Syntactic Interoperability in Microservice-based Architectures

MASTER THESIS

Johannes Lukas Jablonski

Submitted on May 31, 2023

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Georg Schwarz, M.Sc.
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, May 31, 2023

License

Erlangen, May 31, 2023

Acknowledgments

I would like to thank the JValue team, especially Georg, for their always quick and helpful support.

In addition, I would like to thank my interview partners without whom it would not have been possible to conduct this research.

I would also like to acknowledge InstaText, Grammarly, and ChatGPT for helping me overcome my deficits in grammar and vocabulary as a non-native speaker.

Finally, special thanks to all of my friends and family who proofread this work.

Abstract

Nowadays, many applications are developed using a microservice-based architecture. In this context, a software product consists of many individually developed services that communicate over a network. As microservices do not necessarily share a common code base, syntactic interoperability remains a major challenge with this architectural style. The objective of this work is to provide a conceptual framework that helps overcome syntactic interoperability issues in microservice-based architectures.

This thesis follows the design science research methodology defined by Peffers et al., 2007. Interviews with industry experts are conducted to gain insight into current problems and state-of-the-art solutions concerning interoperability. These insights are processed to create a categorization system. The categorization system's applicability is demonstrated in a case study within the JValue project. Finally, the categorization system is validated through triangulation and member checking to confirm its credibility.

We have found six categories with two to three characteristics each that are important when developing solutions to overcome interoperability in microservice-based software projects. The categorization system proves itself as a valuable tool for analyzing and improving software processes with respect to interoperability issues in microservice-based architectures.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

API Application Programming Anterface

CI/CD Continuous Integration and Continuous Deployment

ETL Extract, Transform, Load

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notation

OData Open Data Protocol

RPC Remote Procedure Call

REST REpresentational State Transfer

SDK Software Development Kit

SOAP Simple Object Access Protocol

TCP Transmission Control Protocol

UML Unified Modeling Language

URL Uniform Resource Locator

XML Extensible Markup Language

1 Introduction

Microservice architecture has gained in popularity over recent years. In contrast to the monolithic approach for software development, applications based on a microservice architecture consist of many independently deployable services. This way of designing applications leads to loosely-coupled and relatively small systems that usually communicate over a network (Lewis and Fowler, 2014). A microservice architecture causes design challenges that are typical for distributed systems, among them service interface design and syntactic interoperability between microservices and a microservice with its client, respectively (Zimmermann, 2017).

Syntactic interoperability is a key challenge of microservices that does not occur to an equal extent in monolithic applications. The latter are developed in a single programming language and are deployed as one executable (Lewis and Fowler, 2014). Therefore, the modules of an application do not communicate over a network and the interfaces cannot get out of sync because they are usually validated by a compiler when the executable is built. In contrast, the services of an application based on a microservice architecture can be and usually are developed in different languages and do communicate over a network (Lewis and Fowler, 2014). Especially independent development of microservices could cause the interfaces to become out of sync and incompatible across different deployments.

Stability due to fixed communication structures is a benefit of monolithic applications and a major disadvantage of microservices. As microservices constitute software products by interacting, communication between them is obligatory. There are different ways to do so, including but not limited to communication types like REpresentational State Transfer (REST), Simple Object Access Protocol (SOAP), and Messaging (Wolff, 2016). Even if such a communication type is determined, it may still not be specified how exchanged data is structured, i.e., which attributes and which data types constitute it. This information has to be exchanged with clients to ensure that actual service communication can be implemented. However, even if the interfaces are perfectly designed internally, a client needs to receive information about what a microservice's interfaces look like, especially whether and how they have changed (Newman, 2015).

The initial objective of this thesis is to find suitable ways to ensure syntactic interoperability in the context of a microservice-based project. This vague goal is explored in more detail by conducting interviews with domain experts so that the following research question can be determined:

> *How can problems and solutions related to internal syntactic interoperability in microservice-based architectures be categorized at a conceptual level?*

A detailed derivation and explanation of the research question of this thesis are given in Chapter 3.

1.1 Microservices

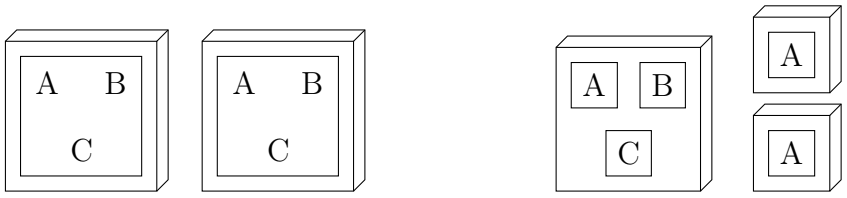
Microservices is a term that describes a possible architecture of a software application (Lewis and Fowler, 2014). They combine concepts of distributed systems and service-oriented architectures to profit from both of their advantages (Newman, 2015). Many building blocks and patterns of microservices existed before the advent of microservices, for example, domain-driven design, continuous delivery, and systems at scale (Newman, 2015). While there is no precise definition of what a microservice-based architecture is, there is a common understanding of what characteristics these typically have (Lewis and Fowler, 2014).

One attribute of microservices is that the services are established around teams of one business capability, where each team has the obligation to create its piece of software but also has the freedom to choose its software setup for that development (Lewis and Fowler, 2014). This approach is used by several big companies (Newman, 2015). It follows that each team can also choose which technologies to use. This can speed up development and can open the door to the usage of new technologies not utilized in any other service before (Newman, 2015).

Each microservice is a cohesive executable, however, as they typically constitute one comprehensive project, they need to communicate with one another. This functions with network calls. Newman, 2015 names *request/response* (e.g., Remote Procedure Call (RPC) or REST) and *event-based* (e.g., message brokers) as two opposing principles of how services can communicate via a network. Each microservice provides an Application Programming Anterface (API) of its choice that ensures that other services can access it for exchanging data (Newman, 2015).

Another key aspect of microservices resolves from the fact that they are developed as autonomous applications: they can be deployed individually one at a time instead of being required to be deployed as an entire huge product at once. Like most other characteristics presented in this section, the advantage of this becomes

clear when comparing it to another common way to architect software: a monolith. Monolithic applications combine all features in one large executable, which has to be fully rebuilt and redeployed even if only a small part changes (Lewis and Fowler, 2014). However, it is common that not each feature of an application is updated and accessed by users in the same amount. Thus it becomes useful that microservices can be deployed independently, as they can be used to scale up the availability of one feature more efficiently than within a monolith (Newman, 2015). Figure 1.1 depicts this difference with an example of three features A, B, and C. Feature A is one that is needed more often than the others. Despite B and C performing as required, the monolith must be scaled with all features together (Figure 1.1a). In a microservice architecture, each feature is implemented in one microservice which makes it possible to deploy more instances of the frequently needed services (Figure 1.1b). These small instances of service A can run on less powerful machines and so upscaling can be designed more cost-effective, often in combination with on-demand cloud systems (Newman, 2015).



(a) Monoliths must be scaled with all features.

(b) Microservices can be scaled individually.

Figure 1.1: Comparison between the scaling of monolithic and microservice-based architectures (Lewis and Fowler, 2014; Newman, 2015). A cuboid represents a physical server; an executable is depicted by a rectangle.

The fact that microservices are independently deployable creates multiple problems. One is that the API that one service expects might not be the one a communication partner provides. This typically arises when breaking changes are introduced, i.e., an API is updated and the new version is not compatible with the old one. Another problem is that multiple versions of one service might coexist at runtime. Newman, 2015 suggests overcoming these issues by trying to avoid introducing breaking changes, however, this is not always feasible. When breaking changes are inevitable it is important to communicate these changes to the clients (Newman, 2015).

To summarize, microservices are technology-independent systems that transfer information over a network. It is important to define how communication is configured, in the form of communication principles (like REST, Open Data Protocol (OData), or plain Transmission Control Protocol (TCP)) and in the form of data representation (like Extensible Markup Language (XML) or JavaScript Object Notation (JSON)). Each microservice has the freedom to be developed

with arbitrary technologies. For example, they can be written in different programming languages like Java, Python, or Scala. Figure 1.2 depicts such a setup of a heterogeneous service environment.

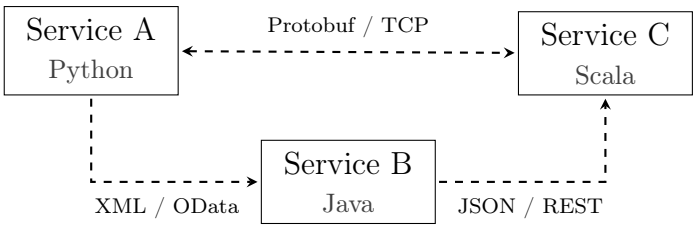


Figure 1.2: Microservices can be developed in different languages and use different protocols.

1.2 Interoperability

There are various definitions and interpretations of the term interoperability, however, a common ground is that interoperability enables communication between multiple systems (Van Der Veer and Wiles, 2008). Section 1.1 demonstrates several challenges in achieving interoperability between microservices. The European Telecommunications Standards Institute published a white paper that presents four different levels of interoperability (Van Der Veer and Wiles, 2008). We use this classification in order to define what level of interoperability should be focused on in this thesis:

- *Technical Interoperability* requires the existence of infrastructure that enables data exchange in the first place and also the existence of communication protocols for that.
- *Syntactic Interoperability* demands that exchanged data is well-defined, which includes data types and their representation.
- *Semantic Interoperability* is achieved when the content of the exchanged data is interpreted the same way by each communication partner. This usually refers to human understanding rather than to the interpretation by computers.
- *Organizational Interoperability* is reached when different organizations can communicate effectively.

This research focuses primarily on achieving syntactic interoperability. Organizational and semantic interoperability, which are mostly relevant when multiple organizations are involved, are considered less important for this study because microservices are typically used to develop a single application within an organization. Technical interoperability is partially satisfied implicitly, as microservices

communicate over a network, indicating the presence of a necessary infrastructure for data exchange. However, agreeing on a common communication protocol and data representation is not a straightforward task. A crucial requirement is that each communication partner is aware of the protocols supported by others and understands the structure of the data being exchanged. This thesis focuses on meeting this requirement, which will be referred to as syntactic interoperability or simply interoperability in the following chapters of this thesis. This focus is justified by the fact that syntactic interoperability builds upon the foundation of technical interoperability (Van Der Veer and Wiles, 2008) and therefore the successful exchange of data types is useless unless a common communication protocol exists. Thus, the two main forms of interoperability that are of major importance for microservice-based architectures are tackled by this thesis under the term syntactic interoperability.

1.3 Structure of this Thesis

The design science research methodology of Peffers et al., 2007 is used in this thesis to structure the path from a vague idea to a specified research question and finally to a solid solution artifact (Figure 1.3). Problem identification (Chapter 2) helps to gain a deeper insight into the overall topic of microservices and the industry's problems and solutions regarding syntactic interoperability. In the objective definition (Chapter 3), these challenges are analyzed superficially so that the research question can be defined: How can problems and solutions related to internal syntactic interoperability in microservice-based architectures be categorized at a conceptual level? A solution artifact for the research question is developed in the step solution design (Chapter 4) and yields a categorization system that helps to evaluate and plan syntactic interoperability solutions. The categorization system is utilized to show its benefit in cooperation with the JValue project in the demonstration step (Chapter 5). To prove the solidity of the developed artifact, it is evaluated using triangulation in Chapter 6. The final activity of Peffers et al., 2007 is communication where Peffers asks to distribute the work, which is done by writing and publishing this thesis. The thesis is concluded and an outlook is given in Chapter 7.

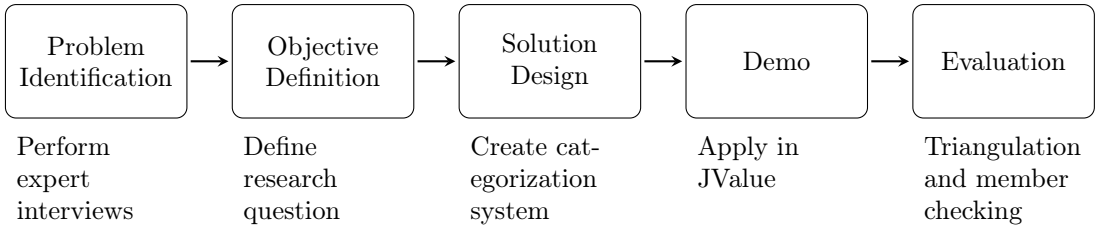


Figure 1.3: Design science steps from Peffers et al., 2007 applied to this thesis.

1.4 How to Read this Thesis

This thesis uses the third person plural (*"we"*) to comply with the style used in many scientific publications. This does not imply that the work is done by others than the author.

There are several diagrams throughout this thesis. They are based on Unified Modeling Language (UML) (Object Management Group (OMG), 2017) but contain simplifications and conventions so that they fit better into the context of this work.

There are four main types of nodes:

- Rectangles with rounded corners, which stand as activity nodes, i.e., process steps (Figure 1.4a).
- Rectangles with sharp corners, which are understood as object nodes and typically represent files or components (Figure 1.4b).
- Rectangles with triangle sides, which represent decisions (Figure 1.4c).
- Cuboids, which stand for hardware units (Figure 1.4d).

There are primarily two relationship edges used:

- Continuous, which stands for a process transition or which indicates that a file is being created or edited (Figure 1.4e).
- Dashed, which represents an accessing dependency. It is typically used to signify that a file is read but not written (Figure 1.4f).

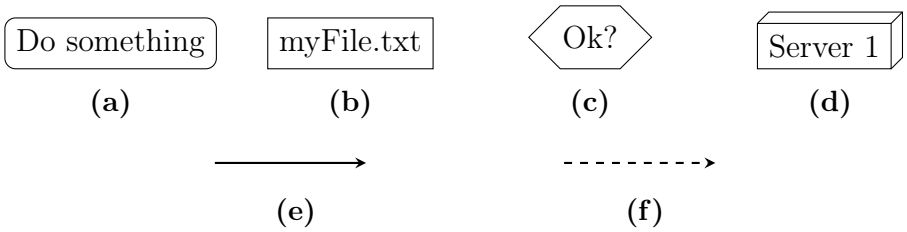


Figure 1.4: Different UML-inspired node and edge types are used throughout this thesis.

2 Problem Identification

The phase problem identification of the design science research methodology defines considerations underlying research and development (Peffers et al., 2007). To do so, we conducted interviews to gain knowledge about the overall topic and to understand the challenges faced by the industry. This chapter first presents the methodology of the interviews conducted and then summarizes the findings to provide a basis for the following tasks of the subsequent work, especially the objective design in Chapter 3.

2.1 Methodology

To learn from real-world projects about the diversity of challenges of syntactic interoperability, we conduct interviews for data collection because they can be a "tool for generating rich data for scientific inquiry" (Schultze and Avital, 2011). More specifically, semi-structured interviews are conducted. Such interviews contain an outline of the survey but still offer the freedom that details of the conversation can be conducted in a non-predefined way (Myers and Newman, 2007). This outline is also called an interview guide (Kallio et al., 2016). Such a guide ensures that all interviews comprise a common pool of topics covered but still provides room for additional off-topic and unplanned information (Schultze and Avital, 2011).

Creating an interview guide requires a "comprehensive and adequate understanding of the subject" (Kallio et al., 2016), which is achieved through a mix of prior experience and an informal literature review. The literature review also includes gray literature to obtain information on current practices in software engineering, as suggested in Garousi et al., 2016.

With the knowledge obtained, a preliminary interview guide was created. It is structured as suggested in Kallio et al., 2016: it starts with gentle warm-up questions, followed by superficial questions before asking for details. An interview ends with questions about open topics not yet covered.

Pilot testing of an interview guide is an important step to ensure that data

collection can be conducted properly (Kallio et al., 2016). Therefore, it is field-tested with one interview (see Appendix B.1). The guide proves useful with only minor adjustments required after this test. This final interview guide is presented in Subsection 2.1.1. All interviews are transcribed and (if allowed) published in Appendix B.

Based on a solid interview guide given, interviewees can be determined. A selection of heterogeneous experts and projects usually covers a broader application domain than a set of similar projects (Robinson, 2014). Interviewees are hand-selected and further interviews are added until theoretical saturation is reached, i.e., when more interviews would not yield new insights (Strauss and Corbin, 2008). The categories by which interview partners are selected are presented in Subsection 2.1.2.

2.1.1 Interview Guide

In this section, the interview guide is presented at an abstract level. A detailed version that is also provided to the interviewees prior to the execution of the interview can be found in Appendix A. The guide contains four sections, ranging from outline-level questions to detailed questions, before concluding the interview with a general question about uncovered topics.

Warm-Up

During the warm-up, each respondent is asked to introduce themselves and briefly describe the context in which interoperability of microservices is required.

Ensuring interoperability on a high level

This part of the interview focuses on the high-level setup. Here, questions are asked about how and how often type information is exchanged between the developers of microservices. In addition, the communication between services should be described, especially with respect to the type of communication (such as REST or GraphQL). This interview phase finishes by evaluating whether the interviewees and their team(s) are satisfied with the solution taken in their project.

Code and Repository Setup

This part of the interview deals with implementation-specific issues. It is about finding out what languages, frameworks, and tools the interviewees and their team(s) use. This section also addresses how syntactic interoperability between microservices is ensured, e.g., through different tests or common type definitions. It also covers the impact of the deployed toolchain on the decision to organize and maintain interoperability in this way.

Outlook

An interview concludes with a question about challenges related to interoperability that have not yet been mentioned. The interviewee is also encouraged to present concepts regarding interoperability that are not covered by the survey.

2.1.2 Interview Sampling

The interviewees are sampled so they cover a heterogeneous set of projects. Therefore, each interview is classified according to the sampling model of Table 2.1 prior to its execution, and it is only performed if it differs enough from already conducted interviews. A heterogeneous set of partners is also called a diversity sample and is desired since it helps reflect the variety of the topic (Jansen, 2010). We reached out to eight possible interview partners; six out of eight responded. All six are considered helpful for this research.

The most important requirement for the interview to be considered beneficial for this research is that microservices are used. However, since interoperability can be an issue with distributed systems in general (van Steen and Tanenbaum, 2017), Interview E[1] was added to provide insight into integrating programs that create a purely non-microservice-based product. The categories and characteristics are based on personal experience and informal literature research.

After six interviews, the research has reached theoretical saturation so further interviews are not expected to yield further significant insights. New interviews are expected to only provide minor new variations to the currently already very heterogeneous software setups. Given that, it is reasonable to stop conducting more interviews (Strauss and Corbin, 2008).

Sampling categories

Six sample categories, each covering two to three characteristics, are established in preparation for interview selection. The categories are created based on our previous experience in the field and literature review.

Microservices can be developed using multiple *programming languages*, even if they span a single project. However, it is also possible to limit the usage of languages to a single one, as it is the case in the JValue project presented in the demonstration (Chapter 5). Another typical example is the combination of two languages: a web-based frontend, usually written in JavaScript, with microservices that use a server-specific language. Introducing more languages increases complexity, but this complexity is not expected to increase significantly once a critical threshold of three languages is reached. The decision not to add more

[1] In the following, the six interviews are referenced as Interview A through Interview F.

Category	Characteristics	Interview					
		A	B	C	D	E	F
Number of Languages	1	X					
	2		X		X		X
	3+			X		X	
Number of Services	1-2	X					
	3-9		X			X	
	10+			X	X		X
Deployment Frequency	daily				X		X
	up to monthly		X	X			
	less	X				X	
Services Deployed Together	arbitrary		X				X
	all	X		X	X	X	
Number of Teams	1	X			X		X
	2-5		X	X			
	6+					X	
Repository Setup	monorepo	X			X		
	multirepo		X	X		X	X

Table 2.1: Interview Sampling: A cross marks that a category applies to a project from an interview.

languages is based on the assumption that companies prefer to leverage the skills of their developers and therefore limit the number of languages applied in the development process.

The *number of services* is used to determine the size of the project. We decided to include very small projects with up to two microservices to assess whether architectures that can hardly be called microservices-based face similar challenges as larger projects do. The number of services that make up a project can easily exceed 100 (Newman, 2015). However, we do not expect to find interview partners developing such a large number of microservices for this master thesis. Therefore, a limit of ten was chosen as a reasonable limit to distinguish between medium and large projects.

Deployment frequency is of interest for ensuring interoperability since it determines how often API definitions need to be aligned. *Daily* is considered an ex-

treme version where great attention must be paid to ensuring compatibility. This must be done even if the deployment frequency is regular but not that often. *Up to monthly* retains the freedom to compensate for slightly irregular cycles such as biweekly to triweekly. Anything beyond this is considered irregular and it is assumed that API customization can be done manually rather than automated.

The *services deployed together* is a controversial subject. There is a distinction between *arbitrary*, where services can be deployed independently and *all*, which means that all microservices are deployed together. On the one hand, the latter feature contradicts the basic characteristics of microservices (Wolff, 2016), but on the other hand, we have experienced that it can ease development, especially when the project is still in its early stages.

The *number of teams* is justified similarly to the number of services. One team makes development easier through short communication paths between contributors. However, microservices can be used to create large applications that require multiple teams. Up to five teams is considered medium, anything above that is considered large in the context of this master thesis.

Today it is standard to use *code repos* (repositories) to keep track of code. In our experience as developers, monorepos are commonly used for application and microservices development. In this type of repository, all code is stored in a single repository, whereas in multirepos code is spread across multiple repositories (Morris, 2020). Monorepos can be a good choice for microservices development (Savkin, 2019), even in large enterprises like Google (Potvin and Levenberg, 2016). Therefore, it is interesting to see if the *repository setup* has an impact on microservices development and if there are software process patterns specific to these setups.

2.1.3 Transcription

All interviews conducted for this problem identification are transcribed and sent to the interviewees to ensure that everything is recorded correctly and that no sensitive information is inadvertently published. This undertaking increases the credibility of the data (Widodo, 2014).

In addition to factual information, other sounds are recorded in the interview. Oliver et al., 2005 categorizes them in "Constraints and Opportunities with Interview Transcription: Towards Reflection in Qualitative Research" as involuntary vocalizations (e.g., coughing, sniffing, or laughing), response tokens (e.g., *Uh huh*, *Hm*), and nonverbal utterances (e.g., head nodding, thought checking). Although these utterances may contain important information (Gardner, 2001), they are mostly considered noise in the context of an interview. The dialogues conducted for this work focus on the neutral facts stated by the partners. Additional vocalizations are added to the transcript only when they provide additional information

such as nodding or *Uh huh* as a form of confirmation.

Slight corrections of grammar, diction, and dialect are made while still ensuring that the factual statement remains unchanged and that the change improves comprehension. Unlike stated in Oliver et al., 2005, they offer no additional insight for this research.

The latter argument also applies to verbal corrections of errors by the interviewer or the interviewee. For example, a correction of a statement in the form of *"this is used in the backend er I mean frontend"* does not provide information about the interoperability of microservice-based architectures. Therefore, these error corrections are suppressed and only the correct sentence is transcribed (in this case: *"this is used in the frontend"*).

2.2 Results

The results obtained from the interviews are prepared from transcriptions of every interview (Appendix B). These textual representations are used to analyze the interoperability setups in the microservice-based architectures.

This section presents the results of the interviews by first summarizing each interview individually. Subsection 2.2.2 assesses the interviews in their comprehensiveness discussing why no further interviews had to be conducted.

2.2.1 Interview Summary

As noted before, the six interviews are referenced as Interview A through F. The letters are assigned according to the order in which the interviews are recorded.

Interview A

The company presented in Interview A works with one team and on average two microservices per project. Their older projects were developed in multiple programming languages, having data types exchanged entirely manually. They tried using code generators to transform from one language to another, but that did not work well enough. In the current setup, the interfaces are shared using a shared library in a monorepo (Figure 2.1). This is possible because they only use one programming language for all of their microservices and their frontend. When data types change, the compiler of the frontend or a service are complaining if the types are not compatible. All services communicate using JSON-based REST. They avoid the need for versioning by building and deploying all services jointly, even if individual services have not changed. This allows them to keep deployment simple, especially since they do not have fixed release cycles. This approach works

well for them. However, the team is looking for better solutions for testing that require less effort.

Figure 2.1: Interview A: Services access type information from a shared library.

Interview B

Interview B presents a project with about eight microservices written in two different programming languages in a multirepo setup. The services communicate via JSON-based REST, RabbitMQ, Webhooks, and WebSockets. For each service, there is an OpenAPI specification file that defines the REST API. Their custom-made parser validates each incoming request against this file and then invokes the respective method.

The OpenAPI specification is also used to publish their API and exchange data types. However, the generators do not function entirely satisfactorily so manual changes are required on a regular basis. This includes bug fixes in the generated code and communication that is beyond REST. These additions are published via an Software Development Kit (SDK) that can be used in other services or frontends (Figure 2.2).

Deployment is happening manually and any subset of services can be selected for updating. This is usually done on a two-week cycle. However, it is possible to deploy an arbitrary subset of services at any time.

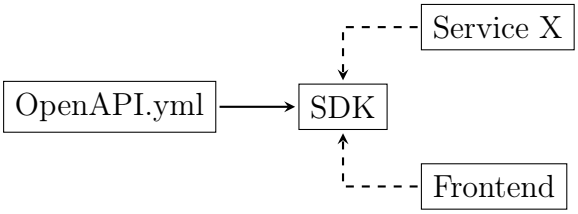


Figure 2.2: Interview B: The `OpenAPI.yml` is used to generate code that is published as an SDK. This SDK is used by other services including the frontend.

Interview C

In Interview C, the partners present how they develop their double-digit microservices. Their multirepo setup works well for their four internal teams. The services communicate mainly via JSON-based REST, but also via GraphQL. However, external services require proprietary data formats that they cannot influence.

Exchanging data types is primarily enacted manually and involves copying and pasting definitions from one repository to another when possible. The same is true for the exchange of REST endpoint definitions. When API specifications such as OpenAPI are provided by one of their customers, the types are manually converted to one of the four programming languages they use. Generators and code sharing are not used as they have had negative experiences with this and the process involved.

New features are usually initialized by stakeholders. However, modifications of the respective API itself are designed and implemented by developers. In cases when OpenAPI specifications are created, they do so by first implementing code that outputs the specification file.

They do not have a rigid development cycle, but the plan is to work on a two- to four-week cycle. If breaking changes are introduced, they are usually addressed in each affected service before a joint deployment is made. Versioning is used only in some cases. Interoperability is ensured through acceptance testing at various stages, including development, quality assurance, and production.

Interview D

In this interview, two projects are presented. The focus is on one of them, as only this one is developed by applying a microservice-based architecture. It spans about 20 services developed by a single team. The services communicate through the frontends mainly through JSON-based REST. The data exchanged between the services is typically encoded using Google Protobuf. Kafka is utilized as a message broker.

The Protobuf data types are first defined in a Protobuf specification and used as input for generating classes of a particular programming language (Figure 2.3a). The API is defined using OpenAPI specifications generated from annotations in Scala code (Figure 2.3b). The annotation library is developed internally. Frontend developers read the specification to manually create TypeScript classes. One problem with this setup is that it is difficult to detect changes before they are merged. These changes must be properly tracked in a ticket system. Between microservices, type definitions are shared through a monorepo.

The microservices do not need to write the connection to other services themselves. Each service publishes an Hypertext Transfer Protocol (HTTP) client as an SDK that ensures that a client invokes the proper endpoint with the correct types. These SDKs are also used to test the service itself.

Deployment is usually done by recreating all services together, but it is also possible to deploy arbitrary services. This is done daily for a development version and once a month for releases.

(a) Language-agnostic Protobuf definitions are used to generate code.

(b) Language-agnostic OpenAPI-specification is created from Scala annotations.

Figure 2.3: In Interview D, code is generated from specification (a) and specification from code (b).

Interview E

The partners of Interview E declare their software as a monolith. Nevertheless, it consists of three independently developed parts that are deployed on different computers and communicate over a network. In addition, the services are also connected to an external infrastructure. Therefore, this project can still provide valuable insights into the interoperability of distributed systems. More than 15 teams are involved in the development.

The parts communicate via a proprietary XML-based protocol. Data types are mainly exchanged manually unless the same programming language is used. Their product is written in three different programming languages.

There are build pipelines to get quick internal feedback. However, the company's product is subject to regulatory requirements that prevent automatic and more regular deployment. This also allows them to forgo versioning and release the entire product, including all three parts, in one piece.

Interview F

In the last interview, a project with about 20 services is presented. These services are developed in a multirepo approach that works better for their developers than a monorepo setup.

The company's single team develops the application in two languages. The services publish OData metadata in XML for the exchange of type definitions which is mainly used by the frontends. There is no automatic solution for JSON-REST-based communication used for communication between services. Information about these endpoints is partially defined in a wiki, but also in chats. In addition to REST, the services also utilize messaging to exchange data. This setup works well because most bugs are discovered by testers who manually validate the product. Automated tests are not enforced and therefore do not cover many parts of the services.

The company has a separate runtime environment for testers. Production versions are released every two to four weeks. The testers and developers track breaking changes that are introduced and if there are some (which they try to avoid), the affected services are deployed jointly. Versioning is not applied often. Currently, tracking these changes is a manual challenge where automated solutions with a communication matrix would be helpful. The services are usually deployed independently.

2.2.2 Aggregated Summary

The interviews show that there are many challenges to ensuring syntactic interoperability in microservice-based architectures. These problems are addressed in several different ways resulting in unique solutions for every project. Also, industry shows that microservices are not always developed according to the theoretical understanding which, amongst other things, includes that microservices are allowed to be developed in a technology stack of choice. However, half of the interviews state that they have shared code that can be used in other services (Interviews A, B, D) which binds them to one programming language.

A further common issue with technology-agnostic solutions is that they need to cover a wide range of different technologies. For example, Interview B mentions that they use a language-agnostic specification with OpenAPI which does not support the definition of other communication protocols or principles like WebSockets or Messaging with RabbitMQ. This causes the effect that they must either introduce a new specification (AsyncAPI is mentioned here) or, which is currently done, go without a formal and agnostic specification of their APIs for these communication types.

Even if a technology-agnostic specification is found, the developers have to rely on support for their programming language or framework. This is typically the case for popular languages, however, there are cases where none exist for their framework (Interview B) or where proprietary formats are used (Interviews C, E), which minimizes the probability that tooling support already exists. Even if tooling exists it may not be good enough for efficient usage and requires manual adjustments of the specification or produced code (Interviews A, B, C). This is one of the reasons why all projects at least partially dispense with such solutions and rely on defining the interfaces manually.

Another characteristic of microservices is that they should be independently deployable, however, tracking when breaking changes are introduced in a service is not trivial. Versioning and therefore keeping a compatible API alive is a common practice (Interviews B, C, F). However, almost all projects try to minimize that additional effort by deploying changed versions together which helps simplify their development setup (Interviews A, C, D, E, F). If non-compatible versions

are deployed nevertheless, errors typically are thrown at runtime.

While errors at runtime might occur regularly, they typically are detected when the application is tested manually in a staging environment (Interviews C, F). Automated testing is often only sparely used and typically covers rather the most important features than ensuring the syntactic interoperability of microservices.

In conclusion, there are many challenges arising from industry regarding syntactic interoperability. Solutions are designed specifically for the software process which also includes violating properties of the microservice theory. However, these violations are justified in their context and improve the developer experience and speed up progress.

3 Objective Definition

The phase objective definition of the design science research methodology should define "what is possible and feasible" (Peffers et al., 2007). This chapter, therefore, declares the main goal of this thesis.

Subsection 2.2.2 concluded that there are many different solutions for similar problems about maintaining interoperability of microservices that all work well in the context they are introduced. For example, most projects have REST-based communication but each project brings its own well-working solution to exchange endpoint information:

- Interview A: Type definitions are put into a shared library.
- Interview B: OpenAPI specifications are defined by hand and used to generate SDKs for clients.
- Interview C: Data types are copied or defined manually.
- Interview D: OpenAPI specifications are created by code annotations and used to manually create code.
- Interview E: *Does not use REST.*
- Interview F: Information about endpoints is stored in a wiki or in chats and added to the client manually.

An initial goal of this thesis has been to create software process suggestions or templates that give recommendations on how to achieve interoperability between microservices while including various development setups. However, the interviews revealed that there are too many different ways to make services interoperable and to design workflows that will ensure successful interoperability for their projects. Therefore, providing concrete process templates is not feasible within the scope of this work.

While concrete process templates are not yet achievable, it is possible to identify the differences and similarities between projects regarding the interoperability of microservices. Categorization systems are one way to accomplish this, leaving

category membership flexible enough for individuals to contribute their knowledge and additional context (Jacob, 2004). Jacob also denotes that boundaries between categories can be loose and summarizes that some entities may be "more representative of a category than others" (Jacob, 2004). Therefore, categorization systems can balance project-specific circumstances and background information mentioned by the interviewees. Categorization systems are less strict than classifications such as taxonomies (Jacob, 2004) and "reduce the infinite differences among stimuli to behaviorally and cognitively usable proportions" (Rosch et al., 1976), which enables their usage by developers and those who can make decisions about the software process. Newman says that "just like many things concerning microservices, it's all about finding the right balance" (Newman, 2015). With that in mind, a categorization system that leaves some space for interpretation and freedom for a user can improve its applicability in real-world projects. This helps users like software architects to balance out to which category their project belongs, even if not all details of a category fit.

The categorization system introduced in this thesis should include dimensions of how interoperability can be described (Section 3.2). These dimensions should be referred to as categories. Further labels do not have to be assigned, i.e. the project-specific combination of characteristics of the dimensions can remain unnamed. The limitations of the developed categorization system are discussed in Section 3.3. Before that, related work is presented in Section 3.1.

3.1 Related Work

A general theme of this work is the analysis of interoperability. Abukwaik and Rombach, 2017 have identified the need to develop approaches for conducting interoperability analysis. The categorization system developed in our work can be used as such and provides support for conceptual and technical issues as demanded by Abukwaik and Rombach, 2017.

There are few activities to develop an analysis tool as called for by Abukwaik and Rombach, 2017. In this sense, the most prominent related work is Valle et al., 2019. They have created a typology for solutions to achieve interoperability at the four levels presented in Section 1.2. Their typology is not limited to microservices. Rather, microservices are considered one possible solution for achieving interoperability between systems. In contrast, our work focuses on problems and solutions related to interoperability when microservices are already chosen as architecture. Moreover, solutions for achieving interoperability are presented in Valle et al., 2019. One of them is the architectural pattern of an adapter that helps transform data from one representation to another. In the exemplary case of an adapter, our work goes a step further by defining ways to find out and ensure what the given and desired data representation needed for such an

adapter actually looks like. Therefore, this work provides solutions to achieve the interoperability methods already known today.

No further scientific publications introducing categorization systems or classifications that address interoperability and include concrete realization strategies have been identified to date. Other papers examine various patterns for establishing interoperability. However, these approaches do not provide information about the issues raised in the interviews.

3.2 Basic Dimensions

This section introduces basic dimensions that are revealed through a first analysis of the interviews. These dimensions are factors that should be considered when creating the categorization system for interoperability in microservices and serve as a starting point for it. It is therefore not required that they are directly represented in the solution itself. However, the final artifact must not block any dimension or characteristics presented in this chapter.

The following sections contain references to the interviews conducted. Unless otherwise noted, the references are only interesting and representative samples of the individual projects. When it is stated that Interview X uses REST, it does not mean that Interview Y does not use it. However, Interview X is considered a typical or notable case.

3.2.1 Driver of Change

Changes to an API are typically initialized by two different groups: either by a client or stakeholder (Interview C) or by the API provider itself (Interviews B, C). However, there are also cases where a change of an interface can be initialized by both the client and the service developers by editing a common codebase (Interview A). The driver of change in this chapter refers to a person or an organizational unit.

3.2.2 Amount of Manual Work

Manual work is prone to error. However, all interviews mention situations where parts of their API are typed manually for client projects. Therefore, the amount of manual work seems to be an important consideration when deciding on a development workflow. This includes not only coding a microservice but also testing it and its integration with other services. Multiple interviews indicated that manual testing is an important aspect of ensuring interoperability between microservices (Interviews C, F).

Another aspect is deployment, whereby it must be ensured that deployed versions are compatible. A common technique for this task is versioning, but most interviewees state that they try to avoid this by only adding new features, not removing or altering existing ones. However, marking a microservice as ready for deployment is usually a manual task.

Every project brings changes at some point. These changes must be communicated to the developers of other services. Monorepo setups that share API definitions are one way to solve this problem. Here, a compiler complains when changes are incompatible, and updated API definitions are immediately available. In multirepos, this task is a bigger challenge and typically less automated.

3.2.3 Technology Support

The diversity of technologies has a significant impact on the software workflow. When the same language is used for multiple services, code sharing has often been reported, usually by exchanging data types or SDKs (Interviews A, B, D).

Microservices, however, should have the freedom to choose a language independent of other services (Lewis and Fowler, 2014). Technology-agnostic solutions are also used, especially for publishing APIs to external customers (B). Therefore, both technology-agnostic solutions and technology-specific solutions have their benefits and disadvantages.

3.2.4 Communication Architecture

There are many different types of communication between microservices (e.g., REST, OData, GraphQL, messaging). The interviews show that it is common to apply multiple types of them within one microservice-based software (Interviews B, C, D, E, F). However, they also reveal that there is no tool that supports all types of communication (Interview B). Therefore, the type of communication is considered to play an important role in integrating a well-functioning software process to ensure interoperability.

The representation of the data exchanged (e.g., XML, JSON, or proprietary formats) is a closely related issue. It seems to have less impact than the type of communication but is important nonetheless.

3.2.5 Level of Type Safety

When API information is exchanged in such a way that multiple services can access it, the level of type safety has not yet been determined. Two main options have emerged: pure data type exchange (Interviews A, D) vs. data type and endpoint exchange (Interviews B, D). Data type refers to how data objects

that are transferred between services are structured, which is usually defined by interfaces or classes in a programming language. For example, it contains information about an attribute of a data object, including the name and data type of that attribute. Endpoint exchange means that API endpoints are fully described. In REST-based communication, an endpoint includes, among other things, the HTTP method, the Uniform Resource Locator (URL), and the incoming and outgoing data types. This implies that endpoint exchange also includes data type exchange.

Pure data type only exchange can be a relatively simple solution if the same programming language is taken (Interview A). However, errors can still occur at runtime if the endpoint is not properly defined within the client. Endpoints can be exchanged by using a generic API specification such as OpenAPI (Interview B) or by publishing SDKs (Interview B, D), even if this is only done internally.

3.2.6 Manual Extensibility

The interviews show that manual adjustments and extensions are the rule, not the exception. For example, OpenAPI is a commonly used method for describing APIs. However, there may be languages for which accessible (i.e., often open-source) generators are not available, especially for proprietary data representations. In addition, there may be concepts that are not supported by OpenAPI like WebSockets (Interview B). Finally, one may use certain language features that are not supported by all generators or that cannot be properly represented by the API description language. Therefore, one should consider how extensible a tool is that supports interoperability.

3.3 Limits of the Categorization System

Categorization systems should be limited in scope so that developers can easily understand them without additional tools (Rosch et al., 1976). It is therefore important to set firm boundaries so that the system can be used as an appropriate and effective tool.

The first limitation is that the categorization system should not include the exchange of semantic definitions. Microservices are a way to form a single application (Lewis and Fowler, 2014) and are therefore used in a semantically enclosed environment. The interviews support this statement, a formal exchange of semantics is not observable in the projects.

Communication to or from outside a project can also be neglected by the categorization. One is usually dependent on a vendor's specification and does not have much impact on it. On the other hand, it is unlikely that an API provider

will offer solutions that are limited in terms of supported technology. The categorization system should therefore focus on the options that are used to ensure interoperability for internally developed services where there is freedom of choice.

The categorization system to be developed should provide up-to-date values over a reasonable period of time. As more and more languages, frameworks, and tools are released each year, a categorization system would quickly become obsolete. However, the availability of tools can drastically change the feasibility of one or another interoperability solution. Tools should therefore only be mentioned as examples, but must not constitute a category of their own.

In summary, this chapter defines the main objectives of this thesis. The main research question is therefore revisited and framed as follows: How can problems and solutions related to internal syntactic interoperability in microservice-based architectures be categorized at a conceptual level?

4 Solution Design

This chapter presents the main artifact of this design science research: a multidimensional categorization system for identifying ways to achieve interoperability in microservices-based architectures. The categorization system is intended to help by raising awareness of how interoperability can be accomplished. Firstly, section Section 4.1 discusses the methodology applied and Section 4.2 presents the results. The categorization itself is explained in Subsection 4.2.1. An explanatory workflow for using the categorization system is presented in Subsection 4.2.2.

The categorization system developed in this thesis is a model for overcoming interoperability problems in microservice-based architectures. There are various definitions of interoperability and classifications regarding interoperability processes from several perspectives (Panetto, 2007). The categorization system designed in this thesis represents a point of view to achieve interoperability that is consistent with the current state of microservices development. Classifications such as this categorization system are critical in creating advanced concepts (Bailey, 1994). This work thus forms the basis for creating more advanced concepts, such as specific processes or implementation proposals.

4.1 Methodology

Categorization systems have similarities to classifications such as taxonomies but are generally less rigorous (Jacob, 2004). In particular, the structural similarity opens up the possibility of using taxonomy construction methods to construct a categorization system for interoperability in this research. A popular method for developing taxonomies in the context of information systems is presented in Nickerson et al., 2013. This publication describes an efficient approach for constructing taxonomies that is applied explicitly in design science research.

The literature review has provided a foundation of expertise on microservices and their interoperability issues and solutions; the interviews have provided sufficient data for further investigation. Given these conditions, Nickerson et al., 2013 proposes to use their empirical-to-conceptual approach. The process template

provided by Nickerson et al., 2013 is altered to meet the requirements and the data available in this thesis. One of these modifications is that this work does not distinguish between an empirical-to-conceptual approach and a conceptual-to-empirical approach. In the latter, the categories are created before the data is analyzed, which is not done in this thesis. Another simplification is that it is not necessary to select a subset of samples to be analyzed for new dimensions. With only having to cover six interviews, all can be examined at once to look for particular characteristics that either recur in multiple projects or show contrasting approaches. The simplified and adjusted process (Figure 4.1) is followed in this solution design.

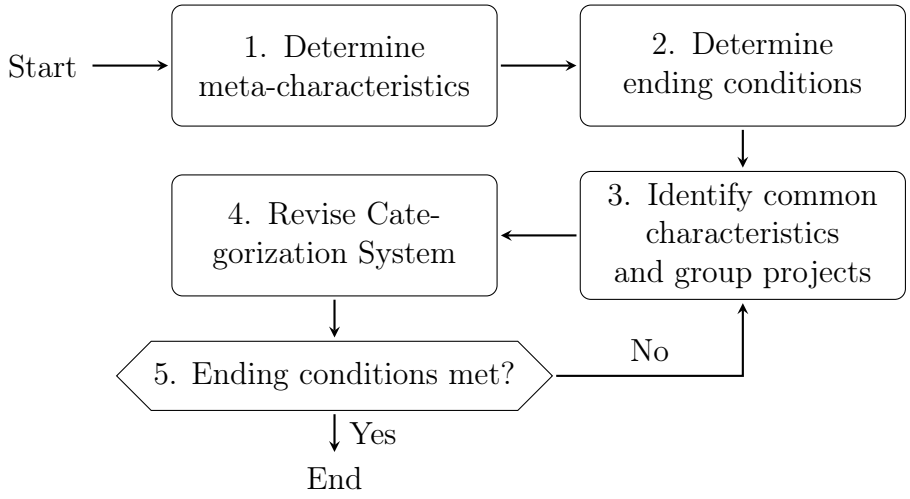


Figure 4.1: The process applied to create the categorization system is a simplified form of the one from Nickerson et al., 2013.

The first step (Step 1) of the process is to define the meta-characteristics that define the main domain from which the categories should originate (Nickerson et al., 2013). The goal of this thesis is to create an up-to-date categorization system that can be used by those who can design a software process around interoperability. Therefore, the potential user group consists of domain experts having an understanding of software architecture, so the main categories and characteristics may assume considerable knowledge in the field of software engineering, microservices, and interoperability. However, the categorization system should operate at a conceptual level and therefore not include characteristics that deal with implementation details.

The next and final preparatory step (Step 2) is to define the ending conditions that define when the artifact is good enough so that the iterative process can be stopped (Nickerson et al., 2013). These ending conditions, both objective and subjective, follow directly from the conditions in Nickerson et al., 2013 and are tailored to the use case of this research:

- Objective
  - All interviews are analyzed.
  - A typical setup of each interview can be categorized. Not every variation needs to be categorized; a representative sample is sufficient.
  - No categories or characteristics are modified during the last iteration.
  - All categories or characteristics are unique.
  - Each characteristic has at least one representative within the interviews.
- Subjective
  - The categorization system contains a reasonable amount of dimensions.
  - The categories are distinctive enough.

After the setup is complete, the iterative design can begin (Step 3). First, the interviews and the projects presented are analyzed to find patterns that repeat between interviews. If a common pattern is found, a category of characteristics is created and all projects are grouped into that category. The new findings are then added to the categorization system by splitting, merging, or redefining the categories for a new version (Step 4).

After revising the categorization system, the ending conditions must be sifted to evaluate their fulfillment. If all end conditions are satisfied, the categorization system is considered complete. Otherwise, another iteration starts.

4.2 Results

The categorization system presented in this thesis is constructed as described in the previous section. Dimensions are split, merged, and refined until the categorization system contains an appropriate number of dimensions and characteristics. In addition, a representative approach from each interview must be able to be described using this classification and each characteristic must be applied at least once (Table A.1). All ending conditions defined in the previous section are fulfilled.

4.2.1 Categorization System

Six different dimensions (further called categories) D_i have emerged during the development of the categorization system, each spanning at least two charac-

teristics C_{ij}. Each enactment of interoperability in a certain project refers to characteristics stemming from the six dimensions.

The characteristics of each dimension are designed to be mutually exclusive. A solution for overcoming interoperability problems can therefore be described by six characteristics, one from each dimension. These categories are influenced by the dimensions presented in Chapter 3.

- D_1: Driver of Change
  - $C_{1.1}$: Code
  - $C_{1.2}$: Specification
  - $C_{1.3}$: Both
- D_2: Level of Static Type Safety
  - $C_{2.1}$: None
  - $C_{2.2}$: Types Only
  - $C_{2.3}$: Everything
- D_3: Completeness
  - $C_{3.1}$: Partial
  - $C_{3.2}$: Full
- D_4: Adoption of Different Versions
  - $C_{4.1}$: Instant
  - $C_{4.2}$: Delayed
- D_5: Deployment of Different Versions
  - $C_{5.1}$: Manual
  - $C_{5.2}$: Automated
- D_6: Technology Support
  - $C_{6.1}$: Internally Used Technologies
  - $C_{6.2}$: Technology-agnostic

D_1: Driver of Change

Driver of Change refers to the origin of a formal definition of a microservice API, especially in terms of change. Literature typically distinguishes between consumer-driven and API-first. The interviews confirm this theory, but a slightly

different distinction has emerged as more important concerning to interoperability in service development: *specification-* vs. *code*-first. These names are chosen since we think that they reflect the actions best. Other commonly used names are design-first instead of specification-first (Lane, 2021) and implementation-first instead of code-first (Postman, 2023).

Specification as the driver of change requires a formal definition of the API. This can be achieved with Pact[1], OpenAPI specification files[2] or similar. Informal requests like *add an endpoint to get user details* are explicitly excluded from this distinction, as this still requires a code change for the API as depicted in Figure 4.2. Such requests may come from a product owner, external consumers, or developers of other microservices.

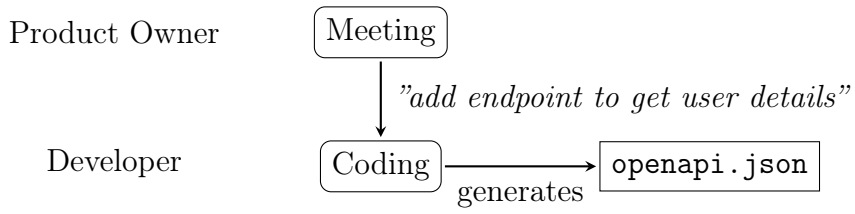


Figure 4.2: Here the category *code-first* applies although the product owner initiates the modification. This is because the formal definition (`openapi.json`) is generated after the code change.

In the *code-first* approach, the microservice (i.e., the code) is updated first and is then used to create formal specifications. Possible methods to accomplish this include code annotations such as the OpenAPI integration in NestJS[3].

The interviews show that usually one or the other approach is chosen. However, solutions such as shared libraries can be considered "neutral ground" because both sides - the consumer and the API developer - can modify them. The same applies to semi-formal solutions like wikis. The characteristic *both* also fits when clients and the server do not share a specification but each side has one which is checked for compatibility or if both share a specification that can be edited by clients and a server.

D_2: Level of Static Type Safety

Three different levels of type safety are inferred from the interviews. There is a level of type safety *none*, in which everything is created manually, without using generators or shared code. The next level is *types only* which includes exchanging interfaces only, for example through shared classes. The highest level

[1]https://pact.io/
[2]https://www.openapis.org/
[3]https://docs.nestjs.com/openapi/introduction

is *everything*, which includes the exchange of interfaces from the previous level and additional communication-specific features such as HTTP methods or the returned HTTP status codes.

None requires the least software setup of all. It can function without a formal specification of an API and with semi-formal, non-machine-readable specifications like definitions in a wiki. Manual work is required in this case to create classes and service implementations for each language taken. These manual tasks are prone to human error and require additional communication to keep track of changes made to an API. One could put certain classes or services in a shared library to make them accessible to multiple clients. However, when these classes and services are not published by the service itself, it is still a manual task that does not provide automated type safety between client and server.

Types only provides additional type safety for clients and servers by agreeing on interfaces used for communication. This can function with minimal effort if the same language is used for a server and all of its clients by accessing the same class definitions, e.g., through a shared library. Based on the interviews, it is clear that this can be a viable solution, provided that it is executed properly and the necessary setup is given. These requirements include using the same language and making the code accessible to multiple projects. However, a fallback solution might be needed if the services involved are implemented in different languages. This occurs especially when communicating with web-based frontends.

Another way to create *types only* type safety is to use generators. This can be done by converting languages directly or by defining them in a language-agnostic way, such as with JSON Schema[4]. It is also possible to derive types from more advanced specifications such as OpenAPI (OpenAPI-Generator Contributors, 2023). So even if documentation exists that includes more than just types, one might be able to choose a different level of type safety. This may allow more freedom in terms of communication customization or may facilitate migrating from a manual solution to the *types only* level, and perhaps even lay the groundwork for *everything* to be type-safe. This setup already requires some configuration and the effort to define *everything* may not be that much higher. Nevertheless, this can be a good alternative when generators for a more advanced solution do not exist or are not satisfactory.

The most automated solution is to define *everything* with formal specification files and use generators to create conforming code. The interviews and case study in Chapter 5 show that this provides a very satisfying developer experience. However, it requires more effort to fully define an API than just to specify the data types. This type of technology-independent definition of APIs can simplify the process of adding new languages and new frameworks to a microservices-based

[4]https://json-schema.org/

product. However, these benefits only arise when high-quality generators and API specifications exist. The interviewees expressed that adjusting generators or the generated result should be expected. In addition, generators may not exist for certain languages or frameworks used. Both of these issues result in the need for manual adjustments or new implementations of generators.

Automated solutions should be created that are technology-agnostic. Interviewees indicate that some language-specific features are not supported by a generator of choice. Also, data types should be defined to be compliant with each language used. For example, in ECMAScript, there are no data types `float` or `double`, but only `number` (ECMA International, 2022). This data type is also commonly used to express integers as well (MDN Contributors, 2023). It might be beneficial to consider an additional step to define the actual number type according to the API description language definition.

This section focuses on generators to solve the task of keeping the API specification and the client or server implementation in sync. However, consistency can also be ensured by validating the implementation, e.g., with PactFlow[5] or automated testing. Emphasis is placed on generators because, according to the interviews, they are used more frequently than validators.

D_3: Completeness

The section *Completeness* primarily focuses on the scope an interoperability solution has: either *partial* or *full*.

Having *full* completeness means that all communication between services is realized with the same level of automated interoperability. Several interviewees pointed out that this is quite a challenge, as existing solutions in the form of generators typically support only a fraction of the existing ways of communicating. One interviewee stated explicitly that further API specifications and generators are needed in addition to OpenAPI to define the communications. OpenAPI only supports HTTP APIs, with WebSockets declared "out of scope" (D. Miller, 2022). Therefore, it is recommended to find out what communication paradigms a project employs before deciding on a particular tool and before deciding on the characteristics of the categories from this chapter. The interviews show that one should expect that a software project has several different interoperability solutions, each covering one category selection. For example, one can choose the *everything* characteristic of the *Level of Static Type Safety* category for the *partial* solution of HTTP APIs and *none* generators for WebSockets. Reasons given by interviewees included insufficient tooling support or the fact that the amount of work required to set up a more exhaustive interoperability solution does not justify the few uses of the communication paradigms.

[5] https://pactflow.io/

It makes sense to add the communication paradigms to the chosen characteristic of this category. Adding that helps documenting and differentiating category selections.

D_4: Adoption of Different Versions

The interviews show two different ways of adopting changes between API versions. Versions can be integrated *instantly* in the same merge request or at least before the affected services are deployed. The more flexible approach is to reflect the changes *delayed* and make the services deployable independently from each other.

Breaking changes can be resolved before a service is deployed when using the *instant* approach without the need for versioning in the API. The interviews and case study in Chapter 5 show that this works particularly well in Monorepos. However, the additional coupling of services and the need to deploy them together contradicts the intent of microservices (Lewis and Fowler, 2014). Nonetheless, this way of developing microservices has proven to be viable, especially for small projects. The claim that services achieve syntactic interoperability can be verified with static analysis, as shown in Chapter 5.

A key property of microservices is that they can be deployed independently (Lewis and Fowler, 2014). The *delayed* approach addresses this point by keeping older APIs alive for a at least reasonable period of time when breaking changes are made. One way to achieve this in practice is to try not to change existing APIs, but only to add new functionality. Since this is not always possible, this problem is often solved by introducing versioning for an API. One can then include this information and notes about deprecation in the API description. OpenAPI, for example, supports versioning and deprecation warnings (OpenAPI Initiative, 2021). This might be the necessary way to go when providing an API to the public.

In addition to just versioning, one might consider adding a compatibility matrix to an API. This will record which services can work together. Depending on the project setup this can range from simple to very complex. The Pact Broker[6] is a tool that automatically checks compatibility and provides first-class support if Pact is used anyway.

D_5: Deployment of Different Versions

Two different approaches to providing services are mentioned in the interviews: *manual* through careful selection of the service to be deployed, and *automatic* where compatible versions are detected without human interaction. While the

[6] https://docs.pact.io/pact_broker

previous subsection described how to keep track of versions, this subsection provides options for deploying them so that incompatible services are prevented from communicating.

In the *manual* solution the person doing the setup must verify that all running (or at least all connected) services are compatible. A compatibility matrix, as described in the previous subsection, can be very helpful therefor. However, the *manual* solution still requires a human expert to check the services and mark them as publishable by hand.

Using an *automated* method requires more setup. Here, versions that can be deployed together are selected automatically. For example, one can use Pact Broker's `can-i-deploy`[7] feature in a Continuous Integration and Continuous Deployment (CI/CD) pipeline to ensure that services are deployed only if they're compatible.

Interviewees indicate that this approach can be very tedious. Depending on the tool chosen, compatibility checkers such as Azure's `openapi-diff`[8] can help develop a solution that matches the desired type of deployment.

This setup can be added to any runtime environment. When employed, a solution must be developed whether it is added to a development, test, production, or any other environment.

D_6: Technology Support

The last dimension *Technology Support* aims at defining whether a *technology-agnostic* solution should be chosen or not. The previous subsections present several implementations such as OpenAPI or JSON Schema that have a neutral API specification. However, solutions that are aligned with the technology used for microservices development can simplify and accelerate this process.

Solutions that focus on *internally used technologies* usually facilitate the development of interoperable microservices. For example, tRPC[9] is a tool that simplifies full-stack application development with TypeScript using a monorepo (tRPC authors, 2022). However, the tRPC authors also point out that this solution should not be chosen if other languages beyond TypeScript are employed or if the API is provided to external clients. This thus limits the freedom for selecting a language of choice per microservice and also leads to immense centralized management that should not exist in microservice-based architectures (Lewis and Fowler, 2014). While not intended by the microservice concept, the focus on *internally used technologies* is employed in several of the projects presented in the

[7] https://docs.pact.io/pact_broker/can_i_deploy
[8] https://github.com/Azure/openapi-diff
[9] https://trpc.io/

interviews. Here, a typical way to integrate services is to distribute code using shared libraries, often combined with a *types only Level of Static Type Safety*.

Integration of new languages or technologies requires other solutions when focusing on *internally used technologies*. An example of defining APIs in a *technology-independent* way is GraphQL[10]. There are also solutions that can be used to describe and serialize data from a service, such as Protocol Buffers[11]. However, for reasonable benefit, one must rely on the availability of generators for each programming language applied.

4.2.2 Integration into a Software Process Design

The interoperability categorization system is intended to provide insight into how syntactic interoperability can be achieved in microservice-based architectures. This can serve as a basis for deciding how comprehensive a project-specific approach to overcoming interoperability issues should be. This section presents a workflow that illustrates the suggested use of the categorization system.

Figure 4.3: Intended usage of the categorization system. The dotted boxes are fully project-specific and therefore not part of this thesis.

The first two steps of the presented software process design are closely related to the categorization system from this thesis. Here one should make a comparison between the current interoperability setup and the desired one. The following two steps are based on these results and are completely project specific. So they cannot be generally supported and guided as it is feasible for a categorization system and are therefore drawn dashed in Figure 4.3.

Existing projects can use the categorization system to find out what the current setup looks like. This helps determine the strengths and weaknesses of current implementations. In order to do this, one should go through each of the categories and make a selection that is the best fit. This step is omitted when designing an interoperability process for a new project.

[10] https://graphql.org/
[11] https://protobuf.dev/

The next step is to decide on the desired interoperability setup. As in the previous step, one should go through each of the categories and make a selection. While this can be done for the whole project, the choices can also be made for parts of the project separately. For example, if a significant portion of the services is written in a single language and framework, it may make sense to use a less technology-agnostic interoperability setup for those services than for others, in order to simplify the overall interoperability setup, as described in Subsection 4.2.1. An example comparison between a current and a desired setup can be found in Table 5.1.

If one knows how interoperability is to be realized in a particular project, it becomes easier to find tools that support exactly this endeavor. One can search for existing tools and check if they support all the desired interoperability characteristics. This can include both proprietary and non-public solutions. Note that this work does not include a list of available tools.

The software process should be designed after defining the interoperability categories and selecting the appropriate tools. Since neither tools nor software processes are suggested by this thesis, software architects can fully customize them to suit their team, its strengths, and its capabilities.

After development has been running for a while, one can restart the process by reassessing the current setup and responding to any changes that might alter the categorization.

5 Demonstration

In this chapter, an application of the categorization system is presented within a case study in cooperation with the JValue project. The purpose of a demonstration step within the design science approach - in this case, it takes the form of a case study - is to exhibit the problem-solving capabilities of the developed artifact (Peffers et al., 2007).

Our categorization system is applied as part of the JValue project[1]. The goal of JValue is to create Extract, Transform, Load (ETL) pipelines to retrieve and process data from various data sources. These can be executed in various runtime environments. In the following, the two runtimes *Simple* and *Kafka* play a role.

JValue operates upon a microservice-based architecture. For demonstration purposes, a simplified model of this architecture is taken that includes features that are planned but not yet implemented (e.g., the Kafka Runtime). These planned features help to understand the rationale behind some implementation details of the case study. All services including the frontend are currently written in TypeScript.

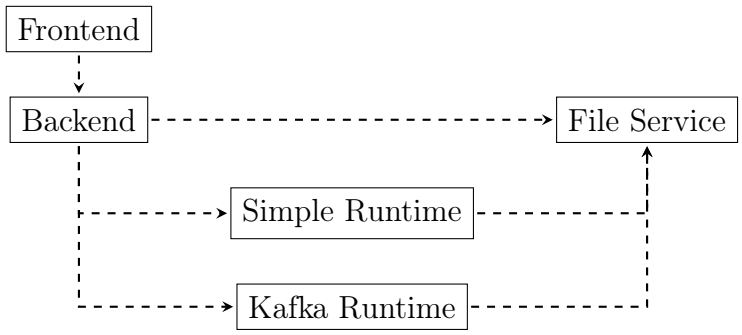


Figure 5.1: Simplified architecture of JValue.

[1] https://jvalue.org/

5.1 Applying the Categorization System

This section describes the process applied in the case study to analyze and improve the current interoperability setup. In general, the workflow proposed in Subsection 4.2.2 is followed.

According to Figure 4.3, the first step of the design process is to categorize the current interoperability setup. JValue operates a code-first approach to generate OpenAPI specifications from various services. These specifications are not exhaustive and are not further processed. The interfaces of the exchanged data are published in a single shared TypeScript library in JValue's monorepo. No other types of communication besides REST calls are used. Also, versioning is not required because all services are always created and deployed jointly. The setup is entirely dedicated to TypeScript and no further languages are currently used or supported with any kind of type exchange.

After analyzing the current configuration, one can use the categorization system to discuss and explicitly note down the state that matches the current directions of the project. An important goal of JValue is to develop a solution for API information exchange that not only works with TypeScript but is also technology-independent. This way, the JValue team has the freedom to use other programming languages besides TypeScript. The API definition should not only be more generic but also include machine-parsable information about the provided REST endpoints that help JValue improve interoperability. One way to accomplish this is to employ generators to create code from the formally defined API specifications. No other changes are required at this time, so the comparison between the current and desired interoperability processes is as summarized in Table 5.1.

Category	**Current Setup**	**Desired Setup**
Driver of Change	Code	
Level of Static Type Safety	Types Only	Types and Endpoints
Completeness	Full (REST)	
Versions	Instant	
Deployment	Manual	
Technology Support	Internal Tech.	Technology-agnostic

Table 5.1: Exemplary juxtaposition of the current and desired interoperability process from the JValue Project.

The next step of the design process proposed in Subsection 4.2.2 is to decide on the tooling to be employed. Only REST-based communication must be sup-

ported since this is the only type of communication that will be used. There should also be support for the frameworks currently in use, namely React[2] with RTK Query[3] for the frontend and NestJS[4] for the services. A significant number of API endpoints are already documented with OpenAPI and also a code-first integration in NestJS[5] is established already. OpenAPI decouples technologies by an intermediate specification format that includes also an API's HTTP methods, HTTP headers, and other information next to the data types that JValue needs to fully define their APIs. That solution also supports the other characterizations of the desired new software process and is therefore the tool of choice to support the interoperability setup.

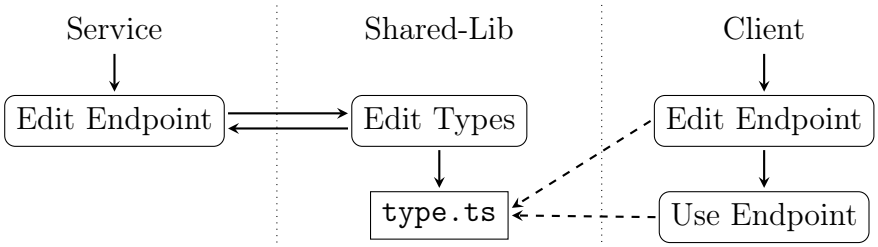
The final step of designing the new software process is to create a new development process. This new workflow is presented in the following section.

5.2 Implementation

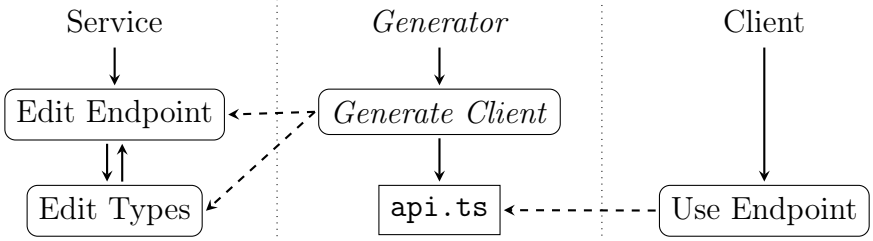
The software process is refactored to reduce manual error-prone work (Figure 5.2). By removing the shared library and replacing it with the client generation mechanism, a microservice developer can now fully stay in the service project and does not need to edit type definitions outside the service project (i.e., in the shared library) anymore. A generator running in the background parses the API endpoint and type information from a service and publishes a ready-to-use file to access the API. This file can be used by a client who does not have to manually create code to access the API.

The new development workflow contains the two different processes presented in Subsection 5.2.2 and Subsection 5.2.3. Each represents an effective and optimized way to reduce interoperability problems while considering the context in which each solution is needed. Nevertheless, both satisfy the previously defined interoperability characteristics. The first option is to create and publish an SDK that allows easy reuse in multiple TypeScript-based services (Subsection 5.2.2). The second way utilizes direct access to the specification (Subsection 5.2.3), which works especially well for the one frontend of JValue, which contains technologies not used anywhere else in the project. These two different processes are examples of how interoperability problems can be overcome and do not represent the variety of valid workflows. However, in our eyes, they represent the most typical and also distinctive use cases in the JValue project and are thus considered well-suited for a case study.

[2]https://react.dev/
[3]https://redux-toolkit.js.org/rtk-query/overview
[4]https://nestjs.com/
[5]https://docs.nestjs.com/openapi/introduction



(a) Old development flow: A developer of a service must edit the shared library and a client developer must create the respective endpoint definition manually.



(b) New development flow: A service developer can focus on the definitions inside the service project. The generator takes work off manual work by creating a client API in the background without manual interactions.

Figure 5.2: In the new workflow, service and client developers stay in the context of the project. Complexity is shifted to the generator that requires no manual interactions during development.

The new workflow is implemented only for the most typical but still distinct examples of JValue, namely the communication between the frontend and the main backend, and the communication from multiple services to the file service. These two examples are considered sufficient for this case study and cover a broad spectrum of applications inside and outside of this project.

5.2.1 Common Setup

Although the two realizations to enable interoperability supported by OpenAPI are quite different, they also have some commonalities. This subsection presents the most important shared setups. There are a few more conventions that developers should follow, however, they mainly provide benefits by standardizing the realization of the rather conceptual implementation details relevant to this thesis. Therefore, they are not listed in the following.

A big common feature is the creation of the OpenAPI specification. As mentioned earlier, a code-first approach is used for this. Decorators are added to the code which provide information about the API endpoints, including but not limited to the HTTP method (`@Get()`), the data type required (`GetRunDto`) and returned (`Run`), the possible status codes (`@ApiOkResponse()` for `200 OK` and

`@ApiNoContentResponse()` for `204 No Content`), and additional textual documentation. A typical example of what a pipeline run retrieval endpoint definition might look like is shown in Listing 1. An excerpt from a corresponding JSON-based specification that is generated from these annotations can be found in Listing 2.

```
// Endpoint definition in a NestJS controller
@Get('instances/:instanceId/runs/:runId')
@ApiOkResponse({ type: Run })
@ApiNoContentResponse()
getRun(@Param() dto: GetRunDto) {
  // ...
}
// Definition of the parameters
export class GetRunDto {
  @IsUUID()
  instanceId!: string;

  @IsUUID()
  runId!: string;
}
```

Listing 1: Annotations are used to document endpoint information in the JValue NestJS services.

```
"/instances/{instanceId}/runs/{runId}": {
  "get": {
    "operationId": "Runs_getRun",
    "parameters": [
      {
        "name": "instanceId",
        "required": true,
        "in": "path",
        "schema": {
          "type": "string"
        }
      },
      // ...
```

Listing 2: The generated `openapi.spec.json` contains the information defined in the code.

Another common setup is the way in which the validity of the specifications is ensured. The specifications are tracked by the version control system used (git[6]). Not tracking them would lead to a significant increase in the complexity of building microservices and, in particular, the corresponding Docker[7] images.

[6]https://git-scm.com/
[7]https://www.docker.com/

Including them in git can still lead to two major problems. First, a developer can change the source code without regenerating the specification which would result in an outdated state. A second problem can occur with git merges. The specification may contain errors if a merge is not done properly. After each push to the version control system, automatic checks are performed to ensure that the specifications are correct and up-to-date. This is done as part of the continuous integration pipeline used by JValue and works as follows: First, all specifications are regenerated. The version control system is also available in the pipeline and is used to check if files have been changed. If this is the case, the specifications created by the developer are not correct, resulting in a failed pipeline (Figure 5.3).

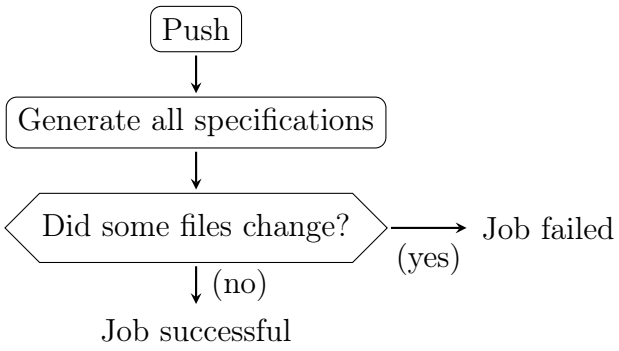


Figure 5.3: An automated pipeline ensures that the specifications in the version control system are correct by rejecting invalid configurations.

The last important common configuration is the point at which the code or specifications are generated. A specification is generated when a service is started in development mode. Code changes cause a restart of the service, which then also updates the specification. Code generation works in a similar way. Here, a specification file is watched so that an OpenAPI generator can be invoked whenever a change is made. This is also linked to the infrastructure already in use, so the daily development flow does not change significantly.

5.2.2 Workflow Using an SDK

Making a service accessible via an SDK is an architectural decision made in other projects as well (Interview B, Interview D). A demonstration of this setup is done for the file service as a service provider. The main reason for this type of implementation is that this service is invoked by multiple clients, two of which are written in TypeScript (backend and simple runtime). Using an SDK allows developers to reuse the generated code and extend it with additional reusable functionality. To achieve this, the SDK accesses the OpenAPI specification of the file service to generate a basic client using a TypeScript code generator[8].

[8]https://openapi-generator.tech/docs/generators/typescript-axios

This client is partly modified and published to be importable by a runtime and the backend (Figure 5.4).

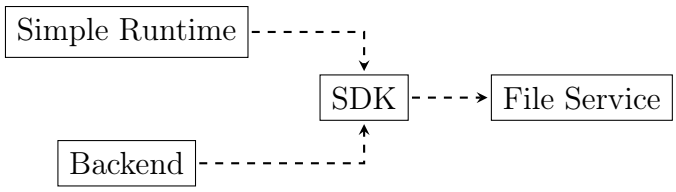


Figure 5.4: Indirect access from file service clients via an SDK.

We have enriched the generated code with three major functionalities before publishing it as an SDK. The first is the addition of automatic authentication headers to each method, which must otherwise be ensured by the caller. The second adjustment is a contravariant redefinition of some method parameters. This allows for more convenient use of the SDK. The last change introduces a fallback class for `FormData`, a class that is still experimental in the NodeJS version used in the project (OpenJS Foundation and Node.js contributors, 2022) and therefore not used in production. The fallback class can be provided as an optional argument when instantiating the SDK to enable NodeJS support. Only the adjusted classes are published and therefore available for a client.

5.2.3 Workflow Using Direct Access

JValue's front end uses RTK Query, a technology not used anywhere else in the application. Releasing a reusable SDK for it would therefore be an unnecessary overhead.

When the frontend is created or launched in development mode, the backend OpenAPI specification is taken as input to a code generator developed by the RTK Query developers[9]. RTK Query provides functions to extend the generated endpoint with additional properties, which in this case enables efficient caching.

This implementation accesses the backend OpenAPI specification directly from the frontend (Figure 5.5). However, the generated and refined code cannot be used in other services or frontends.

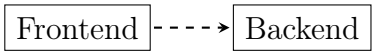


Figure 5.5: Direct access to the backend from the frontend.

[9]https://redux-toolkit.js.org/rtk-query/usage/code-generation#openapi

6 Evaluation

As a final step of the design science research method, it must be determined whether the developed artifact, in this case the categorization system, is indeed a viable solution to the previously defined problem (Peffers et al., 2007). Four different methods are used for this evaluation to assess the categorization system from separate viewpoints. In Section 6.1, the methodology for these methods is presented, while the subsequent discussion of the results can be found in Section 6.2.

1. **Requirements fulfillment:** The completion of the objectives from Chapter 3 is verified.
2. **Categorization system survey:** The categorization system as the primary artifact is assessed through member checking and data triangulation (with feedback from the interviewees and JValue members).
3. **Case study survey:** The implementation of the case study is also evaluated by surveying JValue developers, focusing on the benefits provided by the categorization system.
4. **Application to new software projects:** The categorization system is applied to projects that have not been involved in its development.

6.1 Methodology

As described in the introduction to this chapter, four different evaluation approaches are used. An explanation of the methodology employed for evaluating the requirements fulfillment and the application to new software projects is omitted because the procedures are straightforward. Nevertheless, they are performed and their results are presented in Subsection 6.2.1 and Subsection 6.2.4, respectively. The section explains the methodology for the categorization system survey (Subsection 6.1.1) and the case study survey (Subsection 6.1.2).

6.1.1 Categorization System Survey

The categorization system is assessed by sending the same survey to the interviewees for member checking and to JValue developers for a second perspective (triangulation). The survey consists of two parts – an assessment of the categorization system in general and a ranking for each individual category.

Member checking ensures that the "[...] conclusions are tested with members of those stakeholding groups from whom the data were originally collected" (Lincoln and Guba, 1985). In this case, member checks are conducted by asking the interviewees questions regarding general opinions about the categorization system and about each category itself. This method is taken from Lincoln and Guba, 1985.

In addition to member checking, triangulation is performed. This involves the evaluation of the categorization system by different stakeholders to ensure that a consensus is reached that provides more robust evidence compared to the case when the results are obtained from only one perspective (Guion, 2002). The different points of view are coming from the interviewees (who are also used for member checking) and the developers from the case study (JValue), both of whom received the same survey that is showcased in the following.

Both groups have access to a preliminary version of the categorization system when completing the survey. The evaluation is used to make minor improvements and clarifications to the artifact, however, these adjustments do not change the general idea of the artifact.

The general assessment consists of quantitative and free text questions. The quantitative questions (numbers 1, 3, 4, and 5 in Appendix C) are scored on a scale with seven response categories, as this number yields reliable and differentiated results while being easy for respondents to use (Preston and Colman, 2000). The questions require the respondent to indicate their degree of agreement with a statement about the categorization system. The free text questions (numbers 2, 6, and 8 in Appendix C) provide space for the respondents to elaborate on their answers.

Each category is ranked on a seven-point scale (question 7 in Appendix C). A respondent may assign the same rank to more than one category. This is allowed so that the respondent would have more freedom of choice in the evaluation, which should lead to more solid results than an order without overlapping ranks, especially since the number of replies is expected to be in the single-digit range.

Each question from the survey on the categorization system is optional, so the result contains only the carefully weighed opinions of the respondents. This also applies to the ranking of individual categories, i.e., one is free to omit the evaluation of a single or of multiple individual categories.

6.1.2 Case Study Survey

The case study survey provides further insight into the impact of the categorization system on the project to which it is applied in the demonstration (Chapter 5). It is sent to the main JValue developers, who are free to choose whether to respond in written form or to answer the questions in a short interview.

The first two questions aim at revealing their opinion about the fulfillment of the set goals:

1. How did the interoperability change, especially regarding the type safety of the endpoints?
2. How technology-agnostic is the new solution?

The categorization system should be a resource to help design and integrate a better interoperability workflow. Therefore, the next question tries to discover whether the developers like the new approach better than the former one:

3. How did the developer experience change?

The last question is asked to evaluate if the categorization system can provide value after it was applied once.

4. Do you plan to continue using this categorization system in the JValue project to document and potentially alter interoperability?

6.2 Results

This section presents and interprets the results of the evaluation. It includes the evaluation of the fulfillment of the objectives (Subsection 6.2.1) and the external evaluation of both the categorization system itself (Subsection 6.2.2) and its application in a case study (Subsection 6.2.3). Finally, insight is also provided into whether and how well the categorization system is applicable to previously not analyzed publications on the interoperability-centric software process of organizations (Subsection 6.2.4).

6.2.1 Requirements Fulfillment

This subsection evaluates how the requirements from Chapter 3 are met. This is done by analyzing each initial dimension and the general objectives defined before.

One goal is to consider the dimensions represented in Section 3.2 and not to block any of the properties represented there. The dimensions *Driver of Change*,

Technology Support, and *Level of Type Safety* exist in the categorization system with a very similar definition and naming as in the objective.

In contrast, the dimension *Amount of Manual Work* already differs quite significantly between the initial starting point and the final result. It is split up and influences other categories like *Level of Static Type Safety* and *Deployment of Different Versions* but is not a category on its own. Another factor in this dimension is the notifications when changes occur. The interviews did not provide enough information on this and it is therefore not part of the final categorization system. Further research on this item may be needed, as it is also mentioned as missing in the categorization system survey (Subsection 6.2.2, Appendix C).

The *Communication Architecture* is an interesting point because it seemed to be quite important after the superficial analysis of the interviews. Yet, it is not part of the final categorization system. A closer examination of the interviews shows that this dimension is a central decision in the realization of communication, but the conceptual process of ensuring syntactic interoperability is quite independent of it. *Communication Architecture* is therefore not a category when assessing solutions and problems of syntactic interoperability but is rather considered as an implementation issue.

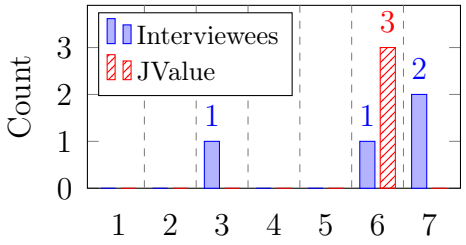
Manual Extensibility is a challenge that must be considered when seeking tooling support and designing a development-oriented interoperability workflow. This dimension is therefore very tool-specific which contradicts the intent of creating a tool-independent categorization system. Extensibility is therefore not represented in the categorization system itself but plays an important role in the design of an interoperability workflow as described in Subsection 4.2.2. In addition, the demonstration included an example of how extensibility can be achieved by publishing a modified version via an SDK (Subsection 5.2.2).

The research questions ("How can problems and solutions related to internal syntactic interoperability in microservice-based architectures be categorized at a conceptual level?") can be answered through the categorization system. It contains the most important topics that came up in the expert interviews while having a reasonable extent. The scope of six categories is a reasonable amount as it is well-processable while having all categories in mind (G. A. Miller, 1956). In addition, it does not contain any tooling suggestions but works only at the conceptual level, which keeps it up-to-date longer than a tool-centric system.

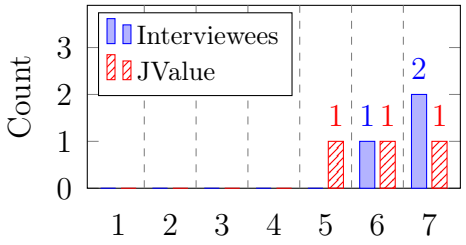
6.2.2 Categorization System Survey

Seven people replied to the survey, including four experts from the interviews and three JValue developers. The answers are depicted in Figure 6.1 and listed in Appendix C.

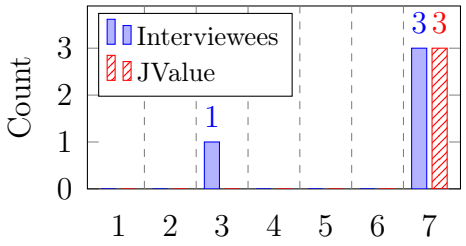
The general feedback assesses the categorization system as a useful and well-structured tool. It is considered (fairly) complete (Figure 6.1a) and useful (Figure 6.1c) by six out of the seven respondents. While there are some individual comments about why certain categorizations are not merged and that some might miss a category, the general opinion about the composition is also positive (Figure 6.1b). All but one respondent can imagine using the categorization system in the future, although the result is not as clear as for the other quantitative questions (Figure 6.1c). One conspicuity is the outlier in questions 1, 4, and 5, all from one respondent. Based on that person's free text responses, it appears that the purpose of the categorization system is not clearly stated. This weakness is addressed before the final publication.



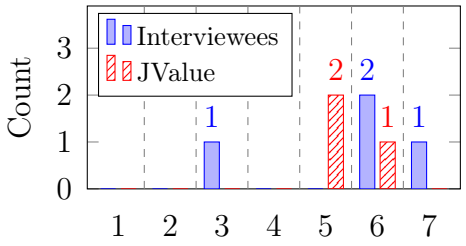
(a) Question 1: The categorization system contains all major categories.



(b) Question 3: The characteristics of the dimensions are allocated in a logically correct place.



(c) Question 4: The categorization system is useful.



(d) Question 5: I can see myself using the categorization system in the future.

Figure 6.1: Results for the quantitative questions with the range from 1 (strongly disagree) to 7 (strongly agree).

The categorization system is perceived as too focused on generators while the possibility of achieving interoperability through testing is omitted. During the interviews, all partners are asked if and how they pursue testing. It is often used as a tool to validate interoperability, however, automated testing is never the sole driver for ensuring interoperability, especially syntactic interoperability. It is also relatively often a task performed by manual testers and rarely extends to an entire server and API, but only to critical parts. So while testing is certainly an important part of software development, the interviews do not justify it being a common alternative to using generators and the like.

Other missing categories include synchronous vs. asynchronous communication, messaging vs. direct communication, and handling large payloads. However, the interviews show that these attributes only matter in implementation and are independent of the conceptual solution and categorization of interoperability issues. When providing feedback, the respondents only had access to a preliminary version of the solution from Chapter 4. Therefore, these categories might seem important from an individual's perspective but are not considered relevant on a conceptual and especially project-independent level.

Ranking from the most important to the least important category is not possible. Figure 6.2 shows that opinions about importance vary widely and no clear patterns are apparent. However, it can be said with caution that *Level of Static Type Safety* (Figure 6.2b) and *Technology Support* (Figure 6.2f) are considered more substantial, as five and six respondents respectively voted that they are at least fairly important. This allows us to conclude that the categories are well-balanced and all are justified to be included in the categorization system.

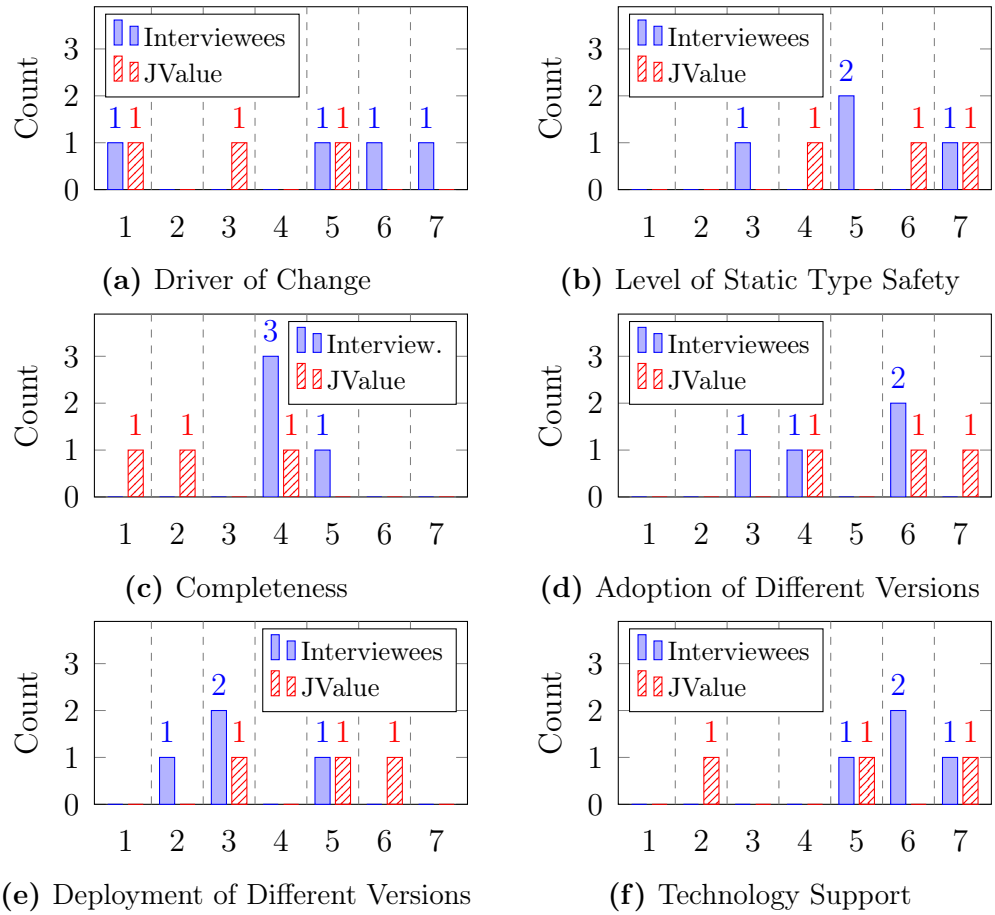
Member checking is concluded as successful. The interviewees as the provider of the initial data do recognize the purpose and validity of the categorization system. There is one exception, which could be explained by an inadequate explanation of the goals and a lack of insights into the other available data. Nevertheless, the member checking proves the credibility of the artifact.

The evaluation reveals no significant differences between the two stakeholder groups of interviewees and JValue developers and so the general opinions about the categorization system largely coincide. Also, the ranking of each individual category shows an agreement, as both groups are divided over the importance of each category. So there may not be an order from the most important to the least important, but both groups settle on that. Consequently, this triangulation verifies the validity of the categorization system.

In summary, this evaluation proves that the categorization system can be of practical use. From the perspectives of the respondents, the main benefit is to reflect on the current setup and to conceptualize alternatives. It can serve as an explicit or implicit mental model to remember all important considerations that are relevant when developing a microservice-based architecture. A more developer-friendly presentation, such as a concise cheat sheet, can improve effectiveness.

6.2.3 Case Study Survey

The evaluation for the demonstration is sent to the three leading JValue developers. They all chose to perform short interviews that are summarized and published in Appendix D.



(a) Driver of Change

(b) Level of Static Type Safety

(c) Completeness

(d) Adoption of Different Versions

(e) Deployment of Different Versions

(f) Technology Support

Figure 6.2: Ranking of the categories from 1 (least important) to 7 (most important).

All interviewees have mentioned that interoperability has improved. By utilizing generators instead of manually creating types, errors can be prevented, particularly since API endpoints are now generated on the client side as well. Type and API definitions have been moved to the service project rather than a shared library, simplifying the setup process and reducing laborious tasks. Furthermore, the API specifications now serve as a single source of truth, enabling clients to access the most up-to-date definitions. However, it is important to note that these benefits only come into play when the setup is properly used, i.e., when the code is annotated properly so a valid OpenAPI specification can be generated. However, developers will immediately detect any errors that result from an incorrect definition.

Another question refers to the technology-agnostic nature of the solution. The current setup is now decoupled from specific programming languages and frameworks. Opting for a widely used standard such as OpenAPI to build the API

specification is a wise decision due to the plentiful tooling support available. However, it is essential that a generator for the employed programming languages is available. If it is not, one can always fall back to develop a custom generator. It is worth noting that the current setup still only supports REST, which remained unchanged in the new implementation as it is not planned otherwise for the case study.

A further question aims to find out if the developer experience has improved. Day-to-day development with the existing setup is now easier, especially since the implementation now immediately reflects changes from the server in all clients, which also helps to detect errors instantly. Therefore, normal usage is easier, facilitating new developers' onboarding. However, profound changes now require more background knowledge and are generally more complex. Good documentation, like the one existing now, is essential for understanding the full setup.

The evaluation concludes by asking whether there are plans to continue using the categorization system as part of the JValue project. It is not expected to be kept up-to-date and beneficial for all developers. According to the interviewees, the main benefit of the categorization system is that it provides a good common ground for discussing major refactorings. Therefore, it is seen as a purpose-driven tool that helps with discussions when the fundamental interoperability setup should be altered. Several respondents mention that they can imagine that this is the case when independent deployment and version adoption are integrated.

In conclusion, the application of the categorization system in JValue can be considered successful. The new implementation is more complex, but this complexity is an accepted trade-off in order to simplify everyday development and enhance the robustness of the services. The categorization system primarily serves the purpose of facilitating discussions on significant changes in the setup, and it is anticipated to be utilized for that specific purpose.

6.2.4 Application to New Software Projects

The categorization system is also applied to publications about software projects of two other organizations: Otto and Zalando, both of which deploy microservices. This subsection is entirely based on Otto GmbH & Co. KG, 2023 and Zalando SE Opensource, 2023 and therefore no further references are given in the text. Both have published API guidelines, which are sufficient to get insight into all but one category of our categorization system. Information about *Deployment of Different Versions* is not found in either policy. However, they enforce backward compatibility of individual services, which makes the use of an *automated* solution relatively easy and therefore likely.

Both approaches are quite similar and show a clear separation between different

communication principles. They primarily include REST as a synchronous type of communication and events as an asynchronous type of communication. Each communication principle has its own guidelines, and so the *Completeness* can be characterized as *partial*. Since REST plays a larger role in this study, it is also the focus of the following analysis.

Both Otto and Zalando focus on a strict *specification-first* approach regarding the *Driver of Change*, which both say improves the quality of API design. Otto calls this "Contract first", while Zalando refers to this as "API first".

Each REST API is fully defined in OpenAPI specification files. These files contain both the input and output data types as well as complete information about the API endpoints, including, for example, HTTP methods and authorization information. There is no enforcement to use generators or the like to ensure static type safety. However, the availability of a fully defined API specification opens up the possibility of setting the *Level of Static Type Safety* to *everything*.

Breaking changes come up at Otto and Zalando as well. The API should implement versioning to ensure backward compatibility when breaking changes are not avoidable. If certain endpoints are removed in the future, they need to be marked formally correct as deprecated in the specification and must send information about when the endpoint will be shut down when further requests are made to that deprecated endpoint. Therefore, the different services are integrated independently from each other, and thus the *Adoption of Different Versions* is *delayed*.

The category *Technology Support* is well defined at Otto and Zalando, as both use a *technology-agnostic* solution with OpenAPI. This gives them the flexibility to choose and alter technologies underlying an API.

In summary, both Otto's and Zalando's approaches to ensuring syntactic interoperability in a microservices-based architecture can be well described with the categorization system. Only one category (*Deployment of Different Versions*) cannot be derived from their publications. These examples indicate that the categorization system is more broadly applicable, as yet unknown software processes can be successfully analyzed with it. Therefore, the artifact does not focus too much on the source dataset available within this thesis.

It is also worth noting that Otto and Zalando are developing microservices on a larger scale than any of the other projects previously analyzed. The fact that the categorization of these two large projects is almost identical suggests that the categorization system has its strengths primarily in smaller projects. In these smaller projects, it is more feasible to deviate from the microservice standard in order to leverage project-specific strengths.

7 Conclusion

This thesis is about categorizing suitable ways to ensure syntactic interoperability in microservice-based architectures. The expert interviews conducted in this research have shown that each microservice-based project has very unique requirements and challenges regarding syntactic interoperability. Therefore, this work does not present concrete software processes as general-purpose templates for establishing interoperability in microservice-based architectures, since this is not yet practically feasible. Instead, it presents a method for categorizing a software setup that is flexible enough to benefit all architects of microservice-based applications. The research question underlying this thesis – How can problems and solutions related to internal syntactic interoperability in microservice-based architectures be categorized at a conceptual level? – can therefore be answered well through the categorization system. Together with the process template from Subsection 4.2.2, the categorization system also provides an interoperability analysis approach that includes conceptual and technical parts as requested by Abukwaik and Rombach, 2017.

The developed categorization system is already considered useful and widely complete. Moreover, by its semi-strict nature as a categorization system, it provides flexibility to be interpreted in such ways that it suits a project definition best. That means that some characteristics do not have to apply completely for a concrete project to be assigned to a category but are left to the disposition of a software architect. The categorization system has also proven valuable when applied to the projects of JValue (Chapter 5), Otto, and Zalando (both Subsection 6.2.4). It is worth mentioning that none of these projects had a direct impact on the design of the developed categorization system. This supports both its value and applicability.

Further research within the context of this categorization system can be done by including tooling suggestions for specific or all category combinations. Several tools and API specification languages are just given as examples. However, a more detailed analysis would facilitate the planning required to decide on tooling that supports a project's software process. For example, OpenAPI has proven to be a good specification language for the use case of the case study (Chapter 5).

Nonetheless, OpenAPI may not be suitable for projects with other characterizations from the categorization system. Therefore, an interesting field for further research would be to find out which tools work best for which use case.

In summary, the developed categorization system is a valuable tool for software architects by facilitating the evaluation of existing software processes and aiding in strategic planning for future improvements. The categorization system can therefore serve as a purpose-oriented tool to support socio-technical challenges related to refactorings and to facilitate communication about them.

Appendices

A Interview Guide

1. Warm-Up Fragen
    - Bitte stellen Sie sich und Ihre Aufgaben in Ihrer Firma kurz vor.
    - In welchem Kontext benötigen Sie syntaktische Interoperabilität? (z.B. für Microservices nach wissenschaftlicher Definition oder für die netzwerkbasierte Kommunikation größerer Services)
2. Interoperabilität sicherstellen
    - Wie verteilen Sie Typinformationen? (z.B. manuelles Erstellen aus einer Dokumentation, OpenAPI-Generatoren, automatisch generierte Stubs, ...)
    - Wie oft synchronisieren Sie diese Typinformation?
    - Wie stellen Sie sicher, dass nur kompatible Versionen gestartet werden?
    - Welche API-Paradigmen benutzen Sie? (z.B. REST oder GraphQL)
    - Wie zufrieden sind Sie mit ihrer Art, die Typinformationen auszutauschen? Wo gibt es noch Verbesserungsbedarf?
3. Code und Repository Setup
    - Mit welcher Art von Repository entwickeln Sie? (z.B. Monorepo, Multi-Repo oder andere Ansätze)
    - Wie und wann stellen Sie die Korrektheit von Typdefinitionen sicher? (z.B. aus OpenAPI Definitionen vor dem Deployment, über PACT broker, spezielle Tests)
    - Wie stellen Sie sicher, dass Sie die korrekten API-Endpunkte für jede Anfrage benutzen?
    - Wie strikt sind die Nutzer an das exakte Modell gebunden (Postel's law)?
    - Wie beeinflussen die benutzten Sprachen und Frameworks Ihre Lösung?
4. Ausblick
    - Welche weiteren noch nicht genannten Herausforderungen stellen sich Ihnen bei der Entwicklung von Microservices?

- Gibt es noch weitere interessante Konzepte im Bereich der Interoperabilität, die noch nicht behandelt wurden?

B Interviews

B.1 Interview A

I [00min 00s]: Hallo Peter. Schön, dass das geklappt hat mit dem Interview. Wir haben uns ja vorher schon kurz darüber ausgetauscht, was in dem Interview alles aufgenommen wird oder gefragt wird und jetzt möchte ich noch einmal explizit fragen, ob du mir der Aufnahme des Interviews einverstanden bist.

A [00min 24s]: Ja klar.

I [00min 27s]: Gut. Dann würde ich dich bitten am anfangen dich kurz vorzustellen, was du machst, welche Rolle du in deiner Firma hast.

A [00min 39s]: Ich bin der Peter Herbst. Ich arbeite jetzt nach dem Informatik Studium als Software Engineer bei einer kleineren Firma. Wir entwickeln hauptsächlich frontend-backend-Architekturen und ich bin da stark im backend tätig, aber ich wirke auch im frontend mit. Also ich bin quasi full-stack Entwickler und ich freue mich heute ein bisschen was aus dem Erkenntnissen dort zu erzählen.

I [01min 10s]: Super, das klingt ja perfekt. Das heißt, wo braucht ihr dann die syntaktische Interoperabilität hauptsächlich?

A [01min 21s]: Wir entwickeln mit microservices, zwar im kleineren Stil, weil wir eben nicht so viele Leute haben, die mitentwickeln. Und wenn die microservices untereinander kommunizieren, da müssen sie sich natürlich auf die Typen einigen. Da benötigen wir sehr viele Schemen, um das umzusetzen und dann natürlich auch im Allgemeinen wenn das frontend schon denn mit dem backend kommuniziert, wird es natürlich auch gebraucht.

I [01min 57s]: Okay, das klingt ja perfekt für meinen Anwendungsfall. Wie viele Services sind da immer involviert? Also gibt es nur frontend Back-End oder sind das auch mehrere microservices?

A [02min 10s]: So im Durchschnitt haben wir im backend ein bis zwei Microservices und eben noch das frontend, also man kann sagen so wir haben drei Akteure die miteinander kommunizieren.

I [02min 24s]: Und von den menschliche Akteuren - habt ihr da mehrere Teams oder beschränkt sich das auf eines?

A [02min 30s]: Wir sind insgesamt fünf Personen und da die Projekte kleiner sind, arbeiten immer etwa zwei bis drei Personen an einem Projekt, also, man kann sagen, wir haben schon mehrere Teams.

I [02min 47s]: Okay, also mehrere Teams. Das bedeutet eine bestimmte Gruppe kümmert sich nur um das backend oder um den einen Service oder greift jeder mal bei einem anderen Thema ein und entwickelt quasi das gesamte Produkt und nicht nur einen microservice?

A [03min 10s]: Aufgrund der kleinen Teamgröße haben wir schon bisschen Rollenverteilung, dass die Leute die sich mit dem backend auskennen auch eher am backend werkeln und die frontend Leute eher am frontend, aber wir haben keine festen Grenzen. Also ich wirke auch im frontend mit und ein frontend-Entwickler wirkt auch am backend mit.

A [03min 33s]: Das heißt für Teams im Sinne von ist SCRUM ist es dann trotzdem eher ein Team für mehrere Projekte, so wie ich das verstanden habe.

A [03min 45s]: Genau.

I [03min 47s]: Perfekt, dann würde ich gleich mal weiter gehen und dich fragen wie ihr denn die Typ-Informationen austauscht, also habt ihr da entweder geteilte Typ-Informationen oder benutzt ihr OpenApi, WSDL oder eine ähnliche Technologie.

A [04min 08s]: Zu Begin hatten wir z.B. noch kein monorepo, da haben wir im Endeffekt die Typen alle kopiert und teilweise hat man dann natürlich auch das Problem, dass wir andere verschiedene Sprachen verwendet haben im backend und frontend. Da haben wir dann mal den Ansatz probiert mit Generatoren, also automatisch aus dem Typ im backend, z.B. den Typ in frontend zu generieren, aber das hat nicht so wirklich gut funktioniert, dann sind wir wieder davon abgewichen. Dann haben wir es manuell gemacht - kopiert und dann manuell abgeändert für die andere Sprache. Und jetzt sind wir aktuell ziemlich stark bei einem Ansatz gelandet, dass wir Monrepos verwenden, wo wir auch im frontend und backend die gleiche Sprache haben. Konkret NodeJS beziehungsweise TypeScript und da haben wir letztendlich dann einfach eine eine shared library, die die Typen enthält, so dass das frontend und das backend darauf zugreifen kann.

I [05min 20s]: Okay, das war ja schon sehr ausführlich, da steigen wir dann später noch mal tiefer rein. Was mich kurz interessiert bei den Generatoren: sind das Generatoren, wie von OpenApi oder waren das einfach nur Generatoren, die von einer Sprache in die andere quasi übersetzt haben?

A [05min 41s]: Da wurde einfach nur von der einen Sprache in die andere übersetzt, also so etwas wie OpenApi haben wir bis jetzt noch nicht verwendet.

I [05min 50s]: Du hast gesagt die Generatoren waren nicht zufriedenstellend in dem Ausmaß, wie ihr euch es erwünscht habt.

A [06min 02s]: Exakt. Da kam teilweise nicht die richtigen Ergebnisse heraus, da musste man sowieso noch einmal nacharbeiten. Daher war das für uns kein sinnvoller Ansatz.

I [06min 15s]: Das heißt, bei eurem alten Projekt war das auch nicht viel automatisierte Arbeit, richtig? Wenn du sagst, du hast die Typen genommen

und dann quasi manuell in die andere Sprache übersetzt.

A [06min 31s]: Genau. Es war sehr viel aufwendige Arbeit, die relativ einfach aber auch war.

I [06min 51s]: Jetzt, hast du gesagt, ihr verwendet ein monorepo mit den gleichen Sprachen - bis du damit mehr zufrieden?

A [06min 03s]: Auf jeden Fall. Es nimmt einiges an arbeit letztendlich ab.

I [07min 09s]: Verwendet ihr dann exakt die gleichen Klassen für die Typen, die aus dem Service 1 kommen für den Service 2, also zum Beispiel für backend und frontend?

A [07min 21s]: Man könnte es theoretisch so machen, aber es gibt immer einige Sonderfälle, die letztendlich dafür sorgen, dass es letztendlich nicht so funktioniert. Teilweise ist es z.B. so, dass wir im backend eine Klassen haben für einen User, die einen Passwort-Hash enthält, aber den wollen wir natürlich nicht ans frontend senden. Das heißt, wir haben immer mehrere Klassen. Vor allem für die Endpunkte haben wir teilweise dann eben eine Klassendefinition, die eine Response darstellt, die von diesem Endpunkt kommt. Und wir haben eine Definition, die eine Anfrage darstellt. Wir haben da relativ viele Klassen, die wir auch wirklich manuell erstellen müssen - das ist schon sehr aufwendig.

I [08min 19s]: Okay. Das heißt, ihr arbeitet dann viel mit Data Transfer Objects und habt da sehr viele verschiedene Definitionen.

A [08min 29s]: Exakt.

I [08min 30s]: Verwendet ihr TypeScript auch auf dem Ausmaße, dass ihr sagen könnt: ihr habt zwar diesem Stammbaum an einem vollständigen Objekt zum Beispiel, aber für dieses DTO sind nur drei von den zehn Attributen erlaubt. Verwendet ihr so etwas?

A [08min 53s]: Es ist ja eine extrem schöne Funktion von TypeScript, dass man basierend auf dem schon existierenden Typen einen neuen Typen erstellen kann, bei dem vielleicht manche properties fehlen oder bei dem manche properties optional sind. Das verwenden wir stark, weil das gewissermaßen ein wenig Typsicherheit bringt. Also im Endeffekt, wenn man dann die Base-Klasse verändert und man vergisst es dann in der Subklasse, dann wird in vielen Fällen sich der compiler dann beschweren.

I [09min 32s]: Okay. Das heißt, ihr verwendet Omit dann explizit, wenn ihr jetzt einen User habt und ihr verwendet in der einen Anfrage eben nur Nutzername und Passwort und ihr fügt dem originalen Interface quasi ein neues Attribut hinzu, dann müsst ihr dieses DTO mit nur Nutzername

und Passwort kurz noch einmal bearbeiten und sagen, dass auch das neue Attribut auch exludiert werden soll. Habe ich das so richtig verstanden?

A [10min 11s]: Genau. Also, wenn man es genau nimmt wäre das eigentlich sogar mit drinnen. Der Vorteil ist eher in der anderen Richtung. Wenn man beispielsweise ein neues Attribut hinzufügt in der Base-Klasse und man benötigt es auch in der Anfrage, dann ist es auch automatisch mit dabei. Also es kommt darauf an. Es gibt die Methode, dass man explizit aus der Base-Klasse properties auswählt. Wenn man dann also eines hinzufügt, ist es in der Base-Klasse nicht mit dabei. Dann gibt es aber auch die Methode, dass man von der Base-Klasse properties entfernt. Und wenn man dann eines hinzufügt, dann ist es mit dabei in der Subklasse. Also je nachdem.

I [11min 03s]: Also sowhl aufbauend als auch quasi die vorhandenen Typdefinitionen verkleinern ...

A [11min 09s]: Genau.

I [11min 10s]: ... je nachdem, wie es für den Ansatz passt.

A [11min 12s]: Exakt.

I [11min 17s]: Verwendet ihr da reines TypeScript oder habt ihr auch andere Frameworks oder Helper-Frameworks, die die Typdefinition dann auch sicherstellen, weil TypeScript an sich ist ja auch erst einmal nur statisch.

A [11min 33s]: Genau. Vor allem beim Netzwerktransfer mit den DTOs können eigentlich keine Garantiren von TypeScript oder sowas gemacht werden. Deswegen verwenden wir eben noch die Frameworks class-transformer und class-validator. class-transformer hat die Aufgabe, aus den properties in JSON-Daten teilweise wieder Typen herzustellen. Ein Beispiel wäre zum Beispiel: ein Date-Objekt aus JavaScript kann nicht über JSON transportiert werden, sondern wird als String transportiert. Und class-transformer kann dann daraus wieder ein Date machen. Und nach class-transformer kommt dann class-validator zum Einsatz. Wir annotieren nämlich unsere gesamten DTO-Klassen mit Annotations von class-validator und können so bestimmen, welche Typen diese properties haben müssen. Und dann kann class-validator das relativ einfach validieren. Das heißt, wenn ein JSON kommt, wo ein property fehlt, aber es ist in der Annomation angegeben, dass es mit drinnen sein muss, dann wird class-validator einen Fehler werfen.

I [12min 52s]: Ah okay, das klingt ja interessant mit der zusätzlichen Validierung auch noch. Wie schon gesagt, OpenApi oder ähnliche verwendet ihr nicht?

A [13min 06s]: Aktuell noch nicht, was wahrscheinlich auch der Projektgröße geschuldet ist. Da ist die Frage, ob es mehr bringt oder im Endeffekt nur mehr Aufwand ist.

I [13min 17s]: Bist du dann zufriedener damit, dass du zwar jetzt nicht eine perfekte Lösung für diese Größe hast, aber schon einmal eine Lösung, die nicht zu viel overhead bietet wie OpenApi?

A [13min 32s]: Ja, auf jeden Fall. Also die Lösung, die wir aktuell haben, vor allem seit dem das Monorepo dabei ist, würde ich sagen ist für unseren Programmieralltag extrem förderlich.

I [13min 43s]: Du hast gerade schon Monorepo angesprochen. Das bedeutet auch, dass die verschiedenen Typdefinitionen immer sehr sehr schnell auch verteilt werden, richtig?

A [13min 54s]: Im Prinzip weiß ja dann jedes Projekt von den neuen Typen wenn ein Typ irgendwo verändert wird, weil eben auf diese ge-shared-e library zugefriffen wird.

I [14min 08s]: Das heißt, wenn ihr jetzt dem Nutzer irgendein neues Feld mitgebt, dann kann ich auch das frontend nicht neu bauen, weil jetzt eben das neue Feld enthalten muss und der Service bekommt auch mit, dass er das neue Feld jetzt benötigt.

A [14min 40s]: Genau. Ich meine, wie du es schon im Beispiel erwähnt hast. Ich meine, im Endeffekt verwenden wir den Typen im frontend und wenn wir dann z.B. ans backend Daten senden wollen, dann wird der TypeScript compiler erkennen, dass da ein field fehlt.

I [14min 51s]: Okay, perfekt. Das bedeutet, sowohl vor Bau als vor Deployment ist bei euch auch instantly sichergestellt, dass die Typen korrekt sind. Klingt ja gut.

A [15min 11s]: Im größten Teil ja. Es gibt nur bei TypeScript das Problem, dass im Allgemeinen in TypeScript Typsicherheit schwierig ist, sobald man fremde libraries mit verwendet, weil diese fremden libraries sich nicht unbedingt an Typen halten müssen. Wenn natürlich jede library absolut typsicher wäre, dann wäre das so. Aber sobald man andere libraries verwendet, können schon Probleme auftreten, die wir tatsächlich auch schon hatten.

I [15min 42s]: Okay. Kannst du dann ein kurzes Beispiel nennen?

A [15min 46s]: Ja klar. Wir haben eine library für die Interaktion mit der Datenbank, also so eine ORM library, die heißt TypeORM. Und es kann zum Beispiel sein, dass wir ein Entity haben, das laut der Entity-Definition die richtigen Typen hat, die wir für unser DTO brauchen, aber theoretisch, wenn TypeORM dann irgendetwas anderes zurückgibt, weil es vielleicht in der Datenbank anders steht oder weil vielleicht ein Bug drinnen ist, also da kann es sämtliche Gründe geben dafür und TypeORM ist einfach eine riesige library, das heißt es kommt tatsächlich auch einmal vor. Und schon

hat mal dann eben nicht mehr diese statische Typsicherheit von TypeScript zur compile-Zeit. Was dann natürlich trotzdem helfen wird, ist die Typsicherheit zur Laufzeit von class-validator. Die würde trotzdem noch bestehen.

I [17min 00s]: Okay. Das heißt ihr müsstet quasi die Entity mit class-validator und dann wahrscheinlich dann auch class-transformer annotieren, so dass die richtigen Typen dann auch herausgesendet werden ans frontend.

A [17min 16s]: Genau. Theoretisch ist man auch denke ich relativ frei, ob man das dann tatsächlich im backend macht, weil dieses validieren ist wirklich relativ Performance-aufwendig. Oder ob man dann auch vielleicht im frontend validiert, aber da haben wir noch keine Musterlösung gefunden.

I [17min 36s]: Okay. Du hast vorhin JSON schon angesprochen. Verwendet ihr dann nur JSON zum austauschen, wahrscheinlich dann auch basiert auf REST, oder verwendet ihr auch mit zum Beispiel GraphQL oder XML-basierte Kommunikation?

A [17min 58s]: Bisher verwenden wir wirklich nur REST und JSON. Bei irgendwelchen XML-Schemen ist wieder das Problem, dass es relativ aufwendig ist und der Mehrwert ist fragwürdig, vor allem bei so kleinen Projekten. Und das gleiche gilt eigentlich für GraphQL auch. Also ich denke, für so etwas brauchst du größere Projekte und größere Teams.

I [18min 22s]: Okay. Um den ersten Teil jetzt einmal kurz abzuschließen: Du bist zwar nicht hundertprozentig zufrieden mit der Lösung, die du gefunden hast, aber es könnte schlimmer sein und gerade für diese Größe von Team - es ist ein entwickelndes Team, relativ wenige Entwickler. Findest du, dass es trotzdem eine gute Lösung ist?

A [18min 52s]: Genau, auf jeden Fall. Für uns haben wir eine Lösung gefunden, die uns erlaubt, wirklich effizient zu programmieren, aber auch sicher.

I [19min 02s]: Okay. Dann würde ich jetzt ein bisschen tiefer in den Code und in die ganze Repositorystruktur hinein steigen. Du hast ja schon erwähnt, dass ihr jetzt in den neueren Projekten reine monorepos verwendet. Und du hast auch vorhin schon gesagt dass es euch super die Vorteile gibt mit den geteilten Typdefinitionen. Gibt es da auch Einschränkungen, die dieses Leben dann auch schwieriger machen im Vergleich zu einem anderen Repository-Setup?

A [19min 42s]: Eine Sache, die natürlich direkt auffällt, ist dass es einfach eine extreme Typexplosion ist. Vor allem wenn man dann auch noch solche Sachen wie schon angesprochen macht, also dass man Typen basiert auf irgendwelchen base-classes. Dann kommt da sehr viel zusammen. Was natürlich in einer einzigen library ist und auf die dann auch wiederum

andere Services Zugriff haben, die es vielleicht gar nicht benötigen. Also das wäre z.B. eine Einschränkung oder ein Problem.

I [20min 20s]: Und die Lösung mit feature-libraries - also wenn jetzt zum Beispiel die Nutzerkommunikation in einer feature-library ist und die Kommunikation für den Wareneingang in einer feature-library - die würde ja auch dann bei euch zu einer Typexplosion, zwar nicht direkt von den Klassen, oder auch von den Klassen, aber vor allem auch zu einer Explosion von libraries führen, richtig?

A [20min 50s]: Genau. Im Endeffekt hätte man trotzdem die gleichen Klassen und man hätte auch noch mehr libraries. Es verringert wieder die Übersichtlichkeit.

I [21min 05s]: Würdest du trotzdem bei einem monorepo bleiben oder wieder zurück zu einem Multi-Repo gehen?

A [21min 14s]: Also nach aktuellem Stand, vor allem bei der Projektgröße würde ich auf jeden Fall bei einem monorepo bleiben. Aber ich sehe schon auch die Vorteile wenn man wirklich größere Projekte hat. Einfach dadurch, dass man sich auf eine kleinere Codebase fokussieren kann und so besser abstrahieren kann quasi.

I [21min 45s]: Du hast schon gesagt, dass ihr sowohl statisch als auch zur Laufzeit die Typinformation checkt, also statisch mit TypeScript und auch class-transformer/class-validator vor allem verwendet. Verwendet ihr auch testing? Wie testet ihr es dann, dass Datentypen so zurückkommen, wie sie es auch sollten? Du hast ja auch erwähnt, dass mit TypeORM z.B. nicht unbedingt das zurückgegeben wird, was du erwartest und was auch statisch TypeScript sagt, was dieser Typ ist.

A [22min 37s]: Genau. Also das wäre natürlich auf jeden Fall wünschenswert, wenn man Tests schreib für jeden einzelnen Endpunkt, auch mit mehreren cases, ob da genau das zurückkommt, was man erwartet. Wenn man das auch mit einer CI/CD-Pipeline kombiniert, direkt sieht, ob man Fehler hat. Aber das große Problem ist einfach der extreme Aufwand, also immer diesen Test-Boilerplate Code zu schreiben und dann die Test-cases und dann kommt teilweise auch Aufwand dazu weil man Test-cases auch updaten muss, weil Typen ändern sich ja prinzipiell auch oft. Und deswegen machen wir aktuell den Ansatz, dass wir so spezielle Tests nur eigentlich nur für wirklich die wichtigen Endpunkte schreiben, wo es vielleicht sicherheitskritisch ist oder wo vielleicht private Informationen geleaked werden könnten. Ein gutes Beispiel wäre eben wieder zum Beispiel so ein User-Endpunkt, der dann plötzlich den Passwort-Hash zurückgibt. Und dann schreiben wir eben explizit einen Test, dass das nicht so ist.

I [23min 46s]: Das heißt, auf Grund eurer Projekt- und eurer Teamgröße reduziert ist das Testing auf die kritischen stellen.

A [23min 57s]: Exakt.

I [24min 00s]: Wenn ihr Tests schreibt - ihr habt zum Beispiel einen fixen Endpoint um den es sich jetzt dreht. Da muss es ja von zwei Seiten getestet werden, d.h. sowohl das backend muss das richtige Ergebnis quasi liefern und das frontend muss z.B. die richtige Anfrage liefern. Könnt ihr da auch die gleichen Klassen verwenden oder sind das zwei stark getrennte Tests?

A [24min 33s]: Also das Problem ist, wenn man das wirklich trennen würde, das würde wieder extremen Aufwand bringen. Aktuell testen wir von der frontend Seite eigentlich gar nicht, weil wir eben auf diesen Validierungsmechanismus von class-validator bauen.

I [25min 07s]: Okay. Wie stellt ihr sicher, dass ihr die korrekten Endpoints verwendet? Benutzt ihr da z.B. kleinere e2e-Tests?

A [25min 18s]: Wir haben das mal gemacht bei unserem ersten Projekt. Da haben wir cypress verwendet und tatsächlich auch den kompletten Aufwand gefahren mit backend hochfahren, Datenbank hochfahren, frontend hochfahren, und wirklich die e2e durchgeführt. Aber der Aufwand im Vergleich zum Nutzen war uns da wieder zu gering, weshalb wir das wieder eingestellt haben. Wir haben da bis jetzt auch keine wirklich gute Alternative gefunden.

I [25min 53s]: Das heißt, ihr seid noch auf der Suche nach einer guten Lösung für die Endpoints. Wahrscheinlich dann auch verbunden mit Tests?

A [26min 02s]: Genau. Wenn sich da was ergibt. Wir sind da offen dafür.

I [26min 06s]: Super, vielleicht ergibt sich ja etwas im Laufe meiner Arbeit.

A [26min 11s]: Das wäre ja wunderbar.

I [26min 14s]: Du hast ja schon erwähnt, dass ihr viele verschiedene DTOs verwendet. Das bedeutet auch, dass jeder endpoint bei euch relativ strikt definiert ist, richtig? D.h. nach postels law - wenn dir dass vielleicht etwas sagt, das sagt aus dass man eben relativ flexibel sein soll in dem was man bekommt. Arbeitet ihr auch danach, oder eher weil das auch wieder ein relativ kleines Projekt ist und die Komponenten recht stark gekoppelt sind könnt ihr dann auch sicher gehen, dass die Interfaces relativ strikt sind und auch so geschickt werden, wie sie erwartet werden?

A [27min 06s]: Genau, wir machen das wirklich hand-craftet. Prinzipiell, wir haben schon die Möglichkeit, dass wir beispielsweise optionale in gewissen DTOs angeben. Aber das machen wir wirklich manuell. Was sind die

großen Vorteile davon wenn man so etwas machen würde? Dann könnte man die Clients ein bisschen flexibler machen, aber da wir im Prinzip auch nur einen Client Programmieren, brauchen wir diese Flexibilität nicht und die vielleicht gewonnene Effizient durch die niedrigere verwendete Bandbreite ist für uns auch nicht wirklich relevant. Also das sehe ich erst in größeren Projekten, die Notwendigkeit.

I [28min 01s]: Genau, das war auch mein Eindruck, dass postels law vor allem bei mehreren Clients eine große Rolle spielt.

Mit eurer Lösung von code-sharing innerhalb von einem monorepo schränkt ihr euch ja auch relativ stark ein auf zumindest die Sprache mit TypeScript, die ihr verwendet. Limitiert euch das auch in den Frameworks, die ihr benutzt?

A [28min 34s]: Im Prinzip sind wir dadurch natürlich schon stark an die TypeScript-Frameworks gebunden und wir verwenden aktuell auch immer ziemlich die gleichen. Also daher würde ich schon sagen, dass es uns einschränkt. Aber ich denke auch, theoretisch beim microservice-Ansatz könnte man ja dann tatsächlich auch im Endeffekt im Nachhinein teilweise Sprachen mixen. Also es wäre ja kein Ding der Unmöglichkeit jetzt zu sagen, wir haben jetzt zwar bisher alle microservices in TypeScript, aber wir brauchen jetzt z.B. einen Machine-Learning-microservice und da da eben die beste Sprache dafür eben wahrscheinlich Python ist, schreiben wir dass jetzt in Python. Prinzipiell wäre das ja machbar. Ich würde sagen, der monorepo-Ansatz ist für uns ganz gut und ganz sinnvoll, aber wenn wir wirklich mal mehr brauchen, dann wäre das auch noch gut machbar.

I [29min 46s]: Ihr seid trotzdem flexibel genug, dass ihr für das frontend React oder Angular verwenden könnt oder beides zum Beispiel und für das backend z.B. NestJS oder express. Ihr seid nur limitiert auf die Sprache und nicht auf das konkrete Framework?

A [30min 05s]: Genau. Vor allem mit dem Ansatz, dass wir class-transformer und class-validator verwenden. Das ist ja auch nicht wirklich gekoppelt an ein Framework.

I [30min 18s]: Ich habe noch eine kurze Frage, die sich noch um die Services allgemein dreht und die ich vorhin vergessen habe zu fragen. Wie sieht es mit dem Deployment aus? Wie oft deployed ihr da ungefähr und wie viele Services deployed ihr da zusammen?

A [30min 42s]: Wir haben jetzt keinen extrem regelmäßigen deployment-cycle. Ich meine, das wiederhole ich jetzt schon extrem oft, aber es ist nunmal einfach so, dass wir kein twitter oder so etwas sind, wo wir wöchentliche bugfixes machen müssen. Wir deployen, wenn wir ein Inkrement haben und

da wird auch alles direkt auf einmal deployed. Auch wenn wir microservices haben, die theoretisch individuell deployed werden könnten, der Aufwand hat für uns aktuell keinen Mehrwert.

I [31min 23s]: Das bedeutet auch, dass ihr jetzt nicht nur die Services neu deployed, die sich geändert haben. Also wenn jetzt z.B. euer Machine-Learning-Service nicht verändert hat und die anderen beiden trotzdem, deployed ihr trotzdem auch den Machine-Learning-Service neu, oder nur die Services, die sich auch wirklich verändert haben?

A [31min 57s]: Also prinzipiell, wenn man sich so die CI/CD-Pipeline anschaut, dann wird ja jeder Service in einen Container gepackt und gebaut. Beim Build-Vorgang wird im Endeffekt tatsächlich trotzdem dann jedes Mal für jeden Service ein neuer Container gebaut, auch wenn sich der Code nicht geändert hat, weil Docker einfach erkennt, dass hier in der codebase etwas verändert wurde und im Endeffekt wird dann wirklich alles neu deployed, ja.

I [32min 40s]: Gut. Dann würde ich hiermit den harten offiziellen Teil schließen. Noch eine kurze kleinere Frage, die wahrscheinlich schon sehr geklärt ist: Wie geht ihr mit versioning um? Du hast ja gesagt, wenn sich Änderungen ergeben, dann deployed ihr alles neu und so wird es wahrscheinlich auch für euch keinen Grund geben, besonderes versioning einzubauen und backwards-compatibility?

A [33min 16s]: Genau. Ich meine, wir deployen natürlich nur, wenn die Version lauffähig ist und lauffähig ist sie eben dann, wenn die Versionen zu einander passen, sage ich einmal, der microservices. Also müssen wir da kein versioning oder so betreiben, dass wir noch alte Versionen aktuell halten.

I [33min 34s]: Das heißt, das geht auch damit ein, dass ihr wahrscheinlich von jedem Service auch nur eine Instanz am laufen habt und nicht z.B. von einem Machine-Learning-Services fünf verschiedene Instanzen.

A [33min 50s]: Exakt. Wir verwenden bisher auch z.B. kein Skalieren der microservices. Wir haben wirklich eine Instanz von microservices.

I [34min 01s]: Okay. Gibt es noch weitere challenges, die du entdeckt hast beim Entwickeln von microservices, die jetzt noch nicht genannt wurden? Du hast ja vorhin schon die Typexplosion genannt.

A [34min 20s]: Also im Allgemeinen, wenn man es jetzt auf Kommunikation zwischen den microservices bezieht, hat man wirklich beim error handling extreme Probleme. Das stelle ich mich jetzt z.B. so vor wenn man irgendwie den case hat mit drei microservices. Microservice A muss calls machen an B und C. Der call an B ist erfolgreich, dann kommt danach der call an C, der ist aber nicht erfolgreich. Dann ist jetzt die Frage, wie handeled man das

jetzt dann? Theoretisch müsste man dann wirklich auch noch Endpunkte im Service B hinzufügen, dass man die Operation rückgängig machen kann oder ähnliches. Also so etwas ist bei der microservices Architektur echt extremst aufwendig und man muss da sehr viele cases durchdenken.

I [35min 14s]: Das heißt, es geht auch viel in die Richtung von Transaktionskontrolle, dass ihr euch sicher sein müsst, dass wenn es eine verteile Transaktion ist, dass die quasi nur als Ganzes durchgeführt werden kann oder dann es ein rollback geben muss.

A [35min 33s]: Exakt. Und das gestaltet sich bei einem Monolith um einiges einfacher, würde ich ein mal behaupten.

I [35min 40s]: Okay. Hast du noch weitere Themen, die du gerne loswerden würdest?

A [35min 48s]: Aktuell fällt mir nichts weiteres ein.

I [35min 53s]: Gut. Dann bedanke ich mich sehr für das Interview. Das wird meiner Arbeit sicherlich sehr weiterhelfen. Ich beende damit auch die Aufnahme. Du kannst noch im Zoom-Raum bleiben, nur der offizielle Teil ist dann hiermit beendet. Vielen Dank.

A [36min 17s]: Alles klar und danke für die interessanten Fragen.

B.2 Interview B

I [00min 00s]: Vielen Dank für die Möglichkeit, das Interview jetzt heute zu führen. Vorab: Bist du damit okay, dass das Audio aufgenommen wird und im Rahmen meiner Masterarbeit verarbeitet wird?

B [00min 15s]: Natürlich.

I [00min 18s]: Sehr gut. Dann würde ich auch gleich mit der ersten Frage starten. Würdest du dich kurz vorstellen, was deine Rolle, deine Aufgabe ist in deiner Firma?

B [00min 30s]: Ich bin Aron Metzig und ich bin jetzt mittlerweile Senior Engineer bei flexperto. Und da hauptsächlich full-stack Zeug, also alles von frontend bis backend, aber auch Plattform und ein bisschen deployment.

I [00min 46s]: Wunderbar, da passt du ja perfekt in die Zielgruppe. Wo benötigt ihr Interoperabilität, also habt ihr richtige Microservices oder eher größere Services, die miteinander dann trotzdem reden müssen über ein Netzwerk?

B [01min 08s]: Bei uns gibt es eigentlich zwei Arten von Interoperabilität. Und zwar einmal die microservices im Sinne von micro-backends und micro-frontends. Bei den micro-backends haben wir mittlerweile echt ein paar Services, die mit der Zeit gewachsen sind. Ein legacy-Monolithen und ein paar modernere Node-backends die mit Moleculer und Express laufen. Bei den frontends handelt es sich eigentlich hauptsächlich um React Apps.

I [01min 41s]: Wie viele Teams sind ja ungefähr involved, also von jeder Größe?

B [01min 49s]: Es sind aktuell ungefähr zwei Teams aktuell à la sechs Leuten, die daran Arbeiten und in den verschiedenen Sprints, die da immer versuchen zu implementieren. Und auch ein bisschen verschiedene Zuständigkeiten und Spezialisierungen auf andere Services haben. Zweieinhalb, das Plattform-Team kann man auch noch zählen, die helfen manchmal aus.

I [02min 09s]: Okay, perfekt. Sehr gut. Weil du gerade von Sprints geredet hast: Deployed ihr dann auch regelmäßig nach dieses Sprints oder ist das enkoppelt von diesen?

B [02min 21s]: Das ist schon relativ entkoppelt. Meistens schon, aber es gibt immer dann doch immer mal wieder einen hotfix oder irgendwas rein und dann wir auch mal zwischendeployed. Meistens ist schon das Ziel von jedem Sprint auch ein Inkrement zu generieren, das auch deployed wird natürlich.

I [02min 38s]: Okay, ja. Da würde ich dann später auch noch einmal zurückkommen. Dann würde ich jetzt gleich weitermachen mit dem higher-level Fragen. Wie tauscht ihr die Typinformationen aus zwischen dem Monolithen und auch zwischen den kleineren moderneren Apps wie du sie genannt hast?

Das heißt, benutzt ihr da eine API description language wie OpenAPI oder definiert ihr die Typen manuell?

B [03min 05s]: Das läuft eigentlich alles über OpenAPI.

I [03min 10s]: WSDL oder ähnliches verwendet ihr auch nicht?

B [03min 14s]: Nein, es ist komplett OpenAPI ausgeneriert.

I [03min 19s]: Wunderbar. Wie automatisiert ist dann euer Prozess, die Typinformation zu erstellen? Das heißt, habt ihr zum Beispiel im backend dann Generatoren? Ich kenne es zum Beispiel von NestJS, dass ich dann eine Controllerdefinition und mir dann die OpenAPI-Definition erstellen lasse. Sieht das bei euch auch so aus?

B [03min 50s]: Das ist leider nicht ganz so elegant leider, weil es ein PHP-Monolith ist mit einem eigenem sehr alten, eher in russischsprachigen Kreisen populären Framworks Namens Yii. Und da geht das nicht mit Annotations und allem womit man das herausgenerieren kann. Es gibt eine große spec.yml, wo man manuell immer alles hineinschreibt. Und dann gibt es einen von uns selber geschriebenen Swagger-parser, der diese yml bei jedem request quasi noch einmal abfrägt, ob das passt. Zum Beispiel, wenn man besondere Rollen hat, die auf den endpoint zugreifen können, wird auch in dieser spec.yml definiert und der selbstgeschriebene Filter filtert die requests dann eben heraus.

I [04min 31s]: Das heißt, das mapping zu den richtigen endpoints findet dann auch zur Laufzeit statt.

B [04min 38s]: Genau. Es gibt schon routes, die manuell gemacht werden. Da ziemlich viel auf white-labelling gesetzt wird, also dass andere routen bei anderen routen anders heißen - deutsche Namen, englische Namen. Oder die haben einfach auch andere spezielle Wünsche. Es ist relativ wichtig, dass vor allem das routing relativ flexibel geblieben ist.

I [05min 00s]: Ah ja. Das ist ein sehr interessanter Punkt dann auch. Das heißt, ihr habt dann trotzdem eine yml-file, in der mehrere unterschiedliche Routen dann zum gleichen endpoint dann auch laufen?

B [05min 18s]: Ja unter umständen schon. Die eigentliche yml-file, mit der eigentlich gearbeitet wird, die proxied dann auch von den anderen Services die yml-files hinein. Also es gibt dann quasi eine generierte yml, also es gibt dann auf live und release eine große Swagger-Definition, die man sich ansehen und auch curl-en kann. Die hat die Information von allen Services. Und jeder Service verwaltet seine Endpunkte in einer mittelgroßen yml-file.

I [05min 52s]: Kann man sich die einzelnen kleinen yml files auch ziehen oder muss man immer mit der großen arbeiten?

B [06min 01s]: Das kommt ganz darauf an was man vor hat damit. Wenn du einen kompletten Client generieren möchtest, also wo alle Parameter vordefiniert sind, das ist alles schon getypt und so weiter, das ist manchmal von der developer experience in zwei Schritte aufgeteilt. Und zwar, dass man sich einmal den Endpunkt abändern muss. Dann muss man das pushen, dann wird ein neuer Client generiert mit der neuen spec.yml. Und erst mit diesem neuen package, das generiert wird, kann sich das frontend den Client herausgenerieren. Jedenfalls für die release environment. Natürlich kann jeder Entwickler selbst sich schon so einmal so einen eigenen *(inaudible)*. Das ist so ein kleiner Zwischenschritt, damit die dev-XP nicht komplett leidet. Das ist auch ganz gut durchautomatiert mit kleineren Skripten und so weiter. So sorgen wir quasi dafür, dass die Endpunkte immer statisch getypt und auch aktuell sind. Natürlich, wenn man einen Endpunkt dann noch anpasst, also einen vorab bestehenden ändert, das passiert das eigentlich kaum. Wenn die APIs auch schon relativ lange bestehen und auch von Kunden recht intensiv genutzt werden, dann kann man die nicht einfach abändern. Das wird dann über header gelöst, dass die auch Rückwärtskompatibel bleiben, dass die alte API deprecated wurde.

I [07min 37s]: Das heißt, versioning läuft bei euch über einfache header.

B [07min 42s]: Ja. Wobei es glaube ich einmal vorgekommen ist seit dem ich da bin, das passiert wirklich super super selten.

I [07min 50s]: Das heißt, ihr baut eigentlich nur neue Routen dazu als dass andere oder ältere veraltet sind oder geändert werden.

B [08min 03s]: Ja. So ein bisschen „never break your userland“. Wie gesagt, die API wird recht intensiv vom Kunden genutzt und das kann man denen nicht einfach zumuten, das einfach zu ändern. Und dann wird schon eher darauf geschaut, dass wir unsere backend-Architektur ein bisschen verbiegen, als dass die ihre Integration noch einmal neu schreiben müssen.

I [08min 22s]: Okay. Redet ihr nur mit JSON-basiertem REST miteinander oder verwendet ihr auch GraphQL?

B [08min 38s]: Nein, wir verwenden eigentlich nur JSON REST. Wir haben auch ein bisschen RabbitMQ asynchrone Kommunikation, da haben wir quasi die Schema-Definition mehr oder weniger hart gecodedet um das so zu halten, wie es soll. Da habe ich auch letztens gesehen, dass es AsyncAPI gibt als Definitionssprache. Das hätte ich vielleicht auch die Tage mal versucht zu integrieren um zu schauen, ob das funktioniert. Und dann gibts eine envelope mit dem Event, mit User, mit dem allem, und der Payload ist dann variabel je nach RabbitMQ route oder topic.

I [09min 28s]: Okay, das heißt hauptsächlich JSON, REST, aber jetzt auch Rab-

bitMQ.

B [09min 34s]: Ja, auch ein bisschen RabbitMQ.

I [09min 38s]: Ich weiß nicht, ob du es gerade gesagt hast oder ich es nicht mitbekommen habe: Läuft das auch über OpenAPI, die Definitionen?

B [09min 46s]: Von RabbitMQ?

I [09min 48s]: Ja. Ich kenne mich damit noch gar nicht aus, sorry.

B [09min 51s]: Da gibt es eben mittlerweile AsyncAPI, die haben so eine Definitionssprache wie OpenAPI. Aber das ist noch releativ neu. Das hätte ich mir über die Weihnachzeit mal angeschaut um zu sehen, ob das was ist. Und dann eben proposed oder nicht, ob wir das einführen wollen. Wir haben da eine sehr breite Spezifikation, in die man alles reinschreiben kann, genommen. Auch versioning mit hineingebaut, natürlich über den topicname und da ist es eher im Code überarbeitet.

I [10min 35s]: Okay. Das wird wahrscheinlich dann eine halbautomatische Lösung werden.

B [10min 45s]: Ja.

I [10min 47s]: Bist du zufrieden, wie es aktuell läuft mit der Interoperabilität und auch gerade mit dem kreieren von Typinformationen und dem schreiben von Typinformationen?

B [10min 57s]: Der Generator ist noch relativ neu. Ich glaube ein halbes/dreiviertel Jahr. Da recht viel Software natürlich davor geschrieben wurde, muss man ab und zu wenn man etwas anfässt immer wieder auf den generierten Code aufpassen, das manuelle fetch oder Axios call umsetzen. Das ist manchmal ein bisschen lästig, aber natürlich auch eine wichtige Aufgabe, dass das langfristig wartbar bleibt und dass sich die Endpunkte noch ändern können. Und die API-Generatoren sind manchmal wirklich nicht gut, muss man ehrlich sagen. Vor allem enums und so funktionieren einfach nicht richtig. Das ist schon sehr schade.

I [11min 42s]: Also auch bei den OpenAPI, über die wir am Anfang gesprochen haben?

B [11min 48s]: Ja, vor allem bei OpenAPI. Vom Workflow finde ich den jetzigen Workflow eigentlich ganz gut, dass die Stabilität vom backend an erster Stelle steht und die Interoperabilität und dass die dev-XP ein bisschen hinten ansteht, aber jetzt auch nicht komplett schlecht ist. Also das ist einfach ein kleiner Zwischenschritt, das Opfer, dass man dann gerne in Kauf nimmt. Und wenn man jemanden anlernt, dann zeigt man das den Leuten ein mal und dann verstehen die, wie das funktioniert.

I [12min 22s]: Das klingt ja schon dann ganz gut.

B [12min 26s]: Das ist dankbar. Das ist ganz dankbar.

I [12min 31s]: Wenn ihr jetzt eine neue Version vom backend oder von den verschiedenen microservices veröffentlicht... Wenn sich ändernde Service deployed werden, deployed ihr dann immer jeden microservice oder auch den Monolithen einzeln, oder schaut ihr dass ihr nur die veränderten neu deployed oder deployed ihr dann immer alle neu zusammen?

B [13min 05s]: Man muss beim deployment ein bisschen aufpassen. Durch das ganze white-labelling müssen wir den Monolithen pro tenant deployen. Das ist dann für jeden Kunden ein deployment sowieso. Die Services werden dann aber nur deployed, wenn es nötig ist. Das wird auch manuell gemacht tatsächlich. Da gibt es keine Pipeline, da es wichtige Kundendaten sind, Versicherungsdaten und so. Das ist natürlich so weit automatisiert wie es geht, aber am Ende schaut immer noch ein Mensch darüber, ob es jetzt wirklich so funktioniert oder nicht und kann im Zweifel noch ein roll-back machen. Und deswegen werden Services deployed, die wichtig sind und die wirklich verändert werden.

I [13min 53s]: Du hast ja gesagt der Monolith ist für jeden Kunden quasi.

B [14min 01s]: Ja.

I [14min 04s]: Und die microservices dann, da gibt es auch verschiedene Instanzen, aber die sind geteilt quasi?

B [14min 11s]: Das sind dann shared-services ja. Der Monolith ist noch nicht geshared, da wird hingearbeitet für cloud-readyness, aber so einen Monolithen umzubauen, das dauert auch ein bisschen.

I [14min 22s]: Das heißt es gibt auch verschiedene oder mehrere Instanzen, die parallel laufen von den microservices?

B [14min 29s]: Ja. Da gibt es ein paar Sachen, weil ein paar tenants einfach nicht wollen, dass sie in der Cloud laufen. Die wollen ihre eigene baremetal Instanz und zahlen auch dafür.

I [14min 44s]: Von der Größenordnung her mit dem Monolithen und den kleineren microservices - in welcher Richtung sind wir da ungefähr?

B [14min 54s]: Meinst du jetzt lines of codes oder Anzahl an Services?

I [14min 57s]: Anzahl an Services. Wenn einer fünffach existiert, dann ist es trotzdem nur ein logischer quasi.

B [15min 04s]: *(takes a look in internal documents – details censored)* Effektiv habe ich jetzt an 6-7 Services mitgearbeitet und ein zwei microfrontends

mit dabei noch. Microfrontends - da arbeiten wir hauptsächlich mit SDKs tatsächlich, die wir für die speziell noch einmal erstellen. Vor allem so etwas wie socket.io, da arbeiten wir ein SKD heraus, das der Client dann einfach aufrufen muss. Da wurde geschaut, dass die Funktionen möglichst generell sind. Also es gibt nicht irgendwie eine große Klasse, die alles für einen macht. Das rendering ist natürlich komplett dann vom Monolithen in dem Falle, also wenn ein Dialog auf geht. Ein gutes Beispiel ist wenn der Kunde einen Experten anruft, dann öffnet sich beim Experten natürlich ein kleiner Screen mit einem decline und einem accept button. Das rendering von den buttons übernimmt der Monolith. Das mapping was auf den buttons passiert logischerweise auch. Aber der eigentlich call wird dann vom SDK ausgeführt. Das SKD ist logischerweise wird ja vom microservices geserved und das ist dann immer die aktuellste Definition von dem Service. Vor allem bei asynchronen calls über socket.io ist das schon sehr sehr praktisch wenn man sich da überhaupt nicht mehr darum kümmern muss wie das geholt wird.

I [16min 53s]: Okay. Das heißt ihr habt so um die sieben Services hast du gesagt?

B [17min 07s]: Es sind 7 bis 8 Services würde ich spontan sagen.

I [17min 15s]: Dann haben wir ja mit den SDKs schon in den Code ein bisschen hineingeschaut. Das würde ich jetzt auch gleich aufgreifen. Das heißt, ihr habt zwei Möglichkeiten. Dass ihr zum einen eine library oder eine SDK veröffentlicht, die ein Client dann benutzen kann. Aber ihr habt auch die Möglichkeit, aus dieser großen YML-file euch Clients zu erzeugen, richtig?

B [17min 41s]: Genau. Für REST Zeug wird eigentlich hauptsächlich das verwendet. Es gibt so ein zwei Beispiele, bei denen der Generator einfach zu schlecht ist für OpenAPI, um das wirklich vollständig zu nutzen. Da Nutzen wir dann auch SDKs, wenn es wirklich komplexe Typinformationen sind. Ein Beispiel wäre zum Beispiel ein Web-Hook-Service. Was in desem Web-Hook dann steht, das ist ja unfassbar Variabel und dass kann man nicht mit Swagger oder OpenAPI wirklich abfangen. Das wird dann über das SDK dann eben geregelt. Wenn man jetzt ein Meeting enden will, dann ist das der Typ der hinein kommt und so weiter. Anders ist es schwierig abzutypen und eben für die frontends. Damit die frontends mit den Services wirklich immer aktuell sind und auch gut zusammenspielen, gibt es eben eigentlich immer eben diese SDKs, die komplett entkoppelt sind von der eigentlichen Implementierung. Und das ist meistens dann wirklich socket.io Kommunikation und kein REST Zeug. REST calls werden eigentlich immer im frontend über den OpenAPI Generator verwendet und domänenspeziferisches Zeug wie socket.io dann eben über SDKs.

I [19min 12s]: Okay. Verwendet ihr auch Remote Procedure Calls?

B [19min 25s]: Was ist das genau?

I [19min 29s]: Du rufst eine Methode auf, die so aussieht, als wäre sie eine lokale Methode. Vor allem in der JAVA-Welt recht gut implementiert.

B [19min 43s]: Nein.

I [19min 46s]: Sehr gut. Wenn es dir nichts sagt wahrscheinlich auch nicht. Testet ihr auch die Interoperabilität mit Ende-zu-Ende-Tests oder teilweise auch mit mocking oder stubbing?

B [19min 57s]: Es gibt mocking und stubbing. Beides, ja. Wir haben ein end-to-end test framework und da wird auch ganz gut gemockt.

I [20min 09s]: Dann auch auf Basis von der Typdefinition aus den yml bzw. SDK?

B [20min 16s]: Ja.

I [20min 18s]: Habt ihr dann eine Möglichkeit, das automatisch zu generieren oder schreibt ihr die Tests manuell aufgrund der Definition?

B [20min 29s]: Wir schreiben die Tests manuell aufgrund der Definition. Das ist dann ein Unit-Tests, ein gemockter. Da wird eher auf die Funktionalität getestet.

I [20min 47s]: Wünscht ihr euch, dass es da automatisiertere Tests gibt, auch quasi wenn du jetzt zum Beispiel wenn du jetzt ein HTTP-request mocken willst, dass es automatisch einen Stub, also eine Beispielimplementierung zum Beispiel, simuliert?

B [21min 07s]: Das wäre schon manchmal ganz nett auf jeden Fall. Das ist jetzt kein muss aber es würde schon manchmal Arbeit abnehmen, ja.

I [21min 14s]: Ja, okay. Ist notiert, vielleicht kommt es ja noch in der Arbeit dran. Bist du zufrieden mit dem Level an Code, was OpenAPI kreiert oder würdest du dir mehr oder weniger wünschen? Also z.B. willst du, dass OpenAPI dir in React direkt einen Service erstellt oder willst du da einfach nur die Klassen haben und die Typinformation und vielleicht auch noch die Endpoints haben?

B [21min 50s]: Ich bin eigentlich ganz zufrieden damit. Es ist wirklich einfach nur ein gewrappter fetch. Das macht es dann ganz angenehm, je nachdem in welchem Framework man arbeitet. Ich kann den gleichen call genau gleich in einem Moleculer backend oder einem React frontend verwenden, ohne mir noch einmal Gedanken darüber zu machen, wie das funktioniert. Und da der meiste Code eh schon legacy ist und man da Zeug eher anpassen muss als wirklich Zeug neu schreibt, ist das viel einfacher weil man sich den Kontext einfach spart, um da jetzt irgendwie ein neues Paradigma

einzuführen, dass man jetzt keine Saga mehr verwendet sondern irgendwie da nochmal einen Services für ein React frontend.

I [22min 44s]: Du hast vorhin schon etwas angesprochen mit validations. Macht ihr das auch automatisch mit den OpenAPI-Definitionen oder ist das noch einmal zusätzlich drauf?

B [23min 06s]: Wie meinst du genau?

I [23min 07s]: Also zum Beispiel wenn ich jetzt vom Client eine Anfrage bekomme und einen bestimmten body, dass der bestimmt formattiert ist, also ein Feld mit dem Nutzernamen enthält, der ist vom Datentyp String zum Beispiel und darf maximal 20 Zeilen lang sein. Ist das auch was, was ihr mit OpenAPI definiert und was dann auch automatisch dann generiert wird für euren Service?

B [23min 31s]: Da ist eine selbstgeschriebene Klasse, die dann über die Parameter drüber geht und eben schaut, ob das passt. Und eben wie gesagt auch die role-based-authentification kontrolliert, das passiert auch mittlerweile damit.

I [23min 47s]: Euer parser, gibt es den dann nur für den PHP-Monolithen oder auch für die kleineren Node-Apps?

B [23min 56s]: Ich glaube die Node-Apps machen das irgendwie automatischer. Ich glaube das macht das framework irgendwie selbst. Da habe ich mir den parser noch nicht angeschaut auf jeden Fall. Da hat es einmal geklemmt, also gehe ich einmal davon aus, dass die Node Dinger das auch machen, aber habe da immer gepasst.

I [24min 22s]: Okay. Ihr habt auch nur zwei Sprachen, also ihr habt nur PHP und NodeJS bzw. dann TypeScript oder JavaScript im Frontend.

B [24min 38s]: Ja genau.

I [24min 40s]: Okay gut. Dass die richtigen Endpoints benutzt werden, stellt ihr dann wahrscheinlich auch sicher mit den SDKs und mit der YML, die ihr veröffentlicht.

B [24min 56s]: Genau. Und peer-reviews.

I [25min 00s]: Wie angepasst sind eure Endpoints an den consumer? Habt ihr relativ generische endpoints, wo der consumer dann quasi nur das mitschickt, was der eigentliche Kunde dann eigentlich braucht, oder habt ihr da für jeden Kunden irgendwie einen speziell angepassten Endpoint?

B [25min 20s]: Das ist schon relativ generisch. Ein paar endpoints sind jetzt nicht für alle Kunden freigeschalten. Das ist dann wirklich so ein Ding, aber eigentlich sind die Endpunkte von der response schon generisch. Es

ist halt was man erwartet, was man bekommt, wenn man fragt, wie das aktuelle Meeting aussieht. Hat man eine Liste von participants und noch Metainformationen.

I [25min 45s]: Und für das Speichern und Neusetzen von Daten: Wie fix seid ihr da auf euer JSON-Schema eingeschränkt quasi? Jetzt nicht unbedingt im negativen Sinne. Es gibt postel's law zum Beispiel wenn dir das etwas sagt. Dass du die Information, die du schickst, relativ flexibel schicken kannst. Hauptsache, die Information ist irgendwie im JSON drin, egal wie eingerückt, in welchem Unterobjekten zum Beispiel die Information ist. Hauptsache, sie ist drin. Oder ist es dann wirklich ein Schema, was ihr erwartet, was der Kunde dann schickt?

B [26min 31s]: Das ist schon sehr schemabasiert. Das ist ein fixes Schema.

I [26min 36s]: Das andere ist euch dann wahrscheinlich auch zu viel overhead, zu viel Entwicklungskosten, die ihr nicht bereit seid zu machen wahrscheinlich.

B [26min 47s]: Ja. Das ist schwierig zu verargumentieren glaube ich.

I [26min 51s]: Ja, okay. Ich denke einmal es gibt da auch noch keinen Grund dazu, so eine Flexibilität da einzubauen.

B [26min 03s]: Nein. Im Zweifel haben wir ja eine öffentliche Swagger-Dokumentation, in der die Payloadinformation ja auch vorgeschrieben ist. Und dann darf der Kunde gerne raus-copy-pasten.

I [27min 13s]: Wie schränken eure Sprachen und eure Frameworks die Lösung ein? Du hast ja schon gesagt, ihr musstet für euren Monolithen einen selbstgeschrieben parser erstellen. Mit NodeJS läuft das ein bisschen automatischer.

B [27min 40s]: Ja. PHP ist da echt ein bisschen nervig, weil das über die spec.yml läuft. Wo man dann erst den Endpunkt schreiben muss, die spec.yml anpassen muss. Und dann auch ein bisschen hofft, dass man da alles richtig gemacht hat, bevor man das zum ersten Mal aufruft. Wobei fairerweise ist es aber in den Node-backends ähnlich. Wobei dann einfach der gleiche Ansatz übernommen wurde. Also das framwork Moleculer hat jetzt auch nicht irgendwelche Annotations, wo es sagt das ist ein REST-Endpunkt und so und es wird einfach an eine Schicht delegiert. Da ist der Architekturstil auch einfach ein anderer, dass alles sehr zentral gelöst ist.

I [28min 28s]: Verwendet ihr code-sharing? Zum Beispiel, wenn ihr einen React-Monolithen und ein NodeJS backend? Oder läuft das dann alles komplett über OpenAPI?

B [28min 41s]: Das läuft eigentlich alles über OpenAPI oder die SDKs dann.

I [28min 53s]: Das heißt, ihr erstellt Code, den ihr dann veröffentlicht. Aber der selbe Code wird nicht unbedingt in euerem backend dann verwendet.

B [29min 03s]: Ja. Wir verwenden schon die Typinformation von unserem backend. Im weitesten Sinne ist es schon das gleiche. So ein core-Modul natürlich und dann vom core ein SDK und dann die eigentlich App dann irgendwie rausgehauen. Das wird schon gemacht, aber das ist weniger Funktionalität und wirklich mit Typinformation.

I [29min 29s]: Welche Repository-Struktur habt ihr? Also verwendet ihr ein Monorepo oder für jedes Projekt/für jeden Service ein eigenes Repo?

B [29min 43s]: Für jeden Service eigentlich ein Repo.

I [29min 45s]: Bist du zufrieden mit der Lösung oder schränkt dich das irgendwie ein oder gibt das Multi-Repo euch irgendwelchen besonderen Vorteile?

B [29min 57s]: Wenn das Tooling passt, bin ich schon ziemlich Fan von Multi-Repo muss ich sagen. Wenn das Tooling nicht passt, ist es die absolute Hölle. Aber das Tooling ist sehr gut tatsächlich und deswegen bin ich eigentlich recht einverstanden damit. PRs überschneiden sich jetzt auch nicht so häufig wie man meint. Und dann hat man mal zwei drei PRs, das kommt schon mal vor in zwei drei Projekten, aber ich mache ja auch nicht das deployment. Und dann muss die Person, die deployed schauen, dass wirklich die Services alles abgeändert werden. Das ist aber auch automatisiert. Also effektiv, wenn das Tooling passt, finde ich Multi-Repo eigentlich am angenehmsten.

I [30min 45s]: Ich schaue jetzt noch einmal ob ich alles abgearbeitet habe. Das heißt ihr verwendet ein Multi-Repo, ihr habt zwei-einhalb Teams hast du gesagt. Wenn ihr deployed, tut ihr die geänderten quasi veröffentlichen, wobei ihr trotzdem immer die backwards-compatibility drin habt.

B [31min 21s]: Rollbacks sind wichtig.

I [31min 25s]: Ja, gegebenfalls Rollbacks. Deployment hast du gesagt geht mit Sprints einher oder öfter oder seltener, aber in der Regel mit Sprints.

B [31min 39s]: Ja, also mit Sprints, aber häufiger.

I [31min 43s]: Okay. Also mindestens mit Sprints. Wie sind die Sprints bei euch? Sind das wöchentliche?

B [31min 49s]: Zwei Wochen sind das aktuell.

I [31min 52s]: Alle zwei Wochen, okay. Ihr verwendet hauptsächlich zwei languages mit JavaScript/TypeScript und PHP. Ob man die jetzt als drei zählt.

B [32min 11s]: Ja, manche Sachen sind auch Misch-Masch, muss man ehrlich sagen. Deswegen passt es schon gut rein dass man es als zwei zählt.

I [32min 23s]: Gut, dann wäre ich jetzt mit dem Hauptteil fertig. Gibt es noch weitere Probleme, die ihr in der Firma habt, im Kontext von Interoperabilität oder weitere Möglichkeiten oder Tools die ihr euch wünscht, die euer Leben vereinfachen?

B [32min 45s]: Tatsächlich wird geschaut, dass möglichst wenige Service-Service-Calls passieren und eigentlich die meistens Events vom User dann getriggert werden. Deswegen ist es eigentlich relativ angenehm, so mit dem Paradigma zu entwickeln. Das führt manchmal auch zu bugs, wenn der User nicht so reagiert, wie man es erwartet oder wenn die Verbindung manchmal abbricht. Aber da hat man irgendwann auch ein Gefühl dafür wann das möglich ist und wann eben nicht. Und deswegen finde ich es eigentlich nicht, nein.

I [33min 23s]: Okay, das heißt du bist sehr zufrieden, das ist ja schön. Dann würde ich das offizielle Interview damit jetzt auch beenden. Bleibe kurz noch in Zoom bitte drin, ich beende kurz die Aufnahme.

B.3 Interview C

I [00min 00s]: Hallo. Vielen Dank für das Interview. Ich habe jetzt die Aufnahme gestartet. Vorab möchte ich noch einmal kurz zum Data-handling Fragen: Seid ihr damit einverstanden, dass ich euren Ton aufzeichne und dann auch im Rahmen der Masterarbeit verarbeite?

C1 [00min 24s]: Ja.

C2 [00min 25s]: Ja.

I [00min 26s]: Sehr gut. Könntet ihr euch kurz einmal vorstellen – zu euch als Person, was eure Aufgaben und responsibilities in der Firma sind?

C1 [00min 42s]: Dann fange ich einfach mal an. Tom mein Name. Ich bin full-stack-developer für eine IoT-Platform. Wir haben broker im Einsatz, wir haben viele kleinteilige microservices im Einsatz, wir haben APIs im Einsatz und ein Frontend. Ich kümmere mich quasi um jeden Teil in dieser Architektur, auch um die Datenbanken quasi. Wir sind gerade auf dem Weg in Richtung Cloud, deswegen kommen da auch ein Haufen neue Technologien und neue Programmiersprachen mit ins Rennen. Wo es dann eben auch interessant ist zu sagen, wie kriegen wir denn für alternative Programmiersprachen auch diese syntaktische Interoperabilität her.

C2 [01min 34s]: Dann mach ich weiter. Ich bin Constantin, bin backend-Entwickler und mein Aufgabengebiet ist primär in der Cloud. Und zwar baue ich eine Infrastruktur auf, die Daten von einem IoT-Gateway abholt, validiert, transformiert und dann normalisiert in verschiedene persistenten storages ablegt. Das Ziel ist, dass es egal ist, ob ich eine relationale Datenbank habe, einen file-storage oder vielleicht einen key-value-storage. Dass ich alles möglichst effizient verarbeiten kann auf der ganzen Pipeline.

I [02min 20s]: Super. Das klingt schon einmal von den Microservice und von allem natürlich sehr interessant. Könnt ihr mir kurz noch einmal sagen, wo ihr die syntaktische Interoperabilität hauptsächlich braucht? Gerade was so Größenordnungen von Services angeht. Wie viele Service habt ihr? Habt ihr eher Microservices im klassischen Sinne oder auch größere Monolithen, die aber trotzdem über ein Netzwerk kommunizieren?

C1 [02min 54s]: Wir haben beides würde ich jetzt einmal sagen. Wir sammeln ja viele Daten von den Maschinen, die wir ausliefern. Das sind alles sehr feingeschnittene microservices. Die senden ihre Daten an den Broker und da geht es dann quasi also schon los mit den topics, die wir da haben, was für Daten da reinkommen und wie wir sie dann auf der anderen Seite in unserem Containern dann wieder abholen und speichern quasi. Andererseits haben wir natürlich einen Haufen APIs, die wir zur Verfügung stellen für

backup-handling, für condition-monitoring. Dadurch, dass wir sogesehen auch ein data-lab haben, die momentan noch direkt auf die Datenbanken zugreifen, wo ich aber der Meinung bin, dieser Datenbankzugriff sollte ihnen weggenommen werden – sie sollten mit der API arbeiten. Da geht es jetzt dann eben los zu sagen: Wie können wir ihnen diese API-Definitionen in ihren Programmiersprachen möglichst einfach zur Verfügung stellen?

I [04min 06s]: Ja, das klingt doch perfekt abgeschnitten auf die Masterarbeit. Wie viele Teams arbeiten ungefähr bei euch mit? Also jetzt keine genaue Anzahl, nur so eine Größenordnung reicht auch aus.

C1 [04min 29s]: Wir haben ein data-lab Team würde ich jetzt einmal sagen. Größenordnung fünf Leute einfach mal in den Raum geworfen. Wir haben das Cloud-Development-Team. Auch einfach mal so fünf, sechs Leute. Wir haben das On-Premise-Team mit zwei bis drei devs und wir haben ein Operations-Team, auch mit vier, fünf Leuten.

C2 [04min 57s]: Wobei man da vielleicht noch ergänzen muss, dass jetzt das nur unsere IoT-Plattform-Teams sind. Es gibt auch Teams außerhalb von unserem Kontext, die Subsysteme entwickeln und wir sammeln auch Daten von denen. Perspektivisch wird es immer mehr werden, dass wir von fremden Systemen, die eigentlich nichts mit unserer Teamstruktur zu tun haben, Daten einsammeln müssen.

I [05min 27s]: Okay. Das heißt zu eurem selber entwickelten Service kommen viele Clients dann auch zu, die ihr wahrscheinlich immer über die Jahre mal auswechselt und neue hinzukommen.

C1 [05min 43s]: Wir entwickeln ja unsere Maschinen immer weiter. Das heißt, es werden andere Geräte verwendet, das heißt, es werden andere Protokolle verwendet. Die neuesten Lieferanten regeln da ihre eigenen Applikationen, für die sie natürlich auch cache haben wollen, mit ein. Die dann natürlich ihr eigenes proprietäres Datenformat dann auch verwenden. Das kommt schon dazu. Zusätzlich kommt hinzu, dass wir natürlich auch auf der Maschinenebene, von der wir – was der Constantin meinte – abgetrennt sind, auch Entwicklungen haben, die Daten für uns bereitstellen. Wo wir nicht in der Hand haben, wie sie gesendet werden.

C2 [06min 24s]: Tendenziell werden die Services eher mehr als weniger. Also es kommen mehr Services dazu als abgeschaltet werden, weil wir einen sehr lifecycle von den Maschinen haben und die Software dementsprechend eigentlich auch einen langen lifecycle hat.

C1 [06min 39s]: Ja.

I [06min 40s]: Das heißt, ihr seid auch was die Services angeht weit im zweistelligen Bereich, oder im zweistelligen Bereich auf jeden Fall.

C1 [06min 50s]: Im zweistelligen Bereich, sagen wir es mal so, ja.

I [06min 53s]: Okay. Wie sieht es bei euch aktuell aus, wie ihr die Typinformationen austauscht? Schreibt ihr die manuell, also wenn ihr verschiedene Sprachen habt, dass ihr manuell in euere Java-Klasse schreibt, in eure TypeScript-Klasse schreibt, oder habt ihr da Generatoren, Standards wie OpenAPI, WSDL, oder ähnliches?

C1 [07min 25s]: Ich schreibe es momentan in den größten Zügen selbst. Das heißt, manchmal kopiere ich mir. Ich arbeite mit data-transfer-objects kann so sagen, die die API bereitstellt. Die kann ich mir so gesehen aus der API selbst auch den Code kopieren und dann quasi in dem Java-Projekt, in dem ich es brauch, mit übernehmen. Beziehungsweise dann auch so zusammenschrumpfen, dass ich nur den Teil habe, der quasi für mich auch interessant ist. In Python mache ich es bis jetzt so, dass mir der Typ eigentlich egal ist, weil ich den Aufbau über den dict-key heraushole. Könnte man noch besser machen, in dem man sie automatisiert bereitstellt, dass es immer der API folgt, die ich habe.

I [08min 27s]: Und wenn ihr mit externen APIs oder Maschinen zum Beispiel redet, woher bekommt ihr da die Typinformation, wenn ihr nicht direkt den Code habt?

C1 [08min 42s]: Ich muss gerade überlegen, wann wir wirklich mit einer komplett externen API reden. Das einzige Beispiel, das mit gerade einfällt, das aber noch nicht produktiv ist, ist die Zusammenarbeit mit einem Service-Center, wo wir Informationen über *(inaudible)*-condition-monitoring austauschen sollen. Da erfolgt der Austausch dann über eine OpenAPI-Spezifikation.

I [09min 13s]: Okay.

C2 [09min 14s]: Bei mir ist es einmal vorgekommen. Wir haben noch einen Online-Shop, der wird extern entwickelt. Und da gibt es eine API, wo man gewisse Sachen abgreifen kann. Du hast einen endpoint gekriegt und das war Trial-And-Error. Du hast quasi Parameter, wie du Daten abholst, aber das Format war nicht gegeben und das war auch nicht so, dass man aus den Variablennamen den größeren Kontext herausziehen konnte.

I [09min 52s]: Okay. Das hat bestimmt viel Spaß gemacht. Die OpenAPI-Definition, die du gerade angesprochen hast Thomas, war das webbasiert oder konntest du das einlesen, also in YML oder in JSON?

C1 [10min 11s]: Sie haben mir quasi eine Datei geliefert. Und ich tue die dann immer mit Postman meistens einlesen.

I [10min 22s]: Und dann quasi einmal versuchen, ein Beispiel quasi hinschicken ...

C1 [10min 31s]: Genau.

I [10min 31s]: ... an den Server und dann kommt ein Beispiel zurück und das baust du dann ein.

C1 [10min 35s]: Genau, das baue ich dann so nach.

I [10min 38s]: Okay. Das heißt, der ganze Prozess ist sehr manuell, ...

C1 [10min 45s]: *(Agreeing Hm-hm)*

I [10min 45s]: ... ohne viel Code-Generation.

C1 [10min 51s]: Zu sagen ist auch noch, weil ich es hier lese: vielleicht ist der Fall bei uns auch sehr besonders, weil wir früher ein monorepo hatten und jetzt eben auf Multi-Repo gewechselt sind.

I [11min 08s]: Ja, gut. Das schreibe ich auf, dazu komme ich gleich noch einmal. Bevor wir dann dazu kommen, zum Code selber, noch ganz kurz: Wie sprecht ihr miteinander? Sprecht ihr über XML, über JSON, benutzt ihr GraphQL zum Beispiel oder wie sieht es bei euch aus?

C1 [11min 38s]: Wir haben denke ich einmal GraphQL und JSON quasi im Einsatz. Obwohl, diese Service-Cloud wollte auch irgendetwas besonderes. OData wollten die erst haben, sind dann aber trotzdem noch einmal umgeschwenkt auf eine normale JSON-Rest-API.

I [12min 11s]: Okay. Wie handelt ihr das dann mit GraphQL? Ich kenne mich mit GraphQL leider noch nicht so gut aus. Wie ich es verstanden habe, kann man ja auch angeben zum Beispiel, dass man nur bestimmte Teile von der Antwort bekommen will, also von dem Objekt zum Beispiel. Wie handelt ihr das da mit der Typsicherheit? Gerade was Java jetzt zum Beispiel angeht.

C1 [12min 47s]: Ich habe GraphQL mit Java noch nicht verwendet. Ich weiß nicht wie ihr das in der Cloud dann macht Constantin, ob da im Frontend dann irgendwie validiert wird, oder ob das egal ist.

C2 [12min 59s]: Das GraphQL-file ist ja im Endeffekt die Spezifikation. Da steht drin, welcher Datentyp das ist. Im frontend bist du eh mit einer dynamischen Sprache unterwegs. Da ist es kein Problem. Du musst da nur wissen, welcher Typ es ist am Ende, aber ich würde vermuten, es gibt auch schon einige Tools, die dir aus der GraphQL-Spezifikation dann deine Klassen generieren, also die gibt es auf jeden Fall. Der Vorteil ist an GraphQL, ist dass es vom Gefühl her immer ein bisschen besser gepflegt ist, weil dein GraphQL file mehr als die Spezifikation ist, sondern das ist irgendwie ein bisschen spezifischer als OpenAPI.

I [13min 53s]: Okay. Verwendet ihr dann da teilweise auch schon Generatoren oder schreibt ihr das trotzdem noch manuell? Also hauptsächlich.

C2 [14min 09s]: Ich war ja quasi immer die Seite, die quasi die Response schickt. Ich habe es tatsächlich immer selber geschrieben. Beziehungsweise ich habe eigentlich gar keine Typen verwendet. In Python gibt es die Möglichkeit, auf dictionaries zu arbeiten. Und dann gibt es typisierte dictionaries, das sind aber mehr oder weniger hints. Da gibt es eine statische Typanalyse. So habe ich das dann sichergestellt, aber im Endeffekt war ich auf den reinen dictionaries unterwegs.

I [14min 51s]: Okay. Was war bei euch zuerst – war zuerst der Server, der etwas bereitgestellt hat oder war zuerst der Client, der zum Beispiel zum Server gesagt hat, ich möchte einen endpoint, wo ich Benutzerinformationen herkriegen kann oder ähnliches? Also consumer-driven, oder stellt eher etwas bereit, was dann jeder Client annehmen muss quasi? Oder eine gute Mischung?

C2 [15min 29s]: Ich hätte jetzt auch tendenziell gesagt sowohl als auch. Also das Ziel im Moment ist es, dass wir generelle APIs zur Verfügung stellen und dann dem Nutzer zu sagen können: Ja schau mal, das gibt es einen endpoint, du kannst dir da ein gewisses set an Daten abholen. Aber in der Vergangenheit ist es eher so passiert dass man quasi gesagt hat, der Client ist gekommen und wollte irgendwelche Daten und dann hat man die bereitgestellt. Ich denke so kann man es formulieren, oder Thomas?

C1 [15min 59s]: Also ich würde sagen, bei uns ist es immer so: Unsere Business-Analysten wollen, dass eine bestimmte Funktion funktioniert. Eine Funktion muss ja nicht zwangsweise über einen endpoint abgebildet werden. Das heißt, ich muss mich dann hinhocken und schauen, wenn ich diese Informationen möchte, kann ich diese schon abgreifen? Wenn nicht, schaffe ich einen neuen endpoint dafür.

I [16min 24s]: Das heißt, ihr bekommt Forderungen von euren Chefs quasi.

C1 [16min 31s]: Naja es sind nicht wirklich Chefs, es sind Teamkollegen von uns, die sich Gedanken machen, wie sie ihr Produkt haben wollen. Was für eine Funktion sie im Produkt haben wollen. Und wir dann als technische Implementierer nenne ich es mal entscheiden dann wie APIs angepasst werden müssen, damit diese Funktion bestmöglich implementiert werden kann.

I [16min 56s]: Okay. Gerade das angepasst werden ist auch für später noch super interessant. Wobei, machen wir es gleich. Wie geht ihr mit Versionierung um, also wenn ihr zum Beispiel einen endpoint habt, bei dem backwards-Kompatibilität nicht unbedingt gegeben ist, erschafft ihr zum Beispiel neue endpoints für eine neue Funktion oder arbeitet ihr mit Versionierung? Wie

sieht es da aus? Vor allem, was jetzt breaking-changes angeht.

C1 [17min 32s]: Da wir nicht viele Externe haben, die unsere APIs verwenden, ist es eigentlich so: Die changes werden auf allen Clients geändert als auch auf der Server-Seite und dann gibt es eigentlich ein gemeinsames deployment.

I [17min 52s]: Okay. Wunderbar, das war auch gleich meine nächste Frage. Also alle Services, die sich geändert haben geployed ihr und wenn sich mal nur ein Server ändert, eine neue Funktion bereitgestellt wird, die noch nicht direkt verwendet wird, dann muss auch nur der eine deployed werden.

C1 [18min 16s]: Genau. In der Cloud ist es natürlich wieder ganz anders, da ja dann automatisch die Versionierung mit dabei ist, oder Constantin? V1, V2, wenn du ein Serverless-API-Gateway nimmst.

C2 [18min 26s]: Ja, also genau. In der Cloud schon. Wir schauen schon darauf, dass es dann Versionierung gibt. Aber ich zumindest versuche es so designen, dass man nicht alle paar Wochen eine neue Version braucht.

I [18min 48s]: Okay. Dann würde ich auch gleich zum nächsten Kapitel springen. Wir haben uns jetzt erst einmal mit oberflächlicheren Themen beschäftigt oder mit Zwischen-Microservice-Kommunikation. Und jetzt würde ich in die einzelnen Repositories reinspringen. Ihr habt schon gesagt, ihr seid weg von Monorepos gegangen und hin zu Multirepos.

C1 [19min 23s]: Da war ich noch gar nicht da als das passiert ist. Das ist glaube ich so ein halbes Jahr vor mir passiert. Also sie hatten ein ganz großes Repository, in dem alles, was mit der Plattform zu tun hatte drin war. Ich denke mal so konnten sie relativ einfach die Versionierung hinkriegen um auf einem Stand zu bleiben. Ich denke, es war der Sache geschuldet, dass die Entwicklungsgeschwindigkeit meiner Meinung nach höher ist, wenn wir mit Multirepos arbeiten.

I [20min 00s]: Das heißt wir wart effektiver, weil zum Beispiel nicht so viele Merge-Konflike kamen oder ähnliches.

C1 [20min 06s]: Genau.

I [20min 07s]: Das heißt, du bist auch zufrieden damit, jetzt ein Multirepo zu haben.

C1 [20min 19s]: Genau, damit bin ich sehr zufrieden.

I [20min 34s]: Ich habe es vorhin schon einmal kurz angesprochen, was die Sicherheit der einzelnen Datentypen oder Klassen angeht, zum Beispiel welche Attribute da drin sind. Stellt ihr auch zur Laufzeit sicher, welche Typen die haben? Also dass zum Beispiel das Feld wirklich ein Integer ist und kein double oder kein String.

C1 [21min 03s]: Dadurch dass wir in der Klasse angeben, dass wir einen Integer wollen, haben wir es ja eigentlich festgelegt.

C2 [21min 20s]: In der Cloud ist es so, also zumindest von GraphQL: GraphQL in der AWS-Cloud ist ein Polymanaged Service. AWS stellt einen Endpoint zur verfügung, da kann hingequeried werden. Und dieser Service stellt auch die Typsicherheit sicher, von beiden Seiten. Sowohl von der Client Seite, wenn der jetzt versucht einen Parameter int zu übergeben, der aber ein String ist, gibt es einen Fehler. Aber es gibt auch einen Fehler, wenn das backend irgendeinen falschen Datentyp schickt, dann gibt es auch einen Fehler, der an den Client durchgegeben wird. Die Typsicherheit ist in dieser Zwischenschicht, auf die ich als Entwickler gar keinen Einfluss hab. Aber das Ziel ist sichergestellt.

I [22min 14s]: Ja. Das ist doch eine coole Lösung. Wie ich es mitbekommen habe, habt ihr auch relativ viele ähnliche Sprachen, die ihr verwendet. Verwendet ihr da auch code-sharing in geteilten libraries oder wie du vorhin gemeint hast Thomas, dass du die Klassen kopierst und so wiederverwendest.

C1 [22min 41s]: Ja, wir haben das teilweise noch drin, dass wir libraries haben dafür. Da bin ich aber tatsächlich weggegangen davon, weil mich der Zyklus, wie ich das rausbringen kann zu sehr stört bei uns. Wenn das besser laufen würde, könnte ich mir vorstellen, dass ich das wieder einbaue. Ich habe letztens auch wieder darüber gelesen. Da müssten aber diese Bibliotheken, die wir da haben doch granularer werden und ich müsste das einfacher herausbringen können. Was mich stört ist ich muss den Release dieser Bibliothek immer manuell anstoßen. Das stört mich.

I [23min 29s]: Wie viele unterschiedliche Sprachen habt ihr ungefähr?

C1 [23min 34s]: Auf der Plattform haben wir Scala, Java, neu hinzugekommen ist jetzt Python. Vom Data-Lab haben wir noch R als Sprache.

I [23min 56s]: Habt ihr Frontends auch mit TypeScript oder JavaScript?

C1 [23min 59s]: Ja JavaScript haben wir, genau. Und dann weiß ich nicht, React ist das TypeScript?

C2 [24min 11s]: Wir verwenden React mit JavaScript aber wollen auch immer mehr in Richtung TypeScript gehen. TypeScript ist da da das, was wir in Zukunft verwenden wollen, aber das ist alles noch in JavaScript.

I [24min 28s]: Also einige verschiedene Sprachen auf jeden Fall. Stellt ihr auch über Testing, also e2e-Tests oder Integration-Tests, stellt ihr auch darüber sicher, dass Interoperabilität an sich sichergestellt ist?

C1 [24min 51s]: Wir haben drei Systeme bei uns. Das ist das Produktivsystem, ein QA-System und wir haben ein Entwicklungssystem. Über diese Systeme hinweg erkennen wir dann eigentlich schon bei den Acceptance-Tests, falls irgendwas nicht interoperabel wäre.

I [25min 14s]: Das heißt, ihr verwendet kein Mocking von Services oder ähnliches.

C1 [25min 18s]: Nein. Wir mocken die API für uns selbst, aber ob das ein kompletter Service gemockt ist - nein, glaube ich nicht.

I [25min 36s]: Ist das etwas, was euch interessieren würde für die Zukunft, dass man zum Beispiel eine automatische API-Definition erstellt und dann auch Mocks oder Stubs von Servern erstellt, die man für das lokale Entwickeln oder für Tests zum Beispiel verwenden würde. Oder sagt ihr zum Beispiel, wir brauchen das eh nicht, unser aktuelles Level reicht.

C1 [26min 06s]: Das ist schon interessent, wenn ich sagen kann, es wird automatisch irgendwie ein docker-Container hochgezogen, wo ich vorher definierte Testdaten quasi bekommen kann, um einfach den Stack auch zu testen, ob auch das REST-Template oder wie auch immer ich den Aufruf mache, dass das auch funktioniert, wie ich es mir vorgestellt habe.

I [26min 34s]: Das heißt, dann auch eher mit einer kleinen Demo-Funktionalität oder mit Stubs. Das heißt ich habe nur zum Beispiel immer wenn ich versuche, mir einen Benutzer zu geben, dann kommt auch immer der gleiche zurück. Wenn ich ein insert bei einen Stub mache, dann ist das nicht unbedingt in den Daten, wenn ich mir zum Beispiel alle Entities geben lasse. Aber ihr wolltet dann schon, dass es eine kleine gemockte Funktionalität vom Service ist als einen Stub.

C1 [27min 12s]: Also ich würde gerne eine kleine Funktionalität dann haben ja.

I [27min 17s]: Gut, schauen wir mal ob sich da was entwickelt.

C2 [27min 25s]: Das ist jetzt zwar nicht in diesem API-Kontext. Aber es gibt mocking libraries, die genau das machen für die ganzen AWS-Services. Also wo ich wirklich sage, ich habe jetzt irgendeinen Bucket, den mocke ich mir und ich kann da files inserten. Und wenn ich dann ne request mache - das ist ja im Endeffekt auch nichts anderes als eine API - „gib mir die objects in einem bucket“, dann bekomme ich auch die objects zurück. Also die Funktionalität ist zum Beispiel bei den AWS-Services vorhanden. Es gibt da immer Listen in der Dokumentation, was supported ist und was nicht. Und dann muss man halt gucken, die Standard-Sachen werden supported.

I [28min 16s]: Habt ihr auch schon einmal mit Remote-Procedure-Calls gearbeitet? Ich kenne es aus der Java-Welt, das gibt es bestimmt auch in anderen Kontexten.

C1 [28min 31s]: Ich habe es mir mal angeschaut, gRPC in unserem Tech-Stack-Umfeld. Aber das war eher so aus Interesse.

I [28min 45s]: Die Endpoints, dass die richtigen Endpoints definiert sind, ist das bei euch dann auch ein manuelles Ding, dass ihr in die Implementierung vom Server und dass ihr dass dann übertragt auf den Client.

C1 [29min 04s]: Ja, genau.

I [29min 10s]: Habt ihr bei euren Sprachen oder bei euren Frameworks irgendwelche besonderen Einschränkungen oder irgendwelche besonderen Features, die ihr mögt? Ihr habt zum Beispiel bei Python angesprochen, dass ihr da relativ flexibel seid, dass ihr einfach ein Dict habt. Oder gibt es da irgendwelche Besonderheiten, die euch auffallen, die ihr in anderen Frameworks vermisst?

C1 [29min 48s]: Also was wir gerne verwenden in Java ist Lombok, auch zur Definition von gettern und settern, dass ich das nicht manuell hinzufügen muss. Das macht die Herstellung von Datenklassen wirklich schön schlank im Code. Der ganze Boilerplate, der kann ja dann irgendwo sein wenn es kompiliert ist, aber ich sehe das nicht im Code, den ich bearbeite, das hat da schon Vorteile, ja.

I [30min 21s]: Ich kenne es zum Beispiel von uns aus der Arbeit, dass wir da auch kleinere Transformationen auch haben. Es kommt zum Beispiel ein Date als String über die Leitung als JSON, dass des dann auch automatisch zu einem Date-Objekt gemappt wird.

C1 [30min 43s]: Genau, ja.

I [30min 45s]: Wenn ihr euch dann vorstellt, ihr habt Generatoren. Wie viel sollen die generieren? Wollt ihr, dass nur die Datenklassen mit Transformationen mit erstellt werden, oder wollt ihr auch, dass Services mit generiert werden? Es ist zum Beispiel in OpenAPI möglich mit Generatoren, sich dann auch Services generieren zu lassen, die getypt sind, die genau den richtigen Endpoint ansprechen, mit den Richtigen Parametern und so weiter. Oder ist das euch zu viel?

C1 [31min 27s]: Das hätte ich schon gerne. Speziell in dem Sinne, dass sich bei uns das viel offener gestalten wird, es werden APIs auch dem Kunden ja zur Verfügung gestellt mit data-as-a-service und dem History, wo wir eben nicht wissen, was nimmt er für eine Programmiersprache dann eigentlich her. Ich denke persönlich, dass es schon für den Kunden einen großen Mehrwert hat, wenn wir ihm zum Beispiel in unterschiedlichen Sprachen, Java, C#, Python, R und JavaScript, einfach etwas an die Hand liefern, wo er einfach eine Bibliothek herunterladen muss und er dann damit dann mit unserem System arbeiten kann.

C2 [32min 16s]: Mit OpenAPI gibt es ja den OpenAPI-Generator, der hat ja glaube ich genau die fünf Sprachen, ich glaube sogar noch mehr, wo du dann Code herausgenerien muss. Ich habe es mir jetzt nicht extrem lange angeschaut, aber beim ersten herumspielen damit ist mir aufgefallen, dass das OpenAPI-file perfekt sein muss, sonst kommt gar nichts heraus. Dann kommt einfach ein Fehler und es funktioniert nicht. Also ich glaube, dass vermutlich so ein Regler, würde ich vom Gefühl her sagen, wie viel musst du im OpenAPI-file generieren, um dann einen Client dann auch daraus zu generieren. Es bringt auch nichts, wenn du ein OpenAPI-file dem Kunden gibst und er einen Client herausgeneriert, der halt Mist ist.

C1 [32min 59s]: Da geb ich dir Recht Constantin, ja.

I [33min 02s]: Ist es euch dann lieber, wenn ihr ein OpenAPI-file definiert und dann zum Beispiel das Skeleton für einen Server schon habt und das Skeleton oder eine SDK für den Client? Oder ist euch der Workflow lieber, wenn ihr einen Client erstellt mit den entsprechenden Endpoints und dass ein parser dann daraus quasi ein OpenAPI-file erstellt?

C2 [33min 34s]: Da haben wir neulich schon einmal darüber diskutiert.

C1 [33min 39s]: Ich tue zur Zeit immer den Server implementieren und dann wird automatisch eine OpenAPI daraus erstellt. Aber das ist halt dieses Paradigma interface-definition-first oder implementation-fist. Das kommt denke ich mal auf den Answendungsfall an. Wenn ich quasi von null anfange würde ich wahrscheinlich sagen, ich schreibe die OpenAPI zuerst, weil ich dann einfach sagen kann, wer mit mir interagiert, dem kann ich das schon einmal schicken und der kann auch schon parallel entwickeln, quasi seine Sachen. Beziehungsweise ich kann mich von Anfang an mit ihm zusammen hocken und sagen, so muss unsere Schnittstelle aussehen. Wenn ich jetzt aber schon in einem vorhandenen Projekt drin bin, dann würde ich wahrscheinlich sagen implementiere ich es und lass mir dann lieber daraus dann eine OpenAPI-Spezifikation generieren.

I [34min 40s]: Du hast kurz angesprochen, dass ihr das auch verwendet in den Generatoren.

C2 [24min 47s]: Ich habe es mal ausprobieren. Es sind Fehler geworfen worden. Ich habe mal ein bisschen versucht die zu fixen. Also es war aus einer existierenden OpenAPI-Spezifikation, die auch ein bisschen älter war. Und dann hatte ich direkt das Problem, dass es einfach nicht funktioniert. Und da ist jetzt die Frage, stecke ich jetzt Zeit hinein, die Spezifikation zu fixen oder schaue ich mir einfach das Swagger an und implementiere es einfach nach. Und da ist die Antwort, auf die ich gekommen bin, dann implementieren wir es nach.

I [35min 33s]: Wie oft kommt bei euch so ein change? Also ein neues Feature zum Beispiel? Und wie hängt es dann auch mit dem Deployment zusammen, gibt es fixe Zyklen die ihr habt oder ist das etwas flexibler?

C1 [35min 49s]: Momentan noch sehr flexibel. Es soll aber entweder einen festen Rhythmus von 2-4 Wochen bekommen oder ein automatisiertes Deployment, sobald wir als Entwickler sagen, es ist bereit für ein Deployment.

I [36min 08s]: Das heißt, auf daily deployment von auch kleineren Services kommt ihr auch gar nicht.

C1 [36min 18s]: Es könnte hinkommen, wenn wir quasi sagen, das was ich gemeint habe, wenn wir als Developer sagen, es wäre bereit zum Deployment. Aber mehrere Deployments pro Tag werden wir nicht fahren. Das wird eher auf wochemäßig herauslaufen.

I [36min 38s]: Vielen Dank für die vielen insights, die ich bekommen habe. Gibt es noch irgendwelche weiteren Probleme oder coole Konzepte, die ihr habt bei euch in der Firma, die auch etwas mit Interoperabilität zu tun haben?

C1 [37min 05s]: Da würde ich jetzt nein sagen. Das einzige, was für dich vielleicht interessant sein könnte, ist dieses Buch *(shows book into the camera)*. Da gibt es ein ganzes Kapitel über reuse-patterns, shared-code oder code-libraries. Da beschreiben die Vor- und Nachteile davon. *(Talking about lending it)*

I [37min 58s]: Ja super. Dann würde ich jetzt den offiziellen Teil beenden und auch die Aufnahme beenden, wenn es von eurer Seite nichts weiter gibt. *(Both shake their head)* Perfekt, dann beende ich jetzt die Aufnahme.

B.4 Interview D

I [00min 00s]: Ja hallo. Vielen Dank für die Möglichkeit, das Interview heute zu führen. Bist du damit einverstanden, dass eine Tonaufnahme von dem Interview gemacht wird?

D [00min 11s]: Ja gerne. Damit bin ich einverstanden.

I [00min 14s]: Sehr gut. Würdest du dich kurz vorstellen? Deine Rolle, dein Aufgabe und vielleicht auch kurz deine Firma?

D [00min 23s]: Genau, also ich bin Felix Schwägerl, ich bin momentan Teammanager bei MID GmbH für das Projekt Smartfacts oder das Produkt Smartfacts muss man sagen. Die MID ist eine mittelständische Firma mit inzwischen 150 Mitarbeitern. Traditionell haben wir uns eher mit Modellierungs-Software beschäftigt und haben dann langsam so den Übergang in das Web in den letzten Jahren geschafft und haben jetzt da auch zwei spezialisierte Produkte entwickelt. Smartfacts ist ein Produkt, das sich damit auseinandersetzt, Informationen aus dem Engineering miteinander zu verknüpfen und also aus verschiedenen Autoren-Werkzeugen oder auch aus verschiedenen Web-basierten Werkzeugen, die eben im Engineering Bereich verwendet werden, wie Requirements-Werkzeuge zum Beispiel. Wir binden uns dann in Smartfacts sozusagen an diese Werkzeuge an, bieten verschiedene Integrationen und verschiedene Möglichkeiten, die Informationen miteinander zu verknüpfen. Das heißt wir haben bei Smartfacts viel mit Schnittstellen, APIs und Libraries zu tun. Bevor ich Teammanager bei Smartfacts war, war ich Senior Softwareengineer für das Produkt Bpanda. Bpanda ist im Prinzip eine interaktive Web-Anwendung für Geschäftsprozessmodellierung, das heißt man kann dann mit BPMN Prozesse erstellen, verwalten, man kann im Prinzip seine ganze Enterprise-Architektur damit dokumentieren, die Prozesse ausführen und so weiter. Bpanda ist jetzt im Gegensatz zu Smartfacts mit weniger externen Schnittstellen versehen, sondern da ist für dieses Interview wahrscheinlich das Interessante, dass Bpanda nach einer strengen Microservice-Architektur entwickelt wird. Das heißt die unterschiedlichen Komponenten von Bpanda sind dann auch nach Domain-driven Design entwickelt worden und kommunizieren eben miteinander, um die komplette Anwendung zu realisiseren.

I [02min 31s]: Vielen Dank, das war ja schon sehr ausführlich. Wie viele Services sind jeweils beteiligt bei den einzelnen Projekten? Also eine ungefähre Größenordnung.

D [02min 45s]: Bei Bpanda sind es jetzt 20 Microservices intern und zusätzlich dann noch ungefähr 5 Services, zum Beispiel die Datenbank und das Frontend und ich sag jetzt mal Komponenten die jetzt noch am Backend dran-

hängen, aber nicht direkt zu der Microservice-Architektur gehören. Bei Smartfacts ist es dementsprechend einfach nur ein Monolith an dem dann auch wieder diese ungefähr 5 externen Services dranhängen.

I [03min 22s]: Okay. Und wie viele Teams arbeiten ungefähr an den einzelnen Projekten für die internen Services?

D [03min 33s]: Bei Bpanda ist es ein Team von 10 internen Entwicklern, die sich ungefähr 50/50 aufteilen auf Frontend und Backend. Bei Smartfacts sind wir momentan 3 Backend-Entwickler, aber da arbeiten wir ganz viel mit externen Freelancern zusammen, die uns dann das frontend schreiben, aber die dann zum Beispiel auch Spezialisten sind für Anwendungen und die Werkzeuge, die wir integrieren, sodass da die Teamgröße ziemlich variabel ist. Zusätzlich haben wir dann jeweils noch für Bpanda und Smartfacts einen Vollzeittester. Im Produktmanagement jeweils noch einen Product-Owner und darüber sozusagen noch einen Head of Productmanagement.

I [04min 20s]: Ja, okay. Vielen Dank. Dann würde ich gleich in den zweiten Teil einsteigen. Welche API-Paradigmen benutzt ihr? Das heißt benutzt ihr zum Beispiel Rest, benutzt ihr GraphQL, oder OSLC habe ich auch gelesen auf der Webseite.

D [04min 47s]: Genau, also prinzipiell ist es bei beiden Anwendungen so, dass wir Rest verwenden. Sämtliche Kommunikation intern und extern wird eben auf Basis des Rest-Paradigmas durchgeführt und wenn wir die Möglichkeit haben, dann auch auf Basis von JSON. Es gibt bei der Anbindung zu externen Services natürlich immer die Möglichkeit, dass dort XML verlangt wird. Und dann haben wir aber noch einige Optimierungen, das ist vielleicht auch noch interessant. Bei Bpanda bei den internen Microservices, da gibt es bestimmte Stellen, an denen wir dann aus Performanzgründen nicht JSON verwenden, sondern Google Protobuf. Google Protobuf ist ein proprietäres Format, das die Daten binär serialisiert. Hat ähnliche Eigenschaften wie JSON, ist aber im Detail manchmal noch, gerade was das Auspacken von Objekten angeht ein bisschen anders zu handlen. Aber es ist dann so, dass diese Protobuf-Objekte aus einer gemeinsame Spezifikationen in einer bestimmen Sprache ausgeleitet werden können in die entsprechenden Zielsprachen.

I [06min 08s]: Okay, interessant.

D [06min 12s]: Weil du es gerade erwähnt hast: OSLC, das ist dann eben der Hauptgrund, weshalb wir bei Smartfacts manchmal auf XML zugreifen, weil OSLC natürlich auf RDF. Und die natürliche Implementierung von RDF ist immer, dass RDF XML verwendet. Genau, das sind eigentlich so diese drei Hauptformate, die wir verwenden.

I [06min 38s]: Und wie tauscht ihr die Typinformationen aus? Verwendet ihr da OpenAPI oder ähnliches?

D [06min 48s]: Da haben wir bei den beiden Produkten unterschiedliche Ansätze. Bei Smartfacts haben wir die Datentypen und die Rest Schnittstellen modelliert in einem Modell dass dann mit einem MID-Produkt, nämlich dem Innovator, modelliert wird. und daraus leiten wir dann die Datenklassen ab, die Rümpfe ab aber eben auch die OpenAPI-Spezifikation, die wir dann eben mit Hilfe von Swagger publizieren, sodass die Frontend-Kollegen die dann mit dazu verwenden können für die Entwicklung. Bei Bpanda ist es so, da gibts um diese microservices herum noch eine Plattform, das ist ein API gateway und für dieses API gateway existiert dann auch eine öffentliche OpenAPI Dokumentation. Die wird generiert aus Annotationen. Da verwenden wir bei Bpanda selbstgestricktes, das hat den Hintergrund, dass wir als Programmiersprache in Bpanda Scala verwenden und für Scala gibt es jetzt nicht wie zum Beispiel für Spring einen vorgegebenen Weg, um zur OpenAPI zu kommen, sondern wir müssen uns das selbst bauen.

I [08min 13s]: Genau. Und das heißt, das es ist eine proprietäre Lösung, um OpenAPI Definitionen zu erstellen?

D [08min 22s]: In beiden Fällen ist es proprietär, selbst entwickelt, es ist keine Open-Source-Library.

I [08min 36s]: Okay. Du hast ja gemeint, dass die Datenklassen aus dem Innovator erstellt werden und das Frontend verwendet dann OpenAPI. Verwendet das Frontend dann auch Generatoren, um aus OpenAPI Code zu erzeugen oder verwenden die auch die Datenklassen?

D [08min 55s]: Nein, die sehen sie die OpenAPI-Typinformationen einfach an und wandeln die manuell dann in TypeScript-Klassen um, also wir verwenden da keine Generatoren.

I [09min 12s]: Das heißt, die Backend-Services sind automatisiert und das Frontend ist manuell.

D [09min 19s]: Richtig.

I [09min 20s]: Okay. Bist du oder ist dein Team zufrieden damit, wie das momentan abläuft?

D [09min 30s]: Ich muss sagen, ich bin hauptsächlich im backend unterwegs. Deswegen kann ich jetzt nicht hundertprozentig in die Lage der Frontend-Entwickler hineinversetzen, aber ich denke es würde ihnen schon helfen, die Möglichkeit zu haben, direkt aus der Swagger OpenAPI diese TypeScript-Klassen auszuleiten. Vor allem dann, wenn es Änderungen gibt. Das ist natürlich so, wenn wir eine Änderung machen und die nicht explizit kom-

munizieren, dann hat das Frontend keine Chance mitzubekommen, dass sich da etwas geändert hat. Da sind wir dann natürlich bei dem Thema syntaktische Interoperabilität denke ich mal.

I [10min 11s]: Genau. Zu den Änderungen: Wie oft werden die ausgetauscht? Du hast schon gemeint, das ist wahrscheinlich ein manueller Aufwand, dem Frontend-Team zu sagen, dass sich etwas geändert hat.

D [10min 26s]: Ja. Ich meine, die Änderungen im Backend können theoretisch jederzeit passieren. Es ist so, wir haben verschiedene Insanzen, eine davon ist unsere Hauptentwicklungsinstanz und von der API-Dokumentation dieser Hauptentwicklungsinstanz holt sich das Frontend im Prinzip die Information, was gerade aktuell ist. Das heißt, wenn wir im Backend ein Feature fertig stellen und das in den Entwicklungsbranch mergen, dann hat sich zu diesem Zeitpunkt die Dokumentation geändern, weil es automatisch deployed wird. Dann ist es so, dass das Frontend kaputt sein kann. Und dann muss es im laufenden Betrieb sozusagen nachziehen. Bei größeren Umstellungen machen wir dann auch manchmal noch den Schritt davor, dass wir erst auf eine feature-branch Instanz gehen, dort das Backend deployen, dann dort dass Frontend nachgezogen wird und dann mergen wir das zusammen zurück, damit eben der laufende Betrieb auf der Entwicklungsinstanz nicht aufgehalten wird.

I [11min 34s]: Wie sieht es zwischen den Microservices, also zwischen den Backends aus? Ist das da schneller synchronisiert oder teilweise instant auch?

D [11min 50s]: Bei Bpanda verwenden wir ein Monorepo, daher erkennt der Compiler im Prinzip sofort, wenn wir jetzt zum Beispiel auf eine property zugreifen, die gelöscht oder verschoben wurde oder so etwas. Da verlassen wir uns sozusagen auf den Scala-Compiler um die Kompatibilität der Versionen sicherzustellen. Was jetzt natürlich das Arbeiten mit dem Frontend angeht haben wir keinen automatischen Mechanismus, da müssen wir uns darauf verlassen, dass wir effizient kommunizieren einfach. Dass dass Ticket-System dokumentiert wird, welche Schnittstellenänderung haben sich ergeben für dieses Feature. Und dann ist natürlich noch der nachgelagerte Integrationstest wichtig.

I [12min 50s]: Das greife ich beides noch einmal in wahrscheinlich ein paar Minuten auf. Mit backwards-Kompatibilität ist bei euch dann hauptsächlich ein Problem für die innere Entwicklung, das bedeutet zwischen euren Backends, euren Microservices und den Frontends. Und die werden dann in der Testumgebung angepasst, bis die Kompatibilität wieder vorhanden ist. Habe ich das so richtig verstanden?

D [13min 35s]: So kann man es ausdrücken, genau. Also für Bpanda hauptsächlich. Bei Smartfacts ist es so, dass die Herausforderung dann bei der Kom-

munikation mit den externen Service liegt. Da ist aber im Prinzip in 90% der Fälle, dass wir die Leser sind von der anderen API. Das heißt wir haben das Problem, wie wir reagieren auf API-Änderungen. Da muss man sagen haben natürlich die APIs, die wir anbinden auch unterschiedliche Mechanismen. Manche haben zum Beispiel versionierte APIs, da ist es noch gut. Da kann man sagen, wir stellen uns auf eine bestimmte Version ein. Dann haben wir aber auch manchmal den Fall, gerade in der OSLC-Welt, dass wir dynamisch abfragen müssen, welche Version hat das Gegenüber gerade. Und daraufhin dann erst wissen, welche Anfragen dürfen wir überhaupt machen und wie sehen die konkreten Objekte aus, mit denen wir da interagieren. Das ist eine ziemliche Herausforderung. Da wäre es natürlich schon schön irgendeinen Mechanismus zu haben, gerade im versionierten Fall sozusagen, die Interoperabilität automatisch sicherzustellen. OSLC ist sowieso ein relativ schwacher Standard. Da können die einzelnen Tools im Prinzip spezifische Erweiterungen machen, auf die wir und dann auch wieder teilweise verlassen. Da müssen wir relativ viel dynamisch abfragen und oft machen wir dann auch einen Cut und sagen okay, wir unterstützen jetzt nur die drei Versionen, die unsere Hauptkunden verwenden.

I [15min 39s]: Nach außen bietet ihr wenige APIs an, habe ich das richtig verstanden so?

D [15min 51s]: Wir hatten schon den Fall, dass Kunden APIs von uns benutzt haben. Zum Beispiel dann für Reporting-Anwendungsfälle. Da ist es jeweils so gelaufen, dass wir dem Kunden die OpenAPI zur Verfügung gestellt haben. Das war denen dann meistens genug. Die haben das scheinbar manuell dann einfach in Datenklassen nachgebildet und verwendet.

I [16min 20s]: Das heißt es ist ein API-first Ansatz, das heißt ihr stellt einen Dienst zur Verfügung und kein Customer-Driven oder Consumer-Driven und ihr stellt auch keine SDK oder so zur Verfügung.

D [16min 37s]: SDKs nicht, nein. Es ist nur so gewesen, um die Spezialschnittstelle zu designen haben wir natürlich schon den Kunden im Boot gehaben und hatten da mehrere Iterationen bis man dann zur finalen Schnittstelle gekommen sind. Weil wir auch nicht immer den Anwendungsfall im ersten Moment voll verstanden haben natürlich.

I [17min 03s]: Dann würde ich auch gleich in den nächsten Teil übergehen. Das heißt, ich würde mich jetzt darauf konzentrieren, wie die einzelnen Services entwickelt werden, das heißt wir gehen gleich über Sprachen, über Repository-Setup, Testing. Das haben wir ja auch alles schon angesprochen. Fangen wir doch gleich mit dem Repository an. Wie sieht da euer Setup aus. Du hast schon Monorepo bei Bpanda angesprochen.

D [17min 38s]: Ja genau. Bei Bpanda verwenden wir für das Backend ein Mono-

repo mit git. Wir verwenden da den git-flow Entwicklungsworkflow. Das gleich gilt für Smartfacts. Bei Smartfacts ist es ja sowieso monolithisch inzwischen, da muss natürlich alles in einem Repo liegen. Multirepos haben wir früher für Bpanda verwendet, das ist vielleicht auch interessant. Jeder Microservice ist dann auch über einen eigenen build-Job gelaufen. Da ist dann ein Docker-Image herausgefallen und diese Images wurden dann deployed. Inzwischen ist es so, wir haben ein Monorepo das dann zum Beispiel auch diese Swagger-Schnittstellen beinhaltet. Es gibt sozusagen einen großen build-Job aber in diesem build-Job werden dann mehrere Docker-Container erstellt und das Deployment ist dann wieder das Gleiche.

I [18min 47s]: Das heißt, wenn sich eine Änderung im Setup ergibt baut ihr auch jeden Service neu und nicht irgendwie nur die, die sich geändert haben.

D [19min 02s]: Meistens ja. Bei Bpanda ist jetzt so, da werden die Services nicht immer direkt deployed sondern eigentlich nur nächtlich. Jetzt ist es aber so, wenn es sich jetzt eine dringende Änderung in deinem Service gibt, dann hat man auch die Möglichkeit, diesen einen Service neu zu bauen und separat zu deployen. Was natürlich dazu führt, dass man aufpassen muss wenn man die Schnittstelle geändert hat und ein anderer Service diese Schnittstelle verwendet. Dann kann es natürlich sein, dass es in dieser Konstellation dann bricht. Man muss dann auch die abhängigen Services dann neu deployen.

I [19min 46s]: Das heißt hauptsächlich alle zusammen, mit der Möglichkeit einzelne und damit auch abhängige Services zu deployen.

D [19min 57s]: Ja.

I [19min 58s]: Du hast auch schon die frequency angesprochen. Das heißt ihr hab tägliche neue Versionen?

D [20min 07s]: Für die Entwicklungsversion. Für die Kundenreleases machen wir dann einmal im Monat auf Basis der Entwicklungsversion. Da findet dann natürlich noch ein finaler Abnahmetest statt und dann ist sozusagen schon eine valide Konstellation von Services gerade auf der Entwicklungsinstanz deployed. Die nehmen wir dann sozusagen als baseline her und bauen daraus das Release.

I [20min 40s]: Bevor wir dann auch noch zum Testen kommen würde ich dich noch kurz Fragen welche Sprachen ihr benutzt ungefähr. Also wie viele verschiedene hauptsächlich.

D [20min 56s]: Eigentlich ganz einfach. In Bpanda verwenden wir nur Scala und bei Smartfacts nur Java. Es gibt einen einzigen Service, der zu diesen Satellitenservices außenrum gehört. Der ist für die Benutzerverwaltung da und der ist in C# geschrieben, das hat aber historische Gründe. Das würden wir wenn wir es neu machen würden in Java machen.

I [21min 24s]: In dem Frontend dann wahrscheinlich TypeScript oder?

D [21min 29s]: Frontend ist TypeScript, Angular. Da bin ich jetzt technisch weiter weg muss ich jetzt sagen. Aber das wir alle mit NodeJS im Prinzip gemacht.

I [21min 51s]: Bezüglich Testen ...

D [21min 57s]: Was ich vielleicht vorher noch sagen muss: Wir verwenden für die Services untereinander in Bpanda..., also es kann sein dass bestimmte Events passieren, auf die andere Services reagieren wollen. Also bei Domain-Driven Design haben wir ja diese absichtliche Redundanz in der Datenhaltung, so dass wenn ich jetzt ein bestimmtes Detail in meinem Service ändere muss das der andere ja natürlich mitbekommen und dafür verwenden wir Kafka als Message-Broker. Diese Kafka-Nachrichten, da kann man ja im Prinzip beliebige Bytes hin und her schicken, aber dafür verwenden wir dann auch wieder diese Protobuf Nachrichten, die wir zum Teil auch für die interne Kommunikation verwenden. Kafka ist jetzt so ein Spezialfall, es ist jetzt nicht API aber es ist trotzdem asynchroner Informationsaustausch.

I [22min 51s]: Das ist ja trotzdem wichtig für die Integration. Schieben wir das Testing kurz nach hinten. Ihr verwendet dann wahrscheinlich im Monorepo auch viel Code-Sharing. Ist die Annahme richtig?

D [23min 15s]: Was bedeutet jetzt Code-Sharing in dem Kontext?

I [23min 17s]: Das bedeutet, wenn ihr ein Monorepo habt, dass ihr eine library habt, in der die Typen quasi sind und der eine Service und der andere Service greift auf diese Typen dann zu. Im Gegensatz zu OpenAPI, wo ein Spec-file generiert wird und daraus dann gegebenenfalls die Klassen.

D [23min 42s]: Ja genau. Also generell haben wir die Möglichkeit libraries zu machen, das wird auch viel genutzt. Und einige dieser libraries sind einfach interne Funktionen, aber es gibt auch für diese Protobuf-Nachrichten entsprechend libraries, wodurch dann auch eben sichergestellt ist, dass es eine gemeinsame Sprache gibt zwischen den Services, die dann auch versioniert wird.

I [24min 14s]: Könnt ihr damit auch die endpoints sicherstellen? Das heißt stellt ihr damit auch sicher, dass die richtigen endpoints benutzt werden?

D [24min 25s]: Ja. Wir haben für jeden Service immer noch einen Http-Client. Services die von anderen Service abhängen verwenden dann den Http-Client dieses Services um darauf zuzugreifen. Die machen jetzt keine plain REST-Aufrufe. Der Service bietet sozusagen einfach die Schnittstellen an als Java-Schnittstellen an, so dass der Fall, dass die endpoints falsch definiert sind nicht vorkommen kann beziehungsweise wenn er vorkommt, ist dann der

aufgerufene Service schuld und nicht der aufrufende.

I [25min 09s]: Das heißt, der aufgerufene Service stellt diese lib-ähnliches oder SDK-ähnliches Ding bereit.

D [25min 18s]: Ja. Man kann es wirklich schon als SDK bezeichnen. Das ist jetzt dann für jeden Service immer so ein kleines mini-SDK.

I [25min 30s]: Verwendet ihr auch Testing, um die Interoperabilität sicherzustellen?

D [25min 38s]: Ja. Bei Bpanda haben wir eben diese zwei Arten von Tests. Unit-Tests und einen Integrationstest. Und beide verwenden dann direkt die generierten API-Schnittstellen aber auch die Datentypen, JSON und Protobuf. Also bei den Unit-Tests sind wir so ein bisschen abgewichen von dem reinen Unit-Testing-Paradigma. Das heißt, wir mocken jetzt zum Beispiel nicht die Datenbank, sondern wir deployen bei dem Test eine echte Datenbank mit. Die meisten Testfälle sind im Prinzip dann REST-Anfragen, die dann über diesen Client laufen, also über dieses SDK, das ich gerade erwähnt habe. Das heißt, das testen wir dann gleich mit. Und dann halt den Service mit seinen jeweiligen Schichten. Dadurch stellen wir dann auch sicher, dass es für den echten Aufrufer konsistent funktioniert. Und im Integrationstest ist es natürlich dann so, da deployen wir dann die einzelnen Container an sich. Und die kommunizieren dann mit den tatsächlichen Schnittstellen miteinander, inklusive Kafka dann auch.

I [26min 57s]: Das heißt, stubbing oder mocking ist bei euch kaum vertreten.

D [27min 03s]: Doch. Also wenn wir in dem einzelnen Service testen, dann ist es ja so, der hat schon Abhängigkeiten zu anderen Services. Und den deployen wir dann natürlich bei den Unit-Tests nicht mit. Das heißt, wenn wir jetzt eine Funktion testen wollen, die einen Aufruf auf einen anderen Service hat, dann müssen wir diesen anderen Service mocken oder stubben.

I [27min 28s]: Und das macht dann auch der Service, der getestet wird und es stellt nicht der gemockte Service etwas bereit, sondern der Service, der under test ist.

D [27min 43s]: Der aufrufende Service mockt sozusagen den aufgerufenen. Oder der Test für den aufrufenden Service. Da haben wir dann auch den Fall: Es gibt Service, die müssen besonders oft gemockt werden und da würde dann diese Mock-Schnittstellen dann auch in eine library verlagern, damit wir einfach keine Code-Duplikation hat.

I [28min 15s]: Verwendet ihr auch Remote-Procedure-Calls oder Stub-Generation in dem Kontext?

D [28min 24s]: Nein. Dadurch, dass wir ja plain REST verwenden, haben wir jetzt keine RPC.

I [28min 35s]: Wie stellt ihr sicher, dass ein Service auch das schickt, was erwartet wird? Ihr habt ja auch viel mit den internen mini-SKD gemacht. Stellt ihr dann so sicher, dass die richtigen Typen benutzt werden? Theoretisch ist es ja möglich, dass jetzt ein Service statt einem int einen String schickt oder ähnliches.

D [29min 07s]: Ja da müssten wir uns darauf verlassen, dass in diesem SDK die Typen korrekt definiert sind. Beziehungsweise dann für komplexe Datentypen, die dann im Payload verschickt werden oder als Response kommen, da haben wir ja dann wiederrum diese JSON oder Protobuf-Typen. Und da haben wir jeweils einen Mechanismus um diese sozusagen automatisch in ein Java-Objekt zu verpacken, also zu marshallen. Also da kann dann sozusagen nichts schief gehen. Aber wieder du gesagt hast, wenn es jetzt um die einzelnen Parameter geht oder auch um die Pfad-Parameter zum Beispiel geht, die dann sozusagen in die URL codiert werden, da ist es dann sozusagen Aufgabe des SDKs, die Typsicherheit herzustellen.

I [30min 12s]: Das heißt, der Aufrufer benutzt die SDKs und dadurch stellt ihr auch sicher, dass der Aufrufer auch eben immer euer festgeschriebenes Schema benutzt und nicht flexible Anfragen stellen kann, wie bei Postel's Law *(explains Postel's Law)*. Das habe ich aber bis jetzt auch noch nicht gehört, dass das jemand verwendet.

D [31min 08s]: Nein, also da verlangen wir schon einfach, dass da exakt die Datentypen verwendet werden, wie sie definiert sind. Zum Beispiel für bestimmte ID-Formate gibt es dann auch reguläre Ausdrücke, dass man sagen kann, es muss jetzt eine Java-UUID sein oder irgendetwas anderes.

I [31min 33s]: Wie schränken euch die Sprachen und Frameworks in eurer Lösung ein? Oder gibt es auch zum Beispiel írgendwelche special Features, die nur bei eurer Sprache oder Framwork existieren?

D [31min 47s]: Eher weniger. Wir haben jetzt bei beiden Produkten eher die Tendenz gehabt kein Framework zu verwenden, sondern eine REST-library und uns sozusagen das Framework selbst zu bauen. Gut, das hatte zum einen historische Gründe. Smartfacts ist schon vor acht Jahren entstanden. Ich glaube es gab schon Spring aber es hat sich noch nicht durchgesetzt. Und es gab diese ganzen OpenAPI-Generatoren noch nicht. Deswegen mussten wir damals noch viel händisch machen. Bei Bpanda war es sozusagen auch die ausdrückliche Entscheidung, kein Framework sondern eine library zu verwenden, also Akka HTTP in dem Fall, weil wir uns nicht einschränken wollten. Deswegen sind wir da komplett flexibel. Wir haben dann Wert darauf gelegt, dass wir unseren eigenen Mechanismen implementieren können,

um die Interoperabilität so sicherzustellen, dass es auch zu unserem Konzept intern passt.

I [33min 00s]: Dann würde ich auch diesen Block des Interview abschließen. Gibt es noch weitere interessante Themen im Bereich von der Interoperabilität von Microservices, die wir jetzt noch nicht genannt habe?

D [33min 30s]: Ja also eine Sache, über die wir uns oft streiten zwischen Frontend und Backend sind die Response-Codes. Oder einfach die Frage, wann ist es ein Client-Fehler, wann ist ein Server-Fehler. Also es ist irgendwie schwer, so etwas wie einen Contract zu definieren. Man hat zwar dann die Datentypen und man hat eine Dokumentation der Schnittstellen, aber was sozusagen dahinter ist, also welche ID darf ich eigentlich einsetzen und was ist das erwartete Verhalten, wenn ich mich auf eine ID beziehe, die noch nicht existiert. Da geht es dann schon ein Level höher von der syntaktischen dann eigentlich in die semantische Interoperabilität. Da wird schon oft gestritten. Wo auf zu wenig Wert gelegt wird, ist wirklich sich die Mühe zu machen, die Fehlerbehandlung dann so zu bauen, dass man sagt, das ist jetzt eindeutig ein Client Fehler, da ist die Request aus irgendeinen Grund nicht zulässig oder es es ist ein Server-Fehler, es hätte eine Antwort kommen sollen, aber der Server hat die falschen Annahmen über die Daten gemacht, die er bekommt. Darüber haben wir wie gesagt oft gestritten. Was vielleicht ein Thema ist, das du nicht angesprochen hast, ist Monitoring. Oder noch ein Thema ist die Art des Deployments erst einmal. Beide Produkte sind ursprünglich als Cloud-Produkt konzipiert worden. Wir hatten eigentlich die Vision, wir stellen da einen dicken Server irgendwo hin und der Kunde holt sich die Software dann as-a-Service. Was bei Bpanda in 90% der Fällen geklappt hat. Bei Smartfacts nur in ungefähr 5% der Fällte, weil Kunden sagen, Cloud ist für uns nicht, wir wollen die Software bei uns in der Firma haben. Gerade wenn es dann um das Thema Toolintegration geht kannst du dir vorstellen, dass es wahnsinnig schwierig wird, Fehler nachzuvollziehen, weil die Kunden dann auch oft andere Versionen der angebundenen Tools verwenden wie die, die wir in der Entwicklung verwenden. Was wirklich schwierig ist, ist die Fehlersuche und Analyse. Ganz oft liegt es dann eben in den Schnittstellen. Ganz oft kommt dann doch irgendein Spezialfall, in dem wir die Schnittstelle falsch aufrufen oder in dem sich eben eine kleine Schema-Änderung zwischen Versionen ergeben hat. Solche Sachen sind ziemlich schwer zu analysieren. Eine Lösung dafür ist eben das Monitoring. Als einfach die Möglichkeit zu haben, nachzuvollziehen, wie viele REST-Request sind jetzt bei einem bestimmten Service angekommen, wie viele waren davon erfolgreich. Wenn Fehler gekommen sind, dann natürlich auch gleich die entsprechende Analysemethode zu haben, Stacktrace et cetera. Das funktioniert in der Cloud natürlich besser als beim Kunden, weil er meistens nicht bereit ist, so eine Monitoring-Umgebung aufzuziehen, auf

die wir dann wiederum zugreifen dürfen. Also das sehe ich so als Hauptproblem. Also es kracht im laufenden Betrieb und wir müssen erst einmal analysieren, woran es liegt.

I [37min 32s]: Vielen Dank. Du hast im Vorfeld kurz angesprochen, dass ihr bei Smartfacts gegen einen Microservice-Ansatz entschieden habt. Was waren dafür die Hauptgründe?

D [37min 50s]: Meiner Meinung nach eignet sich der Microservice-Ansatz gut für die Cloud. Wenn man einfach einen dicken Service hinstellen kann, der dann auch skalierbar ist, der von unterschiedlichen Kunden verwendet wird. Da lohnt es sich dann auch. Wenn man jetzt bedenkt, man hat jetzt 20 Microservices. Jeder Microservices verbraucht dann zwei bis vier Gigabyte an speicher, dann lohnt es sich einfach, eine große Maschine hinzustellen. Bei Smartfacts ist dann eben die Tendenz dann eben in Richtung on-premise gegangen. Da sind zwar auch Kunden dabei, die viele Benutzer haben, also große Konzerne. Aber die sehen trotzdem nicht ein, da irgendwie einen 64GB Server da hinzustellen nur für diese Anwendung. Deswegen sind wir dann eben zurück zum Monolithen gegangen. Wo man das Ganze dann natürlich einfacher auch deployen kann. Gerade wenn man das Deployment nicht monitoren kann. Es gibt ja dann doch immer wieder den Fall, dass es zwischen den Services knirscht oder dass ein Service abstürzt und es sich dann zu einem bestimmten Fehler führt. So etwas lässt sich in der Cloud gut überwachen. Man kann schnell darauf reagieren. Wenn das Ganze dann beim Kunden liegt, der dann nicht weiterkommt, dann ist der Microservice-Ansatz einfach aus unserer Sicht für unsere Anforderungen zu viel Overhead gewesen. Es gibt ja irgendwie auch das Sprichwort „Microservices erst ab 100 Entwicklern“. 100 finde ich jetzt ein bisschen hoch, aber so schön die Architektur ist und es ist eine gute Idealvorstellung, aber in der Realität entstehen viele neue Probleme dadurch, für die man einfach auch Manpower braucht. Das muss sich dann natürlich alles lohnen.

I [39min 59s]: Ja perfekt, vielen Dank auch für den Eindruck. Hast du noch weitere Themen, die du ansprechen willst? Ansonsten wären wir jetzt mit dem Interview durch.

D [40min 16s]: Ich gehe mal kurz noch in mich. Nein, also ich denke gerade diese Thematik mit Kafka habe ich noch erwähnt. Nein. Ich bin alles los geworden. GraphQL sehe ich gerade, das haben wir uns mal kurz angeschaut. Das wäre auch wirklich eine Sache, die in der Theorie recht interessant ist. Aber das wäre dann auch wieder so etwas, entweder man stellt komplett um oder man belässt es bei REST. Da waren wir einfach zu spät dran.

I [40min 58s]: Okay. Perfekt, dann würde ich das Interview hiermit beenden. Vielen Dank.

D [41min 05s]: Danke auch, war interessant.

B.5 Interview E

The interviewee(s) did not consent to the publication of the transcript.

B.6 Interview F

The interviewee(s) did not consent to the publication of the transcript.

C Evaluation Survey

The initial category names are used in the survey. After the evaluation, the following renaming was done:

- *"Exhaustiveness of Generators" to "Level of Static Type Safety"*
- *"Completeness of Generators" to "Completeness"*

1. The categorization system contains all major categories.

O	O	O	O	O	O	O	|	O
1	2	3	4	5	6	7		
Strongly disagree						Strongly agree		Can't decide

2. Are there categories missing or unexpected?

3. The characteristics of the dimensions are allocated in a logically correct place.

O	O	O	O	O	O	O	|	O
1	2	3	4	5	6	7		
Strongly disagree						Strongly agree		Can't decide

4. The categorization system is useful.

O	O	O	O	O	O	O	|	O
1	2	3	4	5	6	7		
Strongly disagree						Strongly agree		Can't decide

5. I can see myself using the categorization system in the future.

O	O	O	O	O	O	O	|	O
1	2	3	4	5	6	7		
Strongly disagree						Strongly agree		Can't decide

6. Why / why not?

7. Sort the categories by importance from 1 (least important) to 7 (most important). It is possible to assign the same value to several categories.

- Driver of Change: _____
- Exhaustiveness of Generators: _____
- Completeness of Generators: _____
- Adoption of Different Versions: _____
- Deployment of Different Versions: _____
- Technology Support: _____

8. Additional comments:

C.1 Evaluation Survey Results

This subsection contains all inputs from the free text survey questions.

2. Are there categories missing or unexpected?

   - No
   - Push vs Pull, Synchronous vs Async, Messaging vs direct, Handling of large payloads
   - I am surprised deployment of new versions and adoption of new versions are separate categories. I can see some differences, but it feels like the categories are strongly related.
   - Some categories are biased on generators. A competing approach would be testing (of any sort) that ensures compatability (no by mechanism, but by "testing effort"). The categories "Exhaustiveness" and "Completeness" however can be mapped to this as well I think, so it is just a matter of naming the categories IMO.

6. Why / why not?

   - Its useful to provide a guidline on what/how changes need to be done.
   - Those dimensions are only covering the basics of what we ask ourselves when we design interactions between (micro-)services.
   - good tool to reflect the current approach and process
   - Gut geeignet, um in einem Projekt den aktuellen Stand der Interoperabilität zu ermitteln und ihn mit den Anforderungen abzugleichen. Dies kann im Anschluss bei der Wahl der Tools und der Konzeption des Setups helfen, um den gewünschten Stand zu erreichen.
   - It seems like a useful way to make sure to think about every dimension and not forget something. Additionally, I learned some new dimensions (especially regarding the deployment dimensions) from reading the categorization system.
   - I think I am most likely to learn from reading it, rather than actively use the categories themselves. It would be useful to have a cheat sheet with longer names/short descriptions for the categories and dimensions to make the system itself more useful.
   - I can see myself using it as a mental model (that already exists implicitly). Making these considerations explicit might help especially unexperienced programmers/designers to find their solution and with communicating it with peers.

I think especially how the categorization changes over time is quite interesting.

8. Additional comments:

   - Nice work!

     Some comments on the characteristics of the categories:

     – D2 "Exhaustiveness" doesn't cover endpoints-only approaches (not sure if they exist in practice, though). In general I see a gap between "Types only" and "Everything".

     – D3 "Completeness" is missing the characteristic "None"

     – D4 "Adoption" might vary throughout the system (special cases) => so this categorization cannot depict the integration strategy of a whole system (which is fine but should be made clear)

     – D5 "Deployment of Different Versions" is misleading as it focuses on the compatibility check and not the deployment (better sth like "Compatibility Check Automation"). Besides, I could imagine semi-automatic approaches as well.

D Evaluation with JValue

In the following subsections, the italic text stands for questions asked by me. All other text stems from the interviewees.

D.1 Interview JValue A

1. How did the interoperability change, especially regarding the type safety of the endpoints?

- It has improved generally.
- Using generators instead of creating type definitions by hand is good.
- There is now less code redundancy.
- The use of OpenAPI requires the developers to describe everything cleanly.
- It is especially good that status codes can be defined with OpenAPI. These status codes are reflected in the generated code and improve the typing.

2. How technology-agnostic is the new solution?

- Works well for this setup with TypeScript.
- Problem 1: It relies on the availability of generators for other languages that one might want to use.
- Problem 2: It only works for REST. Other communication methods require an alternative solution.

3. How did the developer experience change?

- Preliminary note: This answer only consists of speculations since there was not enough time to deeply test the implementation.
- Pleasantly surprised by how it currently functions, especially regarding file watching. This means that changes made in the monorepo during development are instantly live and available in the client code. Errors are therefore detected immediately.
- Disadvantage: The database needs to be running in order to generate the OpenAPI specification. However, this is not a major concern as it usually runs during development anyway.

4. Do you plan to continue using this categorization system in the JValue project to document and potentially alter interoperability?

- Preliminary note: I can only speak for myself and offer speculations.

- It is useful for contextualizing the standpoint and then planning anew if it needs change. Examples of changed requirements can be:
  - Introducing a service with a new programming language.
  - Usage of other protocols besides HTTP.
  - Independent deployment of services with possibly separate versions.
- If the usage shows that there are some categories missing, one can simply add new categories for their own use.

D.2 Interview JValue B

Preliminary note: All answers are provided with the disclaimer that I did not work extensively with it; I (Interviewee B) only briefly examined it.

1. How did the interoperability change, especially regarding the type safety of the endpoints?

- One large change is not defining the types manually but autogenerating them.
- Manual work is error-prone and sometimes skipped because it is too laborious.
- There is now a single source of truth from which is generated and that ensures that the data types are always up to date.
- Biggest change is that also the endpoints are generated, i.e. which URL is called and such things. This is now connected with less manual error-prone work.

2. How technology-agnostic is the new solution?

- OpenAPI specification is a good solution for that since it is a popular standard with a lot of generators for several different programming languages.
- Aside from that, I like it when widespread purpose-built technology is used that we could extend on our own but where also a lot of plugins already exist.

3. How did the developer experience change?

- Divided opinion. It is a trade-off that is worth it.
- Developer experience is better on a higher level if one just wants to use and slightly edit the API. One does not have to worry about the type definitions and API endpoints; it is hidden in the generated code. We have a lot of

students that work on the project for a short amount of time. They profit from the well-defined code.

- But in-depth changes are probably more complex and also getting a deep understanding is now harder. This is probably done by people that stay longer in the team and can read up on that.
- It is important to have good documentation including diagrams on that (which is done in the demo implementation). It is important to get a high-level-overview before dealing with in-depth changes.

4. Do you plan to continue using this categorization system in the JValue project to document and potentially alter interoperability?

- Personally, I do not know if anyone has such plans.
- I found it very useful to learn about these things, especially by thinking about them in categories. There were also some new points that I have not considered yet.
- It is likely that we come back to it when we want to get away from the block deployment.
- So the categorization system will probably be considered when bigger changes are planned.
- It would be helpful to have a concise cheat sheet of the categorization system with an overview of the categories, maybe even with checkmarks to share with others.

D.3 Interview JValue C

1. How did the interoperability change, especially regarding the type safety of the endpoints?

- It has improved.
- Previously the type definitions are placed in a shared library and not in the API itself which was difficult for onboarding.
- Now also the endpoints are specified (i.e., the URLs, HTTP methods, parameters, and so on) which helps with the integration.
- A condition for that is that one uses the tool properly, i.e. one must not forget annotations. However, one will probably notice that quickly.

2. How technology-agnostic is the new solution?

- It is decoupled because of the specification file.

- It is very technology-agnostic in terms of service technologies, i.e. programming languages and frameworks.
- However we are restricted to HTTP with OpenAPI, we cannot simply switch to gRPC or similar. However, that did not change compared to before.
- Therefore, the solution is now better in that regard.

3. How did the developer experience change?

- It improved much since the code is now more cohesive.
- The models are now there where they are defined and in the project where they are used.
- A well-working generator is required for that. This caps the technology independence.
- The learning curve is not significantly higher than it was before when only using the code annotations and code generators.
- However, changing the code generation setup requires more knowledge.

4. Do you plan to continue using this categorization system in the JValue project to document and potentially alter interoperability?

- It is an explicit representation of a mental model that many developers already have similar to that in their heads.
- Probably not used to show it to every new developer.
- Its main use is as a purpose-driven tool that is used in discussions, e.g. for refactoring.

E Interviews Applied to the Categorization System

Category	Characteristics	Interview					
		A	B	C	D	E	F
D_1 (D.o.C.)	Code			X	X	X	X
	Specification		X				
	Both	X					
D_2 (Type Safety)	None			X		X	X
	Types Only	X					
	Everything		X		X		
D_3 (Completeness)	Partial		X	X	X		X
	Full	X				X	
D_4 (Adoption)	Instant	X		X	X	X	
	Delayed		X				X
D_5 (Deployment)	Manual		X	X		X	
	Automated	X			X		X
D_6 (Technology)	Internally	X			X		
	Agnostic		X	X		X	X

Table A.1: Categorization of a representative setup from each interview.

References

Abukwaik, H., & Rombach, D. (2017). Software interoperability analysis in practice: A survey. *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, 12–20. https://doi.org/10.1145/3084226.3084255

Bailey, K. D. (1994). *Typologies and taxonomies: An introduction to classification techniques*. SAGE Publications. https://uk.sagepub.com/en-gb/eur/book/typologies-and-taxonomies

ECMA International. (2022). ECMAScript 2022 Language Specification (13th ed.). https://www.ecma-international.org/wp-content/uploads/ECMA-262_13th_edition_june_2022.pdf

Gardner, R. (2001). *When listeners talk: Response tokens and listener stance*. John Benjamins. https://www.jbe-platform.com/content/books/9789027297426

Garousi, V., Felderer, M., & Mäntylä, M. V. (2016). The need for multivocal literature reviews in software engineering: Complementing systematic literature reviews with grey literature. *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. https://doi.org/10.1145/2915970.2916008

Guion, L. A. (2002). Triangulation: Establishing the validity of qualitative studies. *University of Florida, FCS6014*.

Jacob, E. (2004). Classification and categorization: A difference that makes a difference. *Library Trends*, *52*(3), 515–540.

Jansen, H. (2010). The logic of qualitative survey research and its position in the field of social research methods. *Forum Qualitative Sozialforschung / Forum: Qualitative Social Research*, *11*(2). https://doi.org/10.17169/fqs-11.2.1450

Kallio, H., Pietilä, A.-M., Johnson, M., & Kangasniemi, M. (2016). Systematic methodological review: Developing a framework for a qualitative semi-structured interview guide. *Journal of Advanced Nursing*, *72*(12), 2954–2965. https://doi.org/10.1111/jan.13031

Lane, K. (2021). *Design first, prototype first, or code first apis?* Retrieved May 25, 2023, from https://apievangelist.com/2021/11/20/design-first-prototype-first-or-code-first-apis/

Lewis, J., & Fowler, M. (2014). Microservices. Retrieved March 1, 2023, from https://martinfowler.com/articles/microservices.html

Lincoln, Y. S., & Guba, E. G. (1985). *Naturalistic Inquiry*. SAGE Publications, Inc.

MDN Contributors. (2023). Number - JavaScript | MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number

Miller, D. (2022). Websockets? · issue #2947 · oai/openapi-specification. Retrieved May 9, 2023, from https://github.com/OAI/OpenAPI-Specification/issues/2947

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, *63*(2), 81–97. https://doi.org/10.1037/h0043158

Morris, K. (2020). *Infrastructure as code* (2nd ed.). O'Reilly Media, Inc.

Myers, M. D., & Newman, M. (2007). The qualitative interview in IS research: Examining the craft. *Information and Organization*, *17*(1), 2–26. https://doi.org/10.1016/j.infoandorg.2006.11.001

Newman, S. (2015). *Building Microservices*. O'Reilly Media, Inc.

Nickerson, R. C., Varshney, U., & Muntermann, J. (2013). A method for taxonomy development and its application in information systems. *European Journal of Information Systems*, *22*(3), 336–359. https://doi.org/10.1057/ejis.2012.26

Object Management Group (OMG). (2017). Unified Modeling Language (UML) Version 2.5.1 [OMG Document Number: formal/2017-12-05]. https://www.omg.org/spec/UML/2.5.1/PDF

Oliver, D. G., Serovich, J. M., & Mason, T. L. (2005). Constraints and Opportunities with Interview Transcription: Towards Reflection in Qualitative Research. *Social Forces*, *84*(2), 1273–1289. https://doi.org/10.1353/sof.2006.0023

OpenAPI Initiative. (2021). OpenAPI Specification Version 3.1.0. Retrieved May 29, 2023, from https://spec.openapis.org/oas/v3.1.0

OpenAPI-Generator Contributors. (2023). Customization | OpenAPI Generator. Retrieved May 29, 2023, from https://openapi-generator.tech/docs/customization/#selective-generation

OpenJS Foundation & Node.js contributors. (2022). Node.js Documentation. Retrieved May 29, 2023, from https://nodejs.org/dist/latest-v17.x/docs/api/all.html#all_globals_class-formdata

Otto GmbH & Co. KG. (2023). OTTO API Guidelines. Retrieved May 29, 2023, from https://api.otto.de/portal/guidelines

Panetto, H. (2007). Towards a classification framework for interoperability of enterprise applications. *International Journal of Computer Integrated Manufacturing*, *20*(8), 727–740. https://doi.org/10.1080/09511920600996419

Peffers, K., Tuunanen, T., Rothenberger, M. A., Chatterjee, S., Cichocki, A., Goles, T., Jeffery, R., & Tuunanen, T. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, *24*(3), 45–77. https://doi.org/10.2753/MIS0742-1222240302

Postman. (2023). *Guide to API-first*. Retrieved May 29, 2023, from https://www.postman.com/api-first/

Potvin, R., & Levenberg, J. (2016). Why Google stores billions of lines of code in a single repository. *Communications of the ACM*, *59*, 78–87. https://doi.org/10.1145/2854146

Preston, C. C., & Colman, A. M. (2000). Optimal number of response categories in rating scales: Reliability, validity, discriminating power, and respondent preferences. *Acta Psychologica*, *104*(1), 1–15. https://doi.org/10.1016/S0001-6918(99)00050-5

Robinson, O. C. (2014). Sampling in interview-based qualitative research: A theoretical and practical guide. *Qualitative Research in Psychology*, *11*(1), 25–41. https://doi.org/10.1080/14780887.2013.801543

Rosch, E., Mervis, C. B., Gray, W. D., Johnson, D. M., & Boyes-Braem, P. (1976). Basic objects in natural categories. *Cognitive Psychology*, *8*(3), 382–439. https://doi.org/10.1016/0010-0285(76)90013-X

Savkin, V. (2019). Misconceptions about monorepos: Monorepo != monolith. Retrieved May 29, 2023, from https://blog.nrwl.io/misconceptions-about-monorepos-monorepo-monolith-df1250d4b03c

Schultze, U., & Avital, M. (2011). Designing interviews to generate rich data for information systems research. *Information and Organization*, *21*(1), 1–16. https://doi.org/10.1016/j.infoandorg.2010.11.001

Strauss, A., & Corbin, J. (2008). Basics of qualitative research: Techniques and procedures for developing grounded theory. *Organizational Research Methods*, *12*(3), 614–617. https://doi.org/10.1177/1094428108324514

tRPC authors. (2022). Further Reading | tRPC. Retrieved May 29, 2023, from https://trpc.io/docs/further-reading

Valle, P. H. D., Garcés, L., & Nakagawa, E. Y. (2019). A typology of architectural strategies for interoperability. *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse*, 3–12. https://doi.org/10.1145/3357141.3357144

Van Der Veer, H., & Wiles, A. (2008). Achieving technical interoperability (3rd ed.). *European telecommunications standards institute*.

van Steen, M., & Tanenbaum, A. S. (2017). *Distributed Systems* (3rd ed.). distributed-systems.net.

Widodo, H. P. (2014). Methodological considerations in interview data transcription. *International Journal of Innovation in English Language Teaching and Research*, *3*(1), 101–107.

Wolff, E. (2016). *Microservices: Flexible Software Architecture*. Addison-Wesley Professional.

Zalando SE Opensource. (2023). Zalando RESTful API and Event Guidelines. Retrieved May 29, 2023, from https://opensource.zalando.com/restful-api-guidelines/

Zimmermann, O. (2017). Microservices tenets: Agile approach to service development and deployment. *Computer Science-Research and Development*, *32*, 301–310. https://doi.org/10.1007/s00450-016-0337-0