Department of Informatics

Technical University of Munich

Master's Thesis in Information Systems

Interactive Visualizations for Supporting the Analysis of Distributed Services Utilization

Daniel Graf Hoyos

Department of Informatics

Technical University of Munich

Master's Thesis in Information Systems

Interactive Visualizations for Supporting the Analysis of Distributed Services Utilization

Interaktive Visualisierungen zur Unterstützung der Analyse der Nutzung verteilter Services

Author:	Daniel Graf Hoyos
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Martin Kleehaus, M.Sc.
Date:	June 15, 2018

Abstract

During the past few years, a fundamental architectural shift has taken place in the field of software development. Classical monolithic systems are progressively transformed into or replaced by Microservice-based architectures. This process has been mainly pioneered and accelerated by large tech companies like Netflix, Spotify or Uber, which describe their experiences publicly and extensively provide open source software. Microservices are the result of a decomposition of the complex internal logic of monolithic applications into small, clearly separated tasks. Each Microservice performs only one independent task and delegates more complex processes further on to other Microservices. While communication between different services takes place over well-defined, standardized interfaces, there is a large heterogeneity in the characteristics of Microservices and their instances like programming languages or run-time environments, respectively. The benefits of Microservices over monoliths include easier scaleability, improved technology fit and better service reusability. They enable agile development, where small teams can rework individual Microservices in shorter iteration cycles. However, the lower inner complexity of a single service is achieved through a higher outer complexity, as hundreds of Microservices can be necessary to replace one monolithic system. In order to thoroughly monitor and document these large service networks, a new tool for Service Architecture Discovery was developed at TUM. This work focuses on the visualization of the obtained Architecture Discovery results. Different views providing information for various stakeholders are implemented in a prototype visualization tool. Evaluation of the implementation is performed in a lab environment providing a basic distributed Microservice infrastructure.

Contents

Outline of the Thesis

CHAPTER 1: INTRODUCTION

In this chapter, the transition from monoloithic systems to the newly emerging Microservice technology is presented. Besides, an overview is given of the thesis and its purpose.

CHAPTER 2: BACKGROUND

This chapter provides a more detailed view on features, characteristics and benefits of Microservice-based architectures compared to monolithic systems. The description of Enterprise Architectures explains the applied EA framework and introduces stakeholders with their respective interests and necessary information. Additionally, layers and artifacts are described, thereby providing an idea of the entities included in the visualized architecture model. Finally, the concept and goals of Distributed Tracing are presented, which underlie the Architecture Discovery approach described in the next chapter.

CHAPTER 3: RELATED WORK

The present thesis is embedded in the context of other thesis work at the SEBIS chair at TUM. This chapter gives an overview about the work on Service Architecture Discovery, Anomaly Detection and an Enhanced Enterprise Architecture Model, performed by Patrick Schäfer, Lukas Steigerwald and Christopher Janietz, respectively. All three of those theses are advised by Martin Kleehaus

CHAPTER 4: SOLUTION APPROACH

This chapter provides the theoretical basis of the visualization for the subsequent prototype implementation. It outlines displayed Entities, used Data Sources and imagined Stakeholders of the visualization prototype. Key decisions are made regarding the provided views and information, aggregation levels and navigation flows.

CHAPTER 5: PROTOTYPE IMPLEMENTATION

The prototype implementation is documented by describing components of the views conceptualized in the previous chapter and summarizing the applied libraries, frameworks and application platform. The chapter also provides code examples and depicts the navigation flow through the implemented prototype.

CHAPTER 6: EVALUATION

For the evaluation of the implemented prototype, a test environment with a basic Microservice architecture is used. Increasingly complex data sets are provided to the prototype and functional correctness of the produced graphs is ensured. Additionally, other Microservice visualization approaches are considered in comparison to the designed prototype.

CHAPTER 7: CONCLUSION AND OUTLOOK

The last chapter discusses potential extensions to the visualization by including more diverse data types or implementing additional features. A short conclusion wraps up the work.

1 Introduction

1.1 From Monoliths to Microservices

In our increasingly digital world, everyone uses countless services and applications offered by various IT companies. From online shopping, social media, streaming of music and movies to platforms for ordering food or transportation, billions of user interactions take place every day. This digitization of the daily life is reflected by the rapid expansion of the corresponding IT companies like Amazon, Spotify or Uber. However, as a consequence of this enlargement, many companies struggled to scale up their IT landscapes at the same speed as their businesses grew. But development speed is crucial to compete in the marketplace, as Netflix Director of Web Engineering Adrian Cockcroft learned [Mauro, 2015b]. This is why Netflix as one of the first global players decided to radically turn over their IT architecture by abandoning their classic, monolithic system for the emerging Microservice technology.

In a traditional monolith, a single IT system contains many IT processes and components of a company in a tightly coupled manner. Such a self-contained application is able to perform all tasks on the way to a specific function by itself, independently from other applications. Such systems had been state-of-the-art software development in the early 2000s, but the limitations of monoliths became especially apparent during rapid expansion of the businesses. As all components in a monolith are tightly coupled, a performance bottleneck in a single process limits the whole application. Likewise, reworking a part of the system requires a redeployment of the full code base and becomes more and more complex with growing code size. Together with the increasing tendency towards agile software development and shorter release intervals, these drawbacks of monoliths fostered the emergence of Microservice-based architectures. In this architectural and organizational software development approach, a complex system is broken down into small entities called Microservices. They communicate via well-defined interfaces and perform only a single, isolated task each. This decoupling allows the independent development, operation and scaling of the individual services, thereby enabling agile development with shorter release cycles.

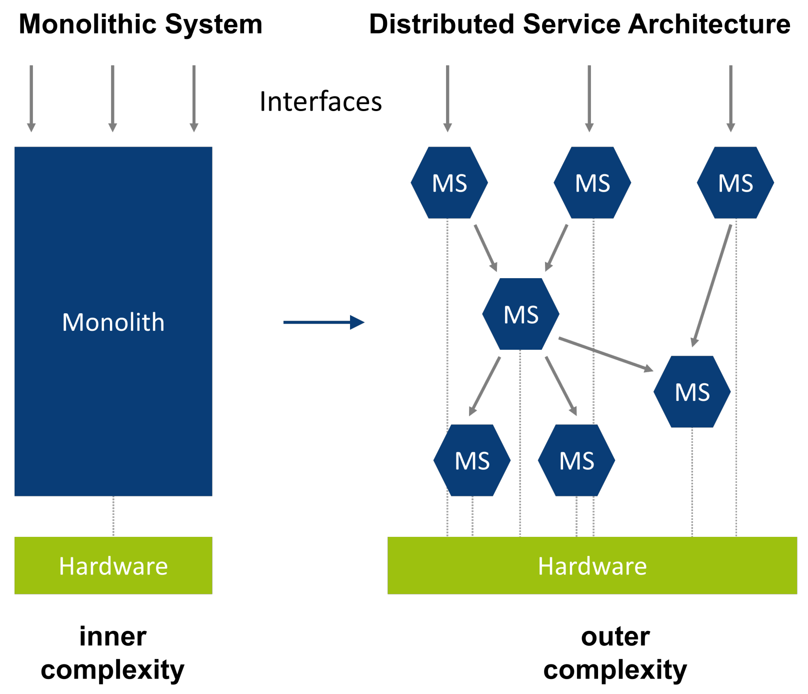


Figure 1.1: Transition from a monolithic system to a Microservice-based architecture

From 2015 on, more and more major global players announced to reorganize their IT systems by converting their monolithic systems into Microservice-based architectures. Netflix was one of the first companies to bet on Microservices and was quickly followed by Amazon, Walmart, Spotify and many others. In a 2016 survey by the German software company LeanIX, more than 100 IT decision-makers from leading companies in the US and Europe were asked for their companies' opinion on Microservices [LeanIX, 2016]. 80% of the surveyed companies planned a transition to Microservices or had already achieved it, while companies that relied mostly on Microservices were able to release new software significantly faster than those not using Microservices.

1.2 Microservice Monitoring and Visualization

While Microservice-based architectures provide a wide range of benefits, they also come at the cost of a highly complex system. A large application may easily consist of several hundred Microservices. This is a result of the transition from the inner complexity of a large, monolithic application towards the outer complexity between individual Microservices performing all the distributed tasks. Due to this complexity, architecture monitoring, recovery and documentation are more challenging than for a monolithic system. Different monitoring approaches have been developed to tackle this issue, but they often lack the connection between the IT infrastructure and business processes [Kleehaus et al., 2018].

At the chair for Software Engineering for Business Information Systems (SEBIS) at TU Munich, a research project is centered around a new tool for monitoring and analyzing distributed Microservice architectures. An Architecture Discovery was developed, which automatically detects Microservice instances and matches them to existing business processes. As a resulting artifact, it creates a an adjacency matrix containing all Microservices, their instances and associated hardware as well as any communication between services. After manual annotation of business processes and mapping to the corresponding user interfaces, business processes are also added to the adjacency matrix.

For a limited number of entities included in the adjacency matrix, the resulting table may directly serve as a depiction of the underlying Microservice architecture. However, with growing data set size, the adjacency matrix quickly becomes complicated to use as read-out. To allow thorough monitoring of an IT architecture and its associated business layer, a meaningful visualization is needed.

To this end, the present thesis describes the conceptualization and implementation of a prototype visualization tool. Based on the needs and interests of different Enterprise Architecture stakeholders, several views on the underlying Microservice architecture are drafted and subsequently realized. Varying aggregation and magnification settings allow to monitor services at the respective level of detail for each use case. Evaluation of the implemented prototype employs a lab environment with a basic Microservice infrastructure to test for functional correctness with increasingly complex data sets.

2 Background

2.1 Microservices

Microservices are a modern architectural design concept being the latest evolution of the well known Service Oriented Architecture (SOA) [Newman, 2015]. As their name already indicates, Microservices still specify as services as defined in the SOA context. But while SOA services provide sometimes large and complex functionality, Microservices break down processes into much smaller generic entities. Such an entity performs only a single independent and isolated business task and in turn delegates more complex tasks further on to other Microservices. Communication among Microservices is typically handled through HTTP web requests using REST as a unified data exchange format. These isolated tasks are executed on separate run-time environments each.

The main difference between Microservices and the original SOA concepts lays in the fact, that Microservices often already know about each other and how to use the the other parties interfaces, while in a SOA scenario services by design need to be discovered at first during run time. Such a Repository and discovery structure might indeed be also implemented in a Microservice based environment but this is far from an obligatory prerequisite [Daya et al., 2016].

Microservices decompose code of large applications into small, separate functional units. Each Microservice has clearly defined boundaries, tasks and dependencies. This forces a low coupling and a high encapsulation. Breaking down large and complex application logic into such functional pieces helps to reduce the com-

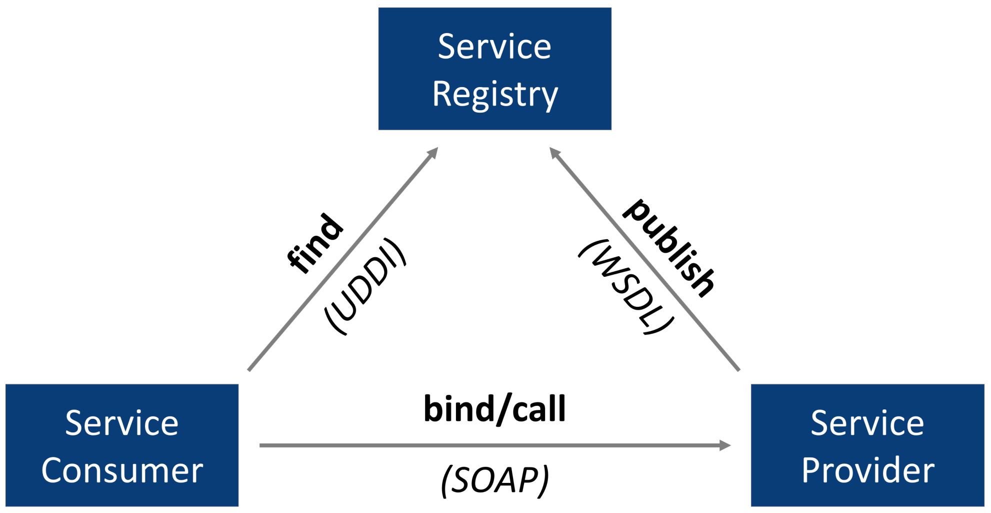


Figure 2.1: Service Oriented Web Service Architecture along the definitions in [Newcomer et al., 2004] and [Matthew MacKenzie et al., 2006]

plexity and interdependency of code and therefore results in appreciable shorter development cycles. Especially parallel development of Microservices is enabled and dependencies are also reduced in terms of software testing [Bakshi, 2017].

This development totally fits the currently prevalent tendency towards more agile software development processes to live up to the demand of shorter release intervals and more flexibility [Fowler and Lewis, 2014].

Not only during the initial design and development stages of Microservices, but also during later maintenance and potential code revision phases the benefits of well-defined Microservices come into play. If a single Microservice is being reworked, only this particular entity needs to be redeployed as long as the provided service interface stays unchanged. Particularly in large settings with various Microservices, this enables nearly continuous operation of business, minimizing downtime by preventing extensive redeployment of largely unchanged code.

As each Microservice is highly specialized on a very specific task, there is a large variation in software architecture, run-time environment and external dependencies (such as databases or other data storage solutions) among Microservices to match their respective needs. Some services may for example utilize classic relational databases while others will benefit from the use of noSQL or time series databases [Hasselbring, 2016].

Similarly, requirements regarding the technical features provided by their host system may differ widely between Microservices, from processing power to large RAM allocations and capabilities to extensive parallel calculations. Thus, optimization of existing Microservices can not only be performed on code level, but also on hardware and visualization levels. In general, different Microservices might even be based on different programming languages as long as they are capable of providing a REST API to interact with.

Based on this possible heterogeneity in terms of technology, Microservices enable developers to experiment with new technologies in a small and restricted environment without needing to go along with or to abandon existing technologies.

2.1.1 Scalability

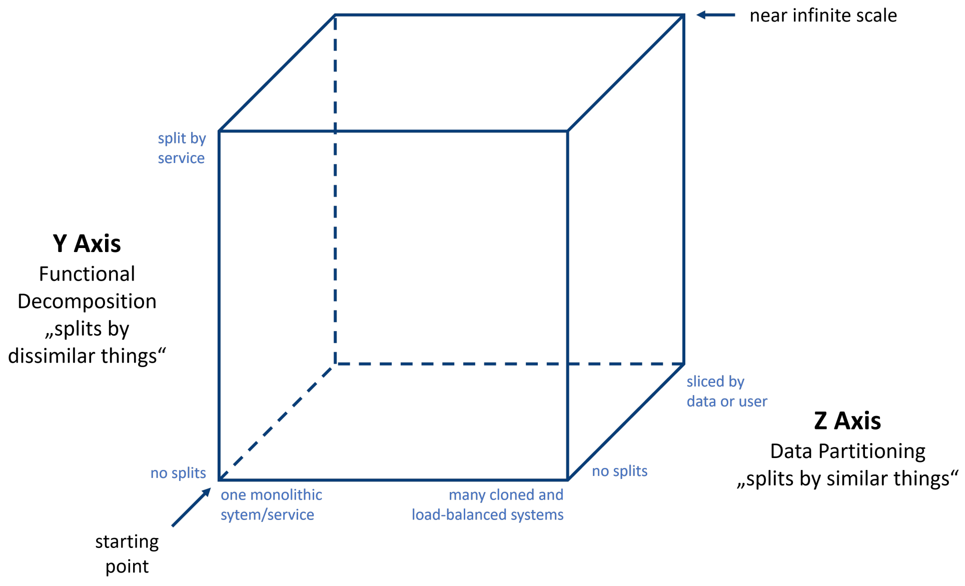


Figure 2.2: AKF scale cube adapted from [Richardson, 2017]

Compared to monolithic systems, a Microservice Architecture allows higher flexibility in terms of scalabilty [Bradley, 2015]. Each Microservice may run on very specific selected hardware (typically on top of virtualization) fitted tightly to the Microservice characteristics. Thereby, performance of single Microservices can be highly optimized. While in traditional, monolithic architectures a performance bottleneck resulted in an upscale of the whole application, in a Microservice architecture only a single functionality needs to be re-scaled.

Regarding the AKF scale cube introduced by Abbott [Abbott and Fisher, 2015], scaling can be performed into three different dimensions, which are illustrated along the edges of a cube (fig.2.2). A duplication or cloning of a system resembles an X-Axis scaling, which is also called horizontal scaling. The second, vertical dimension (Y-axis) relies on functional decomposition and therefore breaks down a large system into smaller, functional parts. Finally, the third, orthographic dimension (Z-axis) is based on data partitioning by contextual factors.

Microservices by design already represent a variant of Y-axis scaling, as they are in fact already a functional decomposition of a potentially larger monolithic system. Furthermore, the use of Microservices reduces the effort for X-axis scaling as well as for Z-axis scaling.

Both types, x- and z-axis scaling can be performed on Microservice level and therefore allow a precise targeting of bottlenecks. Operating multiple instances of a single Service runs smoothly and plain load balancing between these instances represents an X-axis scaling. In contrast, a data partitioning between multiple instances of the same Service would result in a Z-axis scaling [Richardson, 2017].

2.1.2 Complexity

Monolithic applications come along with high code complexity and require large maintenance efforts over time. In contrast, each individual unit in a distributed Microservice environment covers comparably low complexity. In return, a high distribution level and short release cycles on Service-level and therefore frequently changing version constellations lead to a considerably higher complexity on Architecture Level. In fact, changing a monolithic architecture into a Microservice architecture does not reduce the overall complexity of the whole system. Instead the added abstraction and communication layers might even further complicate the system.

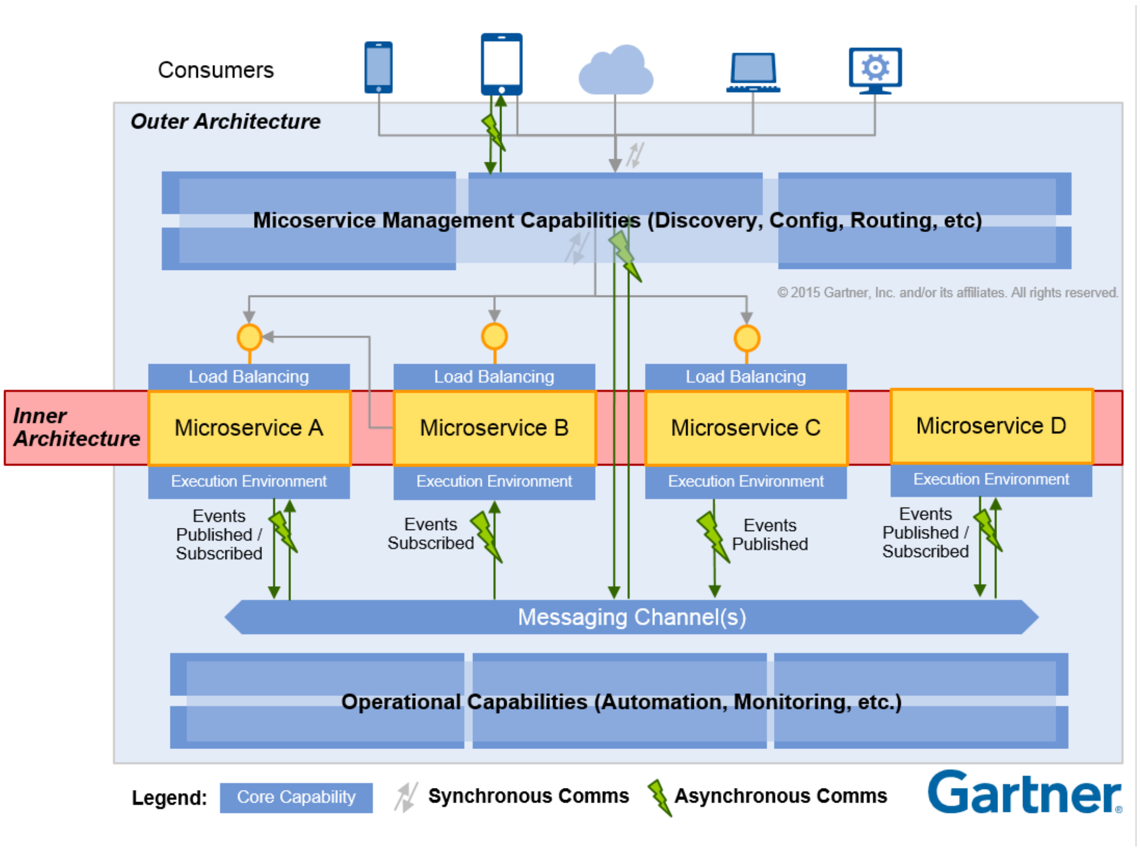


Figure 2.3: Breakdown of Inner and Outer Architecture in a Microservice Architecture Environment [Olliffe, 2015]

Gary Olliffe, Research Director at Gartner, framed a reasonable differentiation between inner and outer architecture of Microservices [Olliffe, 2015]. The *inner architecture* correlates to the implementation architecture within a specific Service, whereas the *outer architecture* covers all inter-service communication and delegation logic, which is further complemented by all efforts required to run and maintain service instances.

Reduced complexity of the inner architecture is what is mostly considered as a main benefit of a distributed service architecture that leads to more efficient development. In fact, it is just shifted towards the outer architecture, where it needs to be addressed likewise. The skills to establish and operate such a distributed architecture are nowadays combined into the job of DevOps, whose assignments involve developmental tasks as well as operational tasks [Wootton, 2014].

2.1.3 Microservices in Action

In modern technology fields, companies often share their technology insights in blog posts, forums and publicly released tech talks. This is how several large tech companies have openly described their movements towards a Service Oriented Application Landscape.

Netflix, a major video streaming provider, was one of the first global players to establish a Microservice architecture. They extensively covered their transition and a bunch of standard use cases and best practices arose from their pioneer work in modern Service-Oriented Architectures [Mauro, 2015a]. Netflix Director of Web Engineering and then Cloud Architect Adrian Cockcroft emphasizes the need for clear modularization and defined boundaries on all levels, ranging from separate data stores for each Microservice through separate builds for individual Microservice to small software development teams. Netflix open-sourced many of their tools, thereby actively contributing to further development of Microservice architectures. Examples are Netflix Eureka as a Microservice Repository and Netflix Conductor, a central orchestrator that manages up to multiple million process flows in a complex Microservice environment [Baraiya and Singh, 2016].

The disruptive mobility intermediary *Uber* started their business with a classical monolithic system, which sufficiently met all requirements as long as they were a small company focusing on a single product in a single city. During their rapid growth, they realized they had to break up their complex system into a distributed Service-Oriented Architecture. Uber Engineering described their movement from a monolithic application towards hundreds of Microservice in late 2015 [Haddad, 2015]. While taking the opportunity of redesigning their application landscape for the purpose of a major cleanup of their codebase, they also faced some newly emerging problems during this extensive transition. For example the huge number of single services led to a lack of obviousness, which they tackled by introducing clear communication standards like mandatory Requests for Commons before introducing a new service to avoid code duplications [Reinhold, 2016].

Other streaming industry players such as *Soundcloud* [Calçado, 2014] and *Spotify* [Novoseltseva, 2017] likewise adjusted their IT strategy, decomposing their year-old monoliths into novel small and independent services. Finally, huge commercial players like *Amazon* [Novoseltseva, 2017] and *Walmart* [Webber, 2016] with vast IT landscapes are turning the tide. They fixed their by now overcharged systems by shifting towards agile development in manageable small teams.

2.2 Enterprise Architectures

In order to successfully pursue business, companies operate a large number of IT systems, applications, digital services and processes. While use of and responsibilities for those systems may be spread throughout the company, top level management still needs to be able to keep track of the entire IT landscape and its development. This top-level view of Enterprise IT is called the Enterprise Architecture. Associated roles and tasks managing the IT landscape are organized as the Enterprise Architecture Management.

2.2.1 Enterprise Architecture Frameworks

Throughout the last three decades, several Enterprise Architecture Frameworks emerged in order to establish a defined set of terms and relationships in the field of Enterprise Architecture. The Zachman Framework was published in 1987 by IT consultant John A. Zachman [Zachman, 1987] and served as a blueprint for multiple following definition sets. Further development of Enterprise Architecture Frameworks was largely driven by US federal agencies introducing the Technical Architecture Framework for Information Management (TAFIM) and the Department of Defense Architecture Framework (DoDAF) by the United States Department of Defense [Department of Defense, 1996, DoD, 2012], the Treasury Enterprise Architecture Framework (TEAF) by the US department of the treasury [Department of the Treasury, 2000] and the Federal Enterprise Architecture Framework (FEAF) by the Federal government of the United States [Council, CIO, 1999].

Another popular framework, The Open Group Architecture Framework (TOGAF), which is in turn based on TAFIM, is developed by The Open Group, a large consortium of businesses and government agencies [The Open Group, 2018a]. TOGAF is nowadays extensively used by large companies around the world, being established at most of *Global 50* and still the majority of *Fortune 500* [The Open Group, 2018b].

Those Frameworks were mostly developed in parallel, adopt features from each others and offer slightly different artifacts, displayed from various points of view, leading to a distinct fragmentation of Enterprise Architecture results [Urbaczewski and Mrdalj, 2006].

2.2.2 ISO 42010

The international standard ISO 42010 was introduced by IEEE in 2011 [ISO42010, 2011]. By defining fundamental terms and abstraction layers, it finally gave a common basis for the scientific discourse as well as for practical implementations and advancements. The standard deals with stakeholders, views, models and relations. Additonally, a comprehensive consideration of Enterprise Architecture Frameworks helps to describe Enterprise Architectures on a basis of mutual understanding.

In the context of ISO Standards, the term *system* is formulated as a broad concept and embraces not only hard- and software, but also includes any configurable and man-made item. This means that processes, procedures, facilities and even human beings are covered by the definition [ISO15288, 2015].

Enterprise Architectures describe a set of systems and their relations among each others. This description proofs to be helpful throughout the whole life cycle of an IT system. It is not only used through planning and implementation phases, but also during operation, maintenance and evaluation periods up to an eventual disassembly stage and potentially a final replacement of the system.[ISO15288, 2015, ISO24748, 2011]

2.2.3 Stakeholders

The Enterprise Architecture does not only describe the IT systems of a company, but also includes the environment surrounding the systems. As such, several different roles and responsibilities in a company are relevant to the Enterprise Architecture.

ISO 42010 gives a rough overview of important stakeholders that should be covered by an architecture description. In this ISO context, all stakeholders are considered in their relations towards the system to be described.

First of all, a system is applied the **users** actually using it, while someone else is responsible for continuous **operation** of aforementioned system. Furthermore, someone **owns** it, whereas potentially another stakeholder **acquired** the system beforehand. Additionally, the system is in turn **supplied** by potentially yet another stakeholder.

From a technical perspective, someone **planned and built** the system, while **developers** actually implemented the system and **maintainers** ensure an ongoing operation and perform inevitable adjustments.

For each of those different stakeholder roles, different concerns are relevant to the architectural description. Varying views in an Enterprise Architecture description allow a satisfying fulfillment of these concerns [ISO42010, 2011].

2.2.4 Architectural Layers

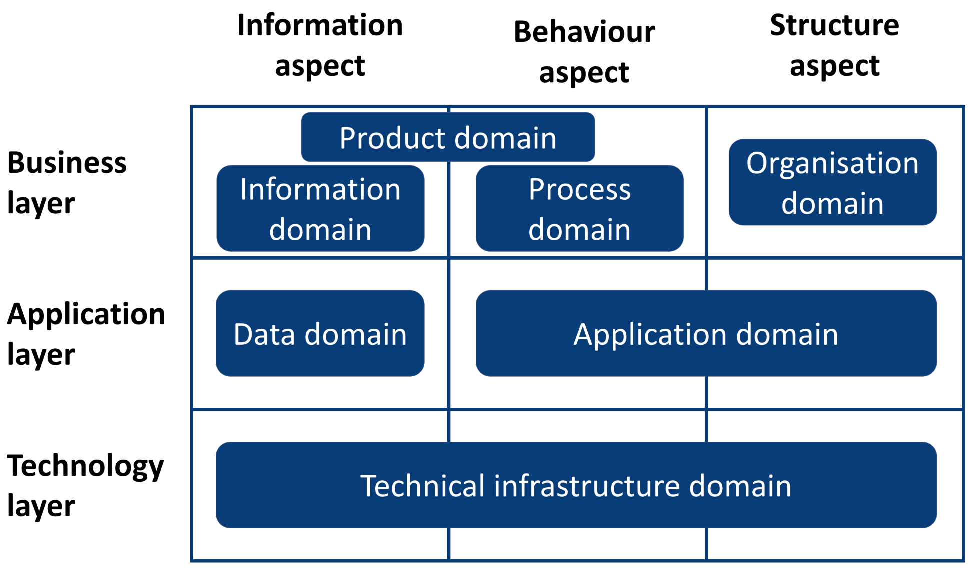


Figure 2.4: Enterprise Architecture layers reproduced from [Jonkers et al., 2003]

In order to structure an Enterprise Architecture in a meaningful way, it is a common approach across the various Enterprise Architecture Frameworks to break an Enterprise Architecture description into three hierarchical architectural layers. These layers differ in scope, abstraction level and information granularity [Jonkers et al., 2003].

On each layer, a set of views examines the respective layer from different angles, fulfilling diverse requirements and needs. From Top to Bottom, layers gradually switch from a pure business perspective towards a solely technical representation. Each layer tackles different topics, also called architectural domains. In the following, the most common layers are shortly described.

Business Layer

The Business Layer is the topmost layer in an Enterprise Architecture. It influences and shapes almost any other of the more technical layers. The business layer is generally the point of origin of all actions undertaken in order to perform as a business. On the business layer, all business-relevant elements are taken into account, covering four different architectural domains.

The organizational domain contains all structural aspects of entities acting in the company as well as their roles organizational setting.

The Process domain deals with all business activities leading to an added value, for example by making products or offering services. In contrast, the Product domain covers all information about the actual products and services and how they shape the overall business behaviour.

The information domain forms the fourth and last domain on the business layer, dealing with all business-relevant information. [Jonkers et al., 2003].

Application Layer

As second important layer, the application layer is the conjunctive link between business and technology worlds.

Its informational aspect is covered by the data domain, which contains all automatically processable information and serves as provider for the business information domain as well as the application domain.

The application domain in turn is responsible for all applications and IT systems operated. These applications in turn support and enable the business processes. The application domain only handles the software side of applications [Jonkers et al., 2003].

Technology Layer

The third and lowermost layer is the technology layer, which is the foundation for the operation of any applications. Only a single domain is covered by the Technology Layer.

The Technical Infrastructure Domain describes all elements providing a functional environment in order to run applications. This includes not only hardware to provide processing power but also virtualization, network and communication infrastructure [Jonkers et al., 2003].

Other Layers

In addition to those three basic layers, some Enterprise Architecture Frameworks further distinguish between business and application layer by introducing a few supplemental layers.

As such, a separate Process layer is introduced to segregate all process related information from the business domains. This new layer handles all matters concerning the implementation and operation of services in order to increase efficiency.

As another additional layer, the Integration layer specifically takes care of the application structure and in general of interactions among applications. The Integration layer helps to evolve a friction-less application implementation [Winter and Fischer, 2006].

2.2.5 Artifacts

A set of artifacts is typically present across Enterprise Architecture Frameworks, helping to describe different domain needs. Along the aforementioned scheme, artifacts can be matched to layers and specific domains. They describe facets of the overall IT landscape and usually confirm the corresponding specifications. As such they comprehensively specify an object of interest from different viewpoints.

As the following artifacts are a superset of multiple Enterprise Architect Frameworks, including all of the specifications in an actual Enterprise Architecture description might result in a redundant and too detailed description. Therefore, several of the artifacts should be slimmed, or moved to lower level architectural descriptions such as for example the software architecture, for an actual Enterprise Architecture implementation.

Strategy Specification

Domiciled in the business layer, the strategy specification covers all matters concerning the company strategy. A companies' strategy specification is defining *what* the company is doing and in which direction it is headed. This does not only include organizational goals and success factors but extends to marketing strategies, mission statements, organization-wide values and business principles [Winter and Fischer, 2006].

Organization and Process Specification

Another artifact of the business layer are the organizational and process specifications. This guideline specifies *how* business should be undertaken. These specifications particularly describe how the organization is structured into business units, their hierarchies and how these units should work and collaborate among each other. Such a specification should also cover metrics to evaluate the overall performance [Winter and Fischer, 2006].

Application and Software specification

In order to support enterprise business activities, the IT alignment is specified in the application specification, emerging from the application layer. It tackles the specification of services, applications and their respective components to support a reasonable application landscape.

The Software specification in turn is describing the same information for the software components inside one application and their functionality as well as the relations to data and input sources and interfaces to provide data.

Technical Infrastructure specification

For the Technology layer a specific description of IT components and actually used hardware meets the relevant needs.

Dependency specification

Additionally to the layer-specific artifacts, there is another set of layer-independent artifacts. These specifications explicitly describe the relations and dependencies between the different layers or at least between multiple specifications.

Possible examples of such dependencies could be an application ownership between an organizational unit (Business layer) and an individual application (Application Layer), the hardware and infrastructure (Technology Layer) usage of a specific Software component, or the relation between strategic goals (Strategy Specification) and performance indicators or metrics (Organization Specification) to track progress towards the goal [Winter and Fischer, 2006].

These cross-specification-dependencies are especially valuable, as they ensure a comprehensive consideration of different aspects in relation to each other.

2.3 Distributed Tracing

As the deployment of large-scale distributed systems catches on lately, a particular need for monitoring and controlling tools emerged. On a classical monolithic system, it is often sufficient to inspect a single log-file in order to identify and understand problems. In contrast, the complexity of the same task can be several magnitudes higher within an environment of distributed systems. Log-files from different machines, potentially running on totally different technical environments, need to be evaluated. Additionally, the sequence of procedures needs to be retraced and matched among multiple services. This requires a re-identification of calls on different systems, for example by matching call time-stamps. Asynchronous calls additionally complicate things, as they can only be reconstructed using the beginning time stamps.

Different solutions that tackle the task of tracing calls in distributed systems have emerged over time. One solution approach has been presented by Google in its Dapper paper [Sigelman et al., 2010]. The core product of Google, the Google Search Engine, is a large construct of sometimes more than a thousand distributed systems involved responding to a single search request. Naturally, there exists a particular interest in understanding relations and activity sequences among this vast amount of Microservices.

Two especially important characteristics of such a distributed tracing tool were identified by Google in the process of designing Google Dapper.

Firstly, distributed tracing needs to allow a ubiquitous deployment, which in turn ensures consistent monitoring without interruptions as far as each and every Microservice can actually be monitored.

Secondly, continuous monitoring of Microservices is a crucial requirement, enabling the monitoring to actually observe anomalies and performance issues on the go, as these might be exceedingly hard to reproduce in hindsight.

In order to meet these two high-level requirements, four major design goals provide a solid foundation for Google Dapper:

Low overhead

Performance is crucial, especially for front-end applications, where response times heavily govern the bounce and exit rates. Therefore, it is essential that any monitoring solution produces little to no additional load, minimizing the negative effects for end users. A too heavy monitoring implementation may easily exceed its benefits by producing too much load.

Application-level transparency

The distributed tracing implementation must not obstruct Microservice development in any ways. As a result, developers should not even need to be aware of distributed tracing and developers should not at all be required to perform any steps or implement and add any code in order to enable distributed tracing. Quite the contrary, distributed tracing should be a ubiquitous feature and tool that does not produce any hassle in development and maintenance.

Scalability

The striven solution for distributed tracing needs to be freely scalable to allow systems of vastly different sizes to successfully perform distributed tracing in their respective system architecture. This may range from small applications relying on only a few distributed services to large-scale products such as the Google search engine, which is based on thousands of services of different size.

Fast data availability

In order to deduct meaningful data from a distributed tracing, it is of particular importance to make the gathered data available as soon as possible.

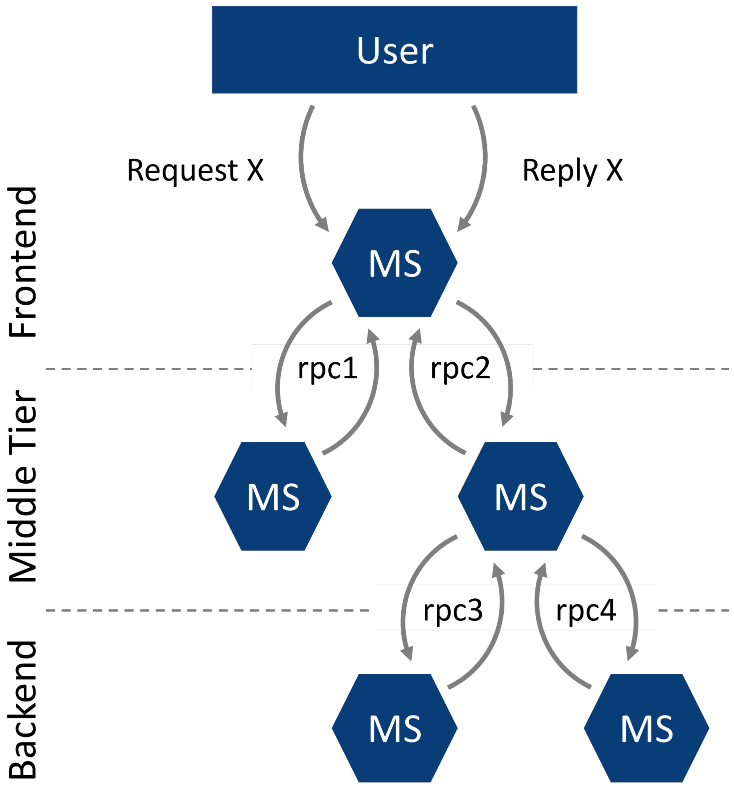


Figure 2.5: Communication sequence of a Microservice based system as example for Google Dapper [Sigelman et al., 2010]. Adapted from [Sigelman et al., 2010]

Figure 2.5 illustrates an exemplary user request in a Microservice based architecture. If all Services in this example are instrumented in a distributed tracing environment, each service will report its own call to the distributed tracing server in form of a span. Each span contains information about the service itself, the id of the parent Service that invoked the method call, the called method inside the service and exact timings from request begin to end. Each span sent by the individual Microservices contains the same *trace id* which matches to a single user request and allows a recognition of related span. Based on this tracing id, the contained parent span id and the annotated timings, it is possible to reconstruct the complete call sequence of such a request as displayed in fig. 2.6 [Sigelman et al., 2010].

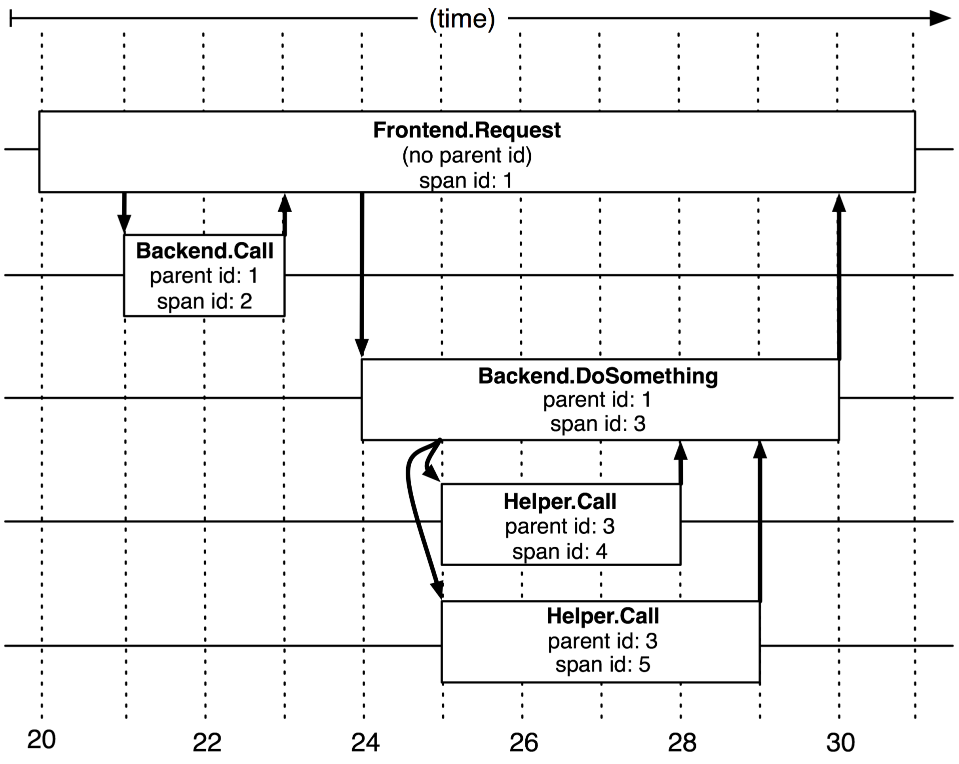


Figure 2.6: Sequential course of the exemplary user request displayed in fig. 2.5 [Sigelman et al., 2010]

3 Related Work

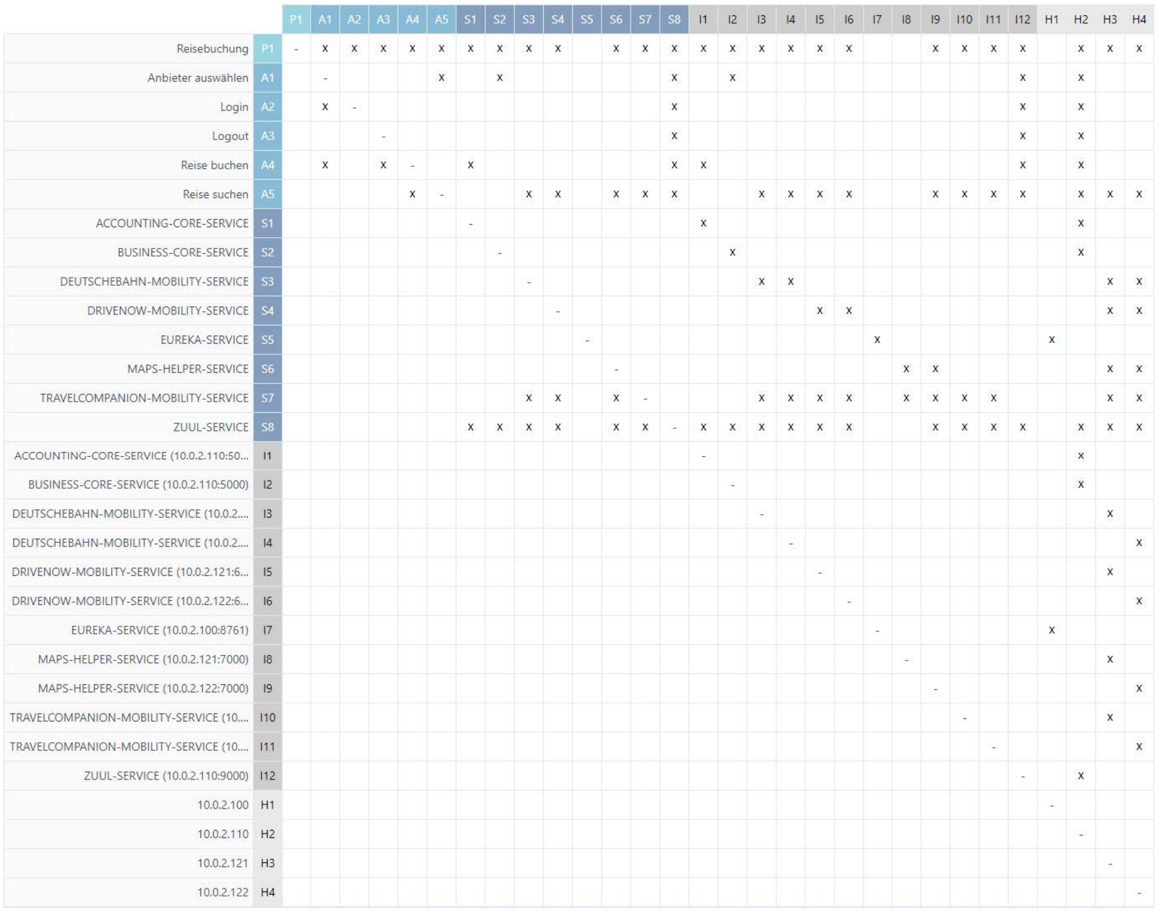
During the realization of this thesis, two other master thesis were written dealing with different aspects of the same Microservice Test Environment. The concept of this thesis is largely built upon the results of those other two works. Another third thesis has just recently been started and will also contribute to the whole complex as described in section 3.3.

3.1 Service Architecture Discovery

In his 2017 master thesis, Patrick Schäfer [Schäfer, 2017] presents a possible procedure to automatically discover Microservice Instances and manually match them to existing Business Processes. The presented solution does not only detect Microservice Instances, but also processes inter-service communication by evaluating the distributed tracing spans produced by zipkin. This helps to gather data about service usage and to enrich this data set with their associated communication dependencies.

In order to reconstruct the request structure, the distributed tracing library enables services to report their call spans as described in 2.3. All these reports from the Microservices may be reassembled afterwards by matching the trace identifier, which is included in each span. Based on this information, not only the order of requests is traceable, but also the timings of services, as well as all instrumented inter-service communications. This information is further useful for the purpose of an Anomaly Detection which is described in section 3.2.

Based on the in this manner observed spans, the prototype designed by Schäfer spreads an adjacency matrix. This adjacency matrix contains all Microservices,



		P1	A1	A2	A3	A4	A5	S1	S2	S3	S4	S5	S6	S7	S8	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10	I11	I12	H1	H2	H3	H4
Reisebuchung	P1	-	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x	x	x	x			x	x	x	x		x	x	x
Anbieter auswählen	A1		-				x		x						x		x										x		x		
Login	A2		x	-											x												x		x		
Logout	A3				-										x												x		x		
Reise buchen	A4		x		x	-		x							x	x											x		x		
Reise suchen	A5					x	-			x	x		x	x	x			x	x	x	x			x	x	x	x		x	x	x
ACCOUNTING-CORE-SERVICE	S1							-								x													x		
BUSINESS-CORE-SERVICE	S2								-								x												x		
DEUTSCHEBAHN-MOBILITY-SERVICE	S3									-								x	x											x	x
DRIVENOW-MOBILITY-SERVICE	S4										-									x	x									x	x
EUREKA-SERVICE	S5											-										x						x			
MAPS-HELPER-SERVICE	S6												-										x	x						x	x
TRAVELCOMPANION-MOBILITY-SERVICE	S7									x	x		x	-				x	x	x	x		x	x	x	x				x	x
ZUUL-SERVICE	S8							x	x	x	x		x	x	-	x	x	x	x	x	x			x	x	x	x		x	x	x
ACCOUNTING-CORE-SERVICE (10.0.2.110:50...	I1															-													x		
BUSINESS-CORE-SERVICE (10.0.2.110:5000)	I2																-												x		
DEUTSCHEBAHN-MOBILITY-SERVICE (10.0.2....	I3																	-												x	
DEUTSCHEBAHN-MOBILITY-SERVICE (10.0.2....	I4																		-												x
DRIVENOW-MOBILITY-SERVICE (10.0.2.121:6...	I5																			-										x	
DRIVENOW-MOBILITY-SERVICE (10.0.2.122:6...	I6																				-										x
EUREKA-SERVICE (10.0.2.100:8761)	I7																					-						x			
MAPS-HELPER-SERVICE (10.0.2.121:7000)	I8																						-							x	
MAPS-HELPER-SERVICE (10.0.2.122:7000)	I9																							-							x
TRAVELCOMPANION-MOBILITY-SERVICE (10....	I10																								-					x	
TRAVELCOMPANION-MOBILITY-SERVICE (10....	I11																									-					x
ZUUL-SERVICE (10.0.2.110:9000)	I12																										-		x		
10.0.2.100	H1																											-			
10.0.2.110	H2																												-		
10.0.2.121	H3																													-	
10.0.2.122	H4																														-

Figure 3.1: Adjacency matrix of the test environment at SEBIS chair TUM. This adjacency matrix already contains observed communications between Microservices as well as manually modelled Business Activities [Schäfer, 2017]

their instances and even the associated hardware on which the instances are operating. Schäfer does not distinct between virtual and physical hardware, as services and service repository are equally unaware if they are running on virtual systems. The mentioned entities are each placed along x and y axis at the same time. The actual fields inside the adjacency matrix are marked with an *X* for each call that was ever observed by the system between the corresponding x and y-axis nodes. This marker stays unchanged, independent of any additional communications between the two nodes concerned.

The only way a marking can be removed occurs when one of the involved nodes is recognized as unavailable, which happens for example at a shutdown or redeploy of a Microservice, its instances or their underlying Hardware. This prevents the adjacency matrix from keeping outdated information that was indeed observed at least once before, but might be no longer valid due to code or infrastructure changes.

To further amend this observed data set, Schäfer offers a tool to manually annotate business processes and their associated business activities steps and map them to the corresponding user request interfaces. By this data model enhancement, connections between a planned and modelled business process and an actually implemented and observed Microservice architecture are established.

Business Activities are added to the scales of the adjacency matrix in line with the already observed components. An example of the resulting adjacency matrix for a small distributed Microservice architecture is displayed in figure 3.1

3.2 Anomaly Detection Based on Distributed Traces

In another 2017 master thesis, Lukas Steigerwald also contributed to the distributed Microservice architecture research project at SEBIS chair, TUM [Steigerwald, 2017]. Steigerwald uses the resulting spans of a distributed tracing tool such as zipkin for a real-time analysis. Utilizing three different analysis algorithms, Steigerwald performs an anomaly detection in order to identify Microservice malfunctions or performance losses during runtime.

By further analyzing the gathered data, a root cause anlysis is performed, revealing the actual originating Microservice that is responsible for a system malfunction or broader performance losses. The results of theses analyses are streamed in real-time to the distributed streaming platform kafka, where other services may consume the gained information. In figure 3.2, a service outage of Service A and B is observed, while the root cause analysis identified Service E as the actual point of failure.

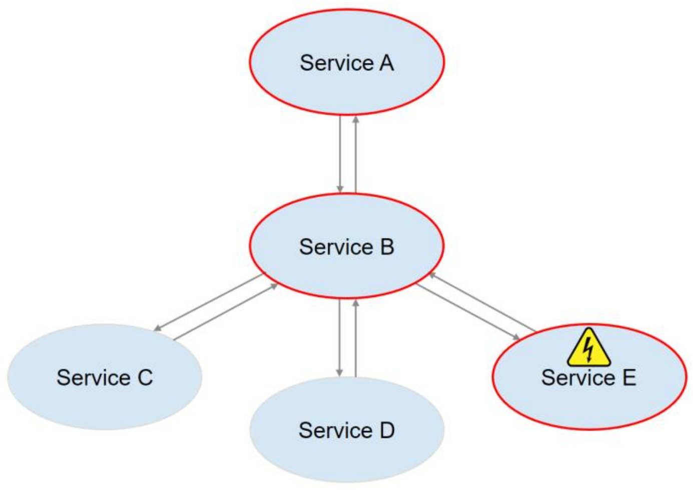


Figure 3.2: A failing Service E caused subsequent failures in Services A and B. The root cause analysis could nevertheless identify Service E as the actual point of failure [Steigerwald, 2017]

3.3 Enhanced Enterprise Architecture Model

There is ongoing research at SEBIS chair at TUM in the field of enhanced Enterprise Architecture models. Specifically, Christopher Janietz has recently started his master's thesis, in which he is developing a solution in order to map high level enterprise architecture objects such as domains and products to low level monitoring information from Microservices [Janietz, 2018]. This combines the proposals of Schäfers work discussed in section 3.1 with the concept of Domain Driven Design and will presumably result in a conceivable data source for the prototype presented in this thesis.

4 Solution Approach

The main focus of this thesis is the visualization of a clear, high-level overview of a distributed Microservice architecture. For this purpose, information from several data sources is collected and aggregated to a more comprehensive summary and visualization. By this means, information with different levels of detail is brought together into a multilayer picture.

4.1 Entities

The in this thesis proposed solution is able to visualize various facets of a living enterprise architecture. The following chapter gives a rough overview of the different entities tackled and characterized by the proposed solution.

Domain

Modern IT landscapes are often structured by their business domain contexts. Inline with Domain Driven Design [Evans, 2004], a separation of concerns helps to partition a large business in smaller functional units, with delegated tasks and responsibilities each. This also allows a clustering of IT infrastructure, business applications and users into such sections with high coupling and dependencies within each one itself, while staying largely independent among each other. These clusters are in turn called domains, providing separate services and products inside the enterprise context, living up with an enterprise strategy and in the long run accounting for financial and competitive market success of a company.

Product

Within each Domain, functionality is again subdivided into smaller parts called Products. This Product partitioning reflects the actual department structure within the domain. Thus, all functionality of one Product is developed by a single development team. In modern agile teams, business accountability and operations are settled alongside in the same team to advocate a culture of short distances and flat hierarchies.

Microservice

Before the adoption of Microservices, each product would have been home to one or more monolithic services. In contrast, through the transformation towards Microservices these occasionally enormous monolithic systems are broken down into a more or less complex network of Microservices communicating among each other (see 2.1).

By decomposing the systems into smaller items, a cross-Product utilization of functionality is suddenly an ease and the enterprise can actually benefit from this newly gained Microservice reusability.

Instance

Either for the purpose of scaling (see 2.1.1), to ensure higher resilience or simply in order to reduce latency times from different places, multiple Instances of a Microservice may be operating independently at the same point in time.

Each instance operates in its own runtime environment and is approachable at its own address and port combination, even if this fact might be hidden behind a load-balancing mechanism. Besides the address, multiple Instances of the same Microservice offer the exact same interfaces providing the same functionality.

API

Each Microservice in turn provides one or more Interfaces to programmatically use its features. These APIs are typically offered as HTTP REST Interfaces, potentially consuming parameters and allowing read, write or manipulation tasks.

Within each Product, some of the Microservices offer public Interfaces that initiate the core functionality. These public Interfaces are the official Endpoints of a Product. External Microservices from other Products may still call any specific Interface of any Microservice in order to use its features.

Microservice Relation

A relation between two Microservices bases on a caller-callee contact between two Microservices. In cases of a recursive call, caller and callee could potentially also be the same Microservice. A call is a single HTTP request on one REST interface in order to trigger an action. These calls might either be synchronously waiting for its answer, asynchronously continuing with their respective work and processing the actual response at a later point in time or only trigger another Microservice action without expecting any response.

4.2 Data Sources

As described in 4.1, domains are the top-level entities inside an Enterprise Architecture. The organizational unit that is responsible for the IT planing and Management (e.g. a designated IT Management department) typically already uses a designated Enterprise Architecture Management tool (EAM) in order to fulfill its responsibilities. Thus, details about user and domain structure can be drawn from the corresponding Enterprise Architecture Management Tool.

Alongside the domain structure, data concerning applications and the actual Microservices as well as their relationships is added. Data about currently active

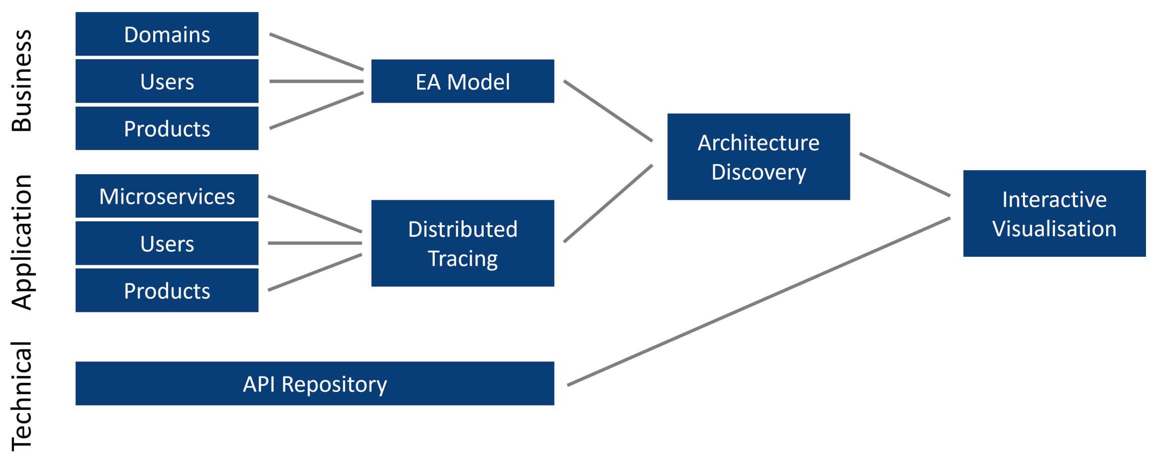


Figure 4.1: The aggregation strategy in order to integrate all required data

Microservices can be gathered from a Microservice Directory Service. Supplementally, performing an architecture discovery helps to learn about relations among Microservices. The information about Microservices can additionally be enhanced with details about the underlying virtualization and hardware layers and their properties. Such data could be retrieved from a architecture discovery service as proposed by Kleehaus [Kleehaus et al., 2018].

As an adequate complementary step, application and technical Microservice data should be combined with the Enterprise Architecture Model. In this case, Applications and Microservices could be matched to their functional business domains as described above. In this way, augmented data sets allow a thorough investigation of an operating environment of distributed Microservices.

Finally as a third, even more low-level data source, functional data about each individual Microservices API can be attached. Modern tools like Swagger are able to extract the signatures of all interfaces a Microservice provides directly from its code. While this does not fully substitute a contextual description of the Microservice, it still provides enough insights to build a basic and useful living documentation.

A main benefit of this data aggregation is based on the possibility to integrate the whole process into a fully operating distributed Microservice environment. Such an integrated analysis solution allows a continuous and mostly automated assessment of the actually existing Enterprise Architecture, in contrast to a one-time analysis which does not allow re-evaluations after adjustments have been made without manually re-iterating the complete analysis process.

Simultaneously, the proposed solution provides a set of helpful tools, instruments and reports to perform a monitoring of the living Microservice environment based on the exact same data sources.

4.3 Stakeholders

The proposed system serves as a valuable data-source for a variety of stakeholders, depicting a Microservice Architecture in its entirety. Nevertheless, each stakeholders role implicates another point of view on a corresponding subset of the data, ranging from high level and non-technical business perspectives down to solely technical interests into Microservice relations and application interfaces. The following summary of roles with illustrates the breadth of the range of varying potential interests.

Enterprise Architect

An Enterprise Architect inherently needs to keep track of the organizations overall IT landscape. To enable an as frictionless as possible interplay of IT systems, the Enterprise Architect especially needs to keep an eye on cross-domain relations. His responsibility does not explicitly lie on preventing such cross-domain relations, but rather on design beforehand and tailoring the IT landscape along the way towards a balance between isolated domain silos and strongly coupled and interdependent domains.

Domain Owner

Within a specific domain, a classification of separate products and their boundaries needs to be performed and thereupon further monitored. In an agile organization environment, Product teams shall act widely autonomous and come to their own decisions. Therefore, a Domain Owner tracks the coupling between products from a technical perspective, while managing the partitioning of Products from a business perspective.

Product Owner

The interests of a Product Owner lie in particular on the overall performance of his product. Thus, he is responsible for the functionality made available by his product, for example in form of one or more APIs provided. Generally, the interest of a Product Owner is focused on managing this functionality and their key performance indicators from a business perspective. The Microservice call stack may nevertheless be of note, as for example already the sheer amount of subsequently called Microservices may significantly increase API response times on product level compared to a more simplified Microservice architecture.

Besides the Product Owners own product, it is of substantial importance to him which other products, possibly even living in another domain are originating requests on services provided within his own Product. Such an external utilization may of course also exist the other way around, when Services from other Products, or Domains are utilized by Microservices from the Product Owners realm. These inbound and outbound requests should by all means be considered at the budget planning for operation and maintenance efforts.

Software Architect

The Software Architect in a product team designs the actual Service structure in order to fulfill the business requirements of the respective product. In order to review and monitor this service structure, Software Architects need comprehensive insights on product level, whereas their particular focus lies on the relations between Microservices and the complexity within the boundaries of a product.

Developer

After the initial implementation of a Microservice, developers are also in charge of Microservice maintenance and further optimization. For this purpose all originators of requests to the specific Microservice are of particular interest, as depending on the actual manners of utilization optimization strategies may vary. Again, the other way around is the set of called Microservices may need to be evaluated for optimization endeavors.

Operator

During operations, a close monitoring of Microservices is inevitable in order to ensure trouble-free business. Operations is focusing primarily on vitality data of individual Microservices and their actual Instances. This data is amended by vitality data of the underlying virtual and physical hardware systems. Beneficial insights can be additionally provided through an automated anomaly detection as proposed by [Steigerwald, 2017] and an accompanying root-cause analysis in order to track down incident causes in an efficient manner.

DevOps

Due to the shift towards more agile processes in software development, in recent years DevOps emerged as a new role aggregating developer and Operation responsibilities into a new, more holistic field of action. Therefore DevOps also share the concerns of developers and operators combined.

4.4 Suggested Views

In order to fulfill the needs of all the varying stakeholders introduced in 4.3, the proposed tool provides multiple views. Users are able navigate freely between views by directly selecting the desired element to focus on within the graphical representation.

In all directed graphs, edges illustrated communication between two entities. These edges are always originating from a caller and are directed towards the correspondent callee. The direction is implied by an arrowhead at the callee-side on the receiving end of an edge.

4.4.1 Enterprise View

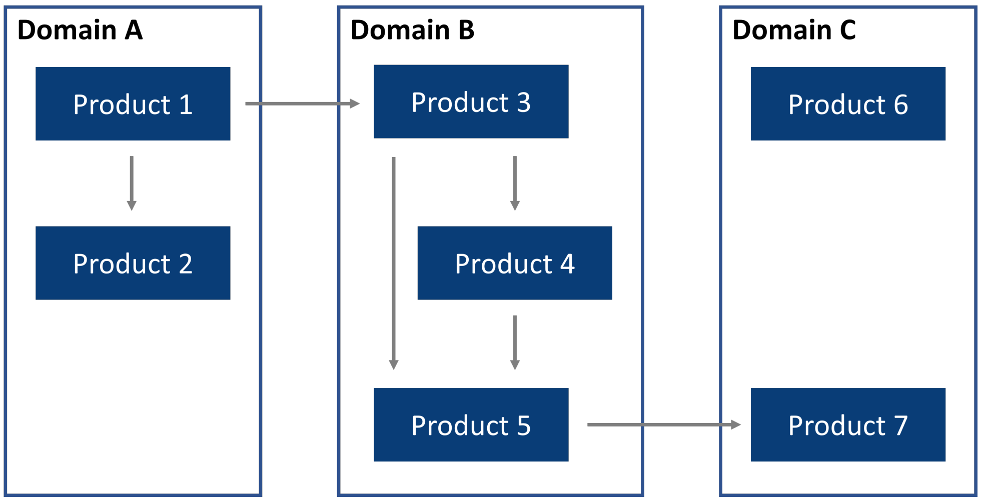


Figure 4.2: Enterprise View with relations between Products

On enterprise level, a greater summary depicts the organizational domain structure. As an adequate software cartographic representation the well-proven form of a cluster map has been chosen [Matthes, 2008]. In the graphical visualization, Domains are represented as compounded nodes, whereas products are illustrated as regular nodes. Products belong always to exact one Domain. Edges between regular product nodes stand for relations among the products. Domains may be displayed side by side for horizontal coexisting entities. While in the case of parent-child relations, domains can be positioned as nested representations respectively.

Products that belong directly to a domain are displayed, alongside potential subordinate domains, inside their respective parent domain.

The enterprise view does not include any Microservices or virtual and physical hardware nodes. Therefore it resides solely on the Business Layer, giving a high-level and organization-wide overview.

To get additional context information, the relations between products, not only existing within the boundaries of each domain, but also crossing domain boundaries, are added. For this purpose any distinct caller-callee relations between two Microservices are select, where both Microservices belong to different products. As the enterprise view does not consider Microservices, multiple relations between the exact same two products are combined to a single relation.

To further adapt the enterprise view for individual needs it is possible to omit relations either inside a domain, between domains or even any relations. Alternatively, users may choose to aggregate relations between products again into superordinate ones depicting relations between domains instead of products.

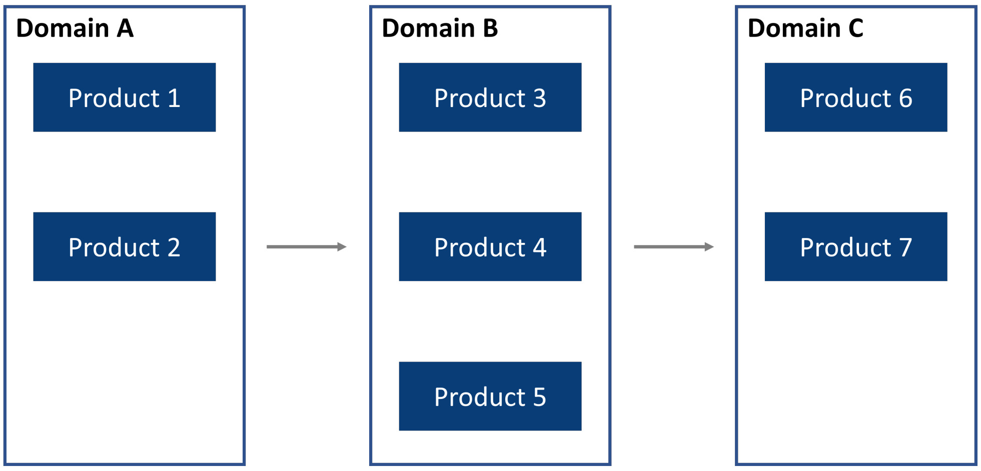


Figure 4.3: Enterprise View with relations between Domains

This enterprise view serves as information basis for Enterprise Architects and could also be used as an executive summary for C-level-Management. Besides the organizational structure, the enterprise view enables the viewer to perform a rough inspection of domain interdependence by examining inter-domain coupling.

Users can navigate to individual Domains and Products by selecting the appropriate visualization entities, leading towards Domain (4.4.2) and Product Views (4.4.3) accordingly.

4.4.2 Domain View

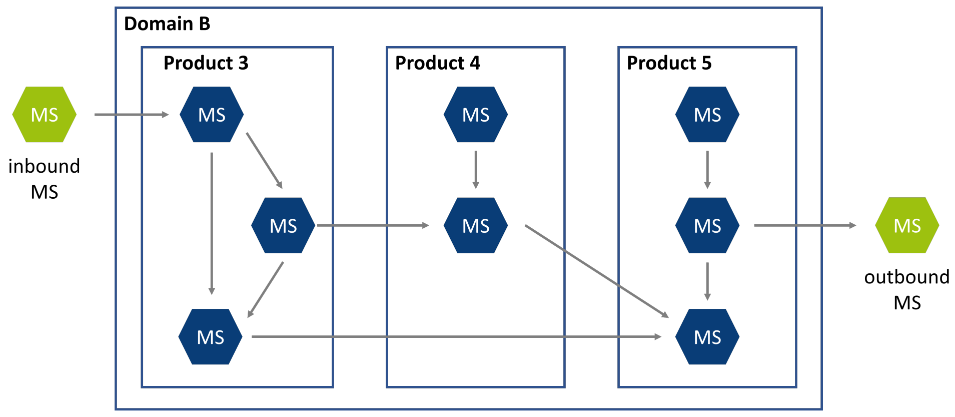


Figure 4.4: Domain View with relations between Microservices

The Domain view describes a single Domain, depicting all associated Products and their Microservices, as well as the relations to other Domains. The visualization shows all Products inside the domain as Compound Nodes, containing all Microservices attributed to the domain. These Microservice-nodes are connected in a directed graph, in which the edges represent caller-callee relations between two Microservices. Again, each edge is an aggregated representation of all communications between the respective caller and callee Microservices. A second edge may exist between two nodes, if both parties have called each other at any point point in time.

Any incoming and outgoing relations to Microservices outside the Domain boundary are added to the visualization to keep the context of the Domain in sight.

The graph settings allow the user to further aggregate relations on product level. Along the lines of the enterprise view features, all edges across Product boundaries are further merged. This helps to identify and evaluate relations across and among products.

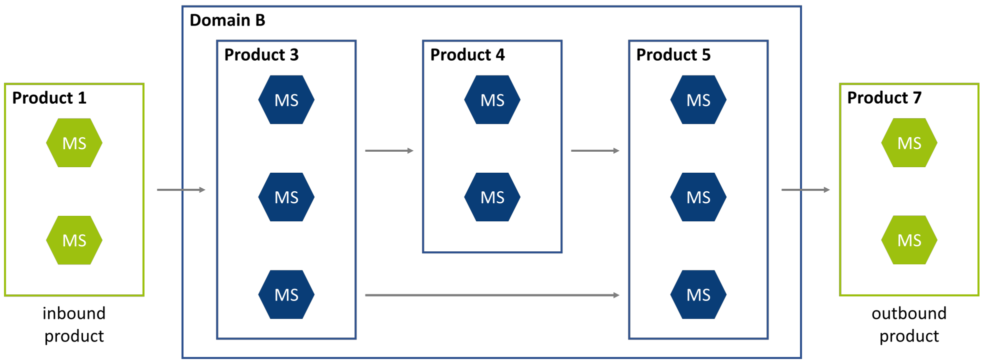


Figure 4.5: Domain View with relations between Products

A Domain Owner can use this view to keep track of products and inter-product communications. Additionally, this view can give insights into how closely products are linked. Relations to other Domains' Microservices show how isolated, or the other way around, how closely coupled the domain is concerning the enterprise context.

For a narrower breakdown of a single Product, users can navigate to the respective Product View (4.4.3). Whereas by selecting a specific Microservice, the user reaches the appropriate Microservice View (4.4.4) for the purpose of a technical in-depth analysis.

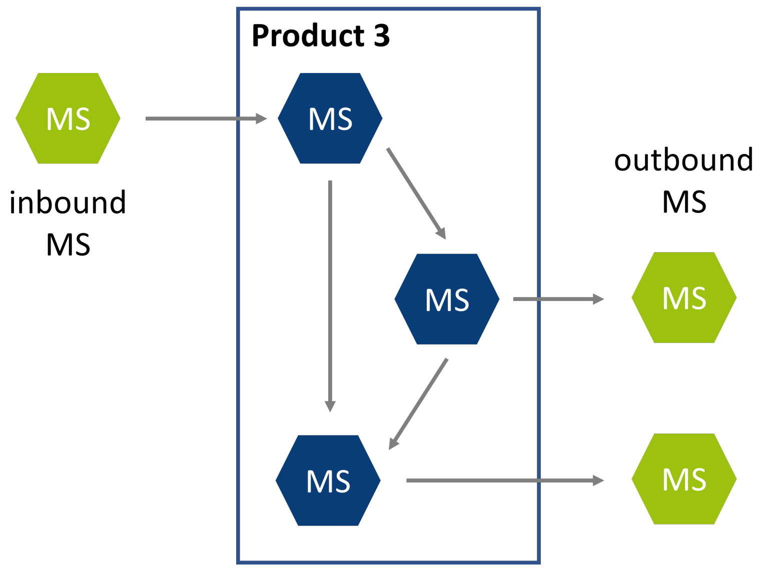


Figure 4.6: Product View

4.4.3 Product View

The proposed solution offers a separate view for each product. This view incorporates details about the Microservice communication flows within the product, as well as inbound and outbound communication with external Microservices. For this purpose each Microservice associated with the product is represented by a node in a directed graph in which edges in turn represent the relations among Microservices. Multiple communication flows between two Microservices are still represented by a single connection per communication direction.

This view displays the architecture of a product, which was originally planned and gets monitored by a Software Architect from a technical point of view. From a business perspective, a product owner can utilize this Product View to either evaluate dependencies and coupling of Microservice inside his Product Team, but also to negotiate the utilization of Microservices provided by other domains or products departments.

Incoming calls, originating from other products and domains, are of particular importance, as the emphasis on them may prevent Product Owners and Software

Architects from potentially abolishing Microservices, that are actually still utilized by other entities within the enterprise.

Users can dive into the technical details of a single Microservice by selecting the appropriate graph node and thus reaching the corresponding Microservice View (4.4.4).

4.4.4 Microservice View

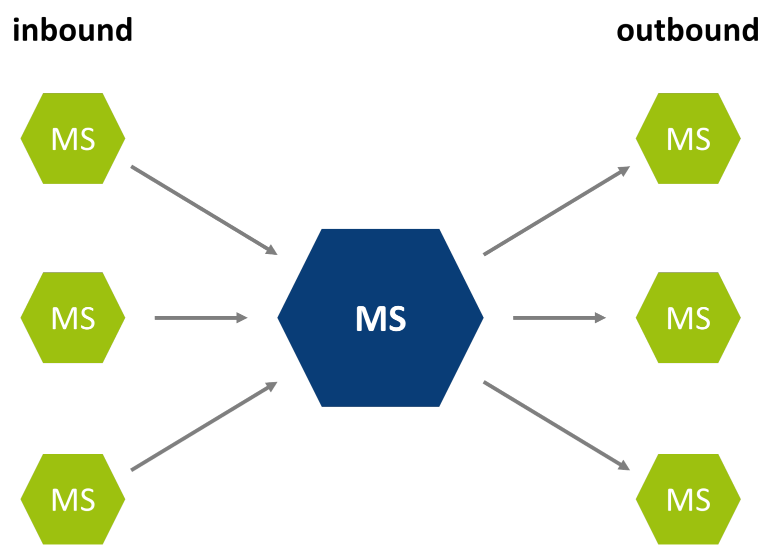


Figure 4.7: Microservice View

Finally, the Microservice View characterizes an actual Microservice. For this purpose all known technical details about the Microservice in question are summarized in one place. In a directed and horizontal oriented Graph the Microservice is displayed in context with all directly related Services, describing its closest environment.

As a central node the particular Microservice is placed in the middle of the graph. On its left side, all Microservices originating calls to the center one are depicted as additional nodes. Within each Microservice-node their affiliation to Products and Domains are denoted for reference. Edges between left-side and center node resemble inbound communications, utilizing the characterized Microservice.

On the right side in turn, all Microservice dependencies of the specific Microservices are specified as nodes. Again product and domain belonging are amended inside the nodes. And edges originating from the central Microservice describe the calls to other entities inside the Enterprise.

Next to this directed graph, technical details about actual Instances of the respective Microservices are listed, enriched with the corresponding hostnames, IP addresses and ports, as well as their current up-time. This data set is directly requested from the Eureka service via REST API.

Another special data box gives information about average response times and further usage statistics. While a third additional info box describes all APIs, provided by the specific Microservice in order to fulfill other Microservices' needs. Details about provided interfaces are obtained from an API documentation, such as Swagger as discussed in 4.2.

This view is especially useful for Development and Operations of a service, as analyzing the Microservice in context of its actual use cases can help in terms of service performance optimization.

Selecting any of the related Microservices nodes leads the user directly to the respective Microservice View (4.4.4), while the product and domain annotations inside each node lead to the correspondent Product (4.4.3) and Domain Views (4.4.2).

5 Prototype Implementation

The solution proposed in this master thesis is realized in a state of the art frontend web application. This web application is based on the conventional web fundamentals Hyper Text Markup Language (HTML) and Cascading Style Sheets (CSS) for representation and styling purposes, accompanied by JavaScript as scripting language in order to handle communications and executing business and application logic. To apply concepts of modern business web applications, the whole project is built upon the Angular application platform, enabling the realization of a Model-View-Control based architecture.

Angular code is written in TypeScript rather than JavaScript, which is in turn a strict superset of the later and fully transpiles into standard JavaScript [Microsoft, 2018]. Alongside neither actual HTML nor CSS has to be written, instead so-called Angular templates are filed with data and translated into HTML markup on run time and the Syntactically Awesome Style Sheets (SASS) language adds several new features to CSS such as variables, inheritance, nesting and loops which in turn are precompiled to standard CSS [Catlin et al., 2018].

The following sections describe the different parts of the web application in detail.

5.1 Data Model

The entities introduced in 4.1 are implemented in TypeScript. Domains, Products and Microservices correspond to the different kinds of nodes to be displayed in the various graphs subsequently and all of them share common similarities and are therefore inherited from a more generic *Component* class.

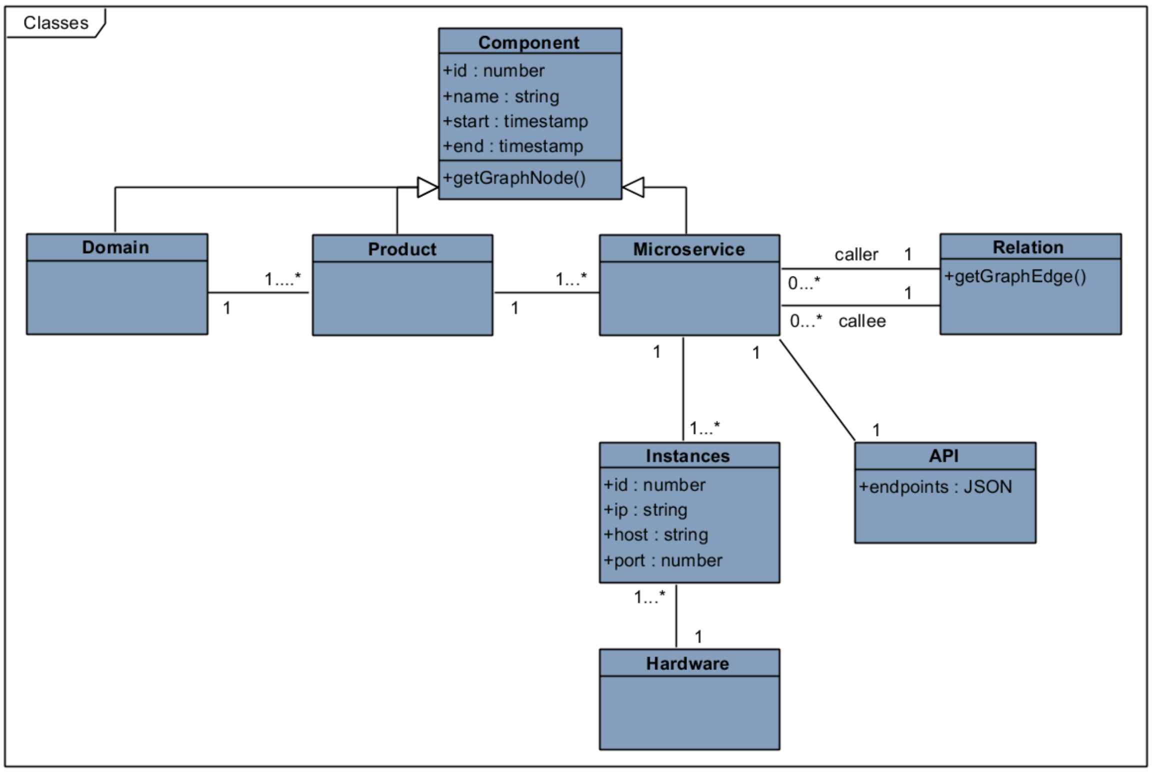


Figure 5.1: Class Diagram for the visualization prototype

A specific *type* field is used to identify the actual class type of such a component while different types can be configured using a custom enum *ComponentType*. As discussed later on in 5.4.1, node styling in graphs also depends on this ComponentType. This enables a later amendment or adaption of this entity structure at little to no expense. Besides the type, the Component class offers member variables in order to store a component *name* as a string, its *id* and the id of a *parent* component as integral numbers. For each of these private members, the Component class offers getter and setter methods. Additionally a method for the export of a ready to use graph node object is provided inside this class. The output of this method matches the node input expected by the graphing library. Furthermore, a static method provides deserialization functionality to convert plain JSON objects to actual TypeScript objects.

Each Microservice belongs to exactly one parent Product, whereas each Product belongs to exactly one parent Domain.

Relations between Microservices are reflected in a specific *Relation* class. At the moment, this class has members for a numerical *id* and most significantly a *caller* and a corresponding *callee* members containing the respective ids of two different Mircoservices. In case of a recursive call on itself, a Microservice would be caller and callee of a relation at the same time. As further discussed in 7.1, the information value and meaningfulness could be considerably enhanced by providing more detailed data about each relation. The relation class offers, just like the Component class before, a static class for the deserialization of plain JSON objects and an equivalent method in order to generate an edge object for the graph library.

To each Microservice belongs an API definition describing the provided API endpoints as well as the corresponding input and output ressource structures. These endpoints are specified using the OpenAPI specification, originally designed under the name *Swagger Specification* by SmartBear Software, which is now developed further by the OpenAPI Initiative which in turn is part of the Linux Foundation.

One of the core features of the Microservice concept is the possibility to scale individual Microservices by duplication. Those duplicated Microservices are all Instances of the same Code running inside an individual environment each. A service repository such as Netflix Eureka keeps track of all Microservices and all currently available Instances of these Microservices. The data about each Instance contains basic information about host, ip address and port as well as uptime information.

5.2 Architecture

The web application is structured into multiple components with the purpose to present the different views to the user. These components are in turn supported

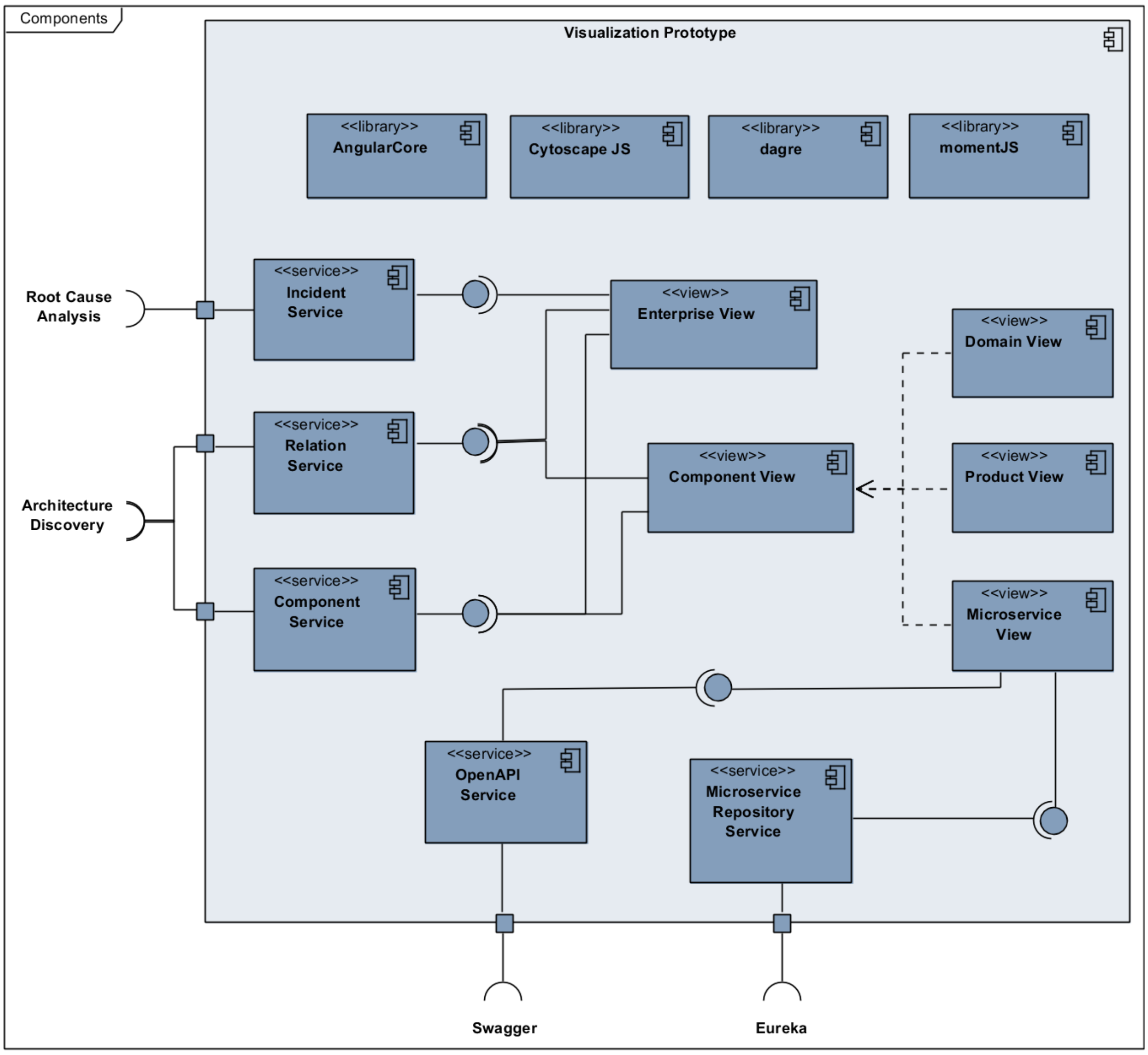


Figure 5.2: Component Diagram for the visualization prototype

by several services, retrieving data from the various data sources and building a data separation layer to facilitate subsequent data source replacements. Domain, Product and Microservice view components are all using the same generic component view as basis which provides the basic layout layout structure as well as connecting to Relation and Component services, whereas the enterprise view is implemented in a separate component as it is different from the other views by not being constructed around a single Component.

Relation and Component services retrieve the results of an Architecture Discovery and provide these in convenient interfaces, offering methods for retrieving a single component by id, all child components of a specified component or even all available components of a certain type at once.

The relation service offers methods for retrieving a unique relation by id and another one for getting all relations of a specified Array of Microservices in a structured way. For this special case relations are separated into inner relation having source and target Microservices included in the provided array, inbound relation when only the target Component belongs to the array and outbound relations when only the source is contained in the specified array. Furthermore the *externalNodes* array contains a set of all source Components of inbound relations as well as all target Components of outbound relations.

```
getRelationsByComponents(components: Component[]): {
  inbound: Relation[],
  inner: Relation[],
  outbound: Relation[],
  externalNodes: Component[]
}
```

Listing 1: TypeScript signature of the getRelationsByComponents() method from the relation service

The enterprise view utilizes additionally an incident service in order to retrieve the results of the root cause analysis displayed as recent problems as described in 5.4.3.

The mostly technical Microservice view utilizes another two services getting instance data from a Microservice repository such as Netflix Eureka , alongside an API service that provides the OpenAPI specification of services.

Cytoscape JS as graph visualization framework and dagre as layouting algorithm are obviously used by both component and enterprise views. These are supplemented by momentJS in order to conveniently handle JavaScript Date objects and naturally the Angular Core components to form a ready for use MVC web application. The last two components are obviously also utilized within the various services.

5.3 Use cases

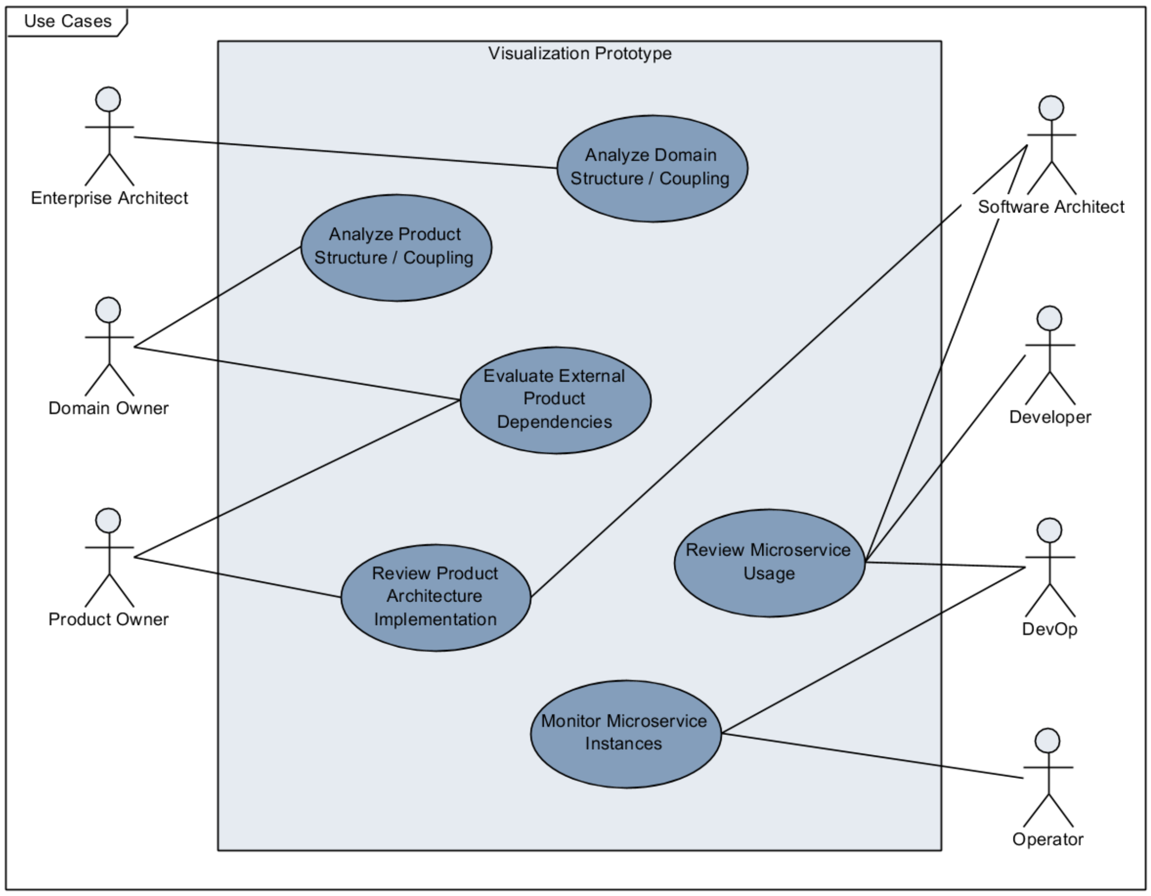

Use Cases
Visualization Prototype
Enterprise Architect
Domain Owner
Product Owner
Software Architect
Developer
DevOp
Operator
Analyze Domain Structure / Coupling
Analyze Product Structure / Coupling
Evaluate External Product Dependencies
Review Product Architecture Implementation
Review Microservice Usage
Monitor Microservice Instances

Figure 5.3: Use Cases for the distributed Microservice architecture visualization prototype

The distributed architecture visualization tool proposed in this thesis shows several different use cases depending on the particular user role and the accordingly analyzed views. Starting at a very technical user role, the following consideration will widen its view up to broad ranged business views.

At the technical position, a Microservice Operator can use the visualization to monitor all instances of a Microservice in his maintenance responsibility. Another technical viewpoint is brought up by developers, who are interested in the communication partners of a specific Microservice in question. This naturally includes callers of the Microservice as well as callees. Based on this communication behaviour, developers can adjust the code of a Microservice in order to optimize it for its actual utilization. A DevOp would combine the positions of Operators and Developers in a single role and can gain additional benefits from knowing both sides of the medal.

A Software Architect can use the Product view to review the distributed Microservice architecture and match it with the initially planned structure. In addition, the Software Architect can inspect a single Microservice and its communication structure in order to identify such servers that should be split up in multiple ones, as their is too much different functionality inside one service, or even the other way around it might be sufficient to merge two Microservices if they show a particularly high coupling.

A Product owner evaluates the coupling of his own Product to others, thereby a Product Owner takes a business perspective and might need to negotiate budgets in order to square usage costs with other Products Microservices.

Domain Owners are again managing the usage of external Microservices from outside the scope of their domain. Further the clustering of functionality in appropriate products and potential adjustments to this structure fall into the domain owners duties. Products with a particularly high coupling might need to be merged, while inflated products with too much diverging functionality should potentially be further partitioned.

As a final, superordinated role, Enterprise Architects need to handle the domain structure and keep track of the overall complexity of a Distributed Microservice architecture. Starting from the enterprise view, a Enterprise Architect may dive into the details of Domains, Products and Microservices to broaden the high-level overview.

5.4 Page Elements

The different views proposed in 4.4 are construct of a series of different view components. Each of these components is based on specific data sources and provides valueable information to the user. This section discusses these varied view components in detail.

5.4.1 Graphs

The core feature of this frontend solution is the visualization of a Microservice architecture in graph form. In order to find a suitable web based graphing library, several requiremants were defined at the outset:

Graph Positioning Such a graphing solution needs to be able to position nodes in a directed graph in a sensible manner, while avoiding intersections of edges to a large extent.

Directed Multigraph Such a framework is supposed to display multiple edges between two nodes, as this is necessary to make mutual communications well recognisable.

Compound Nodes In order to display the hierarchical nature of Domains, Products and Microservice enclosing each other, so-called *Compound Nodes*, which are able to contain other nodes again.

Navigatable The graph should be presented in such a way, that the user is capable of navigating it easily to be able to keep the overview and be able to recognize details at the same time, even in large and complex graphs.

Convenient API In order to enable user interactions directly with the graph, the graphing solution needs to provide a comprehensive API.

Adjustable Styling And last but not least, a wide range of styling features is required to display varying node types in a clearly distinguishable and yet pleasent way.s

Several different graph libraries were discussed during this thesis, but eventually a decision for cytoscape JS was made. Cytoscape JS is the JavaScript based web-offshoot of the Java based and therefore client side visualization platform cytoscape. Which in turn is specialized on the visualization of complex network-like structured data. Cytoscape JS, licensed under the MIT open source license, is funded by the U.S. National Institues of Health and the National Center for Research Resources [Franz et al., 2016]. It offers a rich API which works similar to the jQuery API to select and manipulate graph data and to adjust styling. Besides that, it relies on a easy to handle data structure for nodes and edges and it offers a CSS based styling approach.

```
{
  'data': {
    'id': '123',
    'title': 'CORE-BUSINESS', // Microservice name
    'parent': '110' // ID of its parent compound node
  },
  'classes': 'microservice', // space separated classes
  'selected': false, // not selected by default
  'selectable': true, // users can select nodes
  'grabbable' : true, // users can rearrange nodes
}
```

Listing 2: Example of a graph node JSON representation

In order to reach the desired node positioning outcome, cytoscape Js does not only offer multiple included onboard layout algorithms, but also a convenient plugin system enabling several third party layout solutions to be included. One of these plugins, dagre, is used for the graphs presented in this thesis. Dagre is developed by Chris Pettitt and it is specialized on tree-like data structures and nearly intersection-free visualizations [Pettitt, 2018].

Graph nodes can be passed to cytoscape JS as an array of node objects. Each node object needs to have at least an id, all other fields are optional. classes are used to style the different component types respectively. Edge objects generally show the same appearance, but have *target* and *source* fields instead of a *parent* field. In all presented graph visualizations users are enabled to freely rearrange any nodes by setting the *grabbable* attribute to true, this feature in turn is not available for edges, as their positions are already defined by their source and target nodes positions. Rearrangements are performed by a simple drag-and-drop movement.

```
{
  data: {
    id: 'e1', // an identifier
    source: '123', // source node id
    target: '321'  // target node id
  },
  'classes': 'relation', // space separated classes
  'selectable': false
}
```

Listing 3: Example of a graph edge JSON representation

By pressing and holding a modifier key, which depending on the used operating system might be any of ctrl, ⇧, ⌘ or Alt keys, users are further able to select multiple nodes simultaneously and can rearrange this selection collectively.

Moreover the whole graph can be moved around by clicking and dragging the graph-background and zooming is performed by using the mouse-wheel. The graph can always be re-positioned to fit inside the viewport by clicking the appropriate button, which in turn calls `cytoscapeGraph.fit()` for this purpose.

The styling of graph elements is performed on base of element types, classes and states. Along the lines of conventional HTML and CSS selectors each element has exactly one type, an arbitrary number of classes and it can adopt different states, describing its current situation.

```
{
  selector: 'node',
  style: {
    'label': 'data(title)',
    'width': 'label',
  }
}, {
  selector: 'node.' + ComponentType.Product,
  style: {
    'shape': 'rectangle',
    'font-size': 12,
  }
}, {
  selector: 'node.' + ComponentType.Product + ':parent',
  style: {
    'text-valign': 'top',
    'padding': '20px',
    'text-margin-y': '18px',
  }
}, {
  selector: 'node.' + ComponentType.Product + ':childless',
  style: {
    'text-valign': 'center',
    'padding': '10px',
  }
}
```

Listing 4: Styling of all Component nodes and specifically Product nodes, either for compound or self-contained Product nodes

As illustrated in Listing 4, styling of a node can be constructed hierarchically by combining several complementary styling selectors. A basic styling for all nodes specifies that all nodes should display their title attribute as label, while scaling their width dynamically in such a way that the label is always fully contained. For any nodes resembling Components of the ComponentType *Product* additional stylings are applied to manipulate the node shape and label font size. the state-selectors `:parent` and `:childless` allow the differentiation between compound nodes and self-contained nodes, whereas compound nodes need some extra styling in order to align their labels appropriately.

Via the cytoscape JS API it is possible to manipulate a set of existing graph elements in different ways. The most convenient way to do so, is by adding or removing classes from graph elements. Cytoscape JS then automatically refreshes the representation accordingly.

```
cytoscapeGraph.on('tap', 'node', function(event) {
    // perform any actions inside here
    angularRouter.navigate(['/service', event.target.id()])
});
```

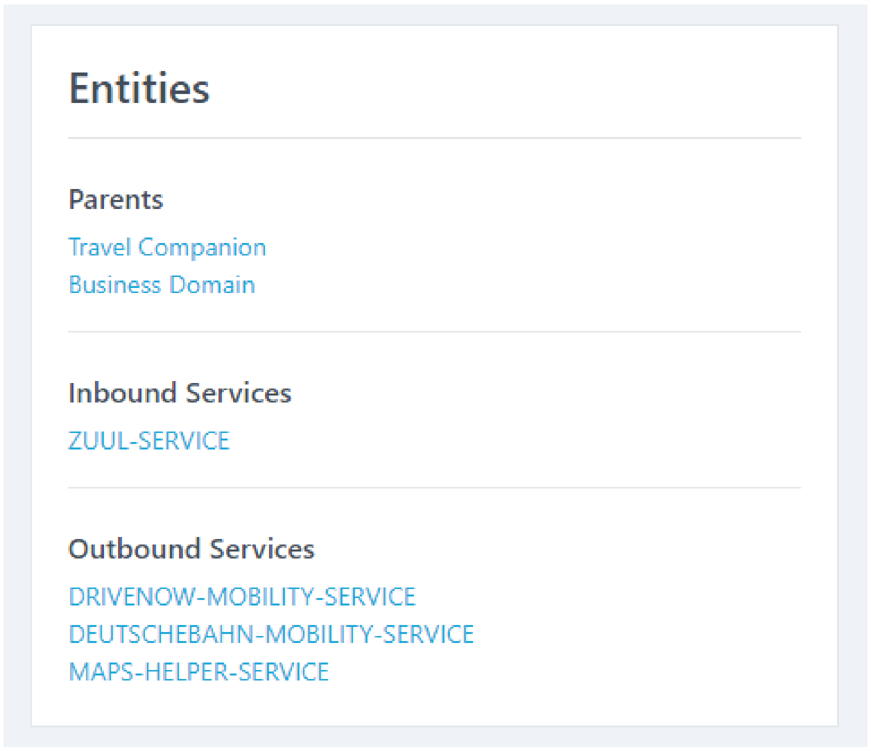
Listing 5: Adding an event listener on graph nodes for taps on mobile devices and clicks on dektop devices

Beyond that, a variety of event listeners are provided by cytoscape JS in order to process any user interactions with the graph. An example for such an event listener is illustrated in code Listing 5. In this example, a listener for any taps or clicks on nodes is registered wich evaluates incoming events and passes the id of the corresponding node on to the angular router in order to navigate the user to the appropriate angular view.

5.4.2 Entity Listings

Each graph is accompanied by a so called entity list which presents all entities visualized inside the graphical representation in a structured manner. Entities are sorted by type and alphabet, providing a useful overview at a glance. To get a visual connection of graph nodes and entity list items, users can hover the entity list items in order to highlight the corresponding graph node. The entity lists are created from the same data as the graph nodes. For Products and Microservices the parent Components are added to the list in a separate section.

To further streamline the user experience, the entity list items are simultaneously acting as links to the corresponding component views.


Entities

Parents

Travel Companion
Business Domain

Inbound Services

ZUUL-SERVICE

Outbound Services

DRIVENOW-MOBILITY-SERVICE
DEUTSCHEBAHN-MOBILITY-SERVICE
MAPS-HELPER-SERVICE

Figure 5.4: Entity list of a Microservice, displaying all graph nodes

5.4.3 Recent Problems

On the enterprise view, a list of recent incidents is presented chronologically . These incidents are based on the results of a root cause analysis as described in the related master thesis of Lukas Steigerwald [Steigerwald, 2017]. Each incident entry links to the specific Microservice, identified by the root cause analysis. As no actual Microservices are displayed inside the enterprise view, its parent Product is highlighted in a red warning color instead. Additionally a colored indicator displays the severeness of the observed incident, ranging from green for minor incidents, along yellow for longer delays up to red for total outages of a service. And to finally be able to classify the problems chonologically, the time span passed since the observed abnormality is displayed alongside in a nicely readable way and gets update in real-time using a special *amTimeAgoPipe* proivided be the momentJS library for Angular templates as in `{{incident.time | amTimeAgo}}` which

```
<div class="card">
  <div class="card-block">
    <h4 class="card-title mb-0">Entities</h4>
    <div *ngFor="let entityGroup of entities">
      <hr class="mb-4">
      <h3>{{entityGroup.name}}</h3>
      <div *ngFor="let entity of entityGroup.entities">
        <a  routerLinkActive="active"
            [routerLink]="['/service', entity.id]"
            (mouseenter)="highlightNode(entity)"
            (mouseleave)="highlightNodeOff(entity)" >
          {{entity.name}}
        </a>
      </div>
    </div>
  </div>
</div>
```

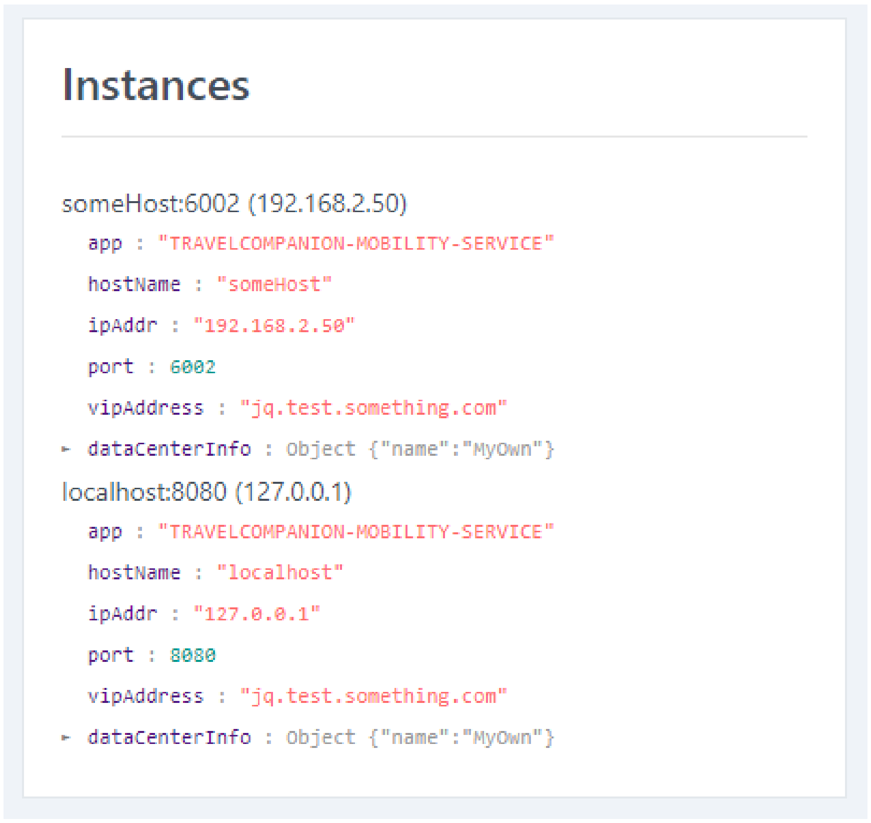
Listing 6: Angular Template for entity lists

result in outputs such as "a few seconds ago", "3 minutes ago" or "12 days ago".

5.4.4 Instances and API Specification

The Microservice view is enhanced with some technical data about the designated Microservice and its Instances. For a quick overview of the Instances JSON data from the Microservice repository is retrieved and displayed in a well organized way. These data sets contain information about hostname, ip address and port as well as current state and uptime of the respective instance.

The OpenAPI specification of that Microservice is also displayed as interactive JSON, which is completely folded up on start up, but allows the user to unfold its deeper layers to retrieve details such as the exact entry points and there respective specifications.



Instances

```
someHost:6002 (192.168.2.50)
  app : "TRAVELCOMPANION-MOBILITY-SERVICE"
  hostName : "someHost"
  ipAddr : "192.168.2.50"
  port : 6002
  vipAddress : "jq.test.something.com"
▸ dataCenterInfo : Object {"name":"MyOwn"}
localhost:8080 (127.0.0.1)
  app : "TRAVELCOMPANION-MOBILITY-SERVICE"
  hostName : "localhost"
  ipAddr : "127.0.0.1"
  port : 8080
  vipAddress : "jq.test.something.com"
▸ dataCenterInfo : Object {"name":"MyOwn"}
```

Figure 5.5: Displaying all Instances of a Microservice based on data from a Microservice Repository

5.5 Navigation Flow

The interactive visualization offers multiple ways to conveniently navigate through a distributed Microservice architecture. The most obvious and convenient way is to click or tap on a component directly inside the graph, which immediately leads the user to the associated component view. This allows both, a zoom into a specific sub component of the currently displayed one and a lateral movement to adjacent components.

In this way, the visualization enables users to move from the enterprise view towards domain and product views, from domain views to product and Microservice views and from Product views also to the associated Microservice views as well as lateral navigation on domain, product and Microservice level.

By design this obviously does not allow any upwards movement to superordinated parent components. To compensate this shortcoming, a designated button is provided directly above the graph to navigate to the direct parent of the currently displayed component. A summary of all provided navigation paths is provided in figure 5.6.

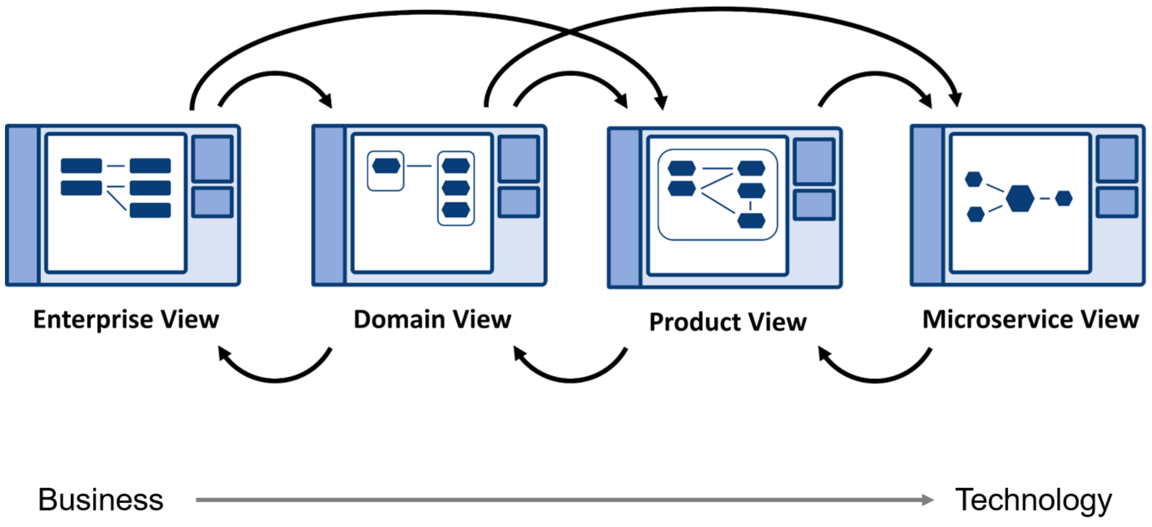


Figure 5.6: All available navigation paths provided among the available views

Next to the actual graph is a corresponding entity list located. This list contains not only links to the same components as shown in and therefore navigable through the graphical representation. Links to all parent components are added supplementary for a further streamlining. In the same way, users are enabled to navigate directly to the originators of all recent incidents displayed on the enterprise dashboard.

This varied navigation is performed by using the router service integrated in Angular. The id of the concerned component is passed to a general component view which retrieves the particular component and chooses and triggers the appropriate view based on the ComponentType. As the angular router parameter is passed via URL, specific views can also be called up by directly opening the appropriate url. This additionally allows other applications to link directly to a desired component views and simplifies the integration with existing applications.

6 Evaluation

6.1 Microservice Test Environment

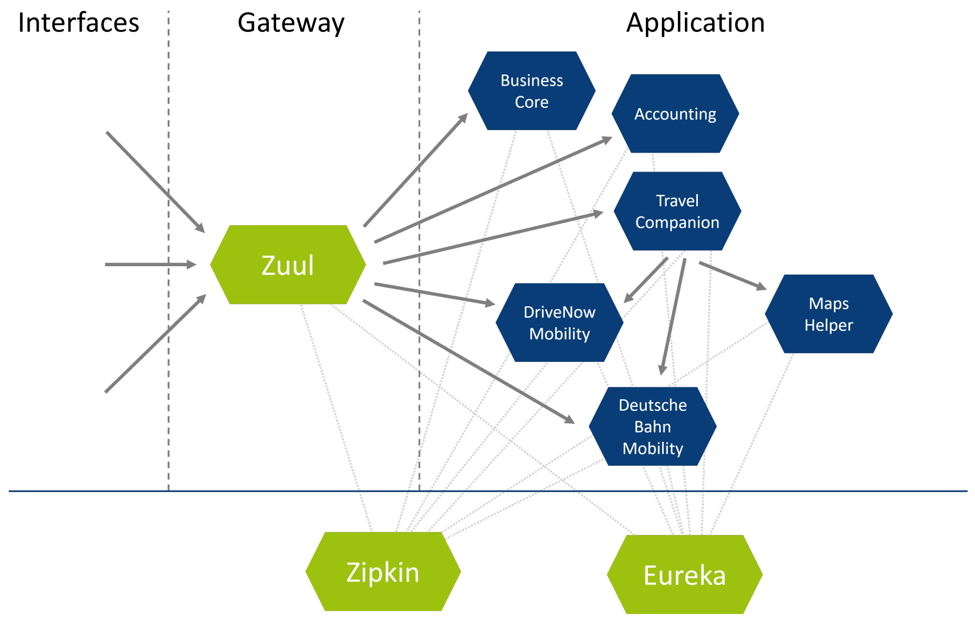


Figure 6.1: Microservice Test Environment at SEBIS chair TUM

For academic purposes, a test or lab environment with a basic distributed Microservice infrastructure was created at the chair for Software Engineering for Business Information Systems (SEBIS) at TU Munich.

The Microservices represent the web application *TravelCompanion*, which could be a typical business use case. Travelers should be able to utilize the TravelCompanion Application in order to find other co-travelers who aim for the same travel destination. Two different travel data provider, *DeutscheBahn* and *DriveNow* car sharing, are integrated for the purpose of calculating suitable travel routes. Subsequently, the application displays any available travel groups along this route that could offer a free spot to split travel expenses.

From a technical point of view, the lab setup features Java based-distributed Microservices, whereas multiple instances may exist of each Microservice. The central gateway service *Zuul* handles all user requests and forwards them to the appropriate responsible Microservice instances. These Microservices in turn divide their tasks further on and call again other services for specific tasks such as travel route calculating. The test environment Microservice structure is illustrated in figure 6.1.

To be able to follow those service calls, each service implements Zipkin [OpenZipkin, 2018], a specific system for distributed tracing. Additionally, all service instances report themselves to Eureka [Netflix, 2018], a Microservice repository server developed by Netflix. These language and dependency choices are solely exemplary, as they only serve the purpose of demonstrating the use case. In a real world application, all choices could be replaced by similar tools.

6.2 Functional Evaluation

In the following sections, an evaluation of the functional correctness of the prototype is described. In multiple steps, several different data sets are provided to the visualization prototype, emulating an architecture discovery service in different environments, starting from the most simple architecture and increasing in data complexity with each step. The produced graph visualizations are checked for correctness in order to ensure functional correctness of the complete prototype. For simplicity, only a subset of the available views is shown in the respective figures.

A Single Microservice

In a first step, a single Microservice along with its embracing product and domain is provided to the component service. A closer look at the enterprise view reveals the *Business Domain* embracing the *Travel Companion* Product, which in turn acts as a parent to the *TRAVELCOMPANION-MOBILITY-SERVICE* Microservice on the Product and Domain views. Figure 6.2 shows the outputs for Enterprise and Product views. Alike, Domain and Microservice view are also showing the expected behaviour.

The navigation paths are also working correctly, as a click on either Domain, Product or Microservice always leads to the associated view.

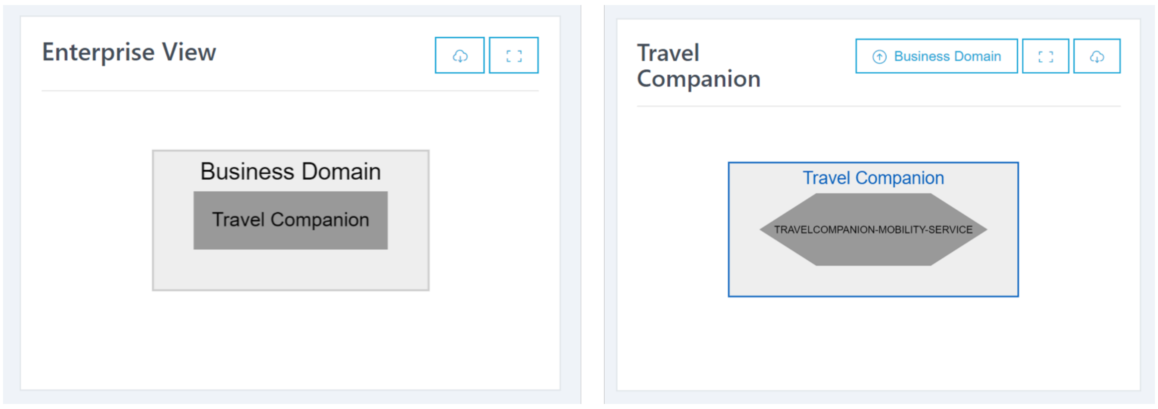


Figure 6.2: A single Microservice with its embracing product is shown in the Enterprise View (left) and Product View (right).

Two Microservices Within The Same Product

In a second step, the *DEUTSCHEBAHN-MOBILITY-SERVICE* is also added to the same Product. Furthermore, a first relation between both available Microservices is amended. The expected output should show no differences for the Enterprise view, as no additional Domains or Products have been added and the only existing relation is solely describing communication inside a Product. All subsequent views show both services including the relation among them, leading from Travelcompanion-mobility-service to Deutschebahn-mobility-service. The corre-

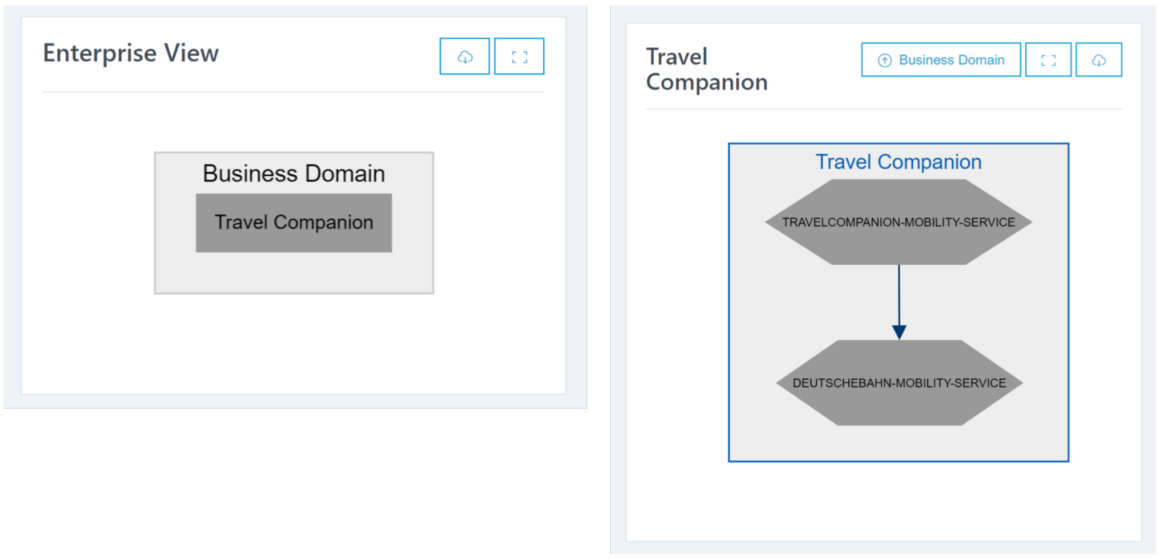


Figure 6.3: Two Microservices within the same product are shown in the Enterprise View (left) and Product View (right).

sponding results are displayed in figure 6.3.

Three Microservices in Two Different Domains

Another Microservices is added for the third step, but this time the service is added inside another Domain. The *MAPS-HELPER-SERVICE* gets also called by the Travel-companion-mobility-service, but lives inside the *Frontend Services* Product belonging to the *Technology Domain*. This results not only in a cross-Product but even in a cross-Domain relation between two services.

As visible in figure 6.4, the call from Travelcompanion-mobility-service to the Maps-helper-service is correctly displayed as a relation between Business and Technology Domain on the Enterprise view. The Product view in turn displays the actual relation directly between both Microservices in questions. The relation towards the Deutschebahn-mobility-service is further on displayed correctly from Domain view downwards.

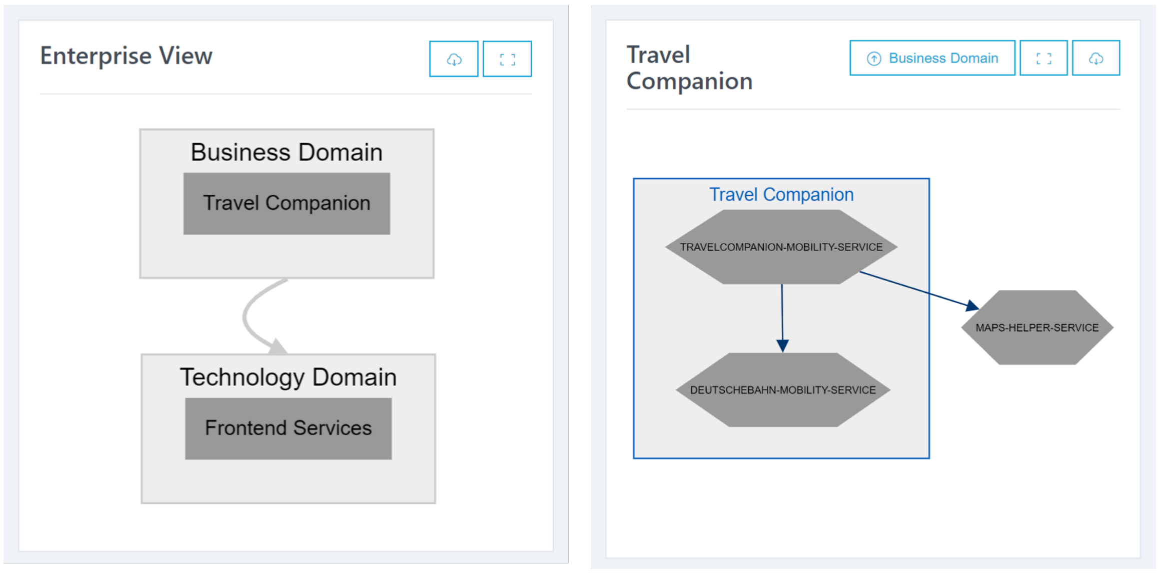


Figure 6.4: The Enterprise View (left) shows two connected Domains with their respective product, while the Product View (right) displays one product with its two internal Microservices and one outbound Microservice.

Switching Between Aggregation Levels

To further adjust Enterprise and Domain views, users can choose the relation aggregation level as discussed in sections 4.4.1 and 4.4.2. On the Enterprise view, the decision is between Domain-level and Product-level aggregation, all relations should switch their source and target Components accordingly.

Multiple relations between two Domains with the same direction are aggregated to a single relation. To observe this behavior, a more complex data model with two additional Products and corresponding nodes as well as additional relations is provided to the visualization. Figure 6.5 clearly shows the desired behavior and proves the correctly applied aggregation.

Similarly, the aggregation switch for the Domain view is successfully tested in figure 6.6. When aggregation is enabled, all relations between the same Products are accumulated and a clean and more tidy Product structure is visible. By disabling the aggregation on Product level, all communication flows between the Microservices are displayed and the detailed relations can be analyzed.

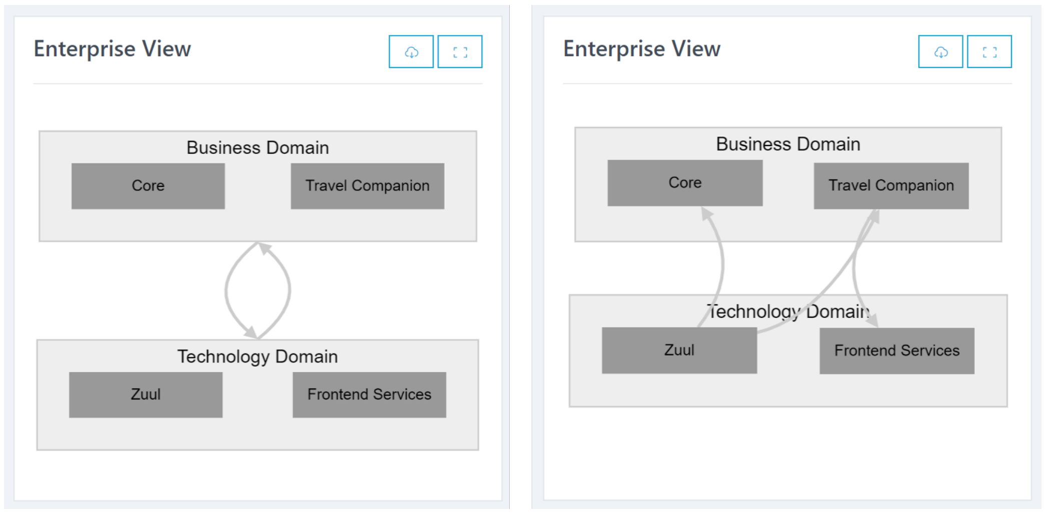


Figure 6.5: The Enterprise View is shown with different aggregation levels for relation aggregation on *Domain* (left) or *Product* (right) level

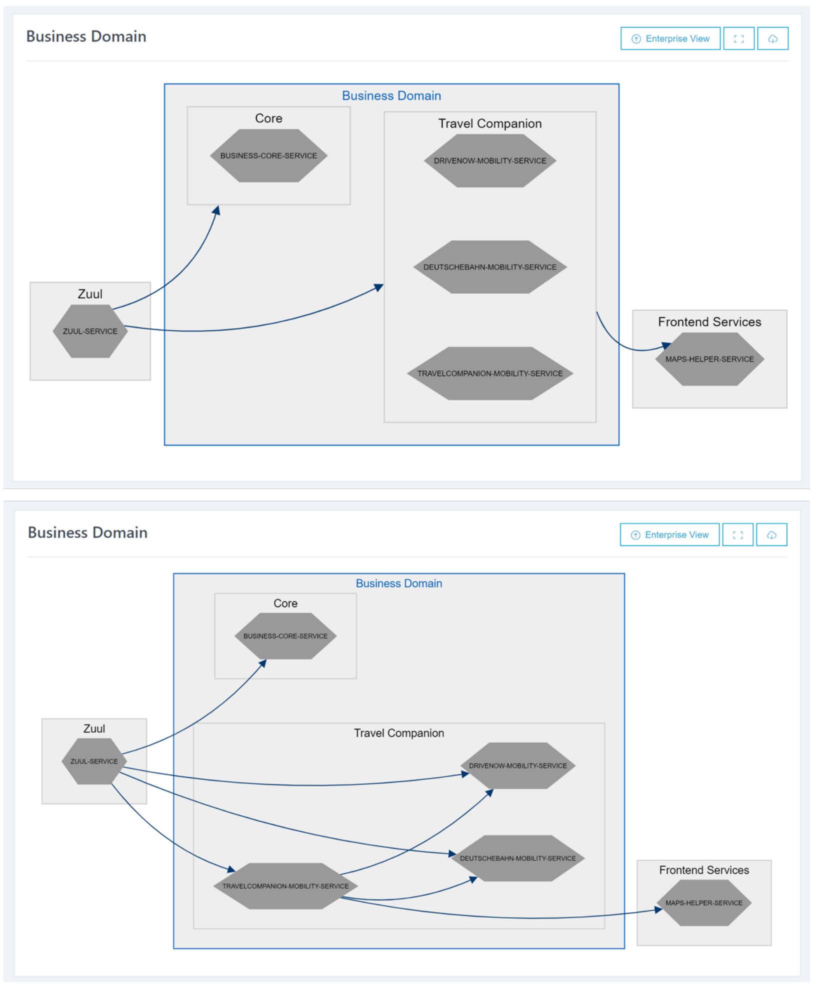


Figure 6.6: The Domain View showing aggregated relations on *Product* (top) or non-aggregated relations on *Microservice* (bottom) level

Additional Info Box: Entities

Besides the graphical representation, all displayed entities should be listed inside an entity list next to the graph view. On hovering the mouse on one of the list entries, the appropriate graph node is highlighted in a light green color for the purpose of a quick and reliable association. The correct behaviour of the entity list is illustrated in figure 6.7. Additionally, the Entity list entries are angular router links, leading to the associated component views, this was also successfully checked for its correct functionality.

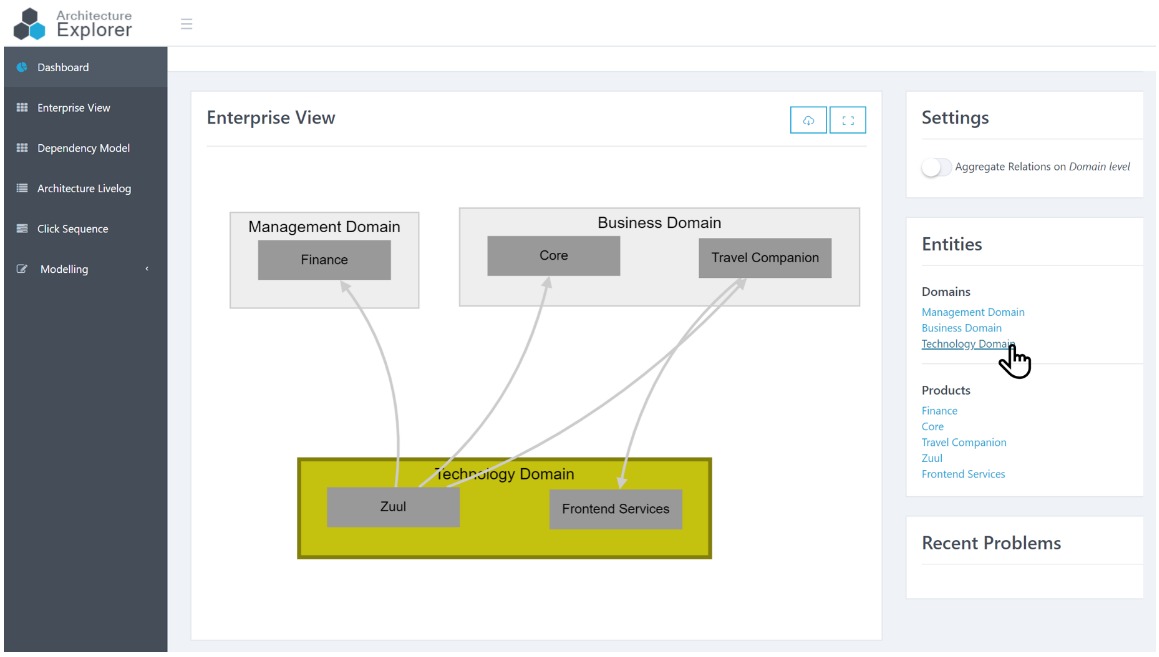


Figure 6.7: The full visualization tool is shown including its navigation possibilities and additional information like the displayed Entities at the Enterprise View Level and a mouse-hovering on a single Entity (highlighted in light green).

6.3 Other Comparable Visualization Tools

There is a range of different tools for the visualization of software systems available. This section look at some of these in comparison to the proposed Microservice architecture prototype.

ExplorViz

ExplorViz did originally visualize system internals and produces automated clustering based on code structures. It is able to impress with it stunning physical 3D cluster models. Nowadays, ExplorViz does also support the visualization of distributed service environments [Zirkelbach et al., 2018] and it additionally offers some useful tools for trace inspections. But in contrast to the solution proposed in this thesis, it does not offer any integration of business entities [Fittkau et al., 2013].

ExTraVis

ExTraVis is a tool specialized on the visualization of application traces, again the focus lies on system internal execution traces. The produced circular bundle graphs are beautiful to look at and at the same time rich in substance giving detailed insights.[Cornelissen et al., 2007] Neither a visualization of distributed architectures nor an amandment of business entities is supported by ExTraVis.

Monitoring Solutions for Distributed Systems

Serveral different providers such as Dynatrace, AppDynamics, Netsil or Instana are commercial players, specialized on the application performance monitoring of distributed Microservice and cloud-based architectures. None of these applications offers an integration with Enterprise architecture software, while several provide features for segmenting, grouping or automated clustering of distributed Services [Dynatrace, 2018, Cisco, 2018, Netsil, 2018, Instana, 2018].

Enterprise Architecture solutions

Enterprise Architecture solutions are a perfect fit for the visualization of Business artifacts, but the integration of real-time Microservice data is for these kinds of software out of scope.

7 Outlook and Conclusion

During the course of this thesis, a first prototype of a visualization of a distributed Microservice architecture and its different layers was designed and developed. In the course of its implementation, varying possible feature additions and suggestions arose. This chapter gives a rough overview of these topics and ideas.

7.1 Additional Data

As the prototype is situated in a web of different data sources it is heavily constrained by the data-extent provided by these data sources. The actual information value of visualizations could significantly be enhanced by extending the data suppliers appropriately. An outline of some possible data extensions is presented here.

Relation Data

In its current form, the Architecture Discovery tool does condense the distributed tracing information into simple binary relations between two Microservices. This means a large amount of potentially interesting data is omitted as the architecture discovery does only state if there is any relation between two Microservices - or not.

But in order to classify relations it would be of interest to be able to identify communication types and therefore to distinguish between synchronous and asynchronous communication or to indicate if a communication is actually invoking a data stream, or just calling another method.

Associated therewith could be a measuring of relation frequencies on which flow weights could be based. This would help to evaluate actual load causes and improve the classification of Microservice couplings. A conceivable way to visualize the data flow to and from a service could be an interactive Sankey diagram, this would even allow users to figure the internal data flow of a Microservice out, as hovering over an input flow could also highlight the associated outgoing flows and vice versa.

Based on such additional flow data it would also be possible to gather and provide profound statistics for Microservice relations, containing information such as average communication timings, failure rates, or even observing daily observed usage spikes to specific times.

For such extensive relation data, a more sophisticated algorithm for relation aggregations would certainly be needed, as relations between products or even domains could no contain a wide variety of different communications which properties would eventually be known.

Microservice Data

Alongside a more sophisticated relationship model, the data concerning Microservices could also be notably broadened by adding additional statistics about each Microservice. The same applies for the individual Instances, as their life data provided by the Microservice Repository is rather limited. By adding more life data about resource usage and connected data sources the tool could emerge to an even more valuable monitoring solution. To conclude Instance data, a direct integration of Instance log files could streamline the failure analyses processes.

Business Data

To additionally improve the integration of business data into the technical information, it could be considerably valueable to annotate a business value to Microservices and Products, which would allow a better assessment of Microservices and especially of failures. Furthermore it could be of similar interest to assess how critical to success a specific Component is.

7.2 Further Features

Beyond the additional information sets mentioned above, a few additional features have been identified, that could also sustain the proposed visualization solution.

Omni Search

A special search feature would help users to navigate the components conveniently. A search for arbitrary components would be included in a single search box, offering auto-complete functionality and displaying results grouped by Component type.

Entry Points

The solution could offer different dashboard like entry points for the various user roles. A Product Owner for example would directly be taken to his associated product view right after authentication. Alongside this feature the visibility of higher level views could be restricted based on a rights management system.

Impact Analyses

Integrating an impact analysis tool into the visualization solution could help to assess the consequences of a specific Microservice failure, or even for routine maintenance outages. Combined with the results of a root cause analysis, currently affected Components could be identified and highlighted accordingly. This would also help to asses and comply with concluded Service Level Agreements.

Advanced Graph Handling

Additional features for the actual graph presentation could include a small minimap to keep track of the current graph viewport in relation to the complete graph, as well as advanced mechnaism to persist user adjusted node layouts across sessions. A problem with such a layout persistence is the complicated alignment of new graph nodes.

7.3 Conclusion

In this thesis, a prototype for the visualization of a distributed Microservice architectures was conceptualized and implemented. The proposed solution allows meaningful illustrations of diverse Microservice relations and their mapping to the respective business entities. Aggregations at different levels serve to reduce the displayed complexity to a reasonable and perceivable extent. Multiple views are tailored to the needs of various Enterprise Architecture stakeholders for Microservice monitoring and documentation. The present prototype provides a solid base for further development including the additional data sources and features described above.

Appendix

List of Figures

List of Listings

Bibliography

[Abbott and Fisher, 2015] Abbott, M. L. and Fisher, M. T. (2015). *The art of scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise, Second Edition*. Addison-Wesley Professional, New York, NY.

[Bakshi, 2017] Bakshi, K. (2017). Microservices-based software architecture and approaches. In *2017 IEEE Aerospace Conference*, pages 1–8.

[Baraiya and Singh, 2016] Baraiya, V. and Singh, V. (2016). Netflix conductor: A microservices orchestrator. `http://techblog.netflix.com/2016/12/netflix-conductor-microservices.html`. Accessed: 2018-06-14.

[Bradley, 2015] Bradley, T. (2015). The challenges of scaling microservices. `www.techbeacon.com/challenges-scaling-microservices`. Accessed: 2018-06-14.

[Calçado, 2014] Calçado, P. (2014). Building products at soundcloud—part ii: Breaking the monolith. `https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-2-breaking-the-monolith`. Accessed: 2018-06-14.

[Catlin et al., 2018] Catlin, H., Weizenbaum, N., Eppstein, C., and et al. (2018). Sass: Syntactically awesome style sheets. `https://www.sass-lang.com/`. Accessed: 2018-06-14.

[Cisco, 2018] Cisco (2018). Application performance monitoring & management | appdynamics. `https://www.appdynamics.com/`. 2018-06-14.

[Cornelissen et al., 2007] Cornelissen, B., Holten, D., Zaidman, A., Moonen, L., Van Wijk, J. J., and Van Deursen, A. (2007). Understanding execution traces using massive sequence and circular bundle views. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pages 49–58. IEEE.

[Council, CIO, 1999] Council, CIO (1999). Federal enterprise architecture framework version 1.1. *Retrieved from*, 80:3–1.

[Daya et al., 2016] Daya, S., Van Duy, N., Eati, K., Ferreira, C. M., Glozic, D., Gucer, V., Gupta, M., Joshi, S., Lampkin, V., Martins, M., et al. (2016). *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach*. IBM Redbooks.

[Department of Defense, 1996] Department of Defense (1996). Technical framework for information management.

[Department of the Treasury, 2000] Department of the Treasury (2000). Treasury Enterprise Architecture Framework, Version 1.

[DoD, 2012] DoD, C. (2012). Dod architecture framework version 2.02. *DoD Deputy Chief Information Officer, Available online at http://dodcio.defense.gov/dodaf20/dodaf20_pes.aspx*.

[Dynatrace, 2018] Dynatrace (2018). Software intelligence for the enterprise cloud | dynatrace. `https://www.dynatrace.com/`. 2018-06-14.

[Evans, 2004] Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

[Fittkau et al., 2013] Fittkau, F., Waller, J., Wulf, C., and Hasselbring, W. (2013). Live trace visualization for comprehending large software landscapes: The explorviz approach. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4.

[Fowler and Lewis, 2014] Fowler, M. and Lewis, J. (2014). Microservices. *Viittattu*, 28. Accessed: 2018-06-14.

[Franz et al., 2016] Franz, M., Lopes, C. T., Huck, G., Dong, Y., Sumer, O., and Bader, G. D. (2016). Cytoscape.js: a graph theory library for visualisation and analysis. *Bioinformatics*, 32(2):309–311.

[Haddad, 2015] Haddad, E. (2015). Service-oriented architecture: Scaling the uber engineering codebase as we grow. `https://eng.uber.com/soa/`. Accessed: 2018-06-14.

[Hasselbring, 2016] Hasselbring, W. (2016). Microservices for scalability: Keynote talk abstract. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 133–134, New York, NY, USA. ACM.

[Instana, 2018] Instana (2018). Instana - dynamic apm for microservice applications. `https://www.instana.com/`. 2018-06-14.

[ISO15288, 2015] ISO15288 (2015). Iso/iec/ieee international standard - systems and software engineering – system life cycle processes. *ISO/IEC/IEEE 15288 First edition 2015-05-15*, pages 1–118.

[ISO24748, 2011] ISO24748 (2011). Ieee guide–adoption of iso/iec tr 24748-1:2010 systems and software engineering–life cycle management–part 1: guide for life cycle management. *IEEE Std 24748-1-2011*, pages 1–96.

[ISO42010, 2011] ISO42010 (2011). Iso/iec/ieee systems and software engineering – architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pages 1–46.

[Janietz, 2018] Janietz, C. (2018). Enhancing enterprise architecture models using application performance monitoring data (work in progress). `https://wwwmatthes.in.tum.de/pages/ybzuz28r8mq9/Master-Thesis-Christopher-Janietz`. Accessed: 2018-06-14.

[Jonkers et al., 2003] Jonkers, H., van Burren, R., Arbab, F., de Boer, F., Bonsangue, M., Bosma, H., ter Doest, H., Groenewegen, L., Scholten, J. G., Hoppenbrouwers, S., Iacob, M. E., Janssen, W., Lankhorst, M., van Leeuwen, D., Proper, E., Stam, A., van der Torre, L., and van Zanten, G. V. (2003). Towards a language for coherent enterprise architecture descriptions. In *Seventh IEEE International Enterprise Distributed Object Computing Conference, 2003. Proceedings.*, pages 28–37.

[Kleehaus et al., 2018] Kleehaus, M., Uludag, O., Schaefer, P., and Matthes, F. (2018). Microlyze: A framework for recovering the software architecture in microservice-based environments. In *30th International Conference on Advanced Information Systems Engineering (CAISE Forum)*, Tallin, Estonia.

[LeanIX, 2016] LeanIX (2016). Survey 2017, beyond agile: Is it time to adopt microservices? `https://www.leanix.net/de/download/Microservices-Study/`. Accessed: 2018-06-14.

[Matthes, 2008] Matthes, F. (2008). Softwarekartographie. *Informatik-Spektrum*, 31(6):527–536.

[Matthew MacKenzie et al., 2006] Matthew MacKenzie, C., Laskey, K., Mccabe, F., F. Brown, P., and Metz, R. (2006). Reference model for service oriented architecture 1.0. *OASIS standard*.

[Mauro, 2015a] Mauro, T. (2015a). Adopting microservices at netflix: Lessons for architectural design. `https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/`. Accessed: 2018-06-14.

[Mauro, 2015b] Mauro, T. (2015b). Adopting microservices at netflix: Lessons for team and process design. `https://www.nginx.com/blog/adopting-microservices-at-netflix-lessons-for-team-and-process-design/`. Accessed: 2018-06-14.

[Microsoft, 2018] Microsoft (2018). Typescript programming language. `https://www.typescriptlang.org/`. Accessed: 2018-06-14.

[Netflix, 2018] Netflix (2018). Netflix/eureka: Aws service registry for resilient mid-tier load balancing and failover. https://github.com/Netflix/eureka. 2018-06-14.

[Netsil, 2018] Netsil (2018). Netsil: Universal observability and monitoring for modern cloud apps. https://www.netsil.com/. 2018-06-14.

[Newcomer et al., 2004] Newcomer, E., Haas, H., Orchard, D., McCabe, F., Ferris, C., Champion, M., and Booth, D. (2004). Web services architecture. W3C note, W3C. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[Newman, 2015] Newman, S. (2015). *Building microservices*. O'Reilly Media, Sebastopol, CA.

[Novoseltseva, 2017] Novoseltseva, E. (2017). Benefits & examples of microservices architecture implementation. https://apiumhub.com/tech-blog-barcelona/microservices-architecture-implementation/. Accessed: 2018-06-14.

[Olliffe, 2015] Olliffe, G. (2015). Microservices : Building services with the guts on the outside. https://blogs.gartner.com/gary-olliffe/2015/01/30/microservices-guts-on-the-outside/. Accessed: 2018-06-14.

[OpenZipkin, 2018] OpenZipkin (2018). Openzipkin - a distributed tracing system. https://zipkin.io/. Accessed: 2018-06-14.

[Pettitt, 2018] Pettitt, C. (2018). dagre - graph layout for javascript. https://github.com/dagrejs/dagre. Accessed: 2018-06-14.

[Reinhold, 2016] Reinhold, E. (2016). Rewriting uber engineering: The opportunities microservices provide. https://eng.uber.com/building-tincup/. Accessed: 2018-06-14.

[Richardson, 2017] Richardson, C. (2017). The scale cube. www.microservices.io/articles/scalecube.html. Accessed: 2018-06-14.

[Schäfer, 2017] Schäfer, P. (2017). Eine prototypische implementierung zur erkennung von architekturänderungen eines verteilten systems basierend auf unterschiedlichen monitoring datenquellen.

[Sigelman et al., 2010] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. (2010). Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc.

[Steigerwald, 2017] Steigerwald, L. (2017). Using distributed traces for anomaly detection.

[The Open Group, 2018a] The Open Group (2018a). The togaf standard, version 9.2.

[The Open Group, 2018b] The Open Group (2018b). Togaf worldwide. `http://www.opengroup.org/subjectareas/enterprise/togaf/worldwide`.

[Urbaczewski and Mrdalj, 2006] Urbaczewski, L. and Mrdalj, S. (2006). A comparison of enterprise architecture frameworks. *Issues in Information Systems*, 7(2):18–23.

[Webber, 2016] Webber, K. (2016). Revitalizing aging architectures with microservices. `https://www.youtube.com/watch?v=SPGCdziXlHU`.

[Winter and Fischer, 2006] Winter, R. and Fischer, R. (2006). Essential layers, artifacts, and dependencies of enterprise architecture. In *2006 10th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW'06)*, pages 30–30.

[Wootton, 2014] Wootton, B. (2014). Microservices - not a free lunch! `http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html`. Accessed: 2018-06-14.

[Zachman, 1987] Zachman, J. A. (1987). A framework for information systems architecture. *IBM Systems Journal*, 26(3):276–292.

[Zirkelbach et al., 2018] Zirkelbach, C., Krause, A., and Hasselbring, W. (2018). On the modernization of explorviz towards a microservice architecture. In *Combined Proceedings of the Workshops of the German Software Engineering Conference 2018*, volume Online Proceedings for Scientific Conferences and Workshops, Ulm, Germany. CEUR Workshop Proceedings.