapter to provide the reader with the needed background information regarding knowledge on topics that are used in this thesis but not specifically described in detail. The next chapter is dedicated to the tools used in this thesis. This includes the example project "Tippkick" and the CDC testing tool "Pact". After this a chapter about consumer-driven contract tests gives the reader a more in-depth explanation on how consumer-driven contract tests work and what functionality and benefits they provide in a software deployment cycle. The thesis continues with observations made when comparing the implementation in a synchronous environment with the implementation in an asynchronous environment. The next chapter is dedicated to analyse these observations and to give some insight on how one could go about enhancing the implementation for asynchronous environments to meet the same level of fluid integration as the synchronous counterpart. Finally there is a chapter which summarizes found results in a concise manner.

2 Preliminaries

This chapter aims to explain some of the commonly used terms in connection to this thesis' topic. It will enable the reader to comprehend the terms, programs and approaches used in this thesis and will further provide a short introduction. At the end of this chapter the reader should have an idea in which environment we are using CDC testing.

At the start we will be introducing microservices because they are the architecture we are using CDC testing on. After that we will define the term "Monolith" and how it compares to a microservice architecture. Following that will be a section about why agile development fits into the idea of microservices and why it makes sense to develop microservices in an agile process. The final section introduces the different ways to test, so the reader has an idea of where CDC testing is to be categorized in a listing of testing stages in a so called testing pyramid.

2.1 Microservices

Microservices are small, autonomous services that work together in a network. They usually are each self-contained and provide a single business capability. [New15]

Example

Before explaining the parts of this definition in detail, the following paragraph provides an overview about microservices with a practical example.

Imagine a webpage that is hosted by an online shop. Various microservices could be used for the fields that are displayed. One microservice could provide information when a certain product is available, another one could handle the price display and another microservice could be providing the information for the product in the form of pictures for example.

Small

”A microservice should be small and focused on doing one thing well” [New15]. This means they are easily maintainable by a single small team because of their size and their scope of functionality. A famous quote by Amazon incorporates this by saying that a team managing a microservice should be no larger than a dozen people and be able to be fed by two pizzas, as stated by Amazon’s notion of the Two Pizza Team [JL14] This is one of the larger sized teams. If we observe the other size of the spectrum there are instances where half-a-dozen people support half-a-dozen microservices. [JL14]

Autonomous

Microservices should be as loosely coupled as possible and have as little dependencies as possible on other services to provide the functionality specified. This includes having their own database to work with, but is not limited to.[1] That does not mean that there is not a microservice in reality that is largely dependant on another service, just that it should not be when designing the ”perfect” pattern for a microservice architecture. Having them loosely coupled enables us to deploy updates to a service without making changes to another service. [New15]

Architecture

While it should now be clear what a single microservice is doing and how it is assembled, we should also establish how the microservice *architecture* works. Microservices usually have the role of either a consumer or a provider. A service either consumes another service and processes that information or it provides information to another service to consume. The communication can either be implemented via the REST-protocol or by asynchronous means of messaging. The synchronous approach is fairly easy to set up and makes for a communication architecture that is easily maintainable. It also has a few drawbacks to its asynchronous counterpart made clear in later chapters. [Rop16] To see what a microservice architecture can look like refer to figure 4.1 on page 23.

[1] This is just a recommendation made by the author. This does not mean that you can not deviate from this.

2.2 Monoliths and Microservices side by side

Monoliths

One refers to a software structure or program as a monolith when it is a single entity program. The program usually has a centralized database that it depends on and the different parts of the program are connected in some way that doesn't allow them to run independently. With a monolithic architecture, a small change to one part of the software can mean that the whole system has to be tested and redeployed. Consider an example: Switching to a different database framework. This directly affects a lot of the other code. The interchangeability of frameworks and libraries is fairly low because it always impacts the whole system. Furthermore you have to deploy the whole monolithic architecture when making changes to it. This includes long testing times because the whole architecture has to be iterated through. Long release cycles are another drawback of this architecture.[New15, pg. 141]

Side by side

When deciding whether to use a monolithic architecture or a microservice-based architecture, there are a lot of factors to be taken into account. This is a quick overview and does not by any means suffice for a complete decision-making process on which architecture to choose.

Property	Monolithic architecture	Microservice architecture
Deployment time	Long	Short
Frequency of updates	Low	High
Quantity	One	Multiple
Initial Cost & Overhead	Low	Higher

Table 2.1: A comparison between microservices and monolithic architecture.

The following points can mostly be found in table 2.1. It shows an informal comparison between microservices and monolithic architecture to describe the main differences in contrast to each other. The first column states the property that is being investigated. The second column refers to a complete monolithic architecture. The third column refers to a single microservice. Both the second and third column use informal descriptors to give a vague idea of what to expect from the corresponding architecture.

While microservices offer more flexibility and interchangeability of modules later on, monoliths are quick and easy to setup because of their reduced complexity regarding their interfaces. The initial overhead is also way lower with monoliths than with microservices. This is important to consider when deciding on a software architecture for a system[2]. Monoliths provide a low frequency in updates and long release cycles because every time a component changes the whole system has to be tested and deployed. Microservices can be deployed independently and take much less time to be tested individually because of the smaller size of each service. Small changes to a service are quicker to be deployed and this provides the user with a higher frequency of updates. Microservices allow for higher flexibility in the choice of components, meaning less effort and cost to update a single service to a newer framework or to have it use a newer database system than it would in a monolithic environment. This means that, even if your software has grown to a size that is very hard to manage, you can still interchange modules and change services with ease. In a monolithic environment this is harder to manage.

[2]Sam Newman gives a good overview of what architecture might be best for you in his book "Building Microservices: designing fine-grained systems" [New15]

2.3 Agile Development, Connection to microservices

As we have established *what* architecture we can use we are now introducing *how* one could develop software. This section provides an introduction to agile development and why microservices are very well suited to be developed in an agile development cycle. This does not mean however that they *have* to be deployed agile, just that they *can* be.

What is agile development?

Agile development is a form of software development which is more responsive to customer needs as opposed to the waterfall model [O'R17]. The scrum method is a widely known example for an agile method. Agile development aims to have short cycle times, early decision-making and recognizes the process of software development as an evolutionary process where change to the product is inevitable and accepted. This means that the common mindset is that the requirements of the product can change regularly throughout the software lifecycle. Good communication and early, regular feedback are essential for this process. [O'R17]

Connection to microservices

Microservices are suited for agile development because of their independence from each other and their short deployment times. The, usually small, team that works on a microservice can further complement the agile development process. Now let us define a few of the central advantages of agile development:

Short cycle times are the case for most microservices, as each team has one service to maintain (which provides a small but sophisticated functionality), thus having a small codebase and a manageable testing pipeline. This results in short cycle times and quick deployment schedules.

Early decision-making also plays a central role in the development of microservices as the team has to know what functionality their microservice has.

The evolutionary process is helped by having assigned each microservice to a small team. Each microservice can be developed on their own (mostly) without the need to contact big project managers for each and every decision. The mostly autonomous workflow is also helping the short cycle times.

Continuous deployment is another important factor. It reduces the overhead of human interaction needed to deploy software. That means that every time a change is made and released a pipeline job will compile the software, run various tests and when this all passes the software will be deployed.

Microservices are a perfect fit for the world of agile development. Their small size gives developers the ability to evaluate their progress fast and to verify their services. Small teams provide the developers with the ability to enable new features and fixes fast without much communication overhead. Microservices embrace the nature of agile development.

2.4 Testing

Testing is a very important part of developing software. There are a lot of different ways to test a software. Testing a microservice architecture differs from a monolithic software in a few ways. The following paragraphs depict which approaches are used in general. It will be clear where the approaches differ between monoliths and microservice.

Types of tests

There are many different approaches and each company or development team has to choose which one works best for them. The testing stages from the testing pyramid displayed in figure 2.1 are listed in the following paragraphs. Each with a short explanation. This specific pyramid shows different levels of testing a microservice. From bottom to top it increases in complexity and and in the time each test takes to complete. The bottom bracket should have the most tests while the top bracket should contain the fewest. [Cle14] Testing pyramids are mostly found in the context of agile development.[Coh09]

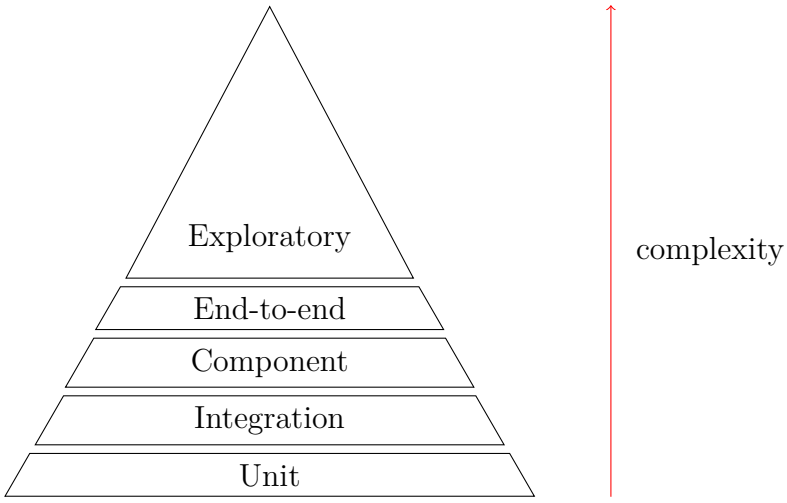

Figure 2.1: Testing pyramid according to a slide show by [Cle14]

As seen in the testing pyramid in figure 2.1 there are many different stages. It is important to note that the tests have different scopes. Unit tests for example only test inside a service, whereas everything up from component tests test between services, so the network of microservices. Further explanations to this can be found in the specific sections to the approaches below.

Unit tests

At the most primitive and basic level there are unit tests. Unit tests typically test functions of a class. They can be thought of as the most basic tests you could write. An example:

Imagine having a function that divides two numbers and returns the result. A unit test is written to test a function with the input parameters `4` and `0`. The test now compares the result and the expected result and fails if they mismatch. In this case the division should fail and give some form of error. This example is restricted to only one function and that function should not be dependant on any other as the test is as atomic as possible.

Integration tests

Integration tests aim to test the communication paths and interactions between components to test for interface defects [Cle14]. The goal is to test the behaviour of the module under test and not the entire subsystem. Furthermore we want to test the interaction between modules. The part of interest here is the interaction between the module and external applications like datastores and other external components. This is the main focus of integration tests. This includes gateway integration and persistence integration tests. [Cle14] Microservice architectures add a new layer of complexity to this form of testing, in comparison to a monolithic architecture, as they introduce more interfaces that need to be tested. The approach of CDC testing tries to make it easier to test these interfaces and to keep consistency between them. Contract tests are a part of integration testing. The difference is that it only regards the integration of the microservice to consumers of that service. This will be discussed in detail in chapter 3 on page 13.

Component tests

Component[3] tests limit the scope of the system under test, use internal code interfaces to manipulate the code interface and use test doubles to isolate the components from other dependencies. [Cle14] The microservice is instantiated with no connection to any network or other interface. Instead it will use in-memory test doubles and datasources. This leads to faster execution time and reduced complexity of the build. [Cle14] This test aims to test the integration of the microservice with the

[3] "A component is any well-encapsulated, coherent and independently replaceable part of a larger system [Cle14]."

other services. This is however not an integration test as they test the integration of the modules *in* a microservice.

End-to-end tests

An end-to-end test tests the complete system and verifies that it meets its requirements. [Cle14] These tests test the complete system and manipulate it through public interfaces such as GUIs and APIs. They take the longest to complete and are the most complex. (See Figure 2.1) As stated in [New15] the earlier mentioned contract tests could indeed replace end-to-end tests when deployed in a specific way[4]. This significantly speeds up the testing process and provides less brittle and flaky tests. [New15]

Exploratory tests

Exploratory tests refer to the manipulation of the software's components by hand. These tests refer to the testers using the software in ways that might have not been included in automatic tests. These tests can help to establish a better understanding of the system and its structure. [Cle14] This testing method is applied to microservices and monoliths alike as it is not coupled to any architecture.

[4]See [New15], page 144, second paragraph.

3 Consumer-driven contract testing

This chapter will provide an overview of the process that is labelled as ”consumer-driven contract testing”. After this chapter the reader has the understanding of how consumer-driven contract testing works and how it differs from a synchronous messaging environment to an asynchronous messaging environment.

First there is an explanation to the usage and the benefits of using CDC testing. The two sections after address the differences of CDC testing in a synchronous environment (REST) and in an asynchronous messaging environment (AMQP) respectively.

CDC testing in the context of microservices

A microservice architecture can have many connections between its microservices. Typically you have producers and consumers. I.e. you have a microservice offering a service and other microservices using/consuming[1] that service. In most cases every microservice has its own team working on it (See 2.1 on page 3). This means that the teams working on the communicating microservices can very easily miscommunicate what they are trying to achieve with their service and how it communicates. There has to be a set of rules for this or at least some form of agreement on how to communicate. This must be written down, as verbal agreements can easily lead to miscommunication. The teams may also forget to communicate when they change their interface in any way, that introduces breaking changes.

Imagine the following scenario: A provider of a person database can send a consumer, the first and last name of a person when it is requesting that item. The response[2] might look like the following:

```
{
    "name": "Max Mustermann"
}
```

[1] The protocol used in this exchange (i.e. REST or an asynchronous message queue) is not relevant for the current chapter and is therefore omitted. Refer to later chapters for an in-depth comparison.

[2] It is assumed that responses are encoded in the JSON format[ECM13]

The consumers expect exactly this response.

What happens if the producer changes the format of his messaging? Imagine the payload of the producer changing to the following:

```
{
    "firstName": "Max",
    "lastName":  "Mustermann"
}
```

The team behind the producer service might think that the new payload makes more sense or enables the consumers to work better with the provided information. The team might change the versioning behind their interactions. They may not realize that introducing these changes breaks the consumer side and many consumers may stop working. The usual methodology for detecting this is running end-to-end tests. These are very time-consuming and require all services to be up and running. The question is what can be done to minimize costs (e.g. time needed to ensure integration) and maximize profits (detecting breaking changes early)?

This is where the concept of consumer-driven contract tests comes into play. The idea is to let the consumers define what format parts of the produced message should look like. They only define these contracts for the parts of the exchange they use, they do not define the complete interaction or interface of the provider. The consumer creates a contract which states that he expects the following:

```
{
    "name": "Max Mustermann"
}
```

This resembles exactly the old payload described by the producer. This way the consumer wants to ensure that the producer is sticking to this format of the payload so it does not introduce breaking changes. The actual format of the generated CDC differs between tools. This is a fictional simplified version of one of those contracts. This contract is generated by the consumer and then exchanged with the provider. The provider receives these contracts from every dependant service to ensure that every dependency can work with the provider.

After these contracts have been exchanged, the provider tests his implementation against all the consumer contracts. This way the provider knows when he is introducing breaking changes for the consumers. He also knows when a feature is obsolete and is omittable from the produced message. This could be done to reduce overhead and increase performance.

Limitations; what CDCs are not suited for

CDCs are only used for validating an interaction between two services. *They should not be used to replace functional tests.* The only way they should be used for, is to resemble the frame in which the interaction takes place. So for example: Instead of testing for a specific name in the database

```
"name": "Max Mustermann"
```

The test should be done for a non-specific string value. This way we don't test for the specific values for a key-value pair, but instead test if the provider gives us the information in the format we need it. Whether or not the provider sends us the correct information is up to the provider's unit tests to determine. For example after this change the providers verification tests should pass with all of the following messages. Only listing a few and not every possibility:

```
"name": ""
```

```
"name": "Max"
```

The provider is not testing against the specific values of its payload, but instead against the format of the payload.

Important to note here, is while CDC testing does provide a great improvement in the testing timeline, it only has limited capabilities. We can't be certain if the provider returns the requested information. Only that he sticks to the format provided by the contract. This also applies to failed interactions or error handling. Those can and should be defined by a contract as well. E.g. does the producer send an error message or does it send a specific HTML code? This is important to remember when employing this type of testing in a testing pipeline.
CDC testing provides us with quick feedback so we catch mistakes early on, but it does not guarantee a fully working network of services and does not replace proper end-to-end tests (in the form we have introduced it). Full end-to-end tests should still be run before deploying software, to reduce the risk of failure. By using the CDC testing mechanic, the development team might not even have to deploy costly end-to-end tests to catch some errors. This way the development team can save time and resources.
Using CDC testing the intensity and quantity of end-to-end tests can be reduced whilst still keeping the same level of risk analysis. Therefore prevention of failure so software can be deployed faster and just as safe.

3.1 CDCs with *synchronous* communication

A widely used scenario for microservices is synchronous communication between services through an HTTP based REST interface. As the REST protocol is commonly used for APIs, it is a good way to start out when using microservices. The following paragraphs will discuss how the synchronous exchange between microservices can look like. Figure 3.1 shows an example of an exchange between consumer and provider.

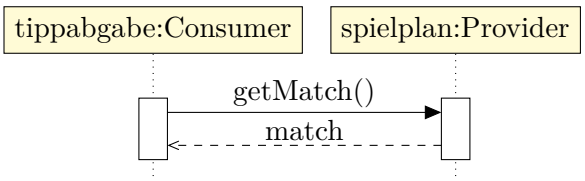

Figure 3.1: Synchronous communication between a provider and a consumer. In this case `tippabgabe` and `spielplan`.

The upcoming sections are split into the consumer and the provider side. This way they are focussed on each side independently from the other, as it would be the case in a project. Judging by the architecture of microservices one would only have access to one microservice. It is often the case that one microservice acts as a consumer as well as a provider.

3.1.1 Consumer

The consumer side is the most commonly used one, because normally there are multiple consumers to one provider service. As a consumer you want to give the provider a CDC which contains every attribute send by the provider that you use. This way the provider won't remove any attributes that the consumer still uses.

Setup

The provider does not need to be reachable by our consumer as we can define the contract without having access to the provider. This is one of the benefits of CDC testing. You can also use the information on the exchange you currently have with the provider. This way we don't have to change any production code. Another benefit is that the testing is based on the current implementation of the exchange.

We start writing tests immediately as we are independent of the deployment status of the provider. We'll define that this is the format the provider will send messages in.

Application

When we are creating a CDC we are establishing which kind of response we expect from the provider. This could be a single field that we expect from the provider. E.g. in our request to the provider we state a user id. This looks like the following:

```
http://example.provider.com/user_id/12345
```

The provider now responds with information of that user. (See 3.2) In our case this is a JSON like response. We only use one field of information, in this case the surname. We want to make sure that our provider always sends this specific part of the information in the response. The consumer only defines the values he is using in the CDC contract. The provider can send much more information that is just disregarded by the consumer. In our CDC we now specify that the provider response has to include a field called `surname` with a matcher because we only want to test if it exists, not which contents it has. If we want an exact matching body take a look at figure 3.3 on the following page.

```
{
    "user-id" : "12345",
    "name" : "Max",
    "surname" : "Mustermann"
}
```

Figure 3.2: The response body of the provider in JSON format.

3.1.2 Provider

The provider is the most important part in a microservice network, as many other parties depend on it. On the provider side the CDCs are getting actively used. A provider tests against multiple contracts to make sure that it fulfils every single contract, so every dependent service can work as expected. If the provider introduces breaking changes to the communication interface, the effected contracts will break in the testing phase. Thus making sure that the provider *does not* introduce changes that inhibit other services (that the provider has a contract with) to work with it.

Setup

The same as seen in 3.1.1 on the preceding page. The provider has to be reachable via HTTP.

```
...
"response" : {
    "status" : 200,
    "headers" : {
        "Content-Type" : "application/json;charset=utf-8"
    },
    "body" : {
        "user-id" : "12345",
        "name" : "Max",
        "surname" : "Mustermann"
    }
}
...
```

Figure 3.3: An excerpt of a generated pact file which only includes a response. Omitted content has been replaced with "..."

Application

As opposed to the consumer, where we had to manually configure the contract generation, we can now let a tool do the work for us. Our provider has to be up and running, so our tool can reach it. Now it can trigger the provider via synchronous means and use the responses to verify all CDCs. An application using a specific tool is found in chapter 5 on page 27.

3.2 CDCs with *a*synchronous communication

One of the more recent approaches for running microservices is to use asynchronous. This is used to counteract the impact of problems that can arise when using synchronous communication.[Rop16] In this chapter we will examine how to best integrate CDC testing into projects. As we don't know exactly how to trigger our provider, our consumer can only define what the expected response is, not what the expected request should be.

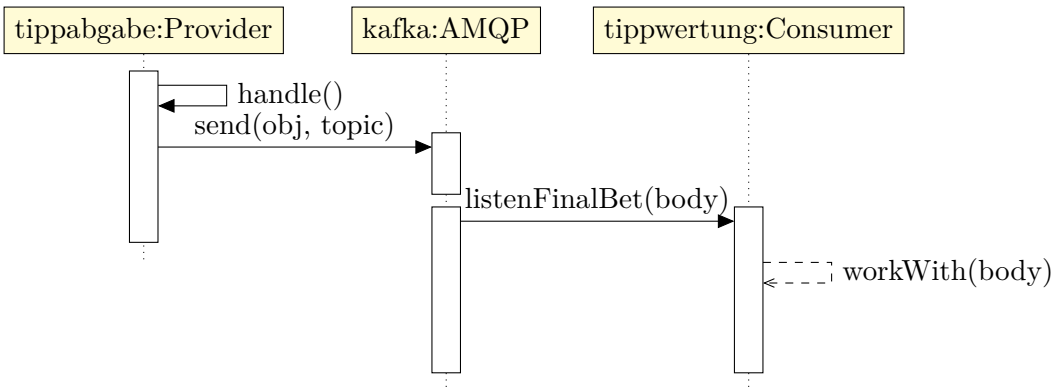

Figure 3.4: The communication between a Provider and a consumer using an AMQP.

As seen in figure 3.4 there is a significant difference to be observed when comparing synchronous (figure 3.1 on page 16) and asynchronous communication. The producer, consumer as well as the message queue could be replaced by any classes with equivalent purpose. The asynchronous communication displays that there is no immediate return from function calls. That is why the lifetimes of the function calls are very short and the first function call does not have a return from the consumer. This enables another advantage of asynchronous communication between microservices:

When a service fails, the error does not bubble up and block the dependant microservices from working. This is described in more detail in a talk from James Roper[3] [Rop16]. He talks about how in synchronous communication a single point of failure can block the whole system as every dependant microservice expects a response and blocks its thread waiting for that response. When using asynchronous communication between microservices a single point of failure does not block the whole system. It simply means that the service is unavailable and we cannot receive information from it. The system does not crash, but it does not offer the full functionality until the failing service has been restored.

[3]Co-creator of the platform Lagom and architect of OSS Lightbend

Preparation

Just like in the synchronous communication in chapter 3.1 on page 16 we *don't* have to have access to both, provider and consumer, to implement CDC testing on either side. We only need access to the side that we are implementing CDC testing for. The following sections are again split in two parts, the consumer side and the provider side.

3.2.1 Consumer

A consumer in an asynchronous messaging environment listens to a messaging queue. In the case of using Kafka, a popular messaging queue tool, the consumer listens to a specific topic. As soon as the producer posts a message to that topic the consumer gets notified and can work with the provided payload. He *consumes* the message.

Setup

As opposed to the mock service that is provided by a CDC tool, e.g. Pact, in a synchronous environment using synchronous REST calls, there will be no mock service provided by Pact in an asynchronous environment, because it is not implemented. The programmer has to specify what the consumer expects from the provider. For this there are several tools provided and the setup as well as the usage is straightforward and well documented [var].

Application

When the stage has been reached where we want to create a CDC it is best to look at what the message handler of the software is expecting to receive from the message queue and to apply that format to the CDC. This way we can make sure that the received payloads resemble what our message handler is actually expecting. This schema has a flaw that can't be overlooked. With this schema we have to edit our testing code, the code generating our CDC, *every* time we introduce changes to the expected payload of the message handler. This is very important because it means that, when we forget to make these changes to the testing code as well, our (unchanged) CDCs will still be correctly verified by the provider against its code base, but the actual communication, tested in end to end tests for example, will fail.

As mentioned before this might introduce human error when instead we would want to keep the testing process as automated as possible. This ensures that our testing

contract breaks the provider as soon as we introduce changes. It is very important to mention that these consumer-sided changes occur infrequently and are introduced mostly by hand anyway. Therefore it is not much more work to edit the generated consumer pact in the same workflow. Another aspect important to mention is that, if a tool that can programmatically create or generate contracts is used, the contracts contents can also be generated from the production code.

If the producer has to be in a certain state (e.g. it has to have one of its databases up and running for our contracts to be verified) we can define this in the contract by using so-called *provider states*. Implementations vary, but the purpose stays the same; Telling the provider in which state he should be in when verifying the contracts given.

3.2.2 Provider

A provider, sometimes also called producer, in an asynchronous messaging environment posts messages to a message queue when triggered by a certain event. This could mean that the provider posts an object every time it gets changed on the provider side, or it could for example post a message to the queue every time it gets requested by a consumer. As we do not know exactly how the provider gets triggered, e.g. what a consumer has to do to provoke a response, we can't validate contracts by simply speaking to the REST interface as we could whilst using synchronous communication. This means that the contract verification on provider side has to be implemented according to the messaging queue that is used. However, our premise does not change. We might handle verifying the interactions differently, but the overall method of CDC testing does not change.

Setup

The provider has to be able to post to an asynchronous messaging queue for our CDC testing scheme in order to work.

Application

We can now trigger our provider depending on what contract we are verifying. As mentioned earlier we have to implement this manually instead of relying on a tool support (See chapter 6 on page 37). When triggering our provider we can then compare the payload that was posted to the messaging queue with the payload described in the contract interaction. If everything matches we have successfully verified our provider.

4 Tools used

This chapter aims to introduce the tools that are used to research the thesis' topic. After reading this chapter the reader has a basic understanding about the tools that are used and why they are used.

The first section introduces the example project that is used to show how CDC testing works in a software environment. The second section gives an overview about the available CDC testing tools, why Pact was chosen and what it has to offer in terms of functionality.

4.1 The example project: `Tippkick`

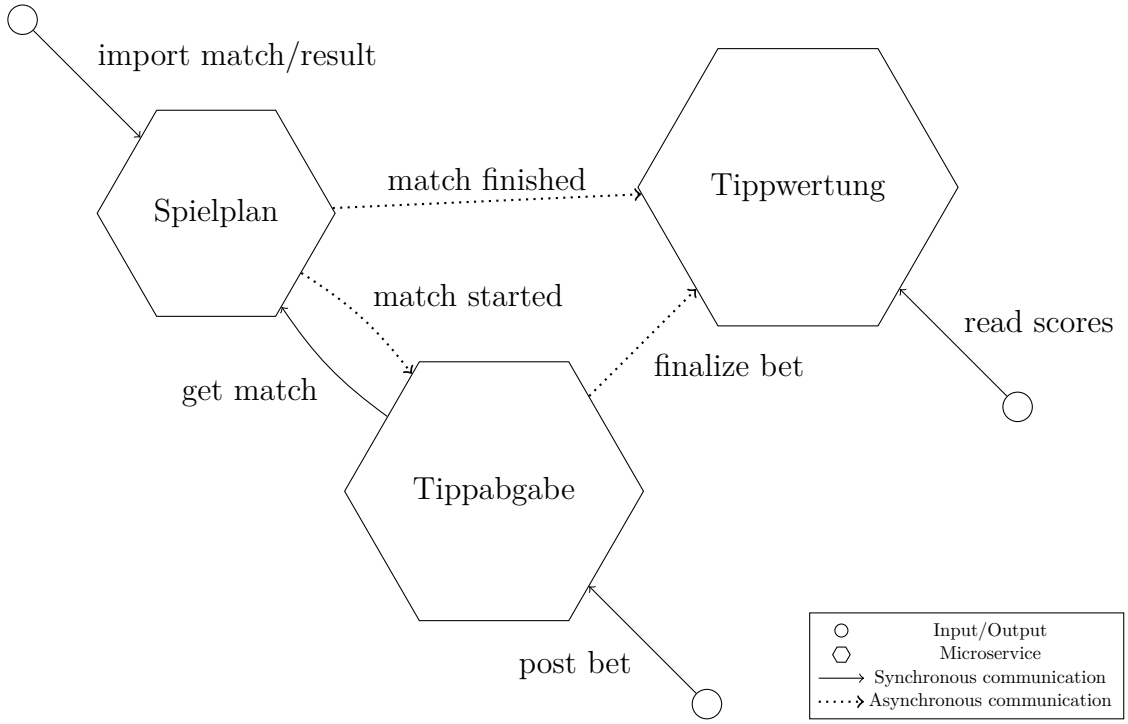

Figure 4.1: The example project with the relations between the three microservices.

The "Tippkick" project is a sample project developed, and provided for this thesis, by Capgemini as a small example application for educational purposes regarding microservices. There are three microservices in the project: `Tippabgabe`, `Tippwertung` and `Spielplan`. These three services aim to provide a betting application for football matches. `Spielplan` maintains all the information regarding the actual matches and notifies the other services of the current status of a match. `Tippabgabe` handles the betting process. You can post a bet to it, and after it is finalized the `Tippwertung`-service will be informed. `Tippwertung` does all the work surrounding whether a bet was successful (your bet was correct) or not (your bet was not correct).

These services communicate through both, synchronous and asynchronous means of messaging (See figure 4.1 on the previous page).

4.2 CDC testing tools and why Pact was chosen

This section aims to introduce the CDC testing tool used by the author. Furthermore it explores other CDC testing tools and why Pact was chosen. In the end there is a short listing showing the capabilities of Pact[1].

4.2.1 CDC testing tools

The most commonly used and discussed tools are Spring cloud contract and Pact. Spring cloud contract is developed by Spring and supports CDC testing for JVM-based applications. It supports CDC testing for synchronous and asynchronous messaging schemes and offers multiple integrations for messaging queues like Apaches Kafka and RabbitMQ for example. Pact is an open-source tool maintained by the Pact foundation. It is implemented in various programming languages[2] and not limited to JVM-based languages. It supports both, synchronous and asynchronous messaging schemes.

It is important to note, that there are ways to exchange the pacts generated by Spring cloud contract and Pact. This is particularly useful if you want to use the asynchronous messaging integrations from Spring cloud contracts while maintaining the flexibility between languages from Pact.

[1] `https://github.com/DiUS/pact-jvm`
[2] A detailed listing can be found at `https://docs.pact.io/feature_support`

Why Pact was chosen for this analysis

Pact was chosen because of its support for multiple languages. This way you can use it for almost any scenario and any provider/consumer build. For example if you want to have Pact verification on an Angular[3] build, which is not a JVM-based framework, Pact would be more suited because it also has an implementation for JavaScript [var].

Spring Cloud Contract does support contract testing, but a little different than the Pact tool. It allows for manual contract generation rather than the code-based generation that Pact ships with. In Spring Cloud Contract the teams working on the provider and the consumer define the contract they are testing against at the beginning. Then both, the provider and consumer(s), test against the same contract. This is another reason Pact was chosen over the Spring Cloud Contract tool as it allows for a more fluent integration of the consumer-driven contract testing approach. As Pact and Spring Cloud Contract have different approaches the choice between the two of them relies heavily on the type of project.

4.2.2 Pact

Pact allows the user to programmatically generate consumer contracts. This means that we can generate contracts at runtime and potentially based on the current codebase. The contracts hold an expected response and the request that will be made to the provider. These contracts are then exchanged with the provider, who then verifies them against itself. The tool triggers the provider with the defined expected request and then verifies the following response with the one defined in the contract.

Pact also offers integration with a so-called Pact broker (provided by the same team that made Pact) which holds all the contracts, offers them to consumer/provider and captures which contracts failed or were successfully verified. More detailed information on the features of Pact are found in the chapters 5 on page 27 and chapter 6 on page 37.

[3] `https://angular.io/`

5 Observations

This chapter aims to provide the reader with observations on the implementation process of CDC testing with Pact. These observations are focused on the differences in implementation of CDC testing using synchronous messaging and asynchronous messaging between services rather than on the subjective experiences made by the author whilst implementing CDC testing.

This is not an experience report but observations made that are subject to the overall process.

The chapter starts by showing what changes in the Pact verification process are done between synchronous and asynchronous testing of interfaces on the consumer and provider side respectively. For each case will be a theory section, explaining what the process looks like in theory, and a practise section, showing by example what aforementioned process looks like. The two big sections are Consumer and Provider, respectively with synchronous and asynchronous subsections. These are further divided to theory and practise.

5.1 Consumer

First will be the differences on the consumer side, starting with the synchronous theory and practise followed by asynchronous theory and practise.

5.1.1 Synchronous

Theory

As described in figure 5.1 on the following page the four steps to synchronous consumer driven contract testing using Pact are:

1. Using the Pact DSL, the expected request and response are registered with the mock service." [var]
2. "The consumer test code fires a real request to a mock provider (created by the Pact framework)." [var]

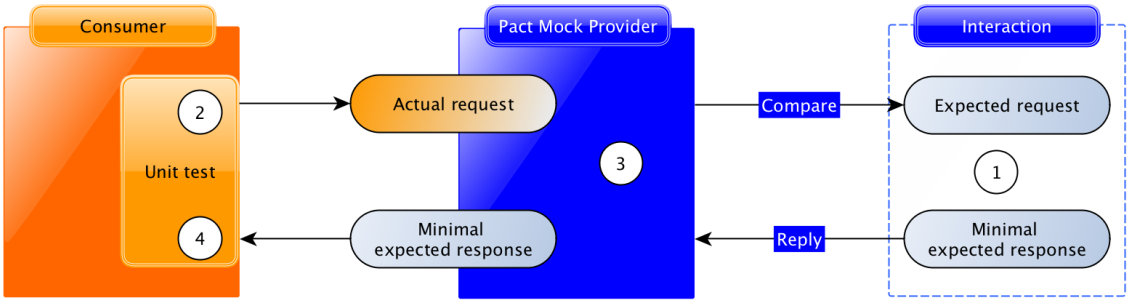

Figure 5.1: The Pact creation/verification process described by the official documentation of Pact[var]

3. "The mock provider compares the actual request with the expected request (from step one), and emits the expected response if the comparison is successful" [var]
4. "The consumer test code confirms that the response was correctly understood" [var]

If all of these steps succeed the Pact verification on consumer side was successful.

As you can see here the Pact tool does most of the test work itself. When writing tests focus has to be put only on the interaction itself by stating the expected request and the minimal expected response. This is because Pact natively supports the REST protocol and handles everything related mostly on its own.

Practise

This subsection aims to give the reader an insight into CDC testing implementation with the Pact tool. The following source code listing (5.2 on the next page) is taken from the Pact-jvm github repository[1] as it is not yet implemented in the `Tippkick` project. A few lines of code are omitted as they are not relevant to this chapter or contain repetitions.

Lines 1 and 2 define the jUnit runner that is used as jUnit 5 is used in this example. Furthermore the provider name and the port on which it can be reached are set.

Lines 4 through 6 define the headers for every Pact interaction. In this case we need to specify that we are using the JSON file format.

Line 8 states the consumer name.

[1] `https://github.com/DiUS/pact-jvm/blob/master/consumer/pact-jvm-consumer-junit5/src/test/java/au/com/dius/pact/consumer/junit5/ArticlesTest.java`

```
1  @ExtendWith(PactConsumerTestExt.class)
2  @PactTestFor(providerName = "ArticlesProvider", port = "1234")
3  public class ArticlesTest {
4    private Map<String, String> headers = MapUtils.putAll(new HashMap<>()
         , new String[] {
5      "Content-Type", "application/json"
6    });
7
8    @Pact(consumer = "ArticlesConsumer")
9    public RequestResponsePact articles(PactDslWithProvider builder) {
10     return builder
11       .given("Articles exist")
12       .uponReceiving("retrieving article data")
13         .path("/articles.json")
14         .method("GET")
15       .willRespondWith()
16         .headers(headers)
17         .status(200)
18         .body(
19           new PactDslJsonBody()
20             .minArrayLike("articles", 1)
21               .object("variants")
22                 .eachKeyLike("0032")
23                   .stringType("description", "sample description")
24                   .closeObject()
25                 .closeObject()
26               .closeObject()
27             .closeArray()
28         )
29       .toPact();
30   }
31
32   @Test
33   @PactTestFor(pactMethod = "articles")
34   void testArticles(MockServer mockServer) throws IOException {
35     HttpResponse httpResponse = Request.Get(mockServer.getUrl() + "/
           articles.json").execute().returnResponse();
36     assertThat(httpResponse.getStatusLine().getStatusCode(), is(equalTo
           (200)));
37     assertThat(IOUtils.toString(httpResponse.getEntity().getContent()),
38       is(equalTo("{\"articles\":[{\"variants\":{\"0032\":{\"description
             \":\"sample description\"}}}]}")));
39   }
40 }
```

Figure 5.2: Contract generating code on the synchronous consumer side

Lines 10 through 30 display the Pact builder used to generate the contract. Line

11 sets the provider state. Lines 12 through 14 define the expected request with the interaction name in line 12, the REST path in line 13 and the HTML method in line 14. Lines 15 through 28 define the response and with it the payload. Line 16 and 17 define the header and the HTML status. In this case `200`. Line 18 and following describe the payload. Further information on how this `PactDslJsonBody` works can be found in the Pact documentation. [var]

Lines 32 through 39 define a jUnit test which determines if the payload was generated correctly. Line 33 defines which method should be used for this test as the `MockServer` is provided to the test. Lines 35 through 38 test if the status code is correct and if the JSON payload was generated correctly.

5.1.2 Asynchronous

Now that the workflow of what happens when using Pact in a synchronous environment has been established we will now look at the asynchronous implementation on the consumer side. The figure from before can be used as an orientation. We will now look at the differences described by the figure and the actual process used in an asynchronous messaging environment.

Theory

In theory the same sequence as for synchronous communication should apply, but there are some changes to the asynchronous implementation. Pact does not offer an implementation for specific messaging queues, but instead has designed the messaging queue testing to be agnostic to the specific implementations offered by Kafka or RabbitMQ for example. Instead of defining a request and an expected response we only define the expected response as we do not have a synchronous request/response system. Further information will be provided in chapter 6 on page 37.

Practise

This subsection aims to give the reader an insight into an implemented CDC testing using the Pact tool. The following source code listing (5.3 on the next page) is taken from the example application "Tippkick" and from the service "tippwertung" which acts as a consumer and has been shortened to not include comments, packages and imports.

```
1  @ExtendWith(PactConsumerTestExt.class)
2  @PactTestFor(providerType = ProviderType.ASYNCH)
3  public class IncomingMessageHandlerTest {
4
5      @Pact(provider = "tippabgabe", consumer = "tippwertung")
6      public MessagePact createPactForTippabgabe(MessagePactBuilder
           builder) {
7
8          DslPart actualPactDsl = LambdaDsl.newJsonBody(o -> {
9              o.numberType("matchId", 0L);
10             o.numberType("ownerId", 0L);
11             o.numberType("hometeamScore", 0);
12             o.numberType("foreignteamScore", 0);
13         }).build().asBody();
14
15         Map<String, Object> metadata = new HashMap<>();
16         metadata.put("Content-type", "application/json");
17         metadata.put("kafka-topic", "tipp");
18
19         return builder.given("NoDataState")
20                 .expectsToReceive("a test message")
21                 .withMetadata(metadata)
22                 .withContent(actualPactDsl)
23                 .toPact();
24
25     }
26
27     @Test
28     @PactTestFor(pactMethod = "createPactForTippabgabe")
29     void testPactForTippabgabe(MessagePact pact) throws Exception {
30         String expectedBody = new ObjectMapper().writer().forType(
               GameBetEvent.class).writeValueAsString(new GameBetEvent());
31         JSONAssert.assertEquals(new String(pact.getMessages().get(0).
               contentsAsBytes()), expectedBody, true);
32     }
33 }
```

Figure 5.3: Contract generating code on the asynchronous consumer side

Lines 8 through 13 define the contract payload. In this case four key-value pairs, where the key is a String and the value of a numeric type. Thus the `o.numberType(...)` to specify this type.

Lines 15 through 17 define the metadata that is to be written into the contract. Line 16 tells the provider which type of content the consumer expects to receive. In this case it's in the `json` format. Line 17 is there to tell the provider on which topic this payload is going to be sent and received. Further information

in chapter 6 on page 37.

Lines 19 through 23 display a return statement which is a contract builder provided by the Pact framework. Line 19 defines the so-called provider state. The state that should be invoked at the provider is in this case the `NoDataState`, meaning that the provider does not have to have any data initialized. Line 20 displays how the interaction between the two services is called. This interaction is called `a test message`.

Lines 27 through 32 define a jUnit test that verifies that the Pact contract payload actually contains the object that is expected to be sent by the provider. In this case a serialization of an object of type `GameBetEvent`.

5.2 Provider

Provider verification has the biggest differences between synchronous and asynchronous in Pact testing.

5.2.1 Synchronous

Theory

With synchronous communication all of the testing is driven by the Pact framework. The Pact runner acts as our consumer by playing the recorded requests from our pact files back to the provider. E.g. it triggers the provider and then records the received answer from the provider. The process of triggering our provider is pretty simple as Pact just making the specific REST calls to it. The response is then compared to what was originally stated by the consumer in our CDC. It can be done in as little as a few lines of code as the provider only has to be set to a certain state and then can be interacted with through the use of REST calls which requires no further interaction as it is provided by the defined contract.

Figure 5.4 on the next page can be used for understanding the described workflow. When following the arrows from top left the workflow of Pact is depicted.

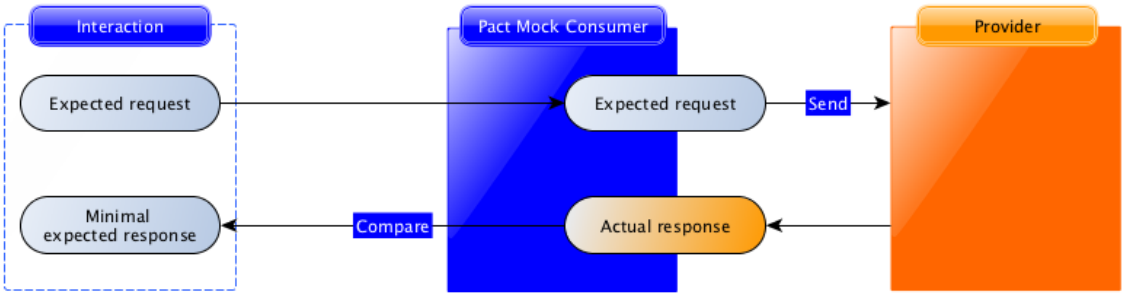

Figure 5.4: The verification process on provider side as stated in the Pact documentation [var]

Practise

This example is a fictional example, so not taken from a live project, but it resembles the structure and the efficiency of reallife provider tests. The author has opted to construct a very minimal example whilst still showing the capabilities of Pact and the different scenarios it can be used with. The following paragraph will explain the code found in listing 5.5 on the following page.

Lines 1 through 3 set up the provider name, the folder where the Pact contracts can be found. One could also use a pact broker or a path to a specific contract.

Lines 5 through 8 are the central part of the contract test with the provider. The command `context.verifyInteraction()` starts the Pact tool which then tests our provider.

Lines 10 through 13 can set up the service. Omitted in this case, but this would be the part where one would start up the service so it can be reached via REST.

Lines 15 through 18 show the method that sets up the provider state ”default”. This would set up databases and/or initial information in those databases for the state ”default”.

Lines 20 through 23 show the method that tears down aforementioned provider state ”default”. I.e. shutting down the database or doing some cleanup.

```
1  @Provider("myService")
2  @PactFolder("pacts")
3  @ExtendWith(PactVerificationInvocationContextProvider.class)
4  public class ContractTest {
5      @TestTemplate
6      void testTemplate(Pact pact, Interaction interaction, HttpRequest
           request, PactVerificationContext context) {
7        context.verifyInteraction();
8      }
9
10     @BeforeAll
11     static void setUpService() {
12       // Set up the service
13     }
14
15     @State("default")
16     public void toDefaultState() {
17       // Set up the state "default"
18     }
19
20     @State(value = "default", action = StateChangeAction.TEARDOWN)
21     public void toDefaultStateAfter() {
22       // Tear down the state "default"
23     }
24 }
```

Figure 5.5: The code used to test contracts against a provider using synchronous messaging

5.2.2 Asynchronous

Theory

While the Pact tool does most of the work in a synchronous environment there are big changes in the asynchronous environment. Due to the vast range of tools available that offer functionality for a messaging queue the Pact tool was designed with no specific messaging queue in mind. [var] This and the fact that an asynchronous interaction does not have an immediate request/response system as a REST interface does lead to Pact not knowing how to trigger our provider. I.e. how to invoke a specific function in our provider to get it to post the message we expect to a messaging queue. Furthermore the Pact tool is not aware of what messaging queue we use and how it can interact with it. The work that has to be put in to get it to working as good as the synchronous counterpart in regards to the functionality is considerably higher. Further information to that in the following chapter 6 on page 37

Practise

This subsection aims to give the reader an insight into an implemented CDC testing using the Pact tool. The following source code listing (5.6 on the following page) is taken from the example application "Tippkick" and from the service "tippabgabe" which acts as a provider and has been shortened to not include comments, packages and imports. The following paragraph will explain the code found in listing 5.6 on the next page.

Lines 1 through 9 set up various things, including settings for Pact in lines 1 to 5 (Name of the provider, authentication for the Pact broker) and for jUnit and Spring in line 6 to 9.

Lines 12 through 16 define fields that are going to be used for the Kafka integration.

Lines 18 through 30 define the method that is executed before every contract verification. In further detail we ensure that our results of the contract verifications are sent back to the Pact broker. In line 22 we tell Pact that our provider is of the type `AmpqTestTarget()` which means that it uses a messaging queue. This line is very important and needs to be included in every Pact provider test that uses asynchronous messaging. Lines 24 to 28 read the `kafka-topic` from the contract. This is mentioned in 6 on page 37. We then set up our Kafka service to run on this kafka topic.

Lines 31 to 34 trigger the Pact tool and start the contract verifications.

Line 36 and 37 are used to invoke the state "`NoDataState`" of our provider. In this case this is just a placeholder.

Lines 39 through 42 display the method that Pact uses to get the message sent by the provider when triggered by the interaction called "a test message". This will be explained in further detail in chapter 6 on page 37.

Lines 44 through 53 loop the payload of the provider through a kafka stream and then reads that stream and returns the payload. This way we make sure that the serialization and deserialization process works and Kafka provides us with the correct payload.

Lines 55 through 74 are used to setup the embedded Kafka. This includes the correct sender with the right serialization properties as well as the correct receiver with the right deserialization properties for the key and for the values of the payload.

```
1  @Provider("tippabgabe")
2  @ExtendWith(PactVerificationInvocationContextProvider.class)
3  @PactBroker(host = "localhost", port = "80")
4  @PactBrokerAuth(username = "postgres", password = "password")
5  @VerificationReports({"console", "markdown"})
6  @SpringJUnitConfig
7  @DirtiesContext
8  @EmbeddedKafka
9  @SpringBootTest
10 public class OutgoingMessageHandlerJunitPactTest {
11
12     @Autowired
13     private EmbeddedKafkaBroker embeddedKafkaBroker;
14     private KafkaTemplate<Long, FinalizeGameBetEvent> sender;
15     private Consumer<Long, FinalizeGameBetEvent> consumer;
16     private String kafkaTopic;
17
18     @BeforeEach
19     void before(PactVerificationContext context) {
20         System.setProperty("pact.verifier.publishResults", "true");
21
22         context.setTarget(new AmpqTestTarget());
23
24         String metaData = context.getInteraction().toMap(PactSpecVersion.V3).get("metaData").
               toString();
25         int indexOfKafkaTopic = metaData.indexOf("kafka-topic=");
26         kafkaTopic = metaData.substring(indexOfKafkaTopic + "kafka-topic=".length());
27         kafkaTopic = kafkaTopic.substring(0, Math.min(kafkaTopic.indexOf(","), kafkaTopic.
               indexOf("}")));
28         setUpKafka(kafkaTopic);
29     }
30
31     @TestTemplate
32     void pactVerificationTestTemplate(PactVerificationContext context) {
33         context.verifyInteraction();
34     }
35
36     @State("NoDataState")
37     public void invokeNoDataState() { System.out.println("NoDataState was called"); }
38
39     @PactVerifyProvider("a test message")
40     public String testTippwertung() {
41         return getKafkaString();
42     }
43
44     public String getKafkaString() {
45         OutgoingMessageHandler handler = new OutgoingMessageHandler(sender);
46         handler.handle(new FinalizeGameBetEvent(0,0,0,0));
47
48         embeddedKafkaBroker.consumeFromAllEmbeddedTopics(consumer);
49
50         ConsumerRecord<Long, FinalizeGameBetEvent> received = KafkaTestUtils.getSingleRecord(
               consumer, kafkaTopic);
51
52         return String.valueOf(received.value());
53     }
54
55     public void setUpKafka(String topic) {
56         embeddedKafkaBroker.addTopics(topic);
57
58         Map<String, Object> senderProperties = KafkaTestUtils.senderProps(embeddedKafkaBroker.
               getBrokersAsString());
59         senderProperties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, LongSerializer.class);
60         senderProperties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, JsonSerializer.class)
               ;
61
62         ProducerFactory<Long, FinalizeGameBetEvent> producerFactory = new
               DefaultKafkaProducerFactory<>(senderProperties);
63
64         sender = new KafkaTemplate<>(producerFactory);
65         sender.setDefaultTopic(topic);
66
67         Map<String, Object> consumerProps = KafkaTestUtils.consumerProps("testGroup", "true",
               this.embeddedKafkaBroker);
68         consumerProps.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
69         consumerProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, LongDeserializer.class);
70         consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.
               class);
71         ConsumerFactory<Long, FinalizeGameBetEvent> cf = new DefaultKafkaConsumerFactory<>(
               consumerProps);
72         consumer = cf.createConsumer();
73         consumer.subscribe(embeddedKafkaBroker.getTopics());
74     }
75 }
```

Figure 5.6: The code used to test contracts against a provider using asynchronous messaging

6 Analysis

This chapter aims to analyse the observations made in chapter 5 on page 27. It will be discussed how the approach to implementing CDC testing differs from synchronous messaging to asynchronous messaging. Further it will be analysed what the current state of CDC testing with asynchronous messaging looks like and how it could be improved. After this chapter it is clear how CDC testing differs from synchronous to asynchronous messaging between services, how one can adapt to these changes and how the asynchronous implementation could possibly be enhanced to further resemble the fluent and seamless integration that the synchronous variant of CDC testing using Pact represents.

6.1 Synchronous to asynchronous messaging

This section will analyse the observations made in chapter 5 on page 27 and will give an idea on how to take these changes into account when using CDC testing with the Pact tool in an asynchronous environment. First subsection will be focussing on the consumer side and the second will be the provider side.

6.1.1 Consumer

We will start by looking at the consumer side of the consumer driven contract testing. While in a synchronous environment we work with a request and response system, in an asynchronous environment we work with a decoupled request response scheme or a response only scheme. Imagine a provider that posts information to a message queue as soon as its internal database changes. While we do not know how to trigger our provider in certain cases, there are nonetheless cases where we can easily trigger our provider. One would be using a REST interaction, but receiving the payload via asynchronous means. Thus we do not have an expected request, but only an expected response that we can define in our contracts.

Because of this our Pact creation/verification process on consumer side changes a fair bit. The expected request is completely left out. Instead we will only provide the tested provider with a string containing the title of our current interaction. E.g.

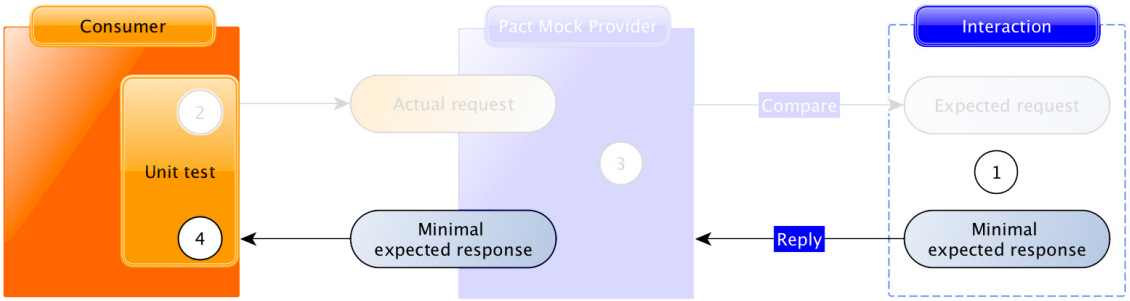

Figure 6.1: Creating a consumer contract in an asynchronous environment. The parts that are displayed transparently are omitted in the interaction.

just a simple name for the interaction. We also specify a provider state [var] that can invoke a specific state on the provider side. E.g. having specific data in its database or setting up the provider in a specific way. Examples for this can be found in 5.3 on page 31.

With all this in mind our four steps when creating a Pact contract (Enumeration 5.1.1 on page 27) condense down to only one step that is lead by the Pact framework. We only describe the expected response in the contract. The process can be found in figure 6.1. There is no automatic verification by the Pact tool that our consumer understands the response correctly. It is now in the hands of the testing department to ensure this. There is no mock provider either, as Pact does not know what to mock. So which messaging queue to use as well as not having the capabilities because of the aforementioned agnostic design. [var]

Further steps would be to post the defined contract interaction to the messaging queue in use and then make sure that our consumer can correctly handle the payload. This would be something that is fairly easy to implement and it would make for a more fluent experience and would be closer to the testing that is done in a synchronous messaging environment. This is a realization that is based on the use of Pact tool for Java and the use of Kafka as a messaging queue. It might differ depending on which language is used and on the framework that is used for asynchronous means of communication.

If the provider can be triggered by the consumer it would be possible for the testing team to define such a testing pipeline that would work in more or less the same way as for the synchronous messaging environment. The only difference would be how the request and response work, but not in the principle of the interaction. We would still have a request/response system as it is the case for a synchronous exchange.

More information = better testing?

Steps that can be made to better prepare for asynchronous contract testing would be to, in the case of using Kafka as a message queue, include the kafka-topic[1] on which the interaction should take place on. This way we have more metadata to test with. The contract test would fail as soon as the two services communicating would communicate on different topics. And maybe in future updates or in self-made enhancements to the tool we can use this information to specifically test the service on this kafka-topic. You can see this being used in line 17 in the code listing 5.3 on page 31.

Of course this has to be agreed on by the teams maintaining the affected services as this could possibly introduce breaking changes to the contract testing if there is testing work done on the metadata of the contract. If there is no interference there is no reason not to include this information as it might be useful for future development and does not add a significantly bigger workload nor a noticeable addition of storage use. The more information one can gather in an interaction in a contract the more secure our contract testing can be, as we have a more detailed interaction between services; even if we don't use all of the information.

Side by side comparison between synch and asynch

The following example will show how the synchronous and asynchronous consumer examples look like side-by-side and analyse the differences and similarities. In figure 6.2 on the following page on the left is the synchronous consumer example (code listing 5.2 on page 29), on the right the asynchronous consumer example (code listing 5.3 on page 31).

These examples differ in quite a few points. The most obvious to point out is that they both use different DSLs provided by the Pact tool to create their expected requests bodies. The synchronous example makes use of the old DSL from Pact which has the structure of a builder and can be hard to read at times because of the excessive nesting using closing calls like `.closeObject()`. The asynchronous example makes use of the newer Pact DSL. It uses a lambda function to better incorporate the auto-indentation provided by IDEs. It also uses simple curly brackets to close an object.

Further differences in this particular section of the code is obviously that they are examples taken from different projects. While the left one uses a complex nested structure with the Pact DSL features `.minArrayLike()` and `.eachKeyLike()`,

[1] The channel to which messages are posted.

the right one uses a less complex payload in their interaction, thus only including calls to the DSL like `.numberType()`.

```
@ExtendWith(PactConsumerTestExt.class)
@PactTestFor(providerName = "
    ArticlesProvider", port = "1234")
public class ArticlesTest {
  private Map<String, String> headers =
      MapUtils.putAll(new HashMap<>(), new
      String[] {
    "Content-Type", "application/json"
  });

  @Pact(consumer = "ArticlesConsumer")
  public RequestResponsePact articles(
      PactDslWithProvider builder) {
    return builder
      .given("Articles exist")
      .uponReceiving("retrieving article
          data")
        .path("/articles.json")
        .method("GET")
      .willRespondWith()
        .headers(headers)
        .status(200)
        .body(
          new PactDslJsonBody()
            .minArrayLike("articles", 1)
              .object("variants")
                .eachKeyLike("0032")
                  .stringType("description"
                      , "sample description
                      ")
                  .closeObject()
              .closeObject()
            .closeObject()
          .closeArray()
      )
      .toPact();
  }

  @Test
  @PactTestFor(pactMethod = "articles")
  void testArticles(MockServer mockServer)
      throws IOException {
    HttpResponse httpResponse = Request.Get
        (mockServer.getUrl() + "/articles.
        json").execute().returnResponse();
    assertThat(httpResponse.getStatusLine()
        .getStatusCode(), is(equalTo(200)))
        ;
    assertThat(IOUtils.toString(
        httpResponse.getEntity().getContent
        ()),
      is(equalTo("{\"articles\":[{\"
          variants\":{\"0032\":{\"
          description\":\"sample
          description\"}}}]}")));
  }
}
```

```
@ExtendWith(PactConsumerTestExt.class)
@PactTestFor(providerType = ProviderType.
    ASYNCH)
public class IncomingMessageHandlerTest {

    @Pact(provider = "tippabgabe", consumer
        = "tippwertung")
    public MessagePact
        createPactForTippabgabe(
        MessagePactBuilder builder) {

        DslPart actualPactDsl = LambdaDsl.
            newJsonBody(o -> {
          o.numberType("matchId", 0L);
          o.numberType("ownerId", 0L);
          o.numberType("hometeamScore",
              0);
          o.numberType("foreigteamScore"
              , 0);
        }).build().asBody();

        Map<String, Object> metadata = new
            HashMap<>();
        metadata.put("Content-type", "
            application/json");
        metadata.put("kafka-topic", "tipp")
            ;

        return builder.given("NoDataState")
            .expectsToReceive("a test
                message")
            .withMetadata(metadata)
            .withContent(actualPactDsl)
            .toPact();

    }

    @Test
    @PactTestFor(pactMethod = "
        createPactForTippabgabe")
    void testPactForTippabgabe(MessagePact
        pact) throws Exception {
        String expectedBody = new
            ObjectMapper().writer().forType
            (GameBetEvent.class).
            writeValueAsString(new
            GameBetEvent());
        JSONAssert.assertEquals(new String(
            pact.getMessages().get(0).
            contentsAsBytes()),
            expectedBody, true);
    }
}
```

Figure 6.2: Side-by-side comparison between the synchronous (on the left) and asynchronous (on the right) consumer implementation

Highlighted in red on the left is the omitted expected request to the provider. It is rather short because the REST interaction is a simple GET-call, thus not transmitting any extra information that could be saved in a contract, except the path that the call is made on. There are another few changes in the first few lines, but they are not as important as they only specify the provider name and type and alike. The actual interaction that is saved in the contract does not contain many changes from

the synchronous to the asynchronous implementation. So for the consumer there is not that much change.

The last paragraph in both code examples is a unit test that tests if the generated contract really contains the payload that we are it expecting to contain. It is important to note that this is testing the contract generation process and is not testing if the consumer can process the payload received from the provider correctly. In a good test it should be tested if the consumer can actually work with the payload so the connection between test code and live code is as close and tightly knitted as it can be. This way we can make sure that if the test passes the code works as well. Otherwise we would have a disconnect between the two and could receive false negatives which we want to avoided at all costs as it lowers the confidence of our tests.

To summarise we can observe that there are not many practical changes on the consumer side. What is important is that we can support the provider in writing its framework for verifying contracts by adding information like the kafka-topic in the meta data of the contract.

6.1.2 Provider

As motivated earlier, the provider experiences the biggest change coming from using synchronous messaging to using asynchronous messaging. This is partly because of the agnostic design of the Pact tool on the provider side. There is no support for a specific messaging queue like Kafka or RabbitMQ but instead a generic implementation to support interactions. This means that there is a lot more to be done before being able to successfully use Pact on an asynchronous provider.

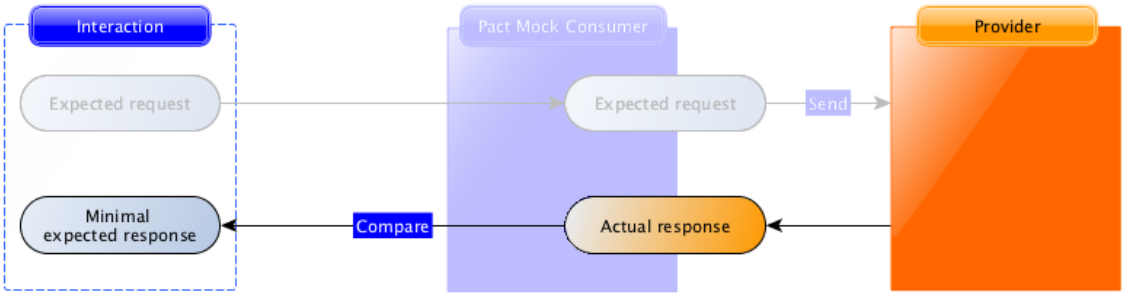

Figure 6.3: Verifying a provider in an asynchronous environment. The parts that are displayed transparently are omitted in the interaction.

Another reason for the big change in provider contract testing is the absence of a well-defined entry point for the provider. It might not be possible to obtain an entry

point as we were able to in the synchronous implementation. We had a well defined REST interface which made it easy to define the entry point. There are different ways our provider could now react to changes, or, to be more precise, different ways for our provider to be triggered to provide new information. It could, to provide a few examples, now only post information when its database changes, from internal influence, or could change when it detects a certain message on a kafka-topic, so an external influence. As Pact can not possibly know which type of provider we are using the team developing the service has to make sure that they are able to trigger the provider. This is displayed in figure 6.3 on the previous page. The parts that are more transparent are omitted in the contract verification process on the provider side.

In the following paragraphs we will look at the code examples from chapter 5 on page 27 and investigate how to incorporate asynchronous testing with Pact to make it as powerful and easy to use as the synchronous implementation.

Side-by-side comparison

The two code examples will be stretched over multiple figures because of their size. In code listing 6.4 on the next page and 6.6 on page 47 are boxes with numbers. These highlight the similarities between the two and will be discussed in the following.

Box 1 is for the source of the Pacts that are going to be tested. In the synchronous example it is a folder named ”pacts”. The asynchronous example is a little more sophisticated as it uses a Pact broker with authentication. Here the credentials are hardcoded, but there are different ways of authentication, including more elegant variants through environment variables and token authentication.

Box 2 is positioned at the method that invokes the Pact tool verification process. There is little to no difference here between the synchronous and asynchronous example. It is possible to manipulate the various Pact tool specific objects that are displayed in the parameters of the method.

Box 3 is at an example provider state method. Same procedure in both, the synchronous and asynchronous, examples.

When we now compare the two code examples we can see that the actual code using the Pact tool does not differ much between synchronous and asynchronous. What does differ however is the functionality of the verification process which was introduced earlier. We now have hardcoded a response in listing 6.5 on page 44. So our testing code does not have any point of interaction to our life production code. What is different as well is that we now have an entry point defined as a method that the Pact tool calls automatically. In this case it is `testTippwertung` annotated

```
1  @Provider("myService")
2  @PactFolder("pacts")
3  @ExtendWith(PactVerificationInvocationContextProvider.class)
4  public class ContractTest {
5      @TestTemplate
6      void testTemplate(Pact pact, Interaction interaction, HttpRequest
       request, PactVerificationContext context) {
7        context.verifyInteraction();
8      }
9
10     @BeforeAll
11     static void setUpService() {
12       // Set up the service
13     }
14
15     @State("default")
16     public void toDefaultState() {
17       // Set up the state "default"
18     }
19
20     @State(value = "default", action = StateChangeAction.TEARDOWN)
21     public void toDefaultStateAfter() {
22       // Tear down the state "default"
23     }
24 }
```

1 2 3

Figure 6.4: Synchronous provider implementation with annotated numbers to guide the reader.

with `@PactVerifyProvider("a test message")` to signal that this is the method we want invoked when the interaction "a test message" takes place.

To try to reach the level of integration of the synchronous implementation with the asynchronous implementation there are a few steps necessary which we will take a look at now.

Testing the provider more thoroughly

The asynchronous example discussed above is a shortened version of 6.6 on page 47 which combines the testing code and life production code by using Kafka. The framed code sequences in code 6.6 on page 47 are the parts that are used to include Kafka in the testing process. This was an effort made to try to include the actual messaging queue into the testing and, when done right, one can trigger the provider

```
1  @Provider("tippabgabe")
2  @ExtendWith(PactVerificationInvocationContextProvider.class)
3  @PactBroker(host = "localhost", port = "80")
4  @PactBrokerAuth(username = "postgres", password = "password")
5  public class OutgoingMessageHandlerJunitPactTest {
6
7      @BeforeEach
8      void before(PactVerificationContext context) {
9          context.setTarget(new AmpqTestTarget());
10     }
11
12     @TestTemplate
13     void pactVerificationTestTemplate(PactVerificationContext context)
       {
14         context.verifyInteraction();
15     }
16
17     @State("NoDataState")
18     public void invokeNoDataState() {
19         // Set up the state "NoDataState"
20     }
21
22     @PactVerifyProvider("a test message")
23     public String testTippwertung() {
24         return "{\"hometeamScore\": 0, \"ownerId\": 0, \"
               foreignteamScore\": 0, \"matchId\": 0}";
25     }
26 }
```

1 2 3

Figure 6.5: Asynchronous provider implementation with annotated numbers to guide the reader. Shortened version.

and compare the expected response just as in the synchronous example, just without the expected request.

The box annotated with a marks the part of the Kafka code where the `kafka-topic` is read from the consumer contracts. This way we can test for the kafka topic in our contract which provides us with another layer of certainty in our test. It can be tested if the provider and consumer communicate on the same kafka-topic, otherwise our test would fail. This is only a suggestion as the code is not fully tested. It does certainly work for simple examples, but further testing is advised. The method `getKafkaString()` invokes the handler of the project itself (triggers it) and then captures the sent payload from the kafka-topic. This example uses the actual testing code as it creates an instance of the so-called `OutgoingMessageHandler` of the service `tippabgabe` and then uses the live test code to generate the payload.

The captured value from the kafka-topic is then used to verify the contract. This way our testing code and live production code are intertwined in a way that allows testing with higher confidence as we do not need to change the testing code when our production code changes. The last box contains the method `setUpKafka` and its sole purpose is to set up the EmbeddedKafkaBroker that is used in the testing code. The Kafka sender and receiver are set up with the parsed kafka topic from the contract.

The code for the Kafka integration takes up a lot of space and writing it can be time intensive. This is especially bothersome if you expect the same fluent integration and minimal effort needed as in the synchronous implementation. Ways to make this process easier could be the refactoring of the Kafka code to another class and then inheriting from that class if your test code allows it. A more elegant variant would probably be to make use of annotation processing. One could write an annotation to the class using Pact for provider verification which provides functionality of a more fluent Kafka integration. This would reduce a lot of the boilerplate code and would allow for faster test creation, thus being more affordable and desirable in software projects.

One could imagine an annotation like `@PactWithKafka`. It could enhance the class using the provider tests with an embeddedKafkaBroker so that one could use it directly without all the setup as seen in the last highlighted code block. Instead one could use the embeddedKafkaBroker directly as seen in the method `getKafkaString()`. This way we could still control the use but do not have to deal with all the setup. This would make the writing of tests less cumbersome. If we would like to keep the flexibility of the used approach one could imagine passing settings in the form of arguments to the annotation. So for example if we would want to define the serializer we could just have something like

```
// --snip--
@PactWithKafka(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG =
    LongSerializer.class)
public class OutgoingMessageHandlerJunitPactTest {
// --snip--
```

to replace a default value. This would represent the code seen in line 59 in figure 6.6 on page 47 which is setting the serializer class for the key of the Kafka sender. With this one could imagine triggering the running provider with a special message on a Kafka stream. What cannot be refactored to a different file is the trigger point of the provider. This is still something that has to be handled by the team writing the tests and it has to be handled for each case individually.

If the provider is triggered externally, by a REST request or a specific message on a Kafka stream, it would however be possible to write a program that takes some of the work off the team writing the tests as you could specify the Kafka

stream in an annotation or the triggering REST request. There is a lot of potential to make the CDC testing process for asynchronous provider less cumbersome and easier accessible.

As it stands now this is the biggest problem with asynchronous contract testing on the provider side: When writing a test you want to focus on writing the test, not on completing the tool you test with.

```
1  @Provider("tippabgabe")
2  @ExtendWith(PactVerificationInvocationContextProvider.class)
3  @PactBroker(host = "localhost", port = "80")
4  @PactBrokerAuth(username = "postgres", password = "password")
5  @VerificationReports({"console", "markdown"})
6  @SpringJUnitConfig
7  @DirtiesContext
8  @EmbeddedKafka
9  @SpringBootTest
10 public class OutgoingMessageHandlerJunitPactTest {
11
12     @Autowired
13     private EmbeddedKafkaBroker embeddedKafkaBroker;
14     private KafkaTemplate<Long, FinalizeGameBetEvent> sender;
15     private Consumer<Long, FinalizeGameBetEvent> consumer;
16     private String kafkaTopic;
17
18     @BeforeEach
19     void before(PactVerificationContext context) {
20         System.setProperty("pact.verifier.publishResults", "true");
21
22         context.setTarget(new AmpqTestTarget());
23
24         String metaData = context.getInteraction().toMap(PactSpecVersion.V3).get("metaData").
               toString();
25         int indexOfKafkaTopic = metaData.indexOf("kafka-topic=");
26         kafkaTopic = metaData.substring(indexOfKafkaTopic + "kafka-topic=".length());
27         kafkaTopic = kafkaTopic.substring(0, Math.min(kafkaTopic.indexOf(","), kafkaTopic.
               indexOf("}")));
28         setUpKafka(kafkaTopic);
29     }
30
31     @TestTemplate
32     void pactVerificationTestTemplate(PactVerificationContext context) {
33         context.verifyInteraction();
34     }
35
36     @State("NoDataState")
37     public void invokeNoDataState() { System.out.println("NoDataState was called"); }
38
39     @PactVerifyProvider("a test message")
40     public String testTippwertung() {
41         return getKafkaString();
42     }
43
44     public String getKafkaString() {
45         OutgoingMessageHandler handler = new OutgoingMessageHandler(sender);
46         handler.handle(new FinalizeGameBetEvent(0,0,0,0));
47
48         embeddedKafkaBroker.consumeFromAllEmbeddedTopics(consumer);
49
50         ConsumerRecord<Long, FinalizeGameBetEvent> received = KafkaTestUtils.getSingleRecord(
               consumer, kafkaTopic);
51
52         return String.valueOf(received.value());
53     }
54
55     public void setUpKafka(String topic) {
56         embeddedKafkaBroker.addTopics(topic);
57
58         Map<String, Object> senderProperties = KafkaTestUtils.senderProps(embeddedKafkaBroker.
               getBrokersAsString());
59         senderProperties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, LongSerializer.class);
60         senderProperties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, JsonSerializer.class)
               ;
61
62         ProducerFactory<Long, FinalizeGameBetEvent> producerFactory = new
               DefaultKafkaProducerFactory<>(senderProperties);
63
64         sender = new KafkaTemplate<>(producerFactory);
65         sender.setDefaultTopic(topic);
66
67         Map<String, Object> consumerProps = KafkaTestUtils.consumerProps("testGroup", "true",
               this.embeddedKafkaBroker);
68         consumerProps.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
69         consumerProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, LongDeserializer.class);
70         consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.
               class);
71         ConsumerFactory<Long, FinalizeGameBetEvent> cf = new DefaultKafkaConsumerFactory<>(
               consumerProps);
72         consumer = cf.createConsumer();
73         consumer.subscribe(embeddedKafkaBroker.getTopics());
74     }
75 }
```

Figure 6.6: Asynchronous provider implementation with annotated numbers to guide the reader. Framed with a box in the source code are parts that are needed for the Kafka integration.

6.2 Future work

This section aims to explore the possibilities of further research in the field that this thesis includes. This can be based on this thesis, but is not limited to. There are two main sections that can be explored. One is independent from the CDC testing tool, focussing on the effects that using CDC testing has on the project it is used on, and the other is highly dependent on the currently available tools, exploring how one can further enhance the testing workflow with them.

CDC testing

An interesting field of research would be the exploration and gathering of metrics concerning CDC testing. This could be a comparison between a sample project with and without CDC testing in a microservice architecture.

Can we replace end-to-end tests with detailed CDC tests? What kind of performance gain can we observe when deploying CDC testing? Can this gain be generified and/or captured with some mathematical approximation? Furthermore it would be interesting how many resources have to be expended to implement CDC testing in a project. A cost benefit analysis could be of interest here. Another very interesting aspect would be the experiences made in the industry. A survey might offer interesting insights into the current distribution of CDC testing as there are little to no statistics available.

Regarding the testing tools

What definitely needs to be explored are the possibilities of implementing specific messaging queue support to Pact for JVM based languages, or other languages for that matter, as the effort needed to implement asynchronous CDC testing is much higher than the implementation process for the synchronous counterpart. The development of an addon implementing/enhancing the suggestions made in this thesis would be of interest. Can a point in implementation be reached where the asynchronous approach is as comfortable to use as the synchronous approach, thus reducing the initial hurdle of asynchronous CDC testing. Another interesting aspect would be the evaluation of the Pact tool itself.

A different approach, regarding JVM-based languages, would be to combine Pact and Spring Cloud Contract, as the latter already has support for specific messaging queues. As already mentioned there exist tools which are used to exchange contracts between the two. An interesting inquiry would be if we lose information in either

direction of the conversion and if it would affect our workflow negatively or at all. Can one, combining the two tools, create a workflow that is as comfortable to use as the synchronous approach only using Pact?

So important aspects to enhance the experience of asynchronous provider testing would be either the development of an addon that adds specific messaging queue implementations to Pact, or a synergy of Pact and Spring Cloud Contract working together to create a good workflow of CDC testing.

7 Summary

This summary will be based on the initial questions asked in the introduction as those are the main results of this thesis. For each question there will be a reference to the chapter or section where the question was answered.

1. What is CDC testing and why do we need it in the context of microservice architecture?

CDC testing is way of formally defining contracts for the specific format of an interaction between two services sharing information. These contracts get generated by the consumer. The provider then takes this contracts and verifies itself with the provided information. (chapter 3 on page 13)

The reason this specific testing approach gets more important in the context of microservices is the sheer amount of interactions and interfaces between these services. While it was easily manageable in a monolithic architecture it can get quite overwhelming in a microservice architecture. The amount of interactions ramps up the quantity and complexity of the end-to-end tests that are used and can easily bloat your testing procedure and make it be very time-consuming. By having the consumer define these tests the testing team of a provider can focus on writing the service and the testing can be done mostly automatic. Furthermore the CDC tests can be run before the costly end-to-end tests and provide an earlier feedback whether or not an interface or an interaction with a service is broken without needing to start up the other service. (section 2.4 on page 9)

2. What are the differences between synchronous and asynchronous CDC testing?

The biggest difference between synchronous and asynchronous CDC testing is that the defined contracts change. Whilst there was both, an expected request and an expected response in the synchronous one, the expected request is completely omitted in the asynchronous contract. (chapter 3 on page 13)

This is because of the structure of an asynchronous exchange which really is not an exchange but a one-way interaction. The provider usually posts a notification/item containing new changes whenever an internal database changes. Other ways a provider could be triggered would include a request via REST without awaiting a result or posting a message to a messaging queue that the provider has subscribed

to. Because of this we cannot assume a default implementation as it was the case with a REST interface. (section 6 on page 37)

3. What tools are there to provide CDC testing capabilities for asynchronous communication between microservices?

There is a very limited selection of tools. Most noticeably are the open source "Pact" tool and the open source tool "Spring Cloud Contract". The Pact team even offers a platform to share the generated contracts between consumer and provider; the so called Pact broker. (4.2 on page 24)

4. Why do we use the Pact tool to provide CDC testing?

The two tools that were mentioned have a very different approach to contract testing as Pact lets the consumers define their contracts programmatically and Spring Cloud Contract works by having both parties (consumer and provider) agree to an already defined contract which they can test against. Spring Cloud Contract also only supports JVM-based languages whilst Pact supports a wide variety of languages. Pact also supports the idea of *consumer*-driven contract tests better. (section 4.2 on page 24)

5. How can we use this tool for CDC testing in an asynchronous environment and how does it differ from being used with synchronous communication between microservices?

In an effort to get the asynchronous workflow to the same level of usability as the synchronous implementation, it was tried to integrate the specific message queue in use (in this case Kafka) into the contract verification process of the provider. This is rather cumbersome as the team writing the tests has to focus on enhancing the tool to fit its needs instead of focussing on writing the tests. Pact does not offer support for specific messaging queues but instead a generic approach to messaging queues. (chapter 5 on page 27)

To make this easier in the future one could refactor the Kafka specific code into another class or have it be used via annotation processing. This way the workflow of creating contract verification tests for the provider could get easier. Another way would be to combine the capabilities of the two showcased tools as they are compatible in a limited way. Spring Cloud Contract offers specific integrations for messaging queues in Java. (chapter 6 on page 37)

List of Figures

List of Tables

Listings

Abbreviations

CDC	**C**onsumer-**d**riven **c**ontract
AMQP	**A**dvanced **M**essage **Q**ueuing **P**rotocol
CI	**C**ontinuous **I**ntegration
REST	**Re**presentational **S**tate **T**ransfer

Bibliography

[CDT18] Cerny, Tomas ; Donahoo, Michael J. ; Trnka, Michal: Contextual Understanding of Microservice Architecture: Current and Future Directions. In: *SIGAPP Appl. Comput. Rev.* 17 (2018), Januar, Nr. 4, 29–45. http://dx.doi.org/10.1145/3183628.3183631. – DOI 10.1145/3183628.3183631. – ISSN 1559–6915

[Cle14] Clemson, Toby: *Testing Strategies in a Microservice Architecture*. https://martinfowler.com/articles/microservice-testing. Version: Nov 2014. – Accessed: 2019-06-17

[Coh09] Cohn, Mike: *Succeeding with Agile: Software Development Using Scrum*. 1st. Addison-Wesley Professional, 2009. – ISBN 0321579364, 9780321579362

[ECM13] ECMA-404, Standard: The JSON data interchange format. (2013)

[JL14] James Lewis, Martin F.: *Microservices*. https://www.martinfowler.com/articles/microservices.html. Version: March 2014. – Accessed: 2019-10-16

[LMM19] Lehvä, Jyri ; Mäkitalo, Niko ; Mikkonen, Tommi: Consumer-Driven Contract Tests for Microservices: A Case Study. In: Franch, Xavier (Hrsg.) ; Männistö, Tomi (Hrsg.) ; Martínez-Fernández, Silverio (Hrsg.): *Product-Focused Software Process Improvement*. Cham : Springer International Publishing, 2019. – ISBN 978–3–030–35333–9, S. 497–512

[New15] Newman, Sam: *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc., 2015

[O'R17] O'Regan, Gerard: *Concise guide to software engineering*. Springer, 2017

[Rop16] Roper, James: *Rethinking REST in a Microservice world*. Recorded talk uploaded to youtube.com. https://youtu.be/NMmKSC794vo. Version: 2016. – Accessed: 2019-08-12

[Sel18] Selleby, Fredrik: *Creating a Framework for Consumer-Driven Contract Testing of Java APIs*. 2018

[var] VARIOUS: *Pact documentation*. git repository. `http://docs.pact.io`. – Accessed: 2019-08-14