Westfälische Hochschule

University of Applied Sciences
Gelsenkirchen Bocholt Recklinghausen

Master Thesis

Evaluation of Contract Testing in a Large Microservice System

Master of Science (M. Sc.)
Akademischer Abschlussgrad: Grad, Fachrichtung (Abkürzung) // Degree (Abbreviation)

Tim Vahlbrock, Bocholt
Verfasserin/Verfasser, Geburtsort // Author, Place of Birth

Praktische Informatik
Studiengang // Course of Study

Informatik und Kommunikation
Fachbereich // Department

Prof. Dr. rer. nat. Ulrike Griefahn
Erstprüferin/Erstprüfer // First Examiner

Prof. Dr. rer. nat. Tom Vierjahn
Zweitprüferin/Zweitprüfer // Second Examiner

6. September 2024
Abgabedatum // Date of Submission

master thesis

t. vahlbrock

Evaluation of Contract Testing in a Large Microservice System

Westfälische Hochschule

MASTER THESIS

Evaluation of Contract Testing in a Large Microservice System

EVALUATION VON CONTRACT TESTING IN EINEM GROSSEN MICROSERVICE SYSTEM

Tim Vahlbrock

6 September 2024

WESTPHALIAN UNIVERSITY OF APPLIED SCIENCES
GELSENKIRCHEN · BOCHOLT · RECKLINGHAUSEN

COMPUTER SCIENCE AND COMMUNICATION
CAMPUS GELSENKIRCHEN

ABSTRACT

The development of a system using a *microservice* architecture aims for an easier and more efficient scaling in modern cloud environments, when compared with a traditional monolithic architecture (Fowler 2014). This simplifies the testing of an individual microservice, but increases the effort and complexity of running *end to end tests* to verify the correctness of the system as a whole (Fowler 2014). *Contract testing* is an alternative approach to testing the interoperability of *microservices*, where the compatibility of *consumer* and *provider* with the shared interface separately from each other (Pact Foundation 2024c). It is advertised as providing as being faster and more stable than *end to end tests* by testing. This thesis evaluates *contract testing* as an alternative to *end to end testing* in a large *microservice* system by implementing *contract tests* for a section of a system and comparing relevant metrics between them. Based on the results it comes to the conclusion that while some hurdles still need to be overcome with further research, the benefits of *contract testing* outweigh it's shortcomings already, making it viable alternative to *end to end testing*.

ACKNOWLEDGEMENTS

I would like to thank the examiners of my thesis, Prof. Dr. rer. nat. Ulrike Griefahn and Prof. Dr. rer. nat. Tom Vierjahn, for taking the time supervise and examine this thesis. Furthermore, I would like to thank my employer for waiving a blocking notice for this work and making it publicly accessible in this way. Finally, I would like to thank my supervisor Stefan Schmeißer and the rest of my colleagues for providing the opportunities and environment needed to create this thesis.

CONTENTS

LIST OF FIGURES

LIST OF LISTINGS

1 INTRODUCTION

Modern cloud environments allow developers to replicate their software to handle changes in load (Fowler 2014). While this can be applied to a software following a traditional, monolithic architecture, this results in all parts of the software being replicated equally. This means that even parts that do not experience as much load, are replicated as often as the ones experiencing the most. The alternative is to split the application into so called *microservices*, allowing sections of the software to be scaled individually. Like any other system, these can be tested using automated tests (Vocke 2018). These do not only prevent releasing or deploying defective versions of a software, but can also be used as a feedback mechanism for the developers. Given a certain coverage and quality of tests, developers can be confident that their changes are functional and did not break other parts of the system, as long as the test suite executes successfully. Tests can be implemented on different levels, covering different amounts of code (Vocke 2018). Tests on low levels cover few lines of code and do not leave the runtime environment, making them fast in execution. Accordingly they can be run frequently even during implementation. To verify interoperability between different system components, like *microservices*, tests on higher levels, like *end to end tests* can be used. As these cover larger sections of code and are likely to include network communication and database access they take longer to execute. In consequence they are run less frequent and late in the development process, giving little and late feedback to developers.

Contract testing is a concept to reduce the reliance on higher testing levels by testing compatibility with the interface at each *microservice* individually (Robinson 2006). The performed requests and send messages are recorded on the client side of the interface and used to create a *contract* that describes the behavior of the interface. The recorded *interactions* are then replayed on the server side at a different point of time and the responses verified to match what is expected by the *consumer*. By establishing the *contract* as a test artifact, the test on the *consumer* and *provider* side can be decoupled from each other both temporarily and technically. The goal of *contract testing* is to reduce or eliminate the amount high level tests like *end to end tests* and therefore the impact of the disadvantages those kinds of tests have. This thesis evaluates, whether *contract testing* is a viable alternative to *end to end*

testing in large *microservice* systems. This is done by performing a reference implementation on an existing, large *microservice* system, evaluating the result and drawing corresponding conclusions for *microservice* systems in general.

The following Chapter 2 describes existing research on *contract testing* and solutions to ensure interface compatibility. The fundamentals of this thesis are elaborated on in Chapter 3, by introducing core concepts of *microservices*, testing, and *contract testing*. Chapter 4 describes the evaluation and implementation concept. This is followed by the description of the implementation in Chapter 5. The results of the implementation are presented in Chapter 6. They are discussed in Chapter 7 and a conclusion is drawn. Chapter 8 gives an outlook given into future development and remaining research.

2

RELATED WORK

The verification of interface compatibility has gained relevance with the adoption of *microservice* architecture due to the increase in interfaces. Nevertheless, the underlying problem has persisted in any form of distributed system before. Thus, existing research deals with this topic and tooling targeting to mitigate this exists. This chapter summaries the state of research and technology and summarizes how *contract testing* and this thesis differentiate from it.

2.1 ORIGINAL CONCEPT

The term *contract testing* was first used by Groß et al (2003) as a form to test component interoperability. The paper proposes integrating tests into component based systems to verify at runtime that the environment of a component matches its expectations. Groß et al recommend this pattern for dynamic and rapidly changing systems, like internet applications. This shows, that the necessity of testing interface compatibility is not a new problem and approaches to implement this have existed longer than recent trends. The described approach assures compatibility at runtime, while modern *contract testing* as discussed in this thesis verifies compatibility during development and before deployment. The underlying motivations are similar, but this difference in the point of execution has a major impact on how compatibility is determined and the associated possibilities and limitations of the approach.

2.2 EXISTING CASE STUDIES

Multiple authors have described case studies, in which *contract testing* was applied to software projects. However, these case studies apply *contract testing* to newly developed sub systems or do not perform an comprehensive evaluation of the implemented solution.

Hernandez (2023) evaluates the usage of *contract testing* on a *microservice* system, but only implemented a proof of concept for *REST*ful interfaces and performed evaluation through surveys. Furthermore, the evaluation was limited to a specific system and no conclusions were drawn on the

applicability of *contract testing* to large *microservice* systems in general. This thesis deals with the application of *contract testing* to a case study as well, but draws conclusions on the applicability to other large *microservice* systems. Additionally, the implementation is not limited to *REST*ful interfaces but also considers *event driven messaging*. The evaluation is performed based on numeric metrics, as surveys might include bias and require a big enough number of participants to be representative.

Both Vu and Labuda (2022; 2019) apply *contract testing* to an existing system, but do not perform a comparison to alternative approaches. Instead, guidelines on the adoption of *contract testing* into an existing system are proposed. The methods and practices used in this thesis influence the evaluation result and may therefore be referred to as guidelines. However, the main focus is on the viability of *contract testing*, compared to alternative approaches to ensure interface compatibility.

Lehvae et al (2019) perform an evaluation of *contract testing* based on its ability to capture intentionally introduced defects. While this metric is relevant for a complete evaluation, other aspects need to be considered as well. This thesis introduces multiple requirements and metrics to determine the viability of *contract testing*.

Nagel et al (2019) apply *contract testing* to an example project and perform a deeper analysis on the concepts used in *contract testing*. An evaluation of *contract testing* against alternatives is not performed. While the concepts used in *contract testing* are relevant for this thesis, its focus is on the viability of *contract testing* compared to alternative approaches to ensure interface compatibility.

2.3 ALTERNATIVE CONCEPTS

Besides *contract testing*, other forms of validating interfaces exist. Tools like *OpenAPI* (The Linux Foundation 2024) allow to specify the types of an interface statically and share them among the *participants* of the interface. The interface specification can be used to generate the client or server code, but can also itself be generated from existing code instead. As these tools operate statically, they can verify the syntactic interface compatibility, but they cannot verify the semantic compatibility. However, they can be used along side other methods to verify compatibility, like *end to end testing*. Similar tools exist to verify the usage of correct types at runtime, like the Hibernate Validator (Hibernate 2024). As incompatibilities can only be caught by detecting malformed requests or responses, these dynamic tools can only be used in combination with *end to end testing*, but not as a replacement. This thesis evaluates whether *contract testing* can be used to not only ensure static, but also semantic compatibility and therefore be an alternative to *end to end testing*.

2.4 SUMMARY

The term *contract testing* has been introduced by Groß et al. While the way modern *contract testing* is used deviates from the original description, the

motivations still apply. Existing case studies are not sufficient to perform a comprehensive evaluation of *contract testing*. Those that do perform an evaluation do not perform a through comparison of *contract testing* to alternative approaches. Alternative approaches to verify interface compatibility exist, but perform validation at a different scope or cannot remove reliance on *end to end testing*.

3 FUNDAMENTALS

This chapter introduces technical terms and essential principles to give an overview of the field covered in this thesis. The concepts of testing, *microservice* architecture, *event driven messaging* and *contract testing* are described in further detail.

3.1 TESTING

To verify correct functioning of a software, it needs to be tested. Automated testing allows to do this without requiring human intervention. To be able to evaluate *contract testing* against other forms of testing, basic testing concepts need to be introduced. This includes the motivation of testing, different testing levels, the testing pyramid, mocking and *test data builders*.

3.1.1 *Motivation*

Traditionally, software products were released once and upcoming features and improvements were marketed as a new, separate product. The ability to distribute software over the internet has made the effort to provide updates so low, that most software is updated multiple times a month (DeBellis and Harvey 2023). Based on this, concepts like *continuous deployment* (Fowler 2013) have emerged, making software changes available as soon as they are implemented. As in this update strategy the user often does not have the possibility to defer updates to a later point, it is even more essential to assure the correct functionality of the software. Doing this with automated tests has proven to be more accurate, faster and reliable than testing manually (Dobles, Martínez, and Quesada-López 2019). In consequence, manual testing is not discussed further. Unless explicitly stated, further references to *Tests* or *Testing* refer to automated tests or automated testing respectively.

Automated tests can provide a fast and simple way for developers to get feedback on whether their changes broke parts of the software. This is not only helpful when developing new functionality, but also when performing refactorings and other improvements. Being able to perform these changes and rely on the tests to verify them allows developers to keep the software

aligned with updated requirements while maintaining good code quality (Jína 2013). Development styles like *Test Driven Development* integrate testing deeply into the implementation process, allowing an initial proof of concept implementation, which is refactored to a well readable and maintainable state afterwards. During refactoring, the repeated execution of tests is used to verify that nothing broke.

3.1.2 *Testing Levels*

To test different levels of a software system, different kinds of tests are used, that each come with different advantages and disadvantages. The most widespread form of categorization divides tests into *unit tests*, *integration tests* and *end to end tests* (Vocke 2018). *Unit tests* are usually written in the same language as the tested code and invoke the classes, methods and functions of a delimitable section of code, referred to as the *unit*, directly. Each *unit test* typically only tests a few tens of lines of code and creates a new instance of the unit to isolate the tests from each other. *Unit tests* usually are able to provide the developers with fast feedback on the correctness of the code, as the direct invocation of source code and isolation makes them very fast.

To verify that multiple units correctly work together *integration tests* are used. The boundaries of which units are or are not tested together in the *integration tests* can usually be drawn in multiple ways. *Integration tests* are usually written in the same language as the source code they are testing. Depending on the amount of code that is tested and additional setup and teardown that is required to get a reproducible test state, *integration tests* are usually somewhat slower than *unit tests*.

End to end tests verify, whether a system as a whole is correctly functioning. It requires the instantiation of all system components, including other services and databases. To prevent test cases from influencing each other, the system needs to be reset to the same state before each test. This becomes harder to achieve the bigger the larger the tested system gets. As *end to end tests* often include interactions with the graphical user interface, they are usually written using specific languages or syntax to be able to describe the simulated user behavior.

In many cases a set of tests cannot be assigned to a single one of the three levels above. Instead, it shares attributes of multiple test levels. This may depend on what is possible with the chosen architecture of the component under test, the design of the test suite and which test levels are well usable in the chosen development process. A lot of the differentiation depends on where the boundaries of a test are drawn, so at which point in the invocation calls or other components are mocked. The larger a system gets, the more possibilities emerge to draw boundaries and the more different kinds of tests are required. Both can also be applied to different kinds of tests within a single testing level, which often results in the introduction of domain specific terminology for different testing suites. For example, one set of *integration tests* might test the integration of different components with each other and therefore be referred to as *Unit Integration Tests* or *Component Integration Tests* whereas another suite of tests verifies the correct integration of multiple

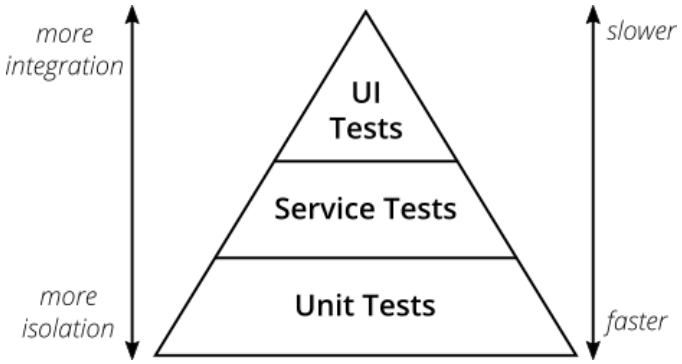

Figure 3.1: The Testing Pyramid according to Cohn (2010), Illustration by Vocke (2018)

```
whenever(
→userService
→→.getUserById("9a9845c4-8a46-4bf9-89a5-ecd3c3b66479")
).thenReturn(sampleUser)
```

Listing 1: Mocking Example in Kotlin

services with each other and therefore be refereed to as *Service Integration Tests*.

3.1.3 *Testing Pyramid*

Tests can not only used be to verify the software before releasing, but can also used as a continuous feedback tool during development. This is only practical, when the feedback is delivered fast, so if the tests are completed quickly and early in development. As *integration* and *end to end tests* are slower than *unit tests*, the amount of tests should decrease with the height of the test level. Cohn (2010) visualizes this concept in form of the testing pyramid, as shown in Figure 3.1. The vertical axes of the pyramid is resembling the amount of code covered by the individual tests of each layer, while the horizontal axes is representing the amount of tests that should be used. Vocke (2018) remarks that the testing pyramid uses different, less accurate terms for some of the testing layers and is overly simplistic. Nevertheless Vocke agrees with the general concept of using multiple testing levels and keeping the amount of tests at higher testing levels low.

3.1.4 *Mocking*

The parts of a system that are tested in different tests and testing layers interact in most cases with other systems or system parts. To be able to test different scenarios and avoid having to manage system parts that are not under test, they are replaced by *mocks* (Fowler 2007). These are placeholder implementations, that can be configured to behave in a specific way and asked, whether certain invocations happened. Depending on the testing

```
class ExampleUsers {
→fun defaultUser(): User {
→→return User(
→→→id = "9a9845c4-8a46-4bf9-89a5-ecd3c3b66479"
→→→firstName = "Jane",
→→→lastName = "Appleseed",
→→→groups = []
→→)
→}
}
```

Listing 2: Object Mother Example in Kotlin

level, *mocks* may have a different form. In *unit testing*, *mocks* are used as replacements for instances of other classes. The creation of the *mock* can be simplified by using a mocking library. Listing 1 shows, how a *mock* for the `userService`, created using the *mockito-kotlin* library (Faber et al. 2024) can be configured to return `sampleUser`, when `getUserById` is called with a certain id. In other testing layers, *mocks* might take a different form. In *contract testing* for example, the requests are send to a local *mock server*, instead of the real one. Similar to the earlier example, it needs to be instructed which response to return if a certain request is received.

3.1.5 *Test Data Builders*

Tests often need instances of specific data classes. These may require a set of constructor arguments to be passed, regardless of whether the values are used in the test or not. This makes tests less readable and less expressive, as they contain information that are not specific to the test case. The creation of test data can be simplified to a certain degree by defining an object mother, a class that contains factory methods to instantiate certain variations of the required class (Schuh and Punke 2001). An example for such a class is shown in Listing 2. Defining object mothers and factory functions makes tests more readable and therefore maintainable, as the creation code is removed from the test file and can be referenced using a name that summarizes its content. Furthermore, each of the factory methods can be used by multiple tests, reducing duplication. However, instances used in the different tests often only differentiate through little changes in the field values. If the field in question is not final, it can be overridden after calling the factory method. If the field is final, each variation needs a new method or requires parameters on the factory method. This is especially relevant if the class to create follows the *Immutable Value Object* pattern, in which all fields are passed to the constructor, but are final (Freeman, Steve and Pryce, Nat 2010, pp. 59-60, 141, 151; Fowler 2016).

Freeman and Pryce (2010, pp. 258–262) propose the alternative approach of using *test data builders*. For each relevant test data class a builder class is created, that implements the *Builder* pattern, but specifies a sane default for each field. To instantiate the class, the test creates an instance of the

```
1 val johnWithFriendJane = aUser()
2 →.named("John")
3 →.hasFriend(aUser().named("Jane"))
4 →.build()
```

Listing 3: TestDataBuilder Usage Example in Kotlin

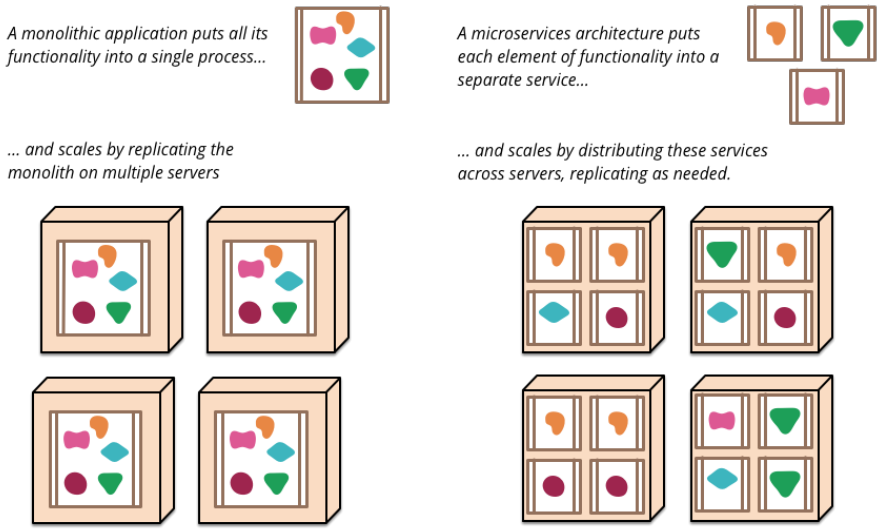

Figure 3.2: Monoliths and Microservices (Fowler 2014)

builder class and calls the `build` method. A test that needs to override one of the values can invoke the appropriate method without having to specify values for all other fields. The ability to only specify the fields that are relevant for this test case makes the test not only well readable, but also very expressive, as the difference to other test cases is explicitly stated. Furthermore, similar instances can easily be created by invoking `build` once, overriding the differing values and calling `build` again.

The readability of the test data creation can further be improved in multiple ways, making the *test data builders* usable as shown in Listing 3. By defining factory functions like `aUser` in Line 1 for the builder classes, noise can be removed from the test data creation and the syntax of test data definition brought closer to natural language. For fields of a non primitive type, additional override methods can be defined. These do not receive an instance of the type, but rather the corresponding builder. This way, the call to the `build` method of the passed builder can be removed from the test and instead be called in the additional override method. In Listing 3 this is shown in Line 3. The override of the `friend` field uses the `hasFriend` method, but does not need to call `build`.

3.2 MICROSERVICE ARCHITECTURE

In the monolithic approach to software architecture all functionality of a software is provided by a single process. With the continuous increase of

software deployments to cloud infrastructure (Gartner 2023), applications need to be able to scale well in this environment. As visualized in Figure 3.2 on the left, monolithic services are scaled horizontally, by replicating the entire process onto multiple servers. Communication between functionalities remains simple, as they are still contained within the same process. However, this assumes that all functionalities require equal replication. Even if a functionality sees little to no use, it is replicated as often as the most used functionality. In consequence, either fewer replicas can be run on the same budget or more money needs to be spend to maintain a set of replicas even though they are not fully utilized.

The concept of the *microservice* architecture offers an alternative to this. It applies the *Single Responsibility Principle* (Martin 2014) to the application as a whole and splits the application into services, with one service for each functionality (Fowler 2014). The communication between these services is then performed by inter process or network communication. This allows to scale functionalities individually as shown in Figure 3.2 on the right. Additionally, the development of the services can be spread easier onto multiple development teams, as the code for the smaller services is easier to oversee and dependencies on other system parts are communicated more clearly as concrete interfaces. There is no concrete definition of how small a *microservice* should be. The phrase *micro* in the name refers to the service having the smallest interface as possible, rather than the size of the source code (Newman, Sam 2019).

Traditionally, any software making requests to other software is referred to as a *Client*, while software handling requests is referred to as a *Server*. However, these terms do not apply well to *microservice* based systems, as services also perform communication between each other. The terms of *consumer* and *provider* can be used to describe the two sides of an interface more precisely. Each component that provides a service on an interface is described as the *provider* of the interface. Every component that uses the service provided on the interface is described as a *consumer* of the interface. As the term is always tied to a specific interface, a component often is a *provider* on one interface and a *consumer* on another. For example, a *HTTP* Server can be a *provider* of a *REST API* that serves aggregated data, but at the same time act as a *consumer* on other interfaces to acquire said data. By using these terms instead of *Client* and *Server* it is communicated more clearly that the term describes the role of a service on a specific interface.

A major hurdle in the development of *microservice* systems is the increased difficulty of testing interface compatibility (Newman, Sam 2019; Smith et al. 2023). In the monolithic architecture interfaces between system components are function and method calls that can be tested using larger *integration tests*. In *microservice* architecture those interfaces are based on network or inter process communication. Therefore, to be able to test the interaction with each other, multiple services need to be instantiated. However, having to test the services together contradicts the core concept of being able to develop the services independently of each other.

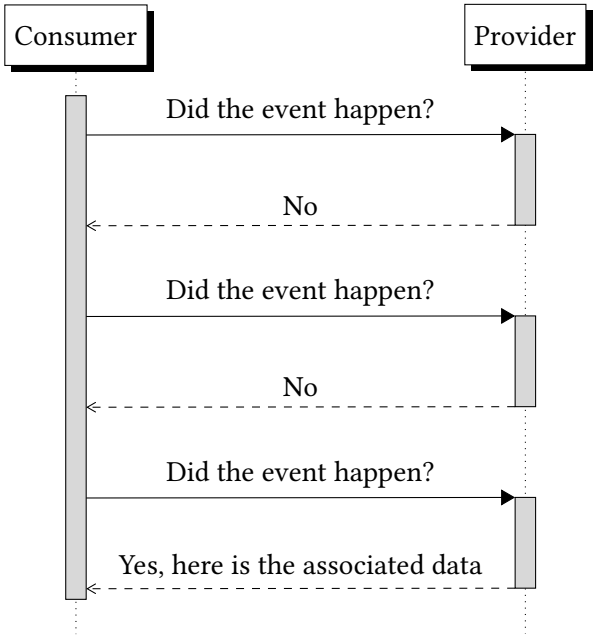

Figure 3.3: Polling for Events

3.3 EVENT-DRIVEN MESSAGING

In *microservice* systems a *consumer* might be interested in specific events that occur on the *provider*, for example if a long running task it has been waiting for was completed (Khattak and Narayanan 2010). It could get informed on this if it keeps polling the *provider* like shown in Figure 3.3. The repeated requests would consume computing resources on both *consumer* and *provider* side, without providing any immediate value to the computation. If the event occurs just after the last poll, the *consumer* is still only verified when it polls the next time. Increasing the polling frequency would reduce this latency, but waste further resources.

Event driven messaging offers an alternative approach based on the *observer* design pattern (Khattak and Narayanan 2010). *Consumers* can register themselves as *observers* of a specific event and unregister when they are no longer interested in the event, like shown in Figure 3.4. Instead of the *consumer* initiating a request to ask for event updates, the *provider* performs requests to all interested *consumers* whenever the event occurs. In *microservices* systems, multiple instances of a *provider* may exist and *consumers* would have to register on each of the *providers* to be notified of all event occurrences. To prevent this unnecessary overhead, *broker services* are used that are responsible for managing the message exchange between *providers* and *consumers*. Instead of registering their interest in a *provider* event, *consumers* register their interest on the *broker*. Similarly, the *providers* notify the *broker* of the event, which then carries out the distribution to the interested *consumers*. This way, *providers* no longer need to keep track of the subscribed *consumers* themselves and *consumers* do not need to keep track of and subscribe to all available *providers*. Additionally, the *broker* is in many implementation able

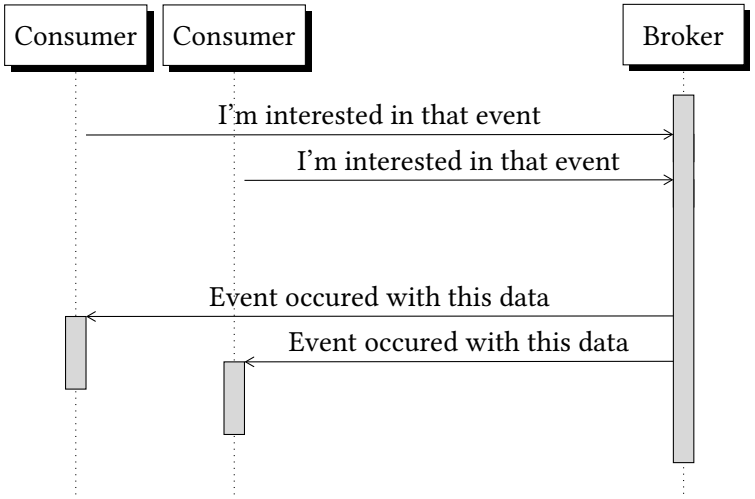

Figure 3.4: Subscribing to Events

to retain the messages until the *consumer* is ready to process them, preventing errors caused by resource exhaustion in the *consumer*.

This ability can also be used to run the system in a different mode of operation. While in the previously described operation mode events are delivered to all subscribed *consumers*, brokers might also be set up to deliver the only to the first *consumer* ready to process it, as shown in Figure 3.5. Instead of sharing information with multiple other *participants*, this operation mode is used to perform more complex tasks that need some time to be processed and should therefore be performed in separate services to prevent blocking other service features.

3.4 CONTRACT TESTING

Most of the main disadvantages of *end to end testing* originate in having to spin up the system as a whole. It requires that a configuration is created that makes the system and tests not only stable, but also reproducible. Due to the amount of code and infrastructure tested, the tests take long to complete and are therefore executed infrequent and late. *Contract testing* aims to resolve these issues by verifying interface compatibility for each *participant* individually, as shown in Figure 3.6. In the first step, the *consumer* of an interface tests its behavior against a mock of the *provider*. Based on this, a *contract* is generated that is made available to the *provider*, which can use it to verify that its own behavior matches the one described in the contract. In perspective to the testing pyramid, *contract testing* tries to reduce the amount of slow, complex, high level tests at the top, by moving the interface verification to the fast, simple, low level ones at the bottom.

To perform *contract testing* for a *REST* interface, tests are written that execute the code responsible for performing the network requests. The network requests are not performed against an instance of the the real provider, but against a local *mock server*. The responses the *mock server* should use are configured within the tests. The *contract* is then generated as a collection of the captured request response pairs referred to as *interactions*. Depending

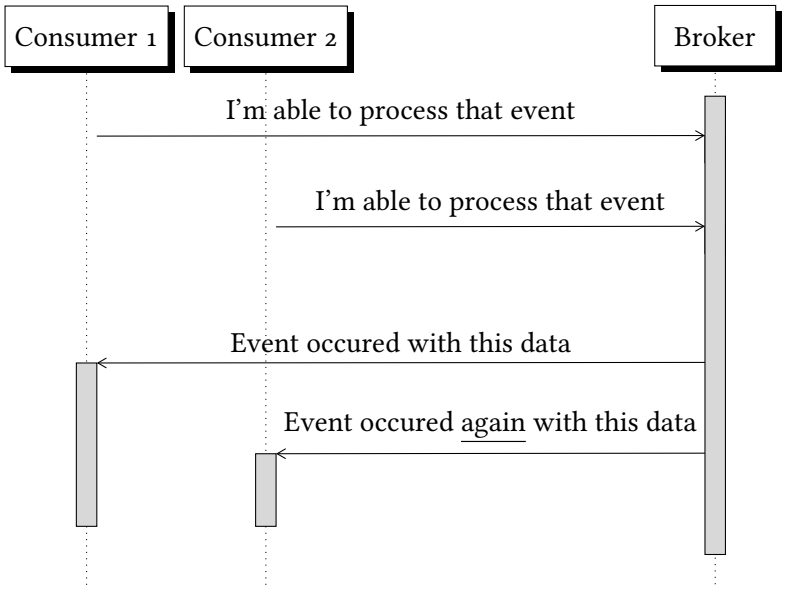

Figure 3.5: Consumers as Workers

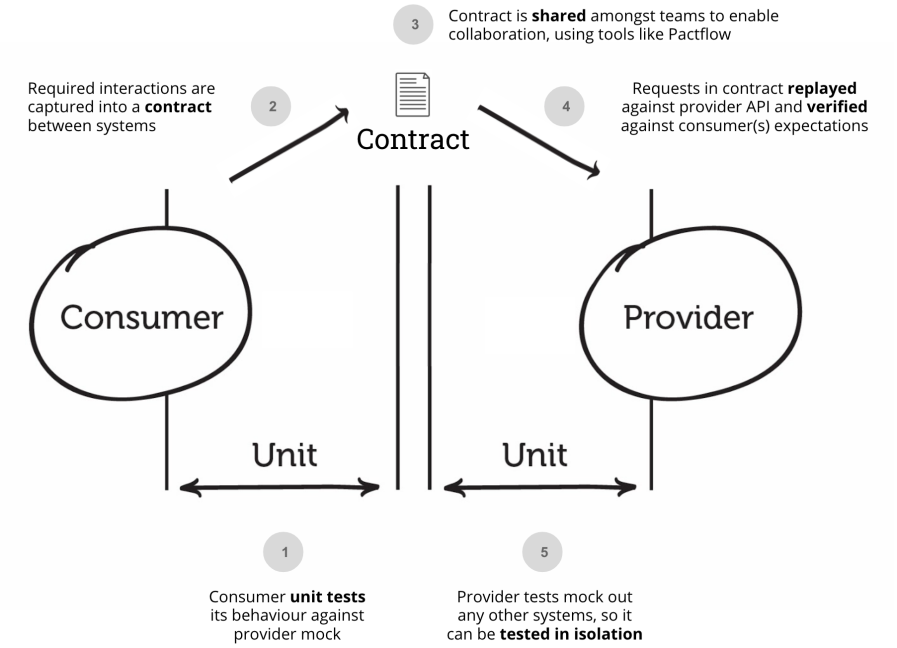

Figure 3.6: How [Consumer Driven] Contract Testing Works (Pact Foundation 2022)

on the implementation, each *interaction* is supplemented with additional information required to further process the *contract*, for example a human readable description of the tested scenario. As no network communication and system setup is required the tests can be executed within a *unit test* runtime. The generated *contracts* are made available to the *provider* repository. This can be done by copying the files, but some *contract testing* frameworks offer a *contract broker* to simplify the exchange. On the *provider* project, all *contracts* for the *provider* can then be used to verify, that the *provider* implementation complies with the *contracts*, by replaying the requests originally made to the mocked interface and comparing the returned responses with the ones expected by the *consumer*.

Contract testing can be performed on interfaces using *event driven messaging* as well. In *event driven messaging* the communication is initiated by the *provider*. As the *consumer* does not sent anything, the interactions cannot be captured using a *mock server*. Instead, a sample message is described in the test and passed to the *consumer's* parsing algorithm, to confirm that this message format is indeed parsable by the *consumer*. To verify that the *provider* generates a message with the same format, a *mock server* could be used to mock the *broker* or the *consumer*. However, *event driven messaging* implementations do not share a common base technology, like *HTTP* being used for *REST* interfaces, a different *mock server* for each implementation would be required. Instead, traditional method mocking is used to capture the generated message shortly before it would have been transmitted.

The described form of *contract testing* is referred to as *consumer driven contract testing*, as the *contract* defining the interface behavior is generated by the *consumer* and verified on the *provider*. The verification on the *provider* can be performed somewhat automatically, because the requests recorded during the *consumer* tests can be replayed to trigger the desired responses on the *provider*. *Consumer driven contract testing* has the benefit that the *consumers* of an interface can state their requirements using *contracts*. The *provider* then knows exactly, which endpoints and fields need to be implemented and unnecessary implementation is avoided. Furthermore the *provider* can use the current versions of the *contracts* to determine which fields and endpoints are in use, which are obsolete, and which *consumers* need to be notified of a breaking change.

The linguistic opposite of *consumer driven contract testing* would be *provider driven contract testing*, which would require the verification to be performed on the *consumer* side using the *contracts* generated on the *provider* side. However, as it is not possible to generate the required function calls to trigger the requests based on an existing *contract*, *provider driven contract testing* is not possible. Instead, there is *bidirectional contract testing* in wich both *participants* generate their own versions of the *contract*, which are then compared for compatibility. This has the benefit that the *provider* does not need to wait for *contracts* to be generated by the *consumer*. On the *provider* side, the *contract* is generated using static tools like *OpenAPI* (The Linux Foundation 2024). Additional dynamic tooling is needed to verify that the *provider* behaves at runtime as the *contract* describes. If these tools are already in place, *bidirectional contract testing* allows for a faster adoption of

contract testing, as no further implementation is required on the *provider* side.

3.5 SUMMARY

A well designed and integrated suite of automated tests provides the ability to repeatedly release reliable software in a viable way. This is especially important for *microservices*, as the services are developed and released independently of each other. The separation of an application into *microservices* allows for easier scaling in cloud environments and separation of concerns. To be able to communicate events through a *microservice* system, *event driven messaging* can be used. *Contract testing* resembles a way of testing the compatibilities of the communication interfaces. In contrast to *end to end testing*, the compatibility is determined on the *consumer* and *provider* separately. In *consumer driven contract testing* the *contract* is recorded on th *consumer* side and verified on the provider.

4 CONCEPT

This chapter describes the concept used to evaluate *contract testing*. It describes the requirements *contract testing* needs to fulfill to be a viable alternative. Based on these, metrics are presented used to evaluate whether *contract testing* fulfills the requirements. Furthermore the case study project is described, on which *contract testing* is implemented to be able to record the metrics. Finally, the framework chosen to simplify the implementation of *contract testing* and its base concepts are introduced.

4.1 REQUIREMENTS

To be a viable testing strategy, and especially a viable alternative to other high level testing strategies, *contract testing* needs to fulfill certain requirements. As the comparison with multiple higher level testing strategies would go beyond the scope of this thesis, the comparison is performed against *end to end testing* as the highest level testing strategy. Findings and arguments can be often be used in similar form with other high level testing strategies. The following sections describe the core requirements, why they are important, and to what extend *end to end tests* and *contract tests* are expected fulfil them.

4.1.1 *Feedback Speed*

Early feedback allows developers to revisit their implementations soon after the implementation to improve them. It also allows developers to practice techniques like *test driven development*, in which tests are used in the iterative improvement of the implementation to verify the validity of the most recent batch of changes. Tests that are slow do not only delay the feedback by being slow themselves. Because they are slow, they are likely to be executed late in the development process and infrequent, for example in a pipeline job that runs only once a day. Developers might then need to revisit an implementation when they already started implementing other tasks.

End to end tests run through the whole system and therefore also include time intensive operations like network transfer, database access and interactions with the user interface. These make *end to end tests* slower than other

testing methods. Besides this, *end to end tests* require the setup, reset and teardown of a test environment, slowing the testing even further. Depending on the setup, developers may not be able to easily execute the *end to end tests* locally.

Contract tests are marketed as faster, as they do not leave the *microservice* and therefore can be run on *unit test* level. The latter also enables a quick and local execution of the *contract tests*, including debugging. This would drastically increase the speed of feedback loops, as developers would no longer receive failure reports after minutes, hours or even days from the development or release pipeline, but can rather execute the tests locally and receive feedback within seconds.

4.1.2 *Stability*

Tests need to be stable to be a reliable development tool. Tests that often fail unnecessarily do not only waste time, but can also no longer be used to assess, whether a change actually broke something. The more often the tests fail unnecessarily, the more likely it is that a failing test result does not represent an actual failure, but rather an instability in the test suite. This may eventually lead to the tests being removed from the pipeline or the tests no longer being enforced to succeed before merging or deploying. It is then no longer noticed, if the tests catch an actual bug in the implementation.

Testing the system as a whole requires a setup of a system instance to be set up for each test run, ideally for each individual test. At the same time, the system under test should be as similar to the production system as possible. This can make *end to end tests* brittle and fail due to environmental issues, rather than actual system faults. *Contract testing* tools suggest, that the execution in the runtime is more stable, as far less environmental setup is required.

4.1.3 *Development & Maintenance Effort*

To be more efficient than testing the software manually, tests need to be developable and maintainable with a reasonable effort. This effort may be reduced by applying best practices to the test setup. Moreover, supporting functionality can be implemented to make the individual tests not only easier to write and read, but also more comprehensive and expressive. Furthermore, the speed at which the required tests can be developed is also influenced by the time it takes to run the tests.

End to end tests are often written using frameworks that rely on a different language or syntactical structure than the production code or other test levels. This slows the development of *end to end tests*, as developers need to develop and maintain the sufficient knowledge on how the tests are written. While *contract tests* might be written in the same language as the production code, they also add new functions and tooling developers need to get familiar with and therefore may increase the time required to write them.

4.1.4 *Defect Detection*

The main goal of all testing is to catch bugs before releasing the software. *End to end testing* verifies, that the services, having been developed independently, are capable to work with each other and to fulfill the required use case. To be able to replace or reduce *end to end testing*, *contract testing* needs to be sufficiently good at catching incompatibilities. Besides component compatibility, *end to end testing* is also able to test use cases and workflows as a whole. While *Contract testing* is unlikely to catch defects in these areas.

4.2 METRICS

Based on the requirements described before, metrics were chosen, which are described in this chapter. These are used to evaluate, whether *contract testing* fulfills the requirements to be a viable alternative to *end to end testing*.

4.2.1 *Feedback Speed*

The feedback speed is measured by checking whether the *contract tests* can easily be run locally and by comparing the execution time of the *contract tests* with the execution time of the *end to end tests*. For reproducibility and accuracy, the execution time is measured by averaging the execution time of the pipeline jobs.

4.2.2 *Stability*

The stability of *end to end testing* and *contract testing* is measured by comparing the share of false-negative *contract testing* jobs in the pipeline with the share of false-negative end-to-end testing jobs in the pipeline.

4.2.3 *Development & Maintenance Effort*

The development and maintenance effort is difficult to measure in an objective way, as this would require to accurately measure how long developers take to develop the specific kind of test. This is possible, but would require accurate and disciplined time tracking by the developers and would also need to observe whether developers get faster at writing tests over time. As the required effort exceeds the capacity of this study, this metric is evaluated in an argumentative way instead, based on the observations made during the implementation.

4.2.4 *Defect Detection*

This metric is measured by comparing the share of *end to end testing* and *contract testing* pipeline jobs that caught incompatibilities and bugs, including the severity of the caught defects. The recorded jobs resemble the true negative pipeline jobs. The false-negative jobs are considered in the Stability

evaluation. False positive jobs are hard to detect if neither test strategy detects the defect. However, as this also means that in that case neither test strategy has a relevant advantage over the other. Therefore false as well as true positive jobs are not considered within the evaluation on their own, but still have an effect on the share false negative and true negative jobs have.

4.3 CASE STUDY

To be able to record the metrics, *contract testing* needs to be implemented on a case study system. To include all aspects relevant for a production grade system, the evaluation is performed on a real *microservice* system, that is targeting productive operation. The system architecture, scale and the used technologies are common across large *microservice* systems and make the case study system a fitting example to perform the evaluation on. Implementing *contract testing* for the whole system would not only be impossible within the time frame available for this thesis, but would also defy performing an evaluation of the viability in the first place. Instead *contract testing* is implemented for a representative subset of the *microservice* interfaces. Similarly, some additional functionality associated with *contract testing* might not be implemented in the scope of this thesis, if it is not required for the evaluation. To assure this partial implementation is an accurate representation of the whole system, the subsystem includes each kind of interface *participant* from the full system. *Consumers* and *providers* can be written in a variety of languages, each potentially requiring own setup and tooling. Consequently, the subset was chosen to contain multiple languages and frameworks to be able to perform a framework and language independent evaluation. By implementing *contract testing* for this subsystem, a confident assumption can be done regarding the effort, benefits and drawbacks for the system as a whole. For the implemented *contract tests* and the existing *end to end tests* the metrics are recorded and compared. These are used to determine whether *contract testing* is a viable alternative to *end to end testing*. The extension of *contract testing* to the rest of the case study system might be performed after the thesis, based on the evaluation result.

The subsystem, as well as the case study system as a whole are subject to confidentiality. To still allow for a public access to this work, the purpose and design of the case study system are not described further. Instead, the following example representation is used as a replacement, that uses the same structure and technology as the subsystem the original implementation and evaluation was performed on. All findings of this thesis have originally been found in the implementation on the original system and only been reproduced on the representation. To allow for better readability and the possibility to execute the representation code, it has been immutably archived at Vahlbrock 2024. Any listings using code from this representation implementation have their line numbers aligned with the full files. As shown in Figure 4.1, the subsystem consists of three services to manage the users in the system. The `User-Service` manages the access to the users of the system. These can be shown using the *GUI*. The `User-Service` manages existing users. The creation of new users is published using *event driven*

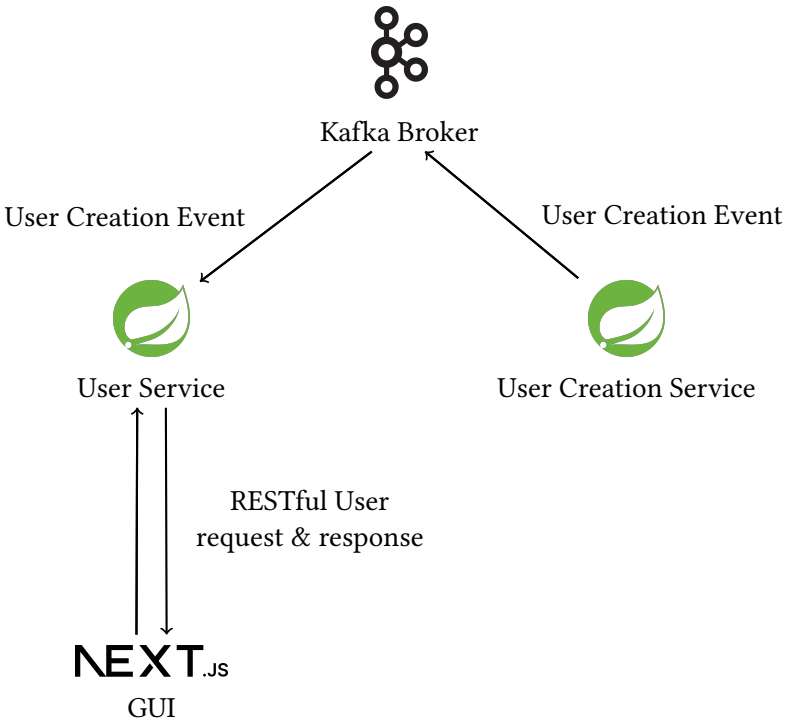

Figure 4.1: Case Study System Structure

messaging by the `User-Creation-Service`. The *GUI* is a *Next.js* application, written in *TypeScript*. Both `User-Service` and `User-Creation-Service` are *Spring Boot* applications written in *Kotlin*. While the *GUI* communicates with the `User-Service` using a *REST*ful *HTTP* interface, the `User-Creation-Service` publishes its events using a Apache Kafka *broker*, which are then consumed by the `User-Service` for processing. The interfaces are described using *OpenAPI* specifications, which are also used to generate *consumer* and *provider* code, as well as the classes used in the interface. The services are deployed on *Kubernetes* in production, as well as development environments. Version control is performed using *Git* with *GitLab*, which also provides the pipeline integration and merge requests used in the development process. The system uses several layers of testing to secure software quality and correctness, including *unit*, *integration* and *end to end tests*.

4.4 CHOICE OF FRAMEWORK

The implementation of *contract tests* without a corresponding framework is possible. The required mechanics can be implemented specifically for a certain system. However, the usage of a framework reduces the implementation and maintenance effort and allows the reliance on proven best practices, as well as usage of existing and future tooling.

To determine a fitting framework it needs to be decided whether a *consumer driven* or *bidirectional* approach should be used. While the existing *OpenAPI* specifications in the case study could have been used for *bidirectional contract testing*, no tools were in place that verified, that the *provider* implementations

match the specifications at runtime. Additionally, *consumer driven contract testing* is promoted with the additional benefit that the *contract* precisely describes, which information is needed by the *consumer* and prevent unnecessary additional implementations in the *provider*. Consequently a *consumer driven* approach was chosen.

The availability of frameworks for *contract testing* is rather limited. Furthermore, the framework needs to support *Kotlin* and *TypeScript*, as well as *event driven messaging* with *Kafka* to be applicable to the case study. The most common frameworks are *Spring Cloud Contract* (Broadcom 2024) and *Pact* (Pact Foundation 2023b). With *Pact* a framework was chosen that has native *Kotlin* and *TypeScript* support, as well as *broker* independent *event driven messaging* support. Aside from *Kotlin* and *TypeScript*, *Pact* offers libraries for several other languages. In contrast to other frameworks, the usage of *consumer driven contract testing* is free with *Pact*. Some of the described formats and procedures are tailored to the usage of *Pact* for this case study, but are used in other *contract testing* frameworks in a similar form.

4.5 SUMMARY

The specified metrics, chosen based on the requirements *contract testing* needs to fulfil, can be used to compare *contract testing* to the existing *end to end tests*. To keep the implementation effort to a reasonable level, while still allowing a comprehensive evaluation, a representative subsection of the case study system was chosen, on which *contract testing* should be implemented for the evaluation. As the case study system itself is subject to confidentiality, a representative example is used to be able to illustrate the implementation and relevant findings. To support the implementation, *Pact* has been chosen as the most fitting *contract testing* framework, as it supports the required languages and frameworks well.

5 IMPLEMENTATION

To be able to record metrics on *contract testing*, it is implemented on a case study project. The following chapter describes, how contract testing was implemented for the different participants, which supporting implementations were made, and which other implementations needed to be made to use *contract testing* in the project.

5.1 REST CONSUMER

The implementation of *HTTP* based *contract tests* with *Pact* is documented more extensively than implementation of *contract tests* for *event driven messaging*. Additionally, the implementation of consumer *contract tests* needs to be performed before the implementation of the *provider* side, when following the *consumer* driven approach. For those reasons, the *contract tests* for the *REST consumer* were implemented first.

The *REST consumer* of the case study project, the *GUI*, is written in *TypeScript* and uses generated code for *API* requests. The code is generated from the schema specification published by the *REST provider*. For the contract testing implementation the *Pact* library for *JavaScript* based languages was included.

5.1.1 *Proof of Concept Implementation*

To make sure that *contract testing* with *Pact* was generally applicable in the case study, it was first implemented for a single *API* call. The chosen endpoint requests a user by its id. To get the test green quickly, this initial test is largely based on the examples provided in the *Pact* documentation (Pact Foundation 2024d) and little optimized. One example for this is the definition of the test data within the same file. The basic configuration of *Pact* is performed by creating an instance of the `PactV4` class, provided by the *Pact* library for *JavaScript* based languages. The configuration is done by passing corresponding parameters, like the *consumer* name and the *provider* name. Furthermore, the directory to export the *contract* needs to be set. The files

```
27  it("handles a request for a user", () => {
28  →return pact
29  →→.addInteraction()
30  →→.given("there is a user")
31  →→.uponReceiving("a request for the user")
32  →→.withRequest(
33  →→→HTTPMethods.GET,
34  →→→`/v1/users/${sampleUser.id}`,
35  →→)
36  →→.willRespondWith(
37  →→→200,
38  →→→(b) => b.jsonBody(sampleUser),
39  →→)
40  →→.executeTest(async (mockServer) => {
41  →→→const configuration = new Configuration({
42  →→→→basePath: mockServer.url,
43  →→→});
44  →→→const api = new UsersApi(configuration);
45
46  →→→const result = await api.getUser({id: sampleUser.id});
47
48  →→→expect(result).toEqual(sampleUser);
49  →→});
50  });
```

Listing 4: REST Consumer Proof of Concept Implementation

written to this directory are shared with the *provider* to perform the *contract* verification.

The core of the proof of concept *contract test* is shown in Listing 4. The instance of the `PactV4` support class allows to describe the *contract* using several chained method calls. For better readability the return types of the method calls ensure, that the methods are always called in the same order. This is started with a description of the state the *provider* needs to be in for this request, as shown in Line 30. When verifying the *contracts* on the *provider* side, these state descriptions are used to set up the required test data and to configure mocks accordingly. The request label set in Line 31 is used in tooling and summarizes the request that is tested in a human readable form. Lines 32 to 35 describe the request on a technical level. The configuration of the *HTTP* Method to use and the *URL* path with which the request is sent always need to be specified, but additional information, like required headers can be set as well. At the beginning of the *contract tests Pact* sets up a *mock server* that responds to the requests triggered by the tests. The request description instructs the *mock server* to expect the described request. It responds to this request with the response that is defined in Lines 36 to 39. The specification of the status code is mandatory. To configure further aspects of a response a callback function can be passed. The function receives a builder object, which it can call different methods on to perform the desired configuration. This allows specific builder methods to configure multiple aspects of the response at once, reducing noise and further improving readability. By example, the `jsonBody` builder method

```
describe("/v1/users/", () => {
→const pact = new PactV4({
→→consumer: "GUI",
→→provider: "User Service",
→→spec: SpecificationVersion.SPECIFICATION_VERSION_V4,
→→logLevel: (process.env.LOG_LEVEL as LogLevel) || "warn",
→});

→it("is able to fetch a user", () => {/* ... */});
});
```

Listing 5: Suite Definition without describePact

```
describePact("/v1/users/", (pact) => {
→it("is able to fetch a user", () => {/* ... */});
});
```

Listing 6: Suite Definition with describePact

used in this example does not only configure the body content, but also sets the `Content-Type` *HTTP* Header to `application/json`. This header is used by *HTTP* clients as an indicator of the data type the body contains. A similar configuration function can be passed to the request description. As the described endpoint is used to provide a user given it's id, the response body is configured to respond with the sample user. Finally, Lines 40 to 49 describe, how functions that perform the request need to be called to trigger the request. As this requires the *URL* to make the request to, *Pact* injects a configuration object into the function, that also contains the *URL* to the *mock server*. This is used to create a configuration object for the *API* and instantiate the required *API* Client as shown in Lines 41 to 44. The client is then used to actually perform the request. The *mock server* verifies that the received request matches the one described. Similarly the request result is verified to have been correctly parsed in Line 48.

5.1.2 *Helpers*

The *Development & Maintenance Effort* (cf. 4.2.3) metric requires *contract tests* to be easy to create and maintain. A major aspect of this is the readability of the tests and the amount of duplication between the tests. Duplication can be reduced and tests be made more readable by extracting common concepts into well named helper functions. By implementing these helper functions after the proof of concept implementation, the creation of the remaining *contract tests* requires less code and can therefore be performed faster.

The *REST consumer* does not use any other interfaces as the one to the `User Service`. Therefore both *consumer* and *provider* name, as well as the folder to save the pact to is identical between all of its *contract tests*. To remove this duplication, a custom test suite function `describePact` was created as a wrapper around the `describe` function, that is used to describe test suites. `describePact` passes an instance of the *Pact* class into the test suite. Listing 5 shows how a *contract test* suite needs to be defined without

```
25  .executeTest(
26  →withApi(UsersApi, async api => {
27  →→const result = await api.getUser({ id: id });
28
29  →→expect(result).toEqual(reify(sampleUser));
30  →})
31  );
```

Listing 7: Usage of withApi

`describePact`. Listing 6 shows how this can be shortened and reduced in duplication through the usage of `describePact`. Furthermore, the existence of this helper method simplifies the maintenance of any setup this specific to the *contract tests*. By example, *JavaScript*'s `fetch` method, which is used to perform network requests, might need to be mocked for regular unit tests, but may not be mocked for *contract tests* as web requests need to reach the *mock server*. If a mocking framework mocked fetch for all unit tests, `describePact` can be used to restore the normal functionality of `fetch` before a *contract test* is executed.

The creation of the *API* Client Instance using the mockServer *URL* as shown in Lines 41 to 44 is similar in every *contract test*. Because the *mock server URL* is only being passed in the callback to the final builder function, the *API* client instance cannot be reused between *contract tests*. An additional helper function, `withApi` simplifies the creation of the *API* client and reduces the associated duplication. As shown in Listing 7 it takes the constructor of the *API* client to generate and the test function as its parameters. The function returned by `withApi` wraps the original test execution function. It receives the *mock server* configuration object and uses it to create an instance of the requested *API* client, before passing it to the test function. The latter no longer needs to contain the setup code, but can rather be reduced to the code that is required to test this specific endpoint.

In addition to the self implemented helper methods there is a library named *jest-pact* (Pact Foundation 2024a) that proposes an alternative *API* to closer integrate with the *jest* (OpenJS Foundation 2024) and *vitest* (Fu, Capeletto, and Vitest contributors 2024) testing frameworks. This library was temporarily adopted and evaluated, but made the code less readable in comparison to the self implemented helper methods and therefore removed again.

5.1.3 *Test Data*

The test data in the original test uses explicit values to describe the response to expect. This requires the provider to return exactly the same values when it is verified. To make the tests more flexible and stable, *Pact* recommends the usage of matchers instead (Pact Foundation 2024b). During verification of the *provider Pact* checks, that the actual value returned by the *provider* fulfills the condition set by the matcher, rather than comparing the actual value. There are simple matchers like `string`, `number` and `boolean` that verify the type of the value. Strings can be limited to a certain pattern using the `date`,

```
const sampleUser: User = {
→id: "9a9845c4-8a46-4bf9-89a5-ecd3c3b66479",
→firstName: "Jane",
→lastName: "Appleseed",
→groups: [{
→→id: "1f2a451b-843c-4e6a-904d-fbee06dc61d7",
→→isAdmin: false,
→}]
};
```

Listing 8: Test Data without Matchers

```
const sampleUser = {
→id: uuid(),
→firstName: regex("\\w{1,50}"),
→lastName: regex("\\w{1,50}"),
→groups: eachLike({
→→id: integer(),
→→isAdmin: boolean()
→})
};
```

Listing 9: Test Data with Matchers

`timestamp`, `datetime` and `regex` matchers. While the `regex` matcher can also be used for numeric values, the `integer` and `decimal` are predefined to limit the values to natural or decimal numbers. For all of the preceding matchers the *mock server* is able to generate sample values, if no explicit example value is passed to the matcher. Additional matchers like `eachLike`, `like`, `eachKeyMatches`, `eachValueMatches` allow matching on arrays and objects. Listing 9 shows how the original test data, shown in Listing 8, can be represented with matchers. Some fields in the case study require their string values to match a certain pattern. To remove the duplication by having the pattern described for each field individually, own matchers for each of the patterns were defined instead. The implementations call the `regex` matcher with the corresponding regular expression. Similarly a `likeEnum` matcher was created that generates a regular expression matcher from the possible values of an enum.

To make sure that the response has been parsed correctly, the *contract tests* need to assert equality between the expected response and the actual response. However, the expected response is no longer equal to the received response, as the field values have been replaced by matchers. To still be able to perform the comparison, the `reify` function is used to extract the example values from the matcher object. As the `reify` method uses the same way of generating the example values as the *mock server*, the resulting object is equal to the expected response object. There are known issues with *Pact's* own implementation of `reify`, in which the restoration of dates from the matcher object generates a random date instead of using the example value.

To fix this, a custom version of `reify` was created based on the original implementation and modified to fix this behavior.

The initial *contract test* defines the test data that is expected in the response within the test case file. While in this case this takes up seven lines in Listing 4, the required definitions can take up a lot more space and reduce the readability of the file when the interface objects are larger. By moving the data definition to its own file, the test file becomes more readable as it only includes the test cases. Moreover, the test data can be used by multiple test files.

To make the usage of test data even more readable, well named factory functions, like described in 3.1.5 were implemented. As some classes are shared in modified form between multiple requests, the *test data builder* pattern was implemented as well. Because *TypeScript* performs type checking differently from *Java*, the builders did not need a separate class, but instead could be implemented by adding a `with` function to the test data that can be called to override a specific property.

Fields for which no value is available on the *provider* may not be transferred at all or be transferred with a value of null. While a missing field is technically not the same as one being set to null, they are usually interpreted the same way. For better readability, from here on fields not transmitted are included when nullable fields are referred to. In that case, the field being null corresponds to it not being transmitted at all. If a *contract* verification would allow the *provider* to leave a field null, the *provider* may never return a response where the field is not null. The case in where the field is not null would then never be checked. Accordingly, the type definitions used for the test data enforce a non null matcher to be defined for every field of the data class. Similarly, arrays are required to always contain at least one item, as otherwise the type of the items in the array could not be verified. This implies that currently the *contracts* do not contain information on which fields are actually nullable and which array type fields can actually be empty. If the different variations of nullable and not nullable fields are critical to be tested, a test needs to be created for each variation, increasing execution time and maintenance cost. This problem has not been solved yet. The *Pact* documentation recommends that this should be solved by making the *consumer* gracefully handle fields that are null but should not be at runtime.

5.1.4 *Final Implementation*

The helper methods and matcher builders were implemented using successive refinement of the proof of concept *contract test*. This also included verifying the generated *contract* on the *provider* side using a proof of concept implementation to make sure that the generated *contract* is correct. Listing 10 shows the final test implementation of the *REST consumer*. Aside from the usage of `describePact` and `withApi` helper methods, as well as the `aUser` factory function, made the test more readable and expressive. In addition, the final test adds an argument to the state, as shown in Line 14. The value can be used by the *provider* to configure the mock more accurately. Compared to the original test file the improved one is more than 30% shorter, making it easier to comprehend the test file structure.

```
8  it("handles a request for a user", () => {
9  →const id = "9a9845c4-8a46-4bf9-89a5-ecd3c3b66479";
10 →const sampleUser = aUser();
11
12 →return pact
13 →→.addInteraction()
14 →→.given("there is a user", { id: id })
15 →→.uponReceiving("a request for the user")
16 →→.withRequest(
17 →→→HTTPMethods.GET,
18 →→→`/v1/users/${id}`,
19 →→)
20 →→.willRespondWith(
21 →→→200,
22 →→→(b) => b.jsonBody(sampleUser),
23 →→)
24 →→.executeTest(
25 →→→withApi(UsersApi, async api => {
26 →→→→const result = await api.getUser({ id: id });
27
28 →→→→expect(result).toEqual(reify(sampleUser));
29 →→→})
30 →→);
31 });
```

Listing 10: REST Consumer Final Implementation

5.2 REST PROVIDER

The implementation of the *REST* of *provider* was started in form of a proof of concept implementation, based on the *contract* created in the *consumer* proof of concept implementation. Supporting tooling and *test data builders* were created to support the implementation of the *contracts* tests of the remaining endpoints.

The *REST provider* of the case study system, the *UserService*, is written in *Kotlin*. It partially relies on code generated using the *OpenAPI* interface specification to serve the *REST* interface. To support the implementation, the *Pact* library for *Java* based *providers* was included.

5.2.1 *Proof of Concept Implementation*

In *consumer* driven *contract testing* the *provider* verification is performed based on the *contracts* provided by the *consumers*. *Pact* verifies the *contracts* by sending the requests recorded in the *interactions* and comparing the response given by the *provider* with the expected one. Each interaction in a *contract* is automatically converted into one verification test case by *Pact*. The *provider* side proof of concept implementation uses the *contract* generated by the *consumer* side proof of concept implementation. To get the *provider* implementation running quickly, the *contract* file was manually copied from the *consumer* repository to the *provider* repository each time it changed. The proof of concept implementation was mostly based on

```
40 @TestTemplate
41 @ExtendWith(PactVerificationSpringProvider::class)
42 @WithMockUser
43 fun pactVerificationTestTemplate(context: PactVerificationContext) {
44 →context.verifyInteraction()
45 }
46
47 @State("there is a user")
48 fun toUserState() {
49 →val sampleUser = UserDto(
50 →→id = UUID.fromString("9a9845c4-8a46-4bf9-89a5-ecd3c3b66479"),
51 →→firstName = "Jane",
52 →→lastName = "Appleseed",
53 →→groups = listOf(
54 →→→GroupReferenceDto(
55 →→→→id = UUID.fromString("1f2a451b-843c-4e6a-904d-fbee06dc61d7"),
56 →→→→isAdmin = false
57 →→→)
58 →→)
59 →)
60
61 →whenever(
62 →→userService.getUserByIdOrNull(
63 →→→any(),
64 →→)
65 →).thenReturn(sampleUser)
66 }
```

Listing 11: REST Provider Proof of Concept Implementation

the example implementations provided by the *Pact* documentation (Pact Foundation 2024f).

Listing 11 shows the core of the proof of concept implementation for the *REST contract tests* on the *provider* side. To be able to create a test for each interaction without having to explicitly define it in the test class JUnit's `TestTemplate` annotation is used, as shown in Lines 40 to 45. The annotation is added to a template method, which is called once for each test case. The difference in execution between test cases is achieved by the parameters that are passed to the test template method. The values for the parameters are provided by an implementation of the `TestTemplateInvocationContextProvider` interface. In this case, *Pact* provides the `PactVerificationSpringProvider`, which supplies each test case with a `PactVerificationContext`. The test template itself only needs to call the `verifyInteraction` method on the context, which then performs the request execution and response verification. To avoid having to pass a valid authentication token with each request the `WithMockUser` annotation is used on the test template method additionally, which adds an authentication token to the request.

The *REST* Controllers are mainly responsible for converting data between interface models and application models. To process transmitted or acquire requested data service methods are called. As the business logic described in the services is not relevant for the request and response structure the services have been mocked for the *contract tests*. Applied to the example

endpoint this means that the `UserController` is responsible for converting the user model used within the *microservice* to the one used on the interface, while the `UserService` is responsible for actually retrieving the user. Note that `UserService` in this example does not refer to the microservice, but to the class in the service layer of the *microservice*. To allow the controllers to generate response data without any actual service layer instances present, the required services are replaced with mocks. Depending on the interaction, a *mock* might be required to return different results. In the example endpoint one interaction might test how a user is transferred when the requested user exists, while another might verify that the *provider* answers with a 404 status code when the expected user does not exist. To allow for this interaction-specific configuration the *provider* state can be modified using state change handlers as shown in Lines 47 to 66. When an interaction requires the *provider* to be in a specific state and therefore require a specific mocking setup this is specified by the *consumer* using the `given` keyword. When this interaction is verified, *Pact* calls the method that is annotated with Pact's `State` annotation and the corresponding state description as the annotations parameter. If no such method is defined, the verification fails. To create the mocks, *mockito-kotlin*, a library to support mocking in *Kotlin* code is used. As shown in Lines 61 to 65, the `whenever` function can be used with a call to `thenReturn` to specify on which call the *mock* should return a specific value. While the argument to `whenever` is the result of a call to the method to mock, rather than a reference to it, mockito is able to use this call to identify the type of calls this mocking configuration should be applied for. To mimic production usage as close as possible, the fake user should only be returned when the id is the expected one. The *mock* continues to return null for other values. To support cases where arguments do not need to be limited to a certain value, *mockito-kotlin* offers argument matchers like `any` to be used instead.

If the application would be fully initialized for the tests, they would be rather slow, as the application would need to be reset to a reproducible state between tests. Besides this, external connections to the database and other *microservices* would need to be available or be mocked, increasing the maintenance effort of the tests. Instead, the test class uses the `WebMvcTest` annotation to only initialize the controller layer of the application, which is responsible for handling incoming requests. As this layer is stateless, the application instance can be reused between tests. The usage of `WebMvcTest` additionally allows to inject requests into the request handling mechanism directly, instead of needing to send them over the network interface, accelerating the tests further. *Pact* supports this way of request injection by allowing to configure a test target, which is responsible for executing the requests. The `MockMvcTestTarget` implementation provided by *Pact* receives the `mockMvc` object, which is injected into the test class by `WebMvcTest`, and uses it to perform the requests described in the *contract* (Pact Foundation 2024f). Two additional `PactFolder` and `Provider` annotations on the test class configure the folder that *Pact* should load the *contracts* from and which *provider* name should be used to filter the available *contracts*.

5.2.2 *Helpers*

The proof of concept implementation limited the *contract testing* to the controller layer of the *microservice* and mocked all other components. The deserialization of the request and serialization of the response are performed by *Spring Boot* internals. *Spring Boot* is a widespread framework and these internals are used in most *Spring Boot* applications (Stack Exchange Inc. 2024, section "Web frameworks and Technologies"). Consequently it is unlikely that they contain any bugs that could be found by *contract testing*. While the controllers are responsible for transforming the data between models, the values itself are usually generated within the service layer. Based on this, the *contract tests* were extend to the service layer, so that only external connections to the database and other *microservices* are mocked. The term *service* does in this case not refer to a *microservice*, but rather a specific kind of application components in *Spring Boot* applications, that contain the business logic of the application. As the services actually assemble the data required by the controller, the state change handlers often needed to mock multiple data sources and therefore became more complex. In some cases services needed to be restructured, as they read from data source before any mocking configuration could be performed. At many endpoints verification took a lot longer to get successful than when testing on controller level, as the service layer logic needed to be understood to set up the correct mocks. In addition, another testing layer is already in place to verify integration between controllers and services. It is therefore recommended to abandon this approach should *contract testing* is adopted and revert to the original approach of limiting the *contract tests* to controller level.

Spring uses an automated *dependency injection* mechanism that injects instances of other required components into each component instance upon creation. Each value to inject is referred to as a *Bean* and each injectable class or value is referred to as a *Bean Definition*. If no *Bean Definition* can be found for a field requesting *dependency injection*, a corresponding error is thrown and the application initialization aborted. The `WebMvcTest` annotation configures the *dependency injection* mechanism to ignore all *Bean Definitions* that are not controllers. As a result, all *Beans* relied upon by the controller level need to be mocked. The controllers exclusively depend on services, that need to be mocked anyway to configure the *provider* state in the state change handlers. However, if a controller is newly created, it is not declared as a *Bean* to be mocked yet. The *contract test* is then not able to instantiate the controller and the *contract* verification fails unnecessarily. To fix this, the controller needs to be marked as a mocked *Bean*, even if no mock configuration is used. This was made redundant by creating a script based on the stale spring-auto-mock project (Inoto 2016) that automatically mocks all *Beans* that are excluded or have no *Bean Definition*. The code was converted to *Kotlin* using a conversion feature of *Intellij IDEA* (JetBrains s.r.o. 2024) and updated to be compatible with *Spring Boot* 3 and *Kotlin*. The `MissingBeansMocker` iterates through the dependencies of all registered Beans on the Bean registry, collecting all dependencies on other *Beans*. For each of those dependencies a *mock* is registered, if no *Bean* is registered to fulfil the dependency.

```
32  @Autowired
33  private lateinit var stateChangeHandlers: MutableList<StateChangeHandler>
34
35  @Autowired
36  private lateinit var mockMvc: MockMvc
37
38  @BeforeEach
39  fun before(context: PactVerificationContext) {
40  →MissingBeansMocker.initMocks()
41
42  →context.target = MockMvcTestTarget(mockMvc)
43  →stateChangeHandlers.forEach { context.withStateChangeHandlers(it) }
44  }
```

Listing 12: Improved Provider Test Implementation

In the proof of concept test, the state change method is defined within the test class. The number of state change methods increases with the number of tests. By creating a test file for each of the *REST* controllers, the test file as well as the files containing the state change methods can be kept short and easy to comprehend. To still be able to register the methods on the test context, a `StateChangeHandler` interface is defined and implemented by the classes containing the state change methods. To automatically collect all state change handlers, without having to import them individually, they are imported using an `Autowired` annotation on a filed with a list type, as shown in Lines 32 and 33 of Listing 12. When the test class is initialized, the `StateChangeHandlers` property includes all classes, that define the `StateChangeHandler` interface. They are registered on the test context using the `withStateChangeHandlers` method on the test context in Line 43.

5.2.3 *Test Data*

The large amount of fields on the classes of the models used by the *provider* would make the manual creation of test data for the *contract tests* very tedious and require modification each time a model class changes. For testing on controller level, the introduction of *objectmothers*, as described in 3.1.5, would have been sufficient, as *contract testing* uses matches and does not require specific values to be present. However, in the extension of the test data to the service layer test data instances often needed specific values in some fields to allow the services to process them correctly. The usage of the *test data builder* pattern would make this easier, but still require the definition of a builder class. In addition to initial implementation efforts, it would require maintenance as well. To circumvent this, a generalized Builder class was created, that is able to automatically select default values by using *reflection* on the class to create an instance of. *Reflection* enables *Java* based languages to access classes, as well as their fields and methods at runtime. If *contract testing* is reset to controller level testing, field manipulation needed less frequently. A switch to using *objectmothers* would likely greater effort than sticking with *test data builders*. In addition, the created *test data builders*

```
1  val sampleUser = aUserBuilder()
2  →.withTypeValue(String:class, "a string")
3  →.with("firstName", "Jim")
4  →.build()
```

Listing 13: Example Usages of `with` Methods

```
7  class BuilderForRestProvider<C : Any>(
8  →targetClass: KClass<C>
9  ) : Builder<C>(targetClass.java) {
10 →init {
11 →→this
12 →→→.withTypeFactory(UserDto::class) { aUserDto() }
13 →→→.withTypeFactory(GroupReferenceDto::class) { aGroupReferenceDto() }
14 →}
15
16 →override fun getNewBuilder(jvmType: Class<*>): Builder<*> =
   ↪  BuilderForRestProvider(jvmType.kotlin)
17 }
```

Listing 14: Project Specific Builder

builders can be adopted in other testing levels as well, offering a uniform way of creating test data across the layers.

Listing 13 shows an example of how the *test data builders* are used from within the test code. The `aUserBuilder` factory function in Line 1 provides an instance of the *test data builder*. The calls in Lines 2 and 3 configure type fallbacks and field overrides. To allow the builder class to be reused between projects, the definition of which defaults to use for which field type is only implemented into the base builder class for primitive types. Additional configuration is performed using calls to `withTypeValue` and `withTypeFactory` as shown in Line 2 are used. To use this configuration for all builders in a project, it can be performed in the constructor of a project specific builder class as shown in Listing 14, which can then be used instead of the base class. To encourage this approach, the base builder class is abstract. If a field contains a non primitive type a new builder instance is created to build a corresponding value. To use the same project specific builder class the instance is created using the `getNewBuilder`, which needs to be overridden by any class inheriting from the base builder class.

Overrides of specific fields can be performed using calls to `with`, as shown in Line 3. The case study project included some field names that reoccurred throughout the project and always needed a string value in a specific format. While this indicates that these fields should use a value class instead (Freeman, Steve and Pryce, Nat 2010, pp. 59-60, 141, 151; Fowler 2016), the necessary refactoring could not be performed in the scope in this thesis due to time constraints. To prevent the default value for these fields from being overridden for each class, the override was included in the project specific builder class, similar to the calls to `withTypeValue` and `withTypeFactory`. As inheriting builders and usages in the test code might still want to override earlier config-

```
6  fun aUserDto() = aUserDtoWith { }
7
8  fun aUserDtoWith(buildUserDto: UserDto.() -> Unit) =
   ↪  aUserDtoBuilderWith(buildUserDto).build()
9
10 fun aUserDtoBuilderWith(buildUserDto: UserDto.() -> Unit) =
11 →BuilderForRestProvider(UserDto::class)
12 →→.with("id") { UUID.randomUUID() }
13 →→.with(buildUserDto)
```

Listing 15: Provider Factory Functions

urations, later calls to any of the `with` functions take precedence over earlier ones. However, as field name based configurations are more precise they are treated as overrides and type based configurations as fallbacks, meaning that type based configurations values are only used when no field name configuration exists.

The *Java* ecosystem provides common utility classes like `Set`, `Map`, `List`. These classes act as containers for other types and are serialized to arrays by *Spring Boot* in requests and responses. As described in 5.1.3, fields of this type need to contain elements as otherwise the *contract tests* would not be able to perform a validation of the content type. The builder class is able to instantiate each of these classes and adds a corresponding content element based on the type information gained by reflection. For enum type fields the first reflected enum value is used as the default. While both implementations could be moved into project specific fallback configurations as well, this was not performed, as these implementations are unlikely to deviate between projects.

For better readability a set of factory functions exists for each target class, as shown in Listing 15. In addition the factory function allow to define all type fallbacks and overrides for a specific target class without defining an own builder class for it. The different factory functions exist to support varying levels of field modifications. `aUser` simply creates a user, without requiring any call to `build`, allowing the creation of a default user with a single line of code, but having no ability to modify any properties. To support easy and type safe modification of non-final properties, `aUserWith` supports the passing of a function that are called after the user is created. The assignment is performed as it would be in a method within the target class, because the instance of the target class, in this case the user, is the context of the function. The same reason limits the usage of this function to non-final properties, as the user needs to be created before the function can be called. To still allow the override of final properties, the `aUserBuilderWith` function does not call build before returning. It exposes the builder to the caller, which can then add additional calls to `with`, before eventually calling `build` itself. While this provides greater flexibility, it also requires more lines of code, making the use of the simpler functions if no modification of final properties is required. In some cases, the project specific configuration is not sufficient to create valid instances for a target class, for example if a string field needs to match a specific format that is validated in the classes constructor. By making the

simplest factory function, like `aUser`, depend on the middle one, which itself depends on the most flexible one, the class specific overriding of properties can be centrally defined in the most flexible function. To also apply these overrides also when the builder of another class needs an instance of the target class, all factory functions are registered as type fallback for the class they produce, as shown in Listing 14, Line 12.

In some cases, generic types cannot be resolved using reflection, for example when they are annotated. This might prevent instantiation of the default value due to the missing type information. The builder contains error handling that allows to determine which fields and classes lead to the errors. The error can be resolved by using this information to explicitly define a default value for the problematic field, which can use the builder pattern again but specify the generic type explicitly. Further, the builder runs into a stack overflow error when instructed to create a recursive type. These errors can be resolved by manually specifying a default value for one of the fields to break the recursion.

Because *Kotlin* lacks a way of limiting a string type to the field names of a class the `with` function uses a type of `String` for the property name. Accordingly, it can not be assured at compile time that the passed property name actually matches a property on the target class. A solution to this was perused with an alternate form of implementation, that is based on a compiler plugin generating a builder for each class. It also provides the additional benefit that the builder is generated once and is only updated when the target class changes. The overhead of analyzing the target classes at runtime, each test run anew is removed and test execution accelerated. The implemented solution is functional and able to create *test data builders* and the corresponding factory function efficiently, but still lacks some configuration abilities. For example, at the current point it is not yet possible to configure overrides for a specific target class. To compile a function that performs said configuration, the builder class would already have to be compiled as it is passed as an argument. However, the builder class needs the configuration function to be already compiled as well, to be able invoke it in its own implementation. This problem can likely be solved by having the configuration function perform the configuration on an interface, instead of the actual builder. The interface is compiled first and implemented by the *test data builder*. The required implementation required to achieve that could not be performed within the scope of this thesis.

5.2.4 *Final Implementation*

Listing 16 shows the final state change handler class. To make the class discoverable to *Spring Boot's* bean system, it needs to be annotated with the `Component` annotation like in Line 13. The usage of the `aUser` builder function in Line 20 reduces the length and improves readability of the method, compared to the original, manual creation. As different *consumers* might use different examples for the user id, the method uses state parameters that are passed along in the *contract* to specify on which user id `getUsers` should

```
13 @Component
14 class UsersStateChangeHandler: StateChangeHandler {
15 →@Autowired
16 →private lateinit var userService: UserService
17
18 →@State("there is a user")
19 →fun toUserState(params: Map<String, Any>) {
20 →→val sampleUser = aUserDto()
21
22 →→whenever(
23 →→→userService.getUserByIdOrNull(
24 →→→→UUID.fromString(params["id"] as String)
25 →→→)
26 →→).thenReturn(sampleUser)
27 →}
28 }
```

Listing 16: REST Provider Final State Change Handler

return the example user in Line 24. The parameters are injected by *Pact* when the state change method is called.

5.3 EVENT CONSUMER

To accelerate the implementation of the *contract tests* for *event driven messaging*, the event *consumer* was only implemented after the final implementations of the *REST consumer* and *provider* were completed. This allowed the usage of the existing tooling, especially as the event *consumer* is the same *microservice* as the *REST provider*. After an initial proof of concept implementation for a single type of message, the findings gained from the *REST* implementation were used to implement helpers and *test data builders* were modified or implemented anew for the event based *contract tests*. Using these, the event *contract tests* were implemented for the remaining message types.

The event *consumer* of the case study project, is the same as the *REST provider*, the *UserService*. The implementation of *consumer* tests requires a different library than the verification of *contracts*. It was added to the *UserService* as well.

5.3.1 *Proof of Concept Implementation*

Like on the *REST provider* tests, annotations are used to include and configure *Pact* for the event *consumer* tests. Listing 17 shows the `userCreationMessage` method of the event *consumer* test class. This method describes the format of a user creation message to be used in the *contract*. To indicate this method as a message description, the message is annotated with the `Pact` annotation, as shown in Line 50. The annotation is also used to configure the *consumer* name. Similar to the *Pact* library for *JavaScript*, the library for *Java* based languages uses a builder pattern to describe the interaction,

```
50  @Pact(consumer = "User Service")
51  fun userCreationMessage(
52  →builder: MessagePactBuilder
53  ): V4Pact =
54  →builder
55  →→.expectsToReceive("A user creation message")
56  →→.withMetadata(mapOf("contentType" to "application/json"))
57  →→.withContent(
58  →→→newJsonBody ({ theObject ->
59  →→→→theObject.uuid("id")
60  →→→→theObject.stringType("initiator", "the string")
61  →→→}).build()
62  →→)
63  →→.toPact(V4Pact::class.java)
```

Listing 17: Event Consumer Proof of Concept Message Description

as visible in Lines 54 to 63. Lines 55 and 56 set a human readable message description, as well as the message metadata. The `withContent` call sets the description of the message content. To verify the message structure, rather than concrete values, the message needs to be defined with matchers, as it is done in the *REST consumer* (cf. 3.1.5). This is done using a lambda based *DSL*, as shown in Lines 58 to 61. The event *consumer* supports several types of user creation messages types. The parsing of fields, that are specific to the types is performed separately from the message fields shared between all message types. To get a quick feedback on viability and hurdles, the proof of concept implementation only verifies the parsing of the shared fields. Line 63 converts the message description to a *contract*.

To verify, that the message described in `userCreationMessage` can indeed be parsed by the event consumer, a sample instance of it needs to be parsed by the message handler. This is implemented in the service layer. As described in 5.2.1, the *REST provider* uses the `WebMvcTest` annotation to limit the application initialization to the controller layer. As *Spring Boot* offers no equivalent annotation to initialize only the service layer, the event *consumer* test uses the `WebMvcTest` annotation, with a configuration to also include services. Even though this also initializes *REST* Controllers and other *REST* mechanics not needed for *event driven messaging*, this is still more performant and simpler to configure than initializing the whole application. The parsing of the message is tested by the `userCreationMessageTest` method of the test class, shown in Listing 18. To be registered as a test, the method is annotated with the `Test` annotation, as shown in Line 65. As a single test class is allowed to contain multiple method descriptions, the method that describes the message for this test needs to be referenced by the `pactMethod` parameter on the `PactTestFor` annotation, as shown in Line 66. To allow the testing of different kinds of messages, the test method receives a list of messages that contains all possible example messages, rather than a single message, as can be seen in Line 68. At this stage only one message description is defined, the list therefore only contains a single element. When following clean code principles error handling should be separated from regular business logic,

```
65  @Test
66  @PactTestFor(pactMethod = "userCreationMessage")
67  fun userCreationMessageTest(
68  →messages: List<V4Interaction.AsynchronousMessage>
69  ) {
70  →doAnswer {
71  →→throw AssertionError(
72  →→→"Unroutable Error detected, error code: ${it.arguments[1]}"
73  →→)
74  →}.whenever(
75  →→this.metrics
76  →).unroutableErrors(
77  →→anyOrNull(),
78  →→anyOrNull()
79  →)
80
81  →assertDoesNotThrow {
82  →→dispatcher.handleUserCreationMessage(
83  →→→messages[0].contentsAsString(),
84  →→→messages[0].metadata as Map<String, String>
85  →→)
86  →}
87  }
```

Listing 18: Event Consumer Proof of Concept Test

by throwing an exception and catching it in a method higher in the call stack (Martin 2008, pp. 46f). The message handler in the case study instead called error handling methods directly when a problem is detected and exited the method using a return. To still be able to detect parsing errors, the proof of concept implementation mocks the `unroutableErrors` method on a *bean* to record metrics, that is invoked whenever an error occurs at this stage of message parsing. The configuration of this mocked method is configured in Lines 70 to 79. The method would normally print the error to the log, making it hard to detect for the test. By configuring the mocked method to throw an assertion error instead, the error is raised back to the test and can be detected as a message parsing failure. To initiate the message parsing, the test invokes the `handleUserCreationMessage` method on the handler responsible for processing the relevant messages, as shown in Line 82. The message content and metadata are taken from the first list element using the corresponding accessors. Tests in JUnit fail, if an exception is thrown and not caught in the test, like the assertion error thrown by the mocked method. By wrapping the handler invocation in a call to `assertDoesNotThrow` it is clearly communicated that the test is successful if the message can be parsed and no further assertions need to be made.

5.3.2 *Helpers*

The message handler starts message verification on the fields independent of the message type, to check if the message can be processed at all. To prevent race conditions by other threads processing the same message, the handler

then tries to save the message to the database and only continues processing if it that succeeds. If another thread started processing the message before, the insertion fails, as the unique message id already exists in the database table. The parsing of type specific field is only executed after the message was saved to the database. As the *contract tests* do not run with a database instance, the database insertion needs to be mocked to test the parsing of type specific fields. In an ideal solution, the testing scope would be defined by a clear line in the application layering, in this case by only mocking non service *beans* like those used to access to the database. For most database operations, the case study uses methods on repository *beans* that can easily be mocked. However, the insertion of the message into the database is performed using several native queries instead, which cannot be mocked easily. Several mocks of the same method would be required, with varying behavior, including management of state. The implementation would tightly couple the test implementation to the productive code, which reduces maintainability. As a result, the service responsible for saving the message to the database is mocked instead.

Both type independent and type specific parsing used inline error logging in combination with early returns to handle message parsing errors. As this adds complexity and redundancy to a method, this is considered bad practice (Martin 2008, pp. 46f). Furthermore, it makes it very hard to test the error handling of the method, as the occurrence of an error is only visible in the log. Clean code practices recommend the raising of exceptions instead. This also improves testability, as error handling and business logic can be tested separately, and thrown errors can be treated as failed tests. Therefore the parsing algorithm of the message type specific fields was refactored to throw errors and a separate method introduced for catching those errors. The new method is called by the parsing algorithm instead of the original parsing method. If an error occurs in type specific parsing it is caught in the new error handling message and prevented from raising further into the logic to insert the message into the database. While this is desired in production, the thrown exception needs to reach the test method to fail the test. As the database insertion that calls the type dependent parsing method is mocked, this is possible by calling the parsing method directly instead of the newly introduced method to catch exceptions. While this works, the parsing of type specific fields should be decoupled from the database insertion to make error handling easier. Due to time constraints this could not be performed within this thesis.

5.3.3 *Test Data*

The data transferred using *event driven messaging* uses fewer data types, but the exchanged objects have more fields and nested objects than those on the *REST* interface. Consequently, *test data builders* are used to build the required test data for the event *consumer* as well. While the builder class is already prevalent in the project from the implementation of the *REST* provider, it cannot be used for *consumer* data generation. The builder produces instances of the target class, but the *consumer* tests need matchers to describe the

```
101 @Pact(consumer = "User Service")
102 fun userCreationMessage(builder: MessagePactBuilder): V4Pact {
103 →builder.expectsToReceive("A CredentialsUserMessage")
104 →→.withMetadata(mapOf("contentType" to "application/json"))
105 →→.withContent(aCredentialsUserMessageDto())
106
107 →builder.expectsToReceive("An LdapUserMessage")
108 →.withMetadata(mapOf("contentType" to "application/json"))
109 →→.withContent(anLdapUserMessageDto())
110
111 →return builder.toPact(V4Pact::class.java)
112 }
```

Listing 19: Event Consumer Final Message Description

```
64 @Test
65 @PactTestFor(pactMethod = "userCreationMessage")
66 fun userCreationMessageMessageTest(
67 →messages: List<V4Interaction.AsynchronousMessage>
68 ) {
69 →assertAll(messages.map { m -> { -> receive(m) } })
70 }
```

Listing 20: Event Consumer Final Test Implementation

message. Based on the concepts used in the *test data builder* a similar builder was developed to generate matchers based on type information gained on reflection. The *REST provider* implementation has already shown, that a compiler plugin is likely to be superior to a reflection based approach. Consequently, the matcher builder has not been developed as project independent as the *test data builder*, because it is likely to be replaced by a compiler-based approach if *contract testing* is adopted.

5.3.4 *Final Implementation*

Listing 19 shows the message description in the final version of the event *consumer contract tests*. To simplify the creation of test data, the test data is generated using the matcher builders, as seen in Lines 105 and 109. Defining each message type within its own method would add a lot of clutter to the test and require the referencing of each description method at the test method. Instead, the message description builder can be used multiple times, as shown in Line 107. The call to `toPact` in Line 111 then converts each message type to an interaction within the *contract*.

Listing 20 shows the final implementation of the test method of the event *consumer contract tests*. Contrary to the original implementation, the method now receives multiple messages within the passed list. If a `for`-loop would be used to test the parsing of the different message types, the test would stop when the first message cannot be parsed. This would hide, whether only the parsing for this message type is defective or if any of the remaining messages

can not be parsed as well. Instead, `assertAll` is used in Line 69, which receives a list of lambda functions. Each lambda function calls the `receive` method with one of the example messages. Any lambda functions can throw an error that is logged in the final output, but the remaining functions are still executed. `assertAll` throws an exception and therefore fails the test, if any of the lambda functions throws an exception. This way, all example messages are tried to be parsed and all errors are listed. However, the testing of all message types in a single test case makes it hard to determine, which message type failed to be parsed in the case of an error. If each message type ran its own, named test case, the message type that failed to be parsed could simply be read of. Without additional supporting implementations however, this would require a separate description and test method for each message type, which drastically reduces readability and increases maintenance effort. In the *REST provider*, the generation of multiple tests from a provided dataset is achieved using the `TestTemplate` annotation on the test method, that, similarly to the current `receive` method is called for each option in the dataset. The idea of providing a similar option for *contract tests* on event *consumers* has been proposed to Pact's development team for *Java* based languages (Pact Foundation 2024e). Until now this has not been taken up for implementation, as it would require some significant changes to the library.

5.4 EVENT PROVIDER

The *provider* implementation for the event *contract testing* verifies the *contracts* generated on the event *consumer* side. Like with the other participants, the implementation was started with a proof of concept, and successively refined until the final implementation was reached.

Like the *REST provider*, the event *provider* in the case study, the *UserCreationService*, is a *Spring Boot* application written in *Kotlin*. Some of the tooling created on the *REST provider* could be reused for the implementation. As the event *provider* does not implement any *consumer contract tests* at the moment, only the *provider* library of *Pact* is needed.

5.4.1 *Proof of Concept Implementation*

Like the *REST provider* requires *contracts* to be generated by the *REST consumer* to verify them, the event *provider* needs *contracts* generated by the event *consumer*. To quickly determine viability and prepare tooling before full implementation, the proof of concept for the event *provider* was developed after the event *consumer* proof of concept was working.

Because in *event driven messaging* the interaction is not initiated by the *consumer*, but by the provider, the interaction cannot automatically be replayed by *Pact* like it was done on the *REST provider*. Instead, each message a *provider* can send needs to be captured using traditional mocking mechanisms. Listing 21 shows the `verifyAUserCreationMessage` method from the class for the event *provider contract tests*. Like the event *consumer*, the class uses a modified version of the `WebMvcTest` annotation to initialize the service layer of the application. To be able to extract the message to verify,

```
59 @PactVerifyProvider("A user creation message")
60 fun verifyAUserCreationMessage(): MessageAndMetadata {
61 →var message: Message<Any>? = null
62 →doAnswer {
63 →→message = it.arguments[0] as Message<Any>
64 →→CompletableFuture<SendResult<String, String>>().also { future ->
65 →→→future.complete(mock())
66 →→}
67 →}.whenever(this.kafkaTemplate)
68 →→.send(any<Message<*>>())
69
70 →this.service.publishCreationOf(aCredentialsUserMessageDto())
71
72 →return MessageAndMetadata(
73 →message!!.payload.toString().encodeToByteArray(),
74 →message!!.headers,
75 →)
76 }
```

Listing 21: Event Provider Proof of Concept Implementation

the *Kafka* method responsible for sending it needs to be mocked and configured to capture the message, as shown in Lines 61 to 68. To generate the required test data, the *test data builder* from the *REST provider* was copied and slightly modified to fit the models used in the event *provider*. It is used to generate the test data required to invoke the methods triggering the message publication as shown in 70. For verification, the captured message is then wrapped in the `MessageAndMetadata` class provided by *Pact* and returned, as seen in Lines 72 to 75.

5.4.2 *Helpers*

The event *provider contract tests* need to capture the transmission of each of the message types defined by the *contract*. As each method types requires a different model class to be instantiated, an individual method is required for each message type. The only difference between each method, however, would be the used model class, causing a lot of duplication between the methods and reducing readability and maintainability. Therefore the publication call and the capturing implementation were extracted to a method called `captureTransmissionOf` that only takes the message to publish and capture as an argument.

5.4.3 *Test Data*

The event *provider* implementation largely makes usage of the existing builder implementation used with the *REST provider*. However, in contrast to the data classes of the *REST* provider, many data classes in the event *provider* do not allow direct access to fields, but rather define getters and setters to access them. In some cases, setting the field through the constructor is not possible as well. To still be able to generate the required classes, the builder

```
63 @PactVerifyProvider("A CredentialsUserMessage")
64 fun verifyACredentialsUserMessage(): MessageAndMetadata =
65 →captureTransmissionOf(aCredentialsUserMessageDto())
66
67 @PactVerifyProvider("An LdapUserMessage")
68 fun verifyAnLdapUserMessage(): MessageAndMetadata =
69 →captureTransmissionOf(anLdapUserMessageDto())
```

Listing 22: Event Provider Final Implementation

was extended to also be able to detected setter methods and set fields using these. This includes setters for regular fields, as well as functions to add elements to the list value of a field, rather than replacing the whole content. In theory, the fields could also haven been set by modifying the access modifiers using reflection, but this could have caused problems with fields that are not data fields themselves, but are set as side effects in other method calls. The changes performed on the builder were refactored and combined back into one common implementation.

5.4.4 *Final Implementation*

Listing 22 shows verification methods of the final implementation for the event *provider contract tests*. The calls to `captureTransmissionOf` in Lines 65 and 69 trigger the generation of the messages to verify in a very readable and maintainable way, as the publication and capturing implementation is centralized in `captureTransmissionOf`. There were no significant changes to the rest of the class.

5.5 PIPELINE CONFIGURATION

The possibility to execute *contract tests* locally simplifies development, as it enables debugging failed tests and trying out potential fixes and improvements quickly. However, to make sure tests executed before merging a feature branch into the main branch, the *contract tests* need to be integrated into the continuous integration pipeline. The *contract tests* are run using the same frameworks like the unit tests, so they can be executed along the existing unit test suite. This however would prevent a publication of valid *contracts* if unit tests fail. The corresponding *provider* implementation would then be delayed until all tests on the *consumer* side are green, even if the failing tests do not relate to the interface at all. The execution in a separate pipeline job allows to treat the *contract testing* result separately from the unit test result. By introducing a common template used by both pipeline jobs, the *contract testing* job can be setup quickly and both jobs maintained with low effort. To be able to separate the *contract testing* files from the regular unit testing files, the files were given a specific file name extension in the *REST consumer*, like `.contract.test.ts` instead of `.test.ts`. In the *Kotlin* based implementations the *contract tests* were placed in specific folders instead.

```
1  {
2  →"$.id": {
3  →→"combine": "AND",
4  →→"matchers": [
5  →→→{
6  →→→→"match": "regex",
7  →→→→"regex":
       ↪  "[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}"
8  →→→}
9  →→]
10 →}
11 }
```

Listing 23: Matching Rule Example

Both methods allow the exclusion or inclusion in the corresponding pipeline job.

5.6 CONTRACT MANAGEMENT

To be able to perform *contract* verification, the *contracts* generated by the *consumer* need to be made available to the *provider*. The *JSON* file containing the contract stores the information recorded by the tests in a serialized form. This includes the *interactions* recorded in the tests, metadata on the tested *consumer* and *provider* pair, as well as versions of the used libraries and the *contract* file schema. Based on the *contract* file, *Pact* needs to be able to generate a request and verify the response given by the provider. This requires the example request made by the *consumer*, as well as the matchers defined for the response. To be able to track changes to the *contract* and show examples for request and response, the *contract* file contains examples and matchers for both requests and responses. Listing 23 shows an example matching rule for the `id` property of a requested user. The rule specifies the path to the property from the request root, as seen in Line 2, how multiple matchers should be combined, as seen in Line 3 and the matchers themselves. As the id of a user is not a simple string, but a `uuid`, the matcher uses a regular expression to check the format of the id, as visible in Line 7.

As the proof of concept implementations featured only two *interactions* and few updates to the *contracts*, the *contract* file generated by *Pact* was manually copied to the *provider* repository. With the increasing number of *contracts* on each interface, the effort of manually copying the file to the *provider* repository each time a change needed to be made on the *consumer* side became tedious and time consuming. Even if the *contract* file is transferred automatically, this form of exchange would likely be rather inconvenient in active development for multiple reasons. For once, the *provider* can verify that the *contracts* are fulfilled, but the validation result is never made available to the *consumer*. Additionally, the compatibility is only checked between the specific versions of the *consumer* and the *provider* used for the particular verification. If the *consumer* publishes a *contract* that is backwards compatible with an older version, this information is only detected, if the new *contract*

version is explicitly verified with the old *provider* version. This compatibility information would be especially useful when deploying the *microservice*. However, it would also require tracking of which *microservice* versions are deployed on which environments.

Pact offers the usage of a *Pact broker* to mitigate these issues. While the cloud hosted variant, *Pact* Flow (Smart Bear Software 2024), offers many additional features and requires no own maintenance or setup effort, it was not applicable to this project due to the pricing. Instead, the open source and free *broker* was used, which is sufficient to mitigate the issues described above. The service is run on the project's *Kubernetes* cluster. *Kubernetes* is an orchestration platform for containerized applications. The setup was significantly simplified by the availability of a *Helm* chart. *Helm* charts are base configuration for common applications and services. To allow the *consumer* pipeline to publish new *contracts* to the *broker* and the *provider* to check them out, as well as upload verification results, the credentials were configured as pipeline variables. To prevent each *microservice* having to define the variables themselves, the pipeline variables were configured on a group containing all the *microservices*, making them available to all of them without further configuration. The `PactFolder` annotations were replaced by `PactBroker` annotations and to use the *broker* using the pipeline variables. To allow the verification of *contracts* locally, the *provider* implementations also received a set of credentials that used in local development. These credentials provide read access only, to prevent an accidental publishing of local verification results and to protect the more sensitive write credentials.

The *Pact broker* requires a version number along with each *contract* of verification result published. The case study system usually uses *semantic versioning (SemVer)*, which consists of three version numbers: one for breaking changes, one for feature updates and one for patches. The *contracts* might change multiple times between releases, even multiple times during development of a single feature. Before adoption of *continuous deployment* only a few *contract* versions are actually released. Therefore commit hash versioning was used for the *contracts* instead. The usage of *SemVer* together with a commit hash suffix would combine the advantage of an easily estimable age of the *contract* with the advantage of having an individual version for each commit, but was not yet pursued due to higher priority of other aspects. To circumvent verification checks between different *contract* versions, the *broker* is able to determine compatibility *contract* versions without executing any additional *contract* verifications. In addition, the *Pact broker* can be used to record deployment of specific *contract* versions to an environment. This can then be used verify that the *contract* of the service version about to be deployed is compatible with the *contracts* of the other deployed services. Aside from these features used to tackle the issues described above, the *Pact broker* additionally supports other features, aiding the development with *contract tests*. For example, the *broker* can be configured to trigger a specific pipeline, if a new *contract* version is released. If the pipeline succeeds, the *provider* supports the *contract* without requiring additional implementations, allowing the *consumer* to complete their implementation faster.

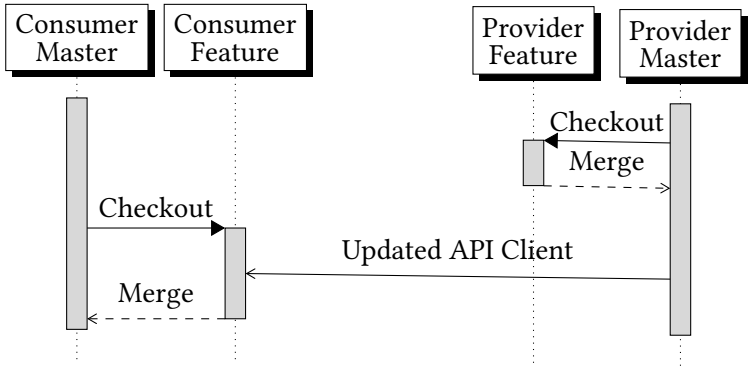

Figure 5.1: Previous, Provider Driven Branching

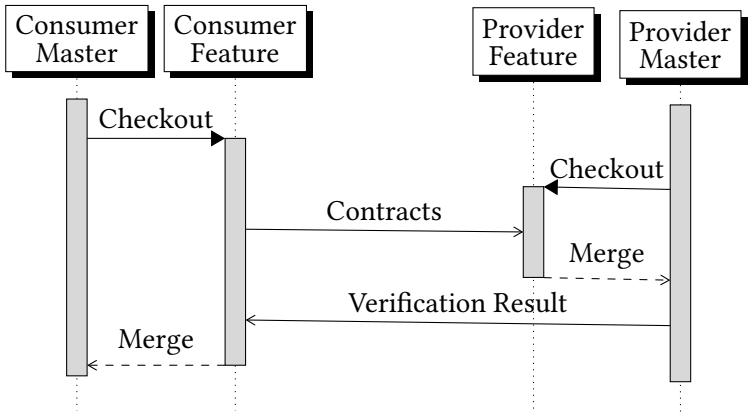

Figure 5.2: Branching with Consumer Driven Contract Tests

5.7 ADOPTING CONSUMER DRIVEN DEVELOPMENT

Before adopting *contract testing*, the case study project developed most features *provider* driven. Figure 5.1 shows a simplified sequence diagram of the branching flow. The endpoints were first implemented on the *provider* side and then uses the modified *OpenAPI* to generate an updated client library version for the *consumer* to use. The *consumer* could still start implementation earlier, but could not finish the implementation before the client library was released. No checks were in place to prevent incompatible *API* versions from being merged onto the main branch or from being deployed onto an environment. A concept of indicating breaking changes using *SemVer* was proposed and planned, but would still have required manual checking for major *API* version updates, which would have slowed deployment and would have been insufficient for the long term goal of *continuous deployment*.

While the implementation of *contract tests* aims to resolve these issues, the chosen, *consumer* driven implementation is incompatible with the existing *provider* driven development approach. The *consumer driven contract testing* expects the *consumer* to start the implementation and generate the contract. In contrast, the current approach expects the *provider* to start the implementation and generate the *API* client to be used by the *consumer*. The implementation of a *bidirectional* approach, compatible with *provider* driven development would have been possible, but was not pursued for multiple rea-

sons. The implementation of bidirectional *contract tests* would have required the implementation of *API* tests that verify, that the *provider* implements the described interface. Moreover, a switch to *consumer* driven development was aspired regardless of *contract testing*, as this development style promises to reduce redundant implementations and to tailor implementations closer to the use case of a feature. When switching to *consumer* driven development, the changes required in the development process temporarily slow down the implementation process as teams need to get familiar with new processes, tools and dependencies. Furthermore, additional changes need to be made to the project to support this development style. For example, the *consumer* can no longer use a client generated from an interface description published by the *provider*, as the *consumer* implementation should occur before the *provider* implementation. Similarly, the usage of generated code no longer provides any significant benefit to the *provider* implementation, if the correctness of the implementation can be verified by contract tests. Because of the amount and effort of the required changes, the customer decided to postpone the evaluation of *consumer* driven development and therefore *contract testing* in the regular development process to the next implementation cycle. As that does not begin within the time frame available for this thesis, the usage of *contract testing* within the regular development process cannot be observed in the scope of this thesis. The following concept of *contract testing* adoption was therefore prepared, but not yet adopted at the time of writing.

The branching model visualized in Figure 5.2 is recommended by *Pact* to be used with *consumer* driven *contract testing*. In contrast to *provider* driven development, the implementation is begun on the *consumer* side. The updated *contracts* are published to the *Pact broker*. Based on these, the *provider* is able to perform the required changes. After successfully verifying the *contracts*, the *provider* implementation is merged onto the main branch. If *contracts* could be merged on the *consumer* side already before the *provider* has completed the implementation, no more *consumer* deployments could be performed until the *provider* merges and deploys an implementation that supports the new *contracts*. In the meantime, more *consumer* implementations awaiting implementation by the provider could be merged, blocking deployment until all *providers* have caught up with the *consumer*. To prevent this situation from occurring, verification results are only published when the *provider* merges to the main branch. *Consumer* branches are blocked from merging, as long as their *contracts* have not been verified. While this leads to longer living *consumer* branches compared to *provider* driven development, it also guarantees, that for each state of the *consumers* main branch a compatible *provider* version exists. Like described in 5.5, an additional compatibility check is performed before deploying a service, as the merge only checks compatibility between the latest version, which might not be deployed to the target environment.

The *consumer* might make changes to the *contract* that do not require implementation on the *provider* side, like using an additional field that is already present, because it is needed by another *consumer* as well. As the *provider* does not open a merge request, the *consumers contracts* cannot be verified. To circumvent this, the *Pact broker* triggers the execution of the verification job on the *provider*'s main branch. This job also includes the

unverified *contracts* and publishes the corresponding verification results. If the *provider* already supports the changed *contracts*, the verification succeeds and the *consumer* branch can be merged without any action required on the *provider* side.

5.8 SUMMARY

As *contract testing* encourages an independent development of the *participants*, the adoption of it can also be performed one service at a time. The implementation was first performed for the *REST* interface and the event interface afterwards. In both cases a proof of concept *contract test* for a single endpoint was first created for the *consumer* side and the *provider* side closely after each other, to make sure that the generated *contracts* are correct. Based on these, additional tooling was created to simplify the remaining adoption. Besides the implementation of the tests itself, the pipeline needed to be configured to run the tests and publish them to a *broker* responsible for managing the *contracts*. As the previous development style was *provider*, instead of *consumer* driven, changes needed to be made to the branching style and the project as a whole as well. Due to the amount of changes and the effort required to implemented them, the inclusion of *contract testing* into regular development could not be evaluated.

6 RESULTS

As many metrics are influenced by specific decisions and supporting implementations, this chapter summarizes them for each metric. Based on the implementation of *contract testing* in the case study the metrics described in 4.2 can be recorded. The recorded results are presented in this chapter as well. Using these results the evaluation can be performed in the following chapter. Because the integration of *contract testing* into the development process was postponed to the next development cycle, the recorded metrics are limited to the feature branches *contract testing* was implemented on. This may lead to different metric values than if *contract testing* was adopted in the regular development process. In some cases the values that would have occurred in regular development can be approximated with workarounds, for example, by manually triggering pipelines to get the average execution time. In other cases the recorded metrics can be used as an initial indicator, but how the recorded results differ from the ones after adoption needs to be estimated argumentatively. How this was dealt with is described for each metric separately.

6.1 FEEDBACK SPEED

Contract testing is marketed as providing faster feedback than *end to end testing*, accelerating development and avoiding unnecessary reiterations on the same feature. The main reason for this is the time the tests require to run, as faster tests can be executed earlier and more often. An important decision in the implementation of *contract testing* to additionally speed up the contract tests was the usage of the `WebMvcTest` annotation in the *Spring Boot* based participants. Instead of initializing the whole application, it only initializes the controller layer. Another aspect influencing the test speed is the scale of the tests. The *REST provider* tests were extended to the service layer, but aside from performance, this approach has shown to be a lot harder to develop and maintain. A reversion to a controller layer only *contract tests* is therefore likely to be performed if *contract testing* is adopted. A similar approach may be possible for the event *consumer* and *provider*, in which only those services are initialized that are needed for message parsing. Alternatively, the parsing of messages may be moved out of the service layer into separate classes

Participant	Job Time	Test Time
REST Consumer	2m 30s	0m 13s
REST Provider	3m 12s	0m 31s
Event Consumer	2m 48s	0m 27s
Event Provider	7m 03s	6m 18s
Sum	15m 34s	7m 29s

Table 6.1: Contract Test Execution Time

entirely. The *contract test* could then be executed without initializing any *Spring Boot* layers at all. The tests may further be accelerated if the *test data builders* are generated by a compiler plugin, as the class inspection would then only need to be performed if the target class changes.

The feedback speed was measured using pipeline executions times. In the case study, the pipeline jobs for *end to end tests* take between one and a half and three and a half hours, depending on the amount of test cases executed. Even if all available tests are executed, only a small subset of the available features is tested. Because of their speed, but also their reliance on the full system to be available, the *end to end tests* are only run a when they are manually triggered.

The amount of pipeline jobs on the feature branches was not large enough to get a reasonable average. Instead, the contract-testing jobs were rerun to get at least 30 runs of the final *contract testing* implementation. The average time required to run the implemented *contract tests* fore each *participant* and interface is shown in Table 6.1. The *Job Time* column refers to the time the whole job took, including runner setup, code checkout, installing dependencies, compiling and running the tests, cleanup and artifact publishing. The *Test Time* column refers to the time the tests itself needed to execute, as reported by the corresponding test tool. While the earlier shows the effect of *contract testing* on the pipeline execution time and therefore where they can be placed in the pipeline order, the latter shows how much time is approximately required to compile and execute the *contract tests* locally during development. The *contract tests* are executed within the same unit testing frameworks as the regular unit tests, and can be run and debugged locally using pre-configured build tool commands.

6.2 STABILITY

A major aspect in the stability of the *contract tests* is their ability to be executed locally. This has further been supported by further implementations, especially in the *Spring Boot* based participants. The implementation of the *MissingBeansMocker* makes it possible to add *Beans* to the *microservice*, without having to explicitly specifying that they need to be mocked in the *contract tests*.

The stability was measured by comparing the share of false negative pipeline jobs of both testing strategies. The last 30 *end to end test* pipeline jobs contain one failed pipeline. This however did not fail due to a detected bug or a false negative, but rather an erroneous pipeline configuration. Similarly,

the pipeline jobs on the feature branches contain many failed *contract testing* jobs that occurred as the *contract tests* were not yet setup correctly. Cases in which *contract tests* were unreliable occurred only once, because *Pact* was not configured correctly for parallel test execution. In some pipelines this caused file locking issues on the *contract* file, which failed the tests. The issue could easily be fixed by disabling parallel test execution. Because the contract tests are stateless, test isolation could be disabled additionally. This combination made the tests even faster than before with parallel, isolated test runners. In the subsequent repeated execution of the pipeline jobs to record the average job time, no further false negatives occurred.

6.3 DEVELOPMENT & MAINTENANCE EFFORT

A major factor in test development and maintenance effort is the readability of the tests. As a result, a considerable amount of implementation effort in this thesis has been spent on ensuring good *contract test* readability and expressiveness. Supporting implementations, especially for test data creation have been implemented for all four kinds of *participants*. The implemented *test data builders* make it easy to create instance of data classes for any test level. By using reflection, instead of individual builder implementations, maintenance can be kept low for the builders as a whole, but it also prevents modifications to the builders being necessary each time the target class changes. The factory functions make the creation of builders even more readable. By generating both builders and factory functions using a compiler plugin, implementation and maintenance effort may be reduced further. In addition, the calls to override defaults would become type safe, making it even easier to identify mistakes during development. Helper functions in the *REST consumer* simplify the creation of *Pact* instances and *API* clients. All of the tooling is structured in a way that makes it possible to extract it to a library, so it can be used by other *microservices* as well, if *contract testing* is adopted.

The test development and maintenance effort cannot be rated numerically in the scope of this thesis, so an argumentative rating is performed instead. The creation of *end to end tests* in the case study system uses specialized tooling and requires access to a specific system instance that runs a full system instance. The *end to end tests* are therefore written by a separate, specialized team. For *contract testing*, the new testing strategy is likely to require some time to get the developers familiar with it. However the created tooling has the potential to heavily reduce the time, developers need to write tests, that verify compatibility between *microservices*.

6.4 DEFECT DETECTION

The defect detection accuracy is largely depended on characteristics of the testing strategy in question. Consequently, little implementations could be performed to improve defect detection directly. However, weak spots in *contract testing*, like the difficulty to represent nullable fields were identified, which is important for developing strategies to handle these hurdles. As

described earlier, in this particular case, the strategy may actually be an additional implementation to generate additional interactions with nullable fields unset.

Contract testing claims to prevent deployment of incompatible services by preventing the merge of unverified *contracts* and tracking the deployed versions. The amount of captured defects was supposed to be measured by counting the amount of true negative pipeline jobs. In the last 30 *end to end testing* pipelines there was none failing because of a defect in the software, although bugs were being reported after deployment. The *end to end tests* cover only cover a portion of the software.

As *contract testing* could not be included into regular development within the scope of this thesis, no true negative pipelines could be recorded. However, the *contract testing* implementation managed to detect that the event *provider* and event *consumer* were using an incompatible version of the interface on their main branches. This issue was known by developers, but prevented deployments of these services to the test environment, as the *consumer* depended on new implementations of the *provider*, but had not implemented support for the breaking changes introduced earlier. If *contract testing* had been implemented earlier it would have prevented the event *provider* from merging the breaking changes before they are adopted by the *consumer*.

6.5 SUMMARY

In summary *contract testing* shows promising results in multiple metrics. Feedback loops can be accelerated and stability increased. Cross version compatibility and default detection accuracy could not be tested extensively but identified incompatibilities on the feature branch already. Test development and maintenance effort could be kept low. In many cases the metrics were improved by supporting implementations.

7 EVALUATION

By discussing the recorded results in this chapter, an evaluation can be drawn, whether *contract testing* fulfills the requirements described in 4.1 and therefore resembles a viable alternative to *end to end testing*. By identifying aspects special to this case study implementation it can be evaluated how the recorded results are applicable to the usage of *contract testing* in other projects.

7.1 FEEDBACK SPEED

The implementation in this case study has shown, that *contract tests* can indeed easily be executed and debugged locally. The *contract tests* of the event *provider* the case study are considerably slower than the other *participants*. As the event *provider* tests are little different from *REST provider* tests in terms of complexity, this can likely be improved to a similar execution time. Depending on the framework in use, the execution time of the *contract tests* needs to be performed in a specific way to benefit from the ability to execute at the fast *unit test* level. In the case study this was achieved by using the `WebMvc` annotation to only initialize the controller and service levels of the *Spring Boot* application, instead of the entire one. Both *end to end tests* and *contract tests* do not cover the entire system at the moment and in both cases it is difficult to estimate the execution time for the whole system. But the *contract tests* have the significant advantage that to check the compatibility of a single service, only the *contract tests* of this service need to be executed. The time measured for the implemented tests indicate, that this can likely be achieved in less than five minutes. With *end to end tests* the whole suite would need to be executed, which takes more than an hour with the current tests already. This, and the fact that no full system is needed allows the *contract tests* to be executed more often and earlier in the development, providing much faster feedback. The recorded times show that when the tests are executed in the pipeline in most cases the majority of the execution time is needed to setup the job environment instead of the actual tests. As this setup is only needed once when executing the *contract tests* locally, they execute even faster after the initial setup, making them well suited to be used during development, like it can be done with *unit tests*.

7.2 STABILITY

Contrary to the claims made by contract testing tools, the *end to end tests* in the case study project did not fail unnecessarily. This may be partially caused by the presence of timers within the *end to end tests*, that prevent them from checking assertions too early. However this strategy also makes *end to end testing* even slower.

While they have not been included in regular development, the implemented *contract tests* have shown to be stable and did not fail any pipelines unnecessarily. This is mainly caused by two main reasons. Firstly, the *contract tests* do not contain any asynchronous operations and therefore avoid tests that fail due to events occurring in unexpected order or not within the expected time frame. Secondly, the tests do not require a full system setup to run, which can cause tests to fail if it is in an unexpected state, for example if some system components are not properly configured.

7.3 DEVELOPMENT & MAINTENANCE EFFORT

The switching to *consumer* driven development and preventing the *consumer* branch from being merged onto the main branch before the *provider* finishes implementation may lead to longer living feature branches in the *consumer* repository. This might introduce merge conflicts that require additional effort to be solved. The scale of the effort is difficult to estimate and likely varies between projects, as changes to the same lines of code, and therefore merge conflicts, might occur at different frequency. This effect may be mitigated, by separating the implementation of changes to the interface from the rest of the feature. While the branch with the interface changes await the corresponding implementation in the *provider*, the feature branch with the rest of the changes may already be merged. For newly introduced fields or endpoints placeholder values may be used.

The *end to end tests* in the case study are developed by a separate, dedicated team, while the *contract tests* would need to be written by the developers. While an adoption of *contract testing* would shift more testing effort to the development teams, effort is reduced in other aspects, as no generated *API* clients need to be maintained and updated anymore. In addition, the faster feedback prevents having to revisit features implemented in earlier iterations. To keep the effort to create and maintain *contract tests* low, the presented implementation introduced several helper functions, especially to create the required test data and corresponding matchers. While the creation of this tooling required a somewhat large effort initially, it reduces the effort of adopting in other *microservices* and maintaining them. In other projects the development of *end to end tests* may be responsibility of the regular development team. Thus, the shift in testing effort might not be as visible as in this case. Nevertheless, is the availability of assisting tooling important, as it does not only reduce maintenance effort but also reduces the entry barrier for developers to get familiar with *contract tests*.

7.4 DEFECT DETECTION

The ability of *contract tests* to capture defects could only be verified to a certain extend, as the *contract tests*, especially pre-deployment compatibility checks were not included in the regular development process. It is to be expected that *end to end tests* capture some defects that *contract tests* do not, because *end to end tests* capture conceptual errors. If a service is expected to send a specific event in a specific scenario, *contract testing* can only verify that the service sends the event in the correct format, but not if it actually sends the event when needed. If this scenario is captured in an exhaustive *end to end test*, it fails if the service does not send the event. The maintainers of *Pact* are aware of this and recommend to reduce the *end to end tests* to those that test the business logic (Pact Foundation 2023a). In a further step, these can be converted to *smoke tests*, which verify that the production system works as intended after a deployment. Should this not be the case, the deployment can be reverted. Converting the *end to end tests* to *smoke tests* eliminates the need to setup and maintain environments needed for *end to end testing*. Furthermore, no more configuration gap exists between the environment the *end to end tests* run and the one the services are deployed to.

Another aspect regarding the ability of *contract tests* to capture defects is the complexity of treating nullable and optional fields. As described before, the *contracts* require all fields to be set to be able to capture the structure of all fields, but this also means that the *contracts* contain no information on which fields are not required. While solutions to this are being investigated, this is one of the major drawbacks of *contract testing* compared to the previous approach of using generated clients.

Nevertheless, the implementation on the feature branch discovered, that the event *provider* and event *consumer* deviated in the models used on the interface failing the *contract tests*. This showed that *contract tests* are in fact able to determine incompatibilities between *contract testing* versions. The pre-deployment checks do not run any *contract tests* themselves but instead rely on existing verification results to determine if a service can be safely deployed. This mechanism is therefore likely to be reliable.

7.5 SUMMARY

In conclusion *contract tests* fulfills the requirements to be a viable alternative to *end to end tests*. During the implementation in the case study project *contract testing* introduced some hurdles. Most of them, like making the *contract tests* and especially the test data creation well readable and maintainable could be solved. One major hurdle that still needs to be solved is how to deal with optional and nullable fields. However, even at the current state the presented benefits outweigh the shortcomings. Especially the accelerated feedback loops offer the potential to improve development and deployment. Thus, it is also recommended to extend *contract tests* to the *REST* of the case study system, including extension to other services as well as adoption of pre-deployment compatibility checks. The consideration of case study

specific aspects within the evaluation allows to conclude, that the discovered results are applicable to other large *microservice* systems as well.

8 OUTLOOK

The goal of this thesis was to evaluate the viability of *contract testing* as an alternative to *end to end testing* for large *microservice* systems. To achieve this, metrics to determine the viability were set and *contract testing* was implemented for a case study project. Based on the recorded metrics *contract testing* could be compared to the existing *end to end tests*. The results were then discussed to determine their applicability to *contract testing* in general. Finally it was determined that *contract testing* can be a viable alternative to *end to end testing*. Consequently, it was also recommended that *contract tests* should be extended to the *REST* of the case study system. Although all relevant aspects were considered, the research field still provides the potential for further studies. This chapter gives an outlook into opportunities for future research on *contract testing* in large *microservice* systems.

8.1 NULLABLE AND OPTIONAL FIELDS

Contract tests require all fields to be set on both the *consumer* and *provider* side. Otherwise the fields could not be integrated into the *contracts*. Similarly arrays may not be empty. This necessity removes any information on nullability or optionality and minimal array sizes from the *contract*. This is especially problematic, as the previously used version of using an *API* schema to generate the *API* implementations did include this information. To include in the *contract*, which fields are optional and which arrays may not be empty, additional *interactions* are needed, in which all nullable fields are null. Depending on the complexities of the models in use, this might result in a high maintenance effort and a lot of duplication. This may be mitigated by being able to generate the second interaction automatically. The helper methods and *test data builder* pattern used in the described implementation could potentially be modified to automatically generate two or more requests with a single description. This would still increase the execution time, but maintain readability and maintainability.

8.2 DETECTION OF UNUSED FIELDS

The usage of *consumer* driven *contract testing* enforces the requirements of the *consumer* to be defined before any implementation is done at the *provider*. As a result, no superfluous endpoints and unnecessary fields are implemented. To identify existing unused endpoints the *contract tests* can be run with coverage. Any endpoint that is uncovered is unused. At the moment there is no automated way to determine which existing fields are unused. It is only possible to use *contract tests* to verify whether the removal of a field was safe after it has been removed. This, however, can also be achieved with *end to end testing*, although the feedback is slower. Future research could investigate a tool automatically identify unused fields, by verifying the *contracts* of all *consumers* and recording the *provider's* responses. The recorded responses can then be compared with all of the *contracts* to identify all fields, that have not been used by any of the *consumers*.

8.3 EXPANSION OF CONTRACT TESTING

Due to the limited time frame of this thesis, *contract testing* was only implemented for a subsection of the system. An adoption of *contract testing* into the regular development cycle was not performed for organizational reasons. Further research may investigate, how *consumer* driven development and *contract testing* can be adopted in the development process and be extended to the rest of the system. Multiple focuses are possible. It could be researched how much the tools developed for the subsystem assist and accelerate the adoption of further services. Alternatively, it could be investigated how the expansion is best performed. The remaining services could adopt *contract testing* all at once. Another method could be a wave like expansion, where each service first implements *provider* tests, when any of its *consumers* implemented its *contract tests*. Afterwards the *provider* could then implement its own *consumer contract tests*.

ACRONYMS

API	Application Programming Interface.
DSL	Domain Specific Language.
GUI	Graphical User Interface.
HTTP	Hypertext Transfer Protocol.
JSON	JavaScript Object Notation.
REST	Representational State Transfer.
URL	Uniform Resource Locator.

GLOSSARY

broker	Service used to transfer data between other services.
consumer	Service that consumes the resource provided by an interface.
continuous deployment	Deployment strategy in which changes are released as soon as they have been added to the main branch.
contract	See *contract testing*.
contract test	See *contract testing*.
contract testing	Testing interfaces using separate tests on *consumer* and *provider* side by using a contract to describe the interface.
end to end testing	Testing a system in its entirety, including all services and database connections.
end to end test	See *end to end testing*.
event driven messaging	Communication that is based on events published by the provider and observed by the consumer.
integration testing	Testing compatibility between selected units of a system or system component.
integration test	See *integration testing*.
interaction	Communication between the *participants* of an interface.
Java	Programming language that compiles to bytecode for the *Java* virtual machine.
JavaScript	Scripting and Programming Language for Web Development.
Kotlin	Programming Language that compiles to *Java*.

microservice	Component of a system, in which each functionality is located in its own component for better scalability.
mock	A replacement for a system component used to be able to run tests without needing the original.
Pact	*contract testing* framework used in this thesis.
participant	Any service that is *consumer* or *provider* on an interface.
provider	Service that provides a resource on an interface.
Spring Boot	Web Framework for *Java* based languages.
test data builder	Design pattern to simplify test data creation, introduced by Freeman and Pryce (2010).
TypeScript	Programming Language that compiles to *JavaScript*.
unit testing	Testing individual units of a system or system component.
unit test	See *unit testing*.

Broadcom (2024). *Spring Cloud Contract Home Page*. URL: https://spring.io/projects/spring-cloud-contract (visited on 09/03/2024).

Cohn, Mike (2010). *Succeeding with Agile: Software Development Using Scrum*. online. booklet. EDS: edsebk:1599331; Quelldatenbank: EDS; Sprache: English; Nachgewiesen in: eBook Index; ddc: 005.1; Resource Type: eBook.; (electronic).

DeBellis, Derek and Nathen Harvey (Oct. 5, 2023). *2023 State of DevOps Report: Culture is everything*. URL: https://cloud.google.com/blog/products/devops-sre/announcing-the-2023-state-of-devops-report?hl=en (visited on 09/03/2024).

DeepL SE (2024). *DeepL Translator*. URL: https://www.deepl.com/en/translator (visited on 09/03/2024).

Dobles, Ignacio, Alexandra Martínez, and Christian Quesada-López (2019). "Comparing the effort and effectiveness of automated and manual tests". In: *2019 14th Iberian Conference on Information Systems and Technologies (CISTI)*, pp. 1–6. DOI: 10.23919/CISTI.2019.8760848.

Faber, Szczepan et al. (2024). *mockito-kotlin Repository*. URL: https://github.com/mockito/mockito-kotlin (visited on 09/03/2024).

Fowler, Martin (Jan. 2, 2007). *Mocks Aren't Stubs*. URL: https://martinfowler.com/articles/mocksArentStubs.html (visited on 09/03/2024).

– (May 30, 2013). *Continuous Delivery*. URL: https://martinfowler.com/bliki/ContinuousDelivery.html (visited on 09/03/2024).

– (Mar. 25, 2014). *Microservices*. URL: https://martinfowler.com/articles/microservices.html (visited on 09/03/2024).

– (Nov. 14, 2016). *Value Object*. URL: https://martinfowler.com/bliki/ValueObject.html (visited on 09/03/2024).

Freeman, Steve and Pryce, Nat (2010). *Growing object oriented software, guided by tests*. 2. print. The Addision Wesley signature series. Upper Saddle River, NJ [u.a.]: Addison-Wesley. ISBN: 9780321503626; 0321503627. URL: http://digitale-objekte.hbz-nrw.de/storage/2010/06/05/file_3/3790740.pdf.

Fu, Anthony, Matías Capeletto, and Vitest contributors (2024). *Vitest Homepage*. URL: https://vitest.dev/ (visited on 09/03/2024).

Gartner (Nov. 13, 2023). *Gartner Forecasts Worldwide Public Cloud End-User Spending to Reach $679 Billion in 2024*. URL: https://www.gartner.com/en/newsroom/press-releases/11-13-2023-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-679-billion-in-20240#:~:text=Worldwide (visited on 09/03/2024).

Groß, Hans-Gerhard et al. (2003). "Built-in contract testing for component-based development". In: *Business Component-Based Software Engineering*, pp. 65–82.

Hernandez, Christian (2023). "Contract Testing: Ensuring Reliable Integrations with Isolated Tests: Support for teams to test in isolation". thesis. Mittuniversitetet, Institutionen för data- och elektroteknik (2023-): Mid

Sweden University. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:miun:diva-48570.

Hibernate (2024). *Hibernate Validator*. URL: https://hibernate.org/validator/ (visited on 09/03/2024).

Inoto, Ruben (May 7, 2016). *spring-auto-mock Repository*. URL: https://github.com/rinoto/spring-auto-mock (visited on 09/03/2024).

JetBrains s.r.o. (June 19, 2024). *Intellij Idea Documentation - Get started with Kotlin*. URL: https://www.jetbrains.com/help/idea/get-started-with-kotlin.html#convert-java-to-kotlin (visited on 09/03/2024).

Jína, Vojtěch (2013). "Javascript test runner". In: *examensarb., Czech Technical University in Prague*.

Khattak, Wajid and Vijay Narayanan (Mar. 27, 2010). *SOA Pattern (#11): Event-Driven Messaging*. URL: https://www.informit.com/articles/article.aspx?p=1577450 (visited on 09/03/2024).

Labuda, Bc Dominik (2019). "Contract Testing in Heterogeneous Distributed Systems". MA thesis. Masaryk University.

Lehvä, Jyri, Niko Mäkitalo, and Tommi Mikkonen (2019). "Consumer-driven contract tests for microservices: A case study". In: *Product-Focused Software Process Improvement: 20th International Conference, PROFES 2019, Barcelona, Spain, November 27–29, 2019, Proceedings 20*. Springer, pp. 497–512.

Martin, Robert C. (Aug. 1, 2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. First edition. Prentice Hall. 464 pp. ISBN: 978-0132350884.

– (May 8, 2014). *The Single Responsibility Principle*. URL: https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html (visited on 09/03/2024).

Nagel, Florian and Martin Leucker (2019). *Analysis of Consumer-driven contract tests with asynchronous communication between microservices*.

Newman, Sam (2019). *Monolith to microservices: evolutionary patterns to transform your monolith*. First edition, second release. Beijing: O'Reilly. ISBN: 9781492047810. URL: https://eu04.alma.exlibrisgroup.com/view/uresolver/49HBZ_WHS/openurl?u.ignore_date_coverage=true&portfolio_pid=5316993420006456&Force_direct=true.

OpenJS Foundation (2024). *Jest Homepage*. URL: https://jestjs.io/ (visited on 09/03/2024).

Pact Foundation (Aug. 30, 2022). *Pact Docs - Introduction*. URL: https://docs.pact.io/ (visited on 09/03/2024).

– (June 2, 2023a). *Pact Docs - FAQ - Do I still need end-to-end tests?* URL: https://docs.pact.io/faq#do-i-still-need-end-to-end-tests (visited on 09/03/2024).

– (2023b). *Pact Homepage*. URL: https://pact.io/ (visited on 09/03/2024).

– (2024a). *jest-pact Repository*. URL: https://github.com/pact-foundation/jest-pact/ (visited on 09/03/2024).

– (2024b). *Matching*. URL: https://docs.pact.io/implementation_guides/javascript/docs/matching (visited on 09/03/2024).

– (2024c). *Pact - Convice Me*. URL: https://docs.pact.io/faq/convinceme (visited on 09/03/2024).

– (July 17, 2024d). *Pact Docs - Consumer Tests*. URL: https://docs.pact.io/implementation_guides/javascript/docs/consumer (visited on 09/03/2024).

Pact Foundation (June 5, 2024e). *Pact JVM Repository - Feature Request: Individual Consumer Test Case per Async Message Interaction*. URL: https://github.com/pact-foundation/pact-jvm/issues/1801 (visited on 09/03/2024).

– (Apr. 22, 2024f). *Pact Spring/JUnit 5 Support*. URL: https://docs.pact.io/implementation_guides/jvm/provider/junit5spring (visited on 09/03/2024).

Robinson, Ian (June 12, 2006). *Consumer Driven Contract Tests*. URL: https://martinfowler.com/articles/consumerDrivenContracts.html (visited on 09/03/2024).

Schuh, Peter and Stephanie Punke (2001). "Objectmother: Easing test object creation in xp". In: *XP Universe*.

Smart Bear Software (2024). URL: https://pactflow.io/ (visited on 09/03/2024).

Smith, Sheldon et al. (2023). "Benchmarks for End-to-End Microservices Testing". In: *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pp. 60–66. DOI: 10.1109/SOSE58276.2023.00013.

Stack Exchange Inc. (2024). *StackOverflow Developer Survey 2024*. URL: https://survey.stackoverflow.co/2024/ (visited on 09/03/2024).

The Linux Foundation (2024). *OpenAPI*. URL: https://www.openapis.org/ (visited on 09/03/2024).

Vahlbrock, Tim (2024). *Archive of the Representation Implementation*. URL: https://archive.softwareheritage.org/browse/revision/8d107b9f6b2f314edaff3831fee9?origin_url=https://github.com/timvahlbrock/master-thesis-listings (visited on 09/03/2024).

Vocke, Ham (Feb. 26, 2018). *The Practical Test Pyramid*. URL: https://martinfowler.com/articles/practical-test-pyramid.html (visited on 09/03/2024).

Vu, Duy Anh (2022). "Harmonization of strategies for contract testing in microservices UI". B.S. thesis. Teampere University.

EIDESSTATTLICHE VERSICHERUNG

Ich, Tim Vahlbrock, versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem Titel

Evaluation of Contract Testing
in a Large Microservice System

selbstständig und ohne unzulässige fremde Hilfe erstellt habe. Die Verwendung von *Large Language Models* beschränkt sich auf *DeepL* (DeepL SE 2024) zur Übersetzung von einzelnen Worten oder Phrasen. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Gelsenkirchen, 03. September 2024

Ort, Datum, Unterschrift // Place, Date, Signature