



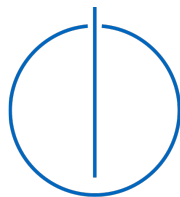
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Information Systems

**Improving the Developer Experience of
API Consumers using Usage Scenarios
and Examples**

Arif Cerit



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Information Systems

**Improving the Developer Experience of API
Consumers using Usage Scenarios and
Examples**

**Verbesserung der Developer Experience von
API Konsumenten durch
Nutzungsszenarien und Beispiele**

Author: Arif Cerit
Supervisor: Prof. Dr. Florian Matthes
Advisor: Gloria Bondel, M. Sc.
Submission Date: September 06, 2019

I confirm that this master's thesis in information systems is my own work and I have documented all sources and material used.

Munich, September 06, 2019

Arif Cerit

Abstract

Working with Application Programming Interfaces (APIs) has become an important part of a software developer's everyday work. Developers (hereafter called "API consumers") heavily rely on API documentation and its different components. Previous work has shown that usage scenarios and examples are key resources to learn and use APIs. While there is a substantial body of knowledge regarding API documentation, only a fraction of it is specifically dedicated to usage scenarios and examples. This body of knowledge lacks research regarding the individual contribution of these parts of the documentation. In particular, there is a need for studies observing the effects of certain resources on developer Experience (devX). Further, the challenges in this field are mostly studied from a developer's perspective. These previous studies create a one-sided view on the problem and leave out the needs of API providers who create the documentation in the first place.

This master's thesis aims to fill this research gap by providing a holistic view including API provider and consumer. We carry out interviews with IT professionals at a large multinational software vendor and elicit knowledge types to include in the documentation. Further, we create API documentation optimized for developer experience based on features from previous research and our interviews. We evaluate this artifact with two groups of software developers who we oppose with the same tasks but different API documentation to study the impact of our optimizations. We observe developer behavior and interview participants to make grounded implications about the effects on the developer experience.

Our findings reveal that improvements to usage scenarios and examples of API documentation show significant positive effects on performance, task success, and perceived usability. Based on our quantitative and qualitative analysis, we give recommendations concerning features in API documentation that seemingly yield improvements to the developer experience. Our results uncover that simply presenting examples does not unfold their full potential. The critical characteristics highly valued by API consumers are the complexity, coverage, or references to other resources. Further, the results indicate that the impact of improvements to API documentation highly depend on the API itself.

Contents

Abstract	iii
List of Figures	vii
List of Tables	ix
1. Introduction	1
1.1. Motivation	1
1.2. Research Questions & Objectives	2
1.3. Research Approach	3
2. Foundations	7
2.1. API Documentation	7
2.2. API Usability	11
2.3. Developer Experience	13
3. Related Work	17
4. Challenges and Knowledge Types	21
4.1. Approach	21
4.1.1. Literature Review	21
4.1.2. Semi-structured Interviews	21
4.2. Concepts in Literature	25
4.3. Challenges	28
4.4. Knowledge in API documentation	30
4.4.1. Quality Attributes & Internal Attributes (Q)	31
4.4.2. Functionality & Behavior (F)	36
4.4.3. Control-Flow (CF)	41
4.4.4. Concepts (C)	42
4.4.5. Directives (D)	44
4.4.6. References (R)	45
4.4.7. Structure (S)	46
4.4.8. Purpose & Rationale (P)	47
4.5. Discussion	47

5. Implementation	55
5.1. Selecting the Features & API	55
5.1.1. Selected Features	55
5.1.2. Project "Compass"	56
5.2. Process & Technologies	59
5.2.1. Designing the API documentation	60
5.2.2. Architecture of the API documentation	61
5.3. Features & Views	64
5.3.1. Overview	64
5.3.2. Basic	65
5.3.3. Advanced	70
5.3.4. Possible extensions	73
6. Case Study	77
6.1. Case Study Design	77
6.2. Detailed Case Description	80
6.3. Results	83
6.3.1. Quantitative Analysis	83
6.3.2. Qualitative Analysis	88
6.4. Recommendations	89
7. Discussion	93
7.1. Key Findings	93
7.2. Limitations	95
8. Conclusion	97
8.1. Summary	97
8.2. Future Work	98
A. Interview Guide - Challenges & Knowledge Types	101
B. Case Study - Material	105
C. Case Study - Results	109
Bibliography	111

List of Figures

1.1. Mapping the research approach to the main questions and chapters. . . .	5
2.1. API providers and consumers in the API value chain (adopted from [6]).	8
2.2. Two approaches exist to develop an API and its specification or documentation (adopted from [48]).	9
2.3. Taxonomy for knowledge types in API documentation [25].	11
2.4. Quality attributes of APIs affecting API consumers (adopted from [44] p.8).	12
2.5. The "UX Honeycomb" depicts the seven aspects of UX [30].	14
2.6. The conceptual framework of devX includes all aspects of cognitive experience [7].	15
4.1. The concept matrix indicates which publication contributed to a concept.	26
5.1. An overview of how Compass is used to manage applications and runtimes.	59
5.2. The process of defining our artifacts included a learning phase and an iterative collaboration and design phase.	60
5.3. The overall architecture of the documentation we built.	62
5.4. Jekyll takes HTML, Markdown and CSS as an input and create a static web page.	63
5.5. GraphQL Playground supports API consumers in creating requests for a given API endpoint.	63
5.6. The overview landing page answers what Compass is on a high-level and what problem it solves.	67
5.7. The components are explained using a high-level diagram with each component.	67
5.8. The most common flows in Compass are illustrated using sequence diagrams.	68
5.9. In getting started, we explain important tools, such as the playground, in detail.	68
5.10. The tutorial contains text segments and simple example requests.	69
5.11. The samples are presented as a list with short titles and the code itself. We only cover the most important queries.	70
5.12. The glossary explains each special term in its context and references further resources in the documentation.	71
5.13. The tutorial starts with a definition of the outcome and explanation of the story.	72

5.14. The samples are guided by textual descriptions with references to the type and object definitions in the specification.	73
5.15. The best practices are specific to GraphQL and the playground. They consist of an explanation and an example to demonstrate the practice. . .	74
5.16. Helper buttons embedded in an examples can support developers in copying and executing of code.	74
5.17. The Twilio API documentation offers an option to instantly rate the current page (accessed on 13.08.19).	75
5.18. Swagger UI offers a playground that includes information about the model and types (accessed on 13.08.19).	76
6.1. The procedure of our case study that was executed with each participant.	82
6.2. Mean durations for each phase of the study. The error bars stand for one standard deviation into both directions.	84
6.3. Total points of participants in the three tasks.	85
6.4. The bars depict success rates of API requests. The dotted red lines represent the mean success rates per group.	86
6.5. Mean relative usage of the documentation sections. The error bars symbolize one standard deviation into both directions.	86
6.6. The SUS score for each participant. The dotted red line symbolizes the mean values of group A and B.	87
B.1. We prepared tables and tally lists to document the times, requests, and features usages.	106
B.2. The SUS consists of ten questions with a Likert scale.	107

List of Tables

4.1. Participants for our semi-structured interviews.	24
4.2. Results of how often the presented problem occurs.	28
4.3. Results of how relevant the presented problem was for our interviewees.	29
4.4. The occurrence of the knowledge types expressed as a fraction of participants and all codings.	31
4.5. All knowledge types with the corresponding implications stated in the previous section.	48
5.1. The criteria to find basic and advanced features for the prototypes.	56
5.2. The implications that we integrate in the basic and advanced artifacts.	57
5.3. The organizational and technical criteria applied to select an API for our implementation.	58
5.4. The differences between the basic and advanced artifacts listed per section in the documentation.	66
6.1. An overview of our participants and their professional experience as software developers.	81
6.2. The three tasks differ in difficulty, topic, and required solution steps.	82
C.1. The summary of the recommendations we stated based on our case study.	110

1. Introduction

This chapter explains why API usage scenarios and examples are an important research area and how our contributions to the field are characterized. Section 1.2 contains the research questions answered by this thesis and explain how they relate to the topic. Lastly, we outline our research approach and the structure of this thesis.

1.1. Motivation

Application Programming Interfaces (APIs) are central to many modern software architectures [32]. They provide high-level abstractions that facilitate programming, support the design of distributed software applications and the reuse of code [38]. The importance of APIs increased when businesses started to offer their products and services over the internet [2]. New software is rarely developed from scratch. Common functionality from existing APIs is used and adapted to the developer's needs. This applies to web APIs such as Google Maps but also Software Development Kit (SDK) APIs like .NET. Therefore, software developers (also called "API consumers") constantly learn and use an unknown API [38]. Usually, the API is published together with documentation containing reference information, examples, tutorials, and other resources created by an API provider. The API documentation serves as the main learning resource for developers and at the same time poses the biggest obstacles and usability issues [24, 39]. Usage examples were mentioned frequently as a major learning obstacle found by [39]. A usage example of an API can be as simple as a code snippet showing how to use the API. Moreover, examples are an important part of usage scenarios or tutorials that demonstrate how a more complex functionality is implemented. Since API documentation serves as the main communication medium between API provider and consumer, the challenges are not restricted to one side. The API providers have little knowledge of how their API and documentation is perceived and used [49]. This results in poor usability on the API consumer side. In this sense, "usability" also includes learnability, efficiency, or usefulness of the API documentation. This holistic view is referred to as "developer experience" (devX) and is an important design goal for API documentation [32].

Even though previous studies revealed that the problems with API documentation are present on the API provider and consumer side, most of the qualitative research is solely focused on API consumers [38, 39, 24, 31, 46]. While researching developers yields essential insights into the needs and potential improvements, it fails to create a holistic

view of the problem. The API provider side includes not only software developers but also software architects and product owners which might be sources of valuable knowledge for API documentation. There is a substantial body of knowledge about API documentation as a whole without a particular focus on individual resources [39, 46]. The results of these studies are difficult to apply practically because they often give general recommendations concerned with all parts of the documentation. Therefore, more research has been conducted focusing on individual resources within the documentation such as examples and tutorials. However, previous research rather investigated the effect of usage examples in general, instead of investigating devX as a whole. There is a broad consensus that usage scenarios and examples are impactful resources, but there is a substantial lack of the fine-grained contribution of different characteristics [28]. Finally, recent studies explicitly call for research regarding observable benefits from "optimized API documentation" [29]. The resulting observations might help to identify what information improved the devX. Studies show that there is an observable effect if the API documentation contains examples [42, 53]. However, existing research does not compare API documentation where optimizations are fine-grained enough to give evidence about the cause. This would more likely satisfy the research gap identified by [28].

This master's thesis fills this gap by conducting interviews including API providers and consumers, revealing effects on devX of optimized API documentation and analyzing the fine-grained contribution of usage scenarios and examples. With semi-structured interviews at a multinational software vendor, we enrich existing findings on usage scenarios and examples with additional empirical evidence. Also, this study contributes with a more holistic perspective by interviewing IT professionals in different roles as well. Further, we build an exemplary API documentation containing optimizations of usage scenarios and examples. Our improvements are based on existing knowledge from relevant literature and include the most promising findings from our interviews. Finally, we conduct a case study with professional software developers to study the effects of our optimizations in a natural setting in practice. As part of this study, we aim to trace the observed effects of our improvements by collecting metrics, observing and interviewing the IT professionals. With this, we show which aspects of devX get affected by the improved documentation. Our most significant findings are presented as concrete recommendations.

1.2. Research Questions & Objectives

This thesis aims to research the devX in API documentation holistically and especially build artifacts based on findings combined with relevant literature. In this section, we formulate three research questions and explain their respective goals.

RQ1: What are the approaches and concepts to create and publish usage scenarios and examples?

The first research question aims to analyze the state-of-the-art in API usage scenarios and examples. To embed our work in the existing knowledge base and to utilize previous findings, a thorough literature review is necessary. We use the insights from answering this question to clarify our contribution to the field and focus our study on gaps in the literature. As part of this thesis we evaluate the concepts which are not yet exhaustively covered in previous work and reoccur in our study. The knowledge about previous research also helps us prevent a re-evaluation of thoroughly studied findings which might be redundant and less valuable for the field.

RQ 2: What are the knowledge types that must be included in usage scenarios and examples?

In addition to findings from previous research, we aim to contribute empirical evidence by conducting interviews at a large multinational software vendor. We organize 14 semi-structured interviews with IT professionals embodying different roles. The interview data is intended to provide further empirical evidence for the knowledge base and propose potentially useful characteristics. We inspect the interviews for knowledge which is valuable for API providers who want to optimize their usage scenarios and examples. Furthermore, our findings are compared to previous literature to show whether we can confirm existing findings or propose new potential improvements. Our answer to this research question is based on the qualitative data from interviews and previous research.

RQ 3: How can usage scenarios and examples be leveraged to improve the developer experience for API consumers?

Even though participants mention requirements in interviews, there is no evidence that we can practically apply every finding and thus improve devX. Since devX includes more aspects than just the perceived usability or learnability, we conduct a case study to observe API consumers in a natural setting to get practical insights. Again, we select professional developers at a software company and confront them with real tasks to be solved using an unknown API. The deeper understanding resulting from this case study is used to formulate recommendations based on multiple data sources with a proven effect on devX.

1.3. Research Approach

In this thesis, we follow a design science research (DSR) approach as proposed by Hevner et al. [12, 11] and Peffers et al. [36]. The DSR paradigm is centered around the design and evaluation of artifacts combining theoretical foundations in a knowledge

base with business needs from an environment including a defined problem. The embodiment of design science in our study uses the three cycle view introduced in [11]. We ensure scientific rigor by grounding this thesis in existing theories, the relevance originates from our research conducted with a software company, and finally, we design and evaluate API documentation based on findings from literature and environment. Figure 1.1 provides an overview of the research design.

In Chapter 2 & 3, we conduct a structured literature review to get an overview of the knowledge base relevant to our thesis. Semi-structured interviews proved to be a valuable method to get a deep understanding of the matter with the possibility to dive deeper into promising topics. As part of the DSR, it is necessary to include relevant knowledge in the design phase. The outcome of this part is a concept matrix presented in Chapter 4. Hereby, we answer our RQ1 which asks for the approaches existing in literature.

In Chapter 4, we carry out semi-structured interviews with IT professionals from a large software vendor. Since we study a practically relevant problem, we aim to create findings originating from the field. We analyze the interviews thoroughly by applying a taxonomy for knowledge in API documentation presented by [25]. The resulting artifact is a set of knowledge types containing patterns observed across multiple interviews. This answers RQ2 in the sense that it provides further evidence for other studies' findings about knowledge in API documentation.

In Chapter 5, this thesis presents the two API documentation developed for our study in collaboration with API providers. One documentation contains only essential features necessary to learn and use an API productively. The other one is optimized for an improved developer experience based on our insights from interviews and the literature. The artifacts serve as a proof-of-concept for our identified features and also enable us to conduct further studies to investigate the difference in devX they make.

Chapter 6 presents an embedded single-case study with software developers to answer RQ 3. A case study is a well-suited research method since it allows us to investigate contemporary phenomenon in their natural setting with only little control over the participant's behavior [52]. In our situation, we confront developers with an unknown API and programming tasks of varying difficulty. This study aims to observe, measure, and understand the actual effects on the devX caused by our optimizations. We collect quantitative and qualitative data to have multiple sources of evidence for our inferences.

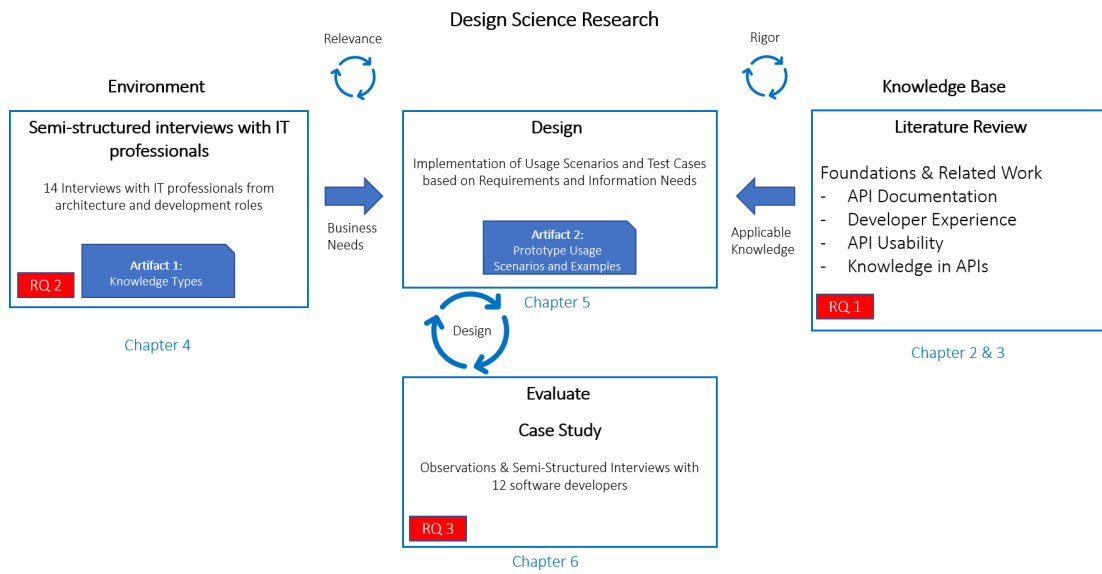


Figure 1.1.: Mapping the research approach to the main questions and chapters.

2. Foundations

This chapter describes the theoretical foundations of this thesis. We aim to explain studies relevant to our domain and build a common understanding of the terminology and concepts we use. First, we explain fundamentals about API documentation and put special focus on some particularly relevant aspects such as knowledge types. Next, we outline some of the theories around API usability. The last section concerns the concepts of user and developer experience.

2.1. API Documentation

API documentation comprises different resources designed to facilitate the learning and use of the API [29]. This section gives a general overview of the fundamentals in API documentation. Further, we put a focus on API usage scenarios and examples. Lastly, the theory behind API knowledge types is presented.

Fundamentals

Following, we explain important terminology concerning API documentation. We present the groups of stakeholders, the common documentation structure, and provide basic lifecycle information.

Stakeholders

There are mainly two groups that are important in the context of API documentation. The **API providers** are responsible for the API in general. In the API value chain, they are described as a group or individuals who create the API and related deliverables. The API providers do not only comprise technical professionals but also roles beyond software development such as business analysts. The **API consumers** or API users are usually software developers who use the API to generate new value. This might be done by creating an app that makes use of the offered API [6]. The actors in the value chain are illustrated in Figure 2.1. The figure shows how API providers use an API to deliver a business asset to API consumers and ultimately to customers [6]. Some studies use the term "API developers" to describe one of the two stakeholders. In a substantial part of the literature, this term stands for the API consumers [25, 29, 28]. Many other studies use "API developer" interchangeably with "API provider" which is contradicting [41, 34, 45]. For these reasons, we do not adopt this description but rather use API consumer and provider as explained.



Figure 2.1.: API providers and consumers in the API value chain (adopted from [6]).

Another important role in the context of API documentation is the technical writer. Software developers write deeply technical documentation. The technical writers create the parts of the documentation which help in actually completing tasks and solving specific problems [26]. For example, this includes the explanation of concepts, tutorials, or conventions. The technical writer is usually part of the API provider team but does not have any code-related tasks [26].

Structure

There is no golden rule on how to design API documentation but some practices can be found in almost any software. Most documentation contains a reference and conceptual part. The **API reference** contains information for each API element and specifically informs API consumers about methods, types, and expected parameters [25]. The information in the reference documentation is mostly intended to explain the communication with the API in software development terms. The reference documentation is a necessary part of the API documentation since it enables developers to solve most of the technical questions [18]. In contrast, the **conceptual topics** in the documentation are more concerned with "guiding the user" and help with the usage that goes beyond single API requests [18]. Common conceptual topics are the following sections [18, 47]:

- Getting started
- Samples and Tutorials
- Authentication and Authorization
- Status and Error Code
- Glossary
- Best Practices

Each topic gives a high-level view of the API by including additional knowledge besides the mere technical specification. For example, code samples and tutorials explain not only the "how" but also "why" of used API elements. This helps API consumers to build a practically relevant knowledge that can easily be transferred to their implementation [18, 25].

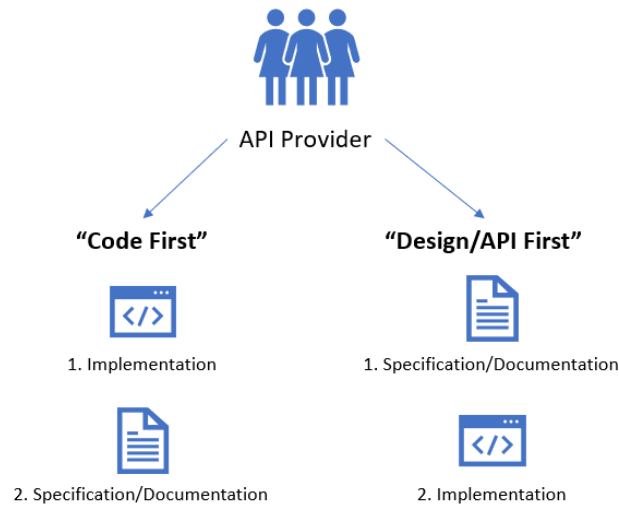


Figure 2.2.: Two approaches exist to develop an API and its specification or documentation (adopted from [48]).

Lifecycle

In practice, two ways mostly dominate the ways of developing an API and related deliverables like specification and documentation. As depicted in Figure 2.2, one approach is called "Code First" where the implementation precedes the documentation. In this variant, the software developers can start very early with the creation of a functional API. Once there is a working version of the API, additional artifacts like the specification or documentation are developed. This results in a short time-to-market for the API but at the cost of good quality [48]. An alternative approach is "API First" or "Design First", where the documentation and structure of the API are created before the actual implementation. Hence, the developers only implement the minimal required code to satisfy the specification. The APIs developed in this approach often have a high quality and provide a great devX because of a sophisticated documentation [48]. Both ideas deliver the same artifacts. However, there are significant differences in devX and quality. For an enhanced devX, the API first is increasingly adopted [10].

API Usage Scenarios and Examples

Multiple studies have consistently shown that API usage scenarios and examples are among the most important parts of an API documentation [38, 46, 28]. However, there is no consensus regarding definitions or relations to other resources. In this section we introduce our notion of usage scenarios and examples together with the literature we used as a foundation.

What are API Usage Scenarios?

The expression "API usage scenarios" generally refer to a sequence of API requests. This sequence is recommended, required, or common to implement a specific functionality [17]. Further, usage scenarios are described as a complex sequence of calls which reflect patterns intended by providers [39]. Exemplary usage scenarios for an API can serve as the entry point for API consumers. As a central component of any API documentation, usage scenarios can help understand the purpose and context of an API and their manual construction requires good knowledge of the domain and API itself [27].

In literature and practice, usage scenarios are presented using formal specification languages like UML or as a sequence of examples in the form of a tutorial [31, 17].

What is the difference to use cases and tutorials?

An API use case is defined as an informal description of what problems can be solved using the API. Usage scenarios provide an executable, formal specification of how a problem can be solved by concretely applying the functions of the API [14].

Most of the key literature in the field uses the terms "tutorial" and "usage scenario" interchangeably [33, 29]. A tutorial is defined as a sequence of examples that implements a non-trivial functionality [39]. This notion of tutorial is close to our definition from the previous paragraph. We adopt this conventionality and use tutorial in the sense of usage scenarios.

What is the relationship between API usage scenarios and test cases or (code) examples?

API test cases and usage scenarios are closely related in this thesis. In our context, test cases serve as API usage examples which is consistent with most of the literature [14, 37]. In contrast to usage scenarios, test cases only cover standard calls to the API. Since usage scenarios cover multiple calls, they can be seen as a concatenation of usage examples [33]. Instead of clearly separating test cases from usage scenarios, we rather regard them as a foundation.

The notion of API usage scenarios in this thesis is in line with the literature. We use the term to describe a complex sequence of API calls to implement a certain functionality within a domain. Usage scenarios differ from use cases in the sense that they are executable and are described formally. They also go beyond mere test cases since usage scenarios include complex interactions.

API Knowledge Types

The knowledge contained in API documentation ranges from general overviews of concepts over technical details about features up to ancillary knowledge to understand

an API [25]. A knowledge type describes a "pattern of knowledge" in API documentation. The categorization of this knowledge into comprehensive categories (also called "taxonomy") provides a valuable foundation for discussions around documentation improvement. Maalej & Robillard (2013) conducted the first empirical study to create a knowledge type taxonomy for API documentation [25]. Figure 2.3 depicts all 12 categories of the taxonomy. We briefly present the types of knowledge and the empirical findings based on over 5000 analyzed API documentation samples.

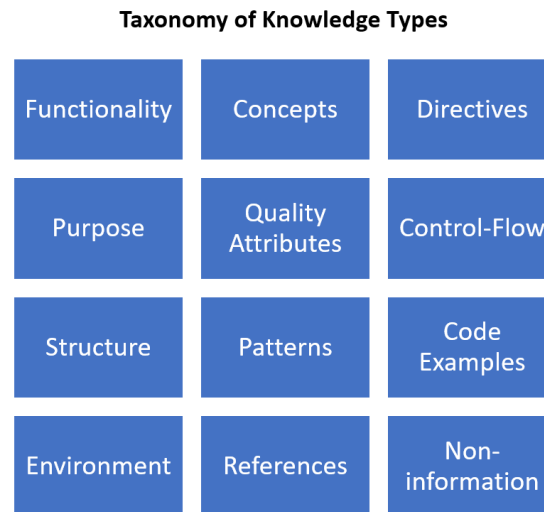


Figure 2.3.: Taxonomy for knowledge types in API documentation [25].

The prevalent knowledge in API documentation concerns API **Functionality** and **Structure**. These two types often occur together to draw a picture of what the API can do and where to find the features. This information is the baseline for implementations based on an API [25]. Less common types of knowledge are **Concepts** and **Purpose**. The **Concepts** refer to terminology and details about the domain and design of the API while **Purpose** knowledge aims to answer the "why" questions of API consumers. Both are necessary to get background information about the API but are rarely provided. The creation of the taxonomy also revealed that a substantial part of the documentation contains "non-information" knowledge, i.e., uninformative boilerplate text [25]. The taxonomy acts as a vocabulary for communication about API documentation. For example, the intuitive definition of "Pattern" does not match the meaning for API documentation. Here, "Pattern" stands for the implementation of certain scenarios. In our work, we use "Usage Scenarios" as a replacement for the "Pattern" knowledge type.

2.2. API Usability

The field of API usability focuses primarily on software developers whereas "classical" usability rather refers to end users [27]. The research in this field has identified quality

attributes and obstacles which especially apply to API consumers. Stylos (2009) presents a set of quality attributes for API usability and connects them to API stakeholders [44]. The API consumers are most affected by the API usability attributes which we depict in 2.4.

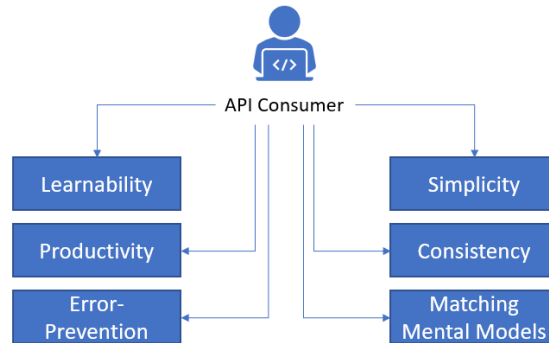


Figure 2.4.: Quality attributes of APIs affecting API consumers (adopted from [44] p.8).

For the following attributes, the trivial definition is sufficient in our context: learnability, productivity, simplicity, consistency. The error-prevention attributes refers to how well the API itself prevents errors [44]. For example, if the user is about to use an API element in a wrong way, the API should prevent that or indicate it at least. The attribute "matching mental models" stands for the ability of an API to work as intended by API consumers [44]. The consumer implicitly builds a mental model of how the API is structured and used. This model is often confirmed or corrected by interaction with the API.

Efficient learning is hampered by obstacles in different parts of the API. A survey with professional software developers revealed that the API learning obstacles mainly divide into the following resources [38]:

- **Resources:** inadequate or absent resources (documentation, examples, tutorials)
- **Structure:** design or structure of API
- **Background:** related to API consumer's prior experiences
- **Technical environment:** technical characteristics of API (hardware, infrastructure, dependencies)
- **Process:** related to the process of working with APIs (time pressure, interruptions)

In Robillard's (2009) study around 38% of respondents mentioned "Resources" as the main learning obstacle especially referring to the API documentation [38]. And within this category, the usage examples were mentioned as the most frequent issue. Further studies confirmed the severe obstacles in API documentation and especially focused

on these issues in interviews. This research resulted in recommendations addressing common challenges [39]:

- Documentation should support where correct usage is not self-evident.
- Examples involving more than one call will be more helpful than single-call examples.
- Examples should demonstrate "best practices" for using an API.
- Mapping scenarios to API elements is particularly helpful ("bridge documentation")
- Explicit statements on performance, error-handling, or side effects are potentially useful for learning an API.
- Boilerplate documentation wastes developer's time and never answers questions ("non-information").
- Developers might look for point of entries. Fragmented documentation can be overwhelming.

The recommendations present the first concrete advice for mitigating the impact of API learning obstacles. Following studies have investigated the documentation issues from a different perspective, for example by asking about the content and presentation [46]. According to a survey with IBM software professionals, the problems caused by the documentation content are more frequent and serious than those coming from the presentation of content. More precisely, ambiguous and incomplete content was found to be a major obstacle for API consumers.

2.3. Developer Experience

"Developer experience" (devX) is an idea influenced by the "user experience" concept (UX) to better understand, analyze, and design environment for software developers [7]. A common assumption in the literature is that an improved devX has a positive impact on software development project success.

The UX covers "a person's perceptions and responses that result from the use or anticipated use of a product, system, or service" [16]. The definition of UX is still evolving but the important part of it is the notion of experience. "Experience" is defined as the encounter with a system with a defined begin and end. The verb "experience" stands for the perceptions and interpretations of an individual during an experience [7]. Thus, the UX definition encompasses various aspects of a system such as the usability, value, or accessibility. Moorville & Sullenger summarized the most important UX aspects in the "UX Honeycomb" depicted in Figure 2.5 [30].



Figure 2.5.: The "UX Honeycomb" depicts the seven aspects of UX [30].

This definition heavily influences the understanding of devX. In "Developer Experience", a "developer" is a person actively engaged in software development and "experience" refers to involvement, not to be confused with "being experienced" [7]. As the perception of UX shifted from user interface design to the whole experience, more aspects got involved in the research. In devX the aspect of the mind and its capabilities became a topic of interest [7]. Fagerholm & Munch (2012) formulated a definition of devX under consideration of the psychological notion of the mind. This resulted in a conceptual framework of devX depicted in Figure 2.6.

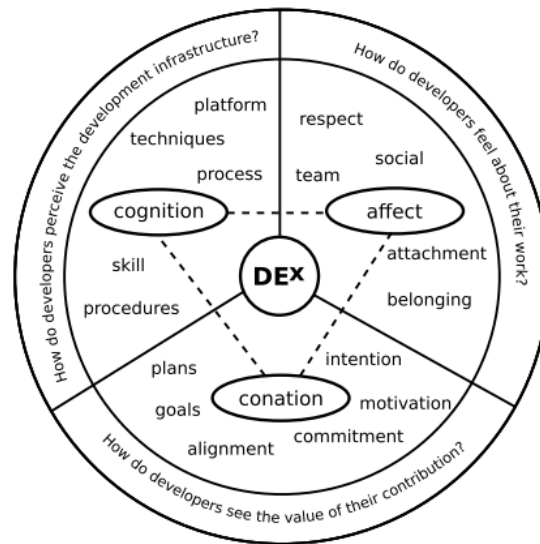


Figure 2.6.: The conceptual framework of devX includes all aspects of cognitive experience [7].

3. Related Work

This chapter presents and summarizes key publications closely related to our work. First, we cover research around problems and knowledge needs of developers in using API documentation. Second, similar papers focusing on enriching documentation with examples are explained.

Problems and Knowledge Needs in API Documentation

Studying the problems with documentation and the needs of stakeholders was investigated in multiple studies. This research area is relevant to this thesis because we aim to use knowledge about API consumers and providers as a foundation for the API documentation.

Nykaza et al. (2002) [35] performed one of the first studies to assess the documentation needs using surveys and interviews with software developers. This study identified that code samples, background knowledge, and an overview are important elements of API documentation. They further revealed that pragmatic documentation resources are perceived as very useful since "no one reads a manual" [35]. This is related to this thesis since we also perform developer interviews on API documentation.

To gain deeper insights into the information needs, **Ko et al. (2007)** [20] observed the activities of professional developers while they solved engineering tasks. The authors found that the different categories of tasks such as writing code or reproducing failures can only be performed well if certain questions are answered. Many knowledge needs were found to be unsatisfied which led to the deferral of tasks. Further, this study shows in detail how problems with matching software to scenarios obstructs developer productivity.

A comprehensive study of learning problems with APIs was conducted by **Robillard (2009)** [38]. Using a survey among Microsoft software developers, the author found five categories of API learning obstacles mostly related to API documentation. Some of the frequently mentioned issues were insufficient examples, missing documentation resources, or inadequate presentation. This paper indicated strongly that API documentation is the biggest obstacle in working with APIs. The paper also yielded an understanding of examples in the API documentation. While examples were perceived as one of the most important resources, it was also the reason for many learning obstacles. Robillard's findings indicated that the use of examples in the API documentation is also connected to other aspects e.g. conceptual knowledge.

Following up on Robillard's (2009) study, **Robillard & DeLine(2011)** [39] confirm the

findings on learning obstacles with a survey and interviews with over 400 software developers. Again, the most important issues were related to API documentation and the different resources in it. The authors propose five factors to consider in the API documentation: intent, code examples, matching APIs and usage scenarios, format, and penetrability of the API. The findings from this paper serve as a comparison baseline for challenges and obstacles found in our work.

The work of **Uddin & Robillard (2015)** [46] focuses on the reasons behind problems with APIs from a software developer perspective. The IBM survey with over 300 participants revealed that more problems in the documentation are concerned with the content rather than presentation. This is relevant as we also explicitly interview stakeholders about problems with documentation. It turns out that API consumers care the most about unambiguous, high-quality content.

Most recently, **Meng et al. (2018)** [29] and **Meng et al. (2019)** [28] conducted studies concerning the knowledge needs and usage behavior of API consumers. Meng et al. (2018) and surveyed software developers about learning and using API documentation. They found that depending on the working style of the developer, different resources are perceived in various ways. The concept-oriented developer is more likely to work through the API overview and getting started sections before implementation attempts are made. In contrast, a code-oriented developer starts with examples to "get a feeling" for the API [29]. This study gives us valuable knowledge about the API consumers and how API providers can satisfy the knowledge needs of both types of developers. Following this study, Meng et al. (2019) researched the usage behavior in an observational study. They collected quantitative data regarding task success and time spent on documentation sections. Further, all participants were interviewed to ask about barriers while working with the API. This study revealed that conceptual information should be an integral part of other API resources such as examples. Besides fast search and clear navigation, the background knowledge and connections within the documentation were stated as useful. The two papers of Meng et al. help us in designing API documentation with knowledge needs and favored sections in mind.

Our study uses comparable methods to qualitatively research the problems and knowledge needs of API consumers. The surveys and interviews conducted in the mentioned studies are valuable foundations for our question design. Also, our findings can be compared to these papers to either provide new empirical evidence or identify discoveries.

Enriching API Documentation with Examples

This thesis concerns itself with using usage scenarios and examples as a means to improve the developer experience. Therefore, the research on examples in API documentation is closely related to our approach.

Nasehi et al. (2012) [34] studied characteristics of good code examples using knowledge

from StackOverflow, a question and answer forum for software developers. The answers on StackOverflow often contain an executable code snippet to solve a problem. Further, the content on StackOverflow is cross-validated by a large community of professional software developers which makes it a valuable and reliable knowledge source. They found that the favored answers from the forum had characteristics such as conciseness, highlighting, and references to other resources. An important conclusion of this study is the strong evidence that examples and descriptions are "inseparable elements" [34]. These attributes of good examples are relevant for this thesis and especially helpful in the design phase of our artifacts.

The paper by **Sohan et al. (2017)** [43] is among the closest related works to our thesis. This study includes a controlled study with software developers in which participants solve tasks using API documentation. Sohan et al. formed two groups of participants with different documentation. The optimization in this study was to have usage examples in one group whereas the other group worked entirely without usage examples. Furthermore, the group without examples criticized the lack of helpful resources which reflects the crucial role of usage examples. The paper proposes four recommendations for usage examples in the API documentation. This is highly relevant work as it shows whether usage examples affect at all. They found that there are substantial differences in successful API calls and task success.

Zhang (2019) [53] applied a mining approach of StackOverflow related to Nasehi et al. (2012) with the difference of automatically generating examples from the collected knowledge. Their approach is named ADECK and enriches existing documentation with generated samples and usage scenarios. They evaluated ADECK with software developers and found that the enriched documentation increased the developer performance and decreased completion times. Another important insight from this paper is that the generation of examples is feasible and can improve existing documentation. Our thesis conducts a similar experiment with the difference of manually creating usage scenarios and examples. However, this study is related to our thesis because it tests slight differences in the documentation using two groups of software developers.

In summary, we see that there were measurable and even significant effects from enriching API documentation with examples. We plan to include the presented study designs and especially their analyzed metrics into this thesis.

4. Challenges and Knowledge Types

This chapter presents the first artifact of this thesis: the different knowledge types to be included in API usage scenarios and examples. Section 4.1 presents how we used semi-structured interviews to collect qualitative evidence from API consumers and providers. Further, we give an overview of existing concepts for API usage scenarios in the literature. In Section 4.4 we analyze the interview findings to extract the knowledge to be included in usage scenarios and examples based on a taxonomy of knowledge types in API documentation. Finally, we discuss the interview findings with regards to the literature.

4.1. Approach

This section describes the approach used to conduct the literature review and interviews in detail. Prior to our interviews, we carried out a literature review to extract existing concepts and ideas. After, we conducted semi-structured interviews with professionals in API provider and consumer roles at a large multinational software vendor. The data from the interviews was used to elicit requirements and find other knowledge types for API usage scenarios and examples.

4.1.1. Literature Review

We conducted a structured literature review following guidelines from Webster & Watson (2002) to find existing concepts and principles on API usage scenarios and code examples [51]. The goal was to incorporate findings from previous research into artifacts designed in this thesis. Also, the concepts found in literature enabled us to assess whether our findings are thoroughly covered in research or novel. Thereby, our findings can provide further evidence from the field for discovered concepts.

The literature search was carried out using the following digital libraries: ScienceDirect, ACM, IEEE Xplore, and Scopus. In total, we analyzed 51 papers primarily from journals and conferences. The results of the literature review are presented in the form of a concept matrix as proposed by Webster & Watson [51]. We will introduce the discovered concepts and explain them with reference to previous work in that field.

4.1.2. Semi-structured Interviews

We carried out 14 semi-structured interviews with IT professionals mostly using open questions mixed with few closed questions. The interview participants were divided

into two groups according to their job profile. In the following we describe the goals and procedure of these interviews, together with the original questionnaire used to guide the conversation with the participants.

Goals

The goals for our interviews were derived from the research question we aim to answer. This idea is adopted from previous research conducted by Runeson & Höst [40] and Adams [1]. The findings from our interviews try to answer RQ2:

RQ 2: What are the knowledge types that must be included in usage scenarios and examples?

To answer the question, we first intend to elicit knowledge mentioned by multiple participants. By this, we gain robust findings which are less likely to be biased by a single participant. Besides answering the research question, we expect the interview findings to deepen our understanding of the problems faced by API providers and consumers. Another goal of the interviews was to collect diverse data with multiple perspectives by interviewing developers as well as architects and product owners. Lastly, we specifically ask about usage scenarios and examples. In summary, we designed the interviews with following goals in mind:

- Elicit knowledge from API consumers and providers mentioned multiple times
- Holistic view on the problems with multiple perspectives
- Specifically ask about usage scenarios and examples

Questionnaire

Our questionnaire consisted of three sections. First, we explained the problem and asked the participant to rate its frequency, relevance, and explain potential causes. Second, we asked participants about API usage scenarios and examples as a resource for learning and understanding APIs. More precisely, we inquired our interviewees about the usefulness, problems, and effects. Finally, we asked what knowledge developers seek in usage scenarios and how hard it is to find. The full English questionnaire can be found in the Appendix A. The original interview guide is in German but it was translated for two English-speaking participant.

Procedure

We used previous research from Adams [1] and Hove & Anda [15] as a guidance for the design and conduct of our semi-structured interviews.

First, we selected interview participants at the aforementioned software vendor. At the time of this thesis, the author was employed at this company, enabling him to easily

communicate with interviewees and schedule interviews. To find suitable participants, we started by asking employees in lead development and architecture roles. In these informal conversations we discussed who were suitable interview partners from a development and architecture or product perspective. Important selection criteria were whether the person worked in projects where our problem at hand was of significant importance. Further, it was important to select participants from different but related roles to get a holistic view on the topic. This approach is in line with some former surveys and interviews on the topic of API documentation [38, 50]. We compiled a list of 14 candidates to interview (see Table 4.1). Out of these 14 participants, seven were in architecture or product owner roles and the remaining seven worked in software development. The interview was planned for 30 to 45 minutes. The participants with IDs A1, D1, and D2 were interviewed twice in the form of a pre-study which explains the higher duration.

The interview invitations were sent out one week before and contained an advance letter with some background information on the topic. The advance letter is attached in the Appendix A. All interviews were carried out face to face on the premises of the study organization. All interviews expect for two, were conducted in German which was the native language for most of our participants.

The interview questions were derived from our previously mentioned goals with RQ2 in mind. The final interview guide consisted of three sections. Beforehand, we asked the participants about their years of professional experience and current role. Each section contained an explanation of the situation and purpose of the questions followed by two to five questions. In total, we asked eleven questions excluding the demographic section. The sessions were recorded and notes of key statements were taken by the author. During the interview with participant A3, there was a malfunction with the recording device which resulted in a loss of the audio file. For this interview, we have only analyzed the field notes which were significantly less insightful than a transcript.

We conducted a qualitative data analysis following the guidelines of research synthesis by Cruzes & Dyba [5]. First, the interviews were transcribed mostly verbatim but omitting fillers (such as "Ähm") and off-topic passages. In total, we recorded more than ten hours of interview material. The coding of interviews is the process of examining and organizing the data by applying labels to passages in the text [4]. We followed an integrated coding approach, i.e. using inductive and deductive methods [4]. Our method is deductive in the sense that we use an existing knowledge type taxonomy as an orientation for our initial coding scheme. We used the knowledge type taxonomy for API reference documentation by Maalej & Robillard [25] to categorize our codes using an existing framework. We also conduct inductive coding by investigating emerging patterns within the categories of the taxonomy. Based on our evidence we try to sub categorize knowledge types where appropriate.

We used the taxonomy as a guideline for our coding approach, however we did not aim to orient our analysis entirely on the knowledge types proposed by [25]. Therefore,

Table 4.1.: Participants for our semi-structured interviews.

ID	Role	Experience [years]	Duration [hh:mm]
A1	Software Architect	12	01:15
A2	Product Owner	15	00:32
A3	Enterprise Architect	12	-
A4	Enterprise Architect	10	00:30
A5	Software Architect	29	00:36
A5	Enterprise Architect	7	00:44
A7	Product Owner	12	00:45
D1	Lead Developer	20	01:05
D2	Senior Developer	15	01:31
D3	Senior Developer	6	00:40
D4	Developer	4	00:50
D5	Developer	2	00:36
D6	Senior Developer	8	00:41
D7	Senior Developer	20	00:38
Mean		12,3	00:39

we excluded four knowledge types from our analysis. The types "Patterns" and "Code Examples" were omitted because almost all information would fall into this category as we explicitly ask about these in our study. According to [25], the "Environment" refers to boundary conditions of the API such as licensing and compatibility. Even if this might affect the documentation indirectly, we claim that this specific knowledge raises many other subjects which are not focus of this thesis. Further, we did not code the "Non-information" items in our interviews. While we acknowledge that substantial parts of API documentation fall into this category, we did not regard it as relevant in the context of API usage scenarios and examples. In contrast to other parts of the API documentation, usage scenarios and examples are more likely to convey any information.

In summary, we conducted the interview round at a large software enterprise to elicit eight types of knowledge with particular focus on API usage scenarios and examples. We interviewed 14 junior and senior professionals in roles such as development, architecture, and product management. Our approach involved first asking generally about the problems with API documentation and second inquiring interviewees about functionality and knowledge that should be part of usage scenarios and examples. The findings of these interviews are classified using a taxonomy for knowledge types in API documentation which is applied for our purposes. Our interviews are supported by a literature review of thoroughly researched concepts for API usage scenarios and examples. The findings of this review are outlined in the following chapter.

4.2. Concepts in Literature

This section explains the results of the literature review in the form of a concept matrix depicted in Table 4.1. In the review we identified concepts relevant in the context of designing effective API usage scenarios and examples. The listed concepts are thoroughly studied and recommended in previous research and therefore serve as a scientific foundation for our own artifact design.

Usage Scenarios: Mapping to API

A central challenge in learning an API is to understand how a certain functionality within a domain matches the elements of the API. In a greater context, mapping a hypothesis about programs to available tools was even identified as a central learning barrier, also called "gulf of evaluation" [21]. Robillard & DeLine observed in a field study on learning obstacles that scenarios mapping to specific API elements are perceived as helpful by developers [39]. The field study found that API structure and the developer background are major influence factors for a successful match between scenarios and API. On a higher level, the mapping between API and scenarios is required to understand how the system relates "upwards" to its problem domain [39].

In a study with developers, Kuhn & DeLine (2012) found that "mapping task to code" forms a major learning category for API consumers during the initial phase of using an API [23]. After developers have successfully made this mental mapping they continue to search more focused.

Furthermore, the work of Meng et al. confirms the mentioned findings from Robillard & DeLine [28]. In addition, the observation study of Meng et al. found that developers seek to connect found concepts to specific code. The implication of this discovery is that conceptual information presented in the form of usage scenarios must also stress the connection to concrete implementations [28].

In summary, previous studies indicate that usage scenarios and examples might have an increased perceived usefulness if they clearly point out their connection to the API elements.

Code Examples: Tests as Examples

Hoffman & Strooper (2003) proposed executable unit tests as code examples in API documentation. They identified benefits of well-written test cases as code examples such as precision, guaranteed consistency with the code, and program validation [13].

Another study by Nasehi & Maurer (2010) discovered that unit tests might support API consumers in shaping a correct "mental model" [33]. There is empirical evidence that a standard API documentation without examples is of limited usefulness when it comes to complicated usage scenarios. Therefore, unit tests developed with common usages in mind can serve as code examples. To our knowledge, there are no other major works in the field of test cases as examples. The activities in this area of research seem to be

4. Challenges and Knowledge Types

Articles	Concepts				
	Mapping Scenarios to API	Tests as Examples	Integrate Concepts & Code	Conceptual Knowledge	Complexity
Hoffman & Strooper (2000)		X			
Hoffman & Strooper (2003)		X			
Ko et al. (2004)	X				
Ko et al. (2007)				X	
Robillard & Deline (2010)	X		X	X	X
Myers et al. (2010)			X		
Nasehi & Maurer (2010)		X	X		X
Ko & Riche (2011)			X	X	
Kuhn & Deline (2012)	X				
Nasehi et al. (2012)			X		X
Watson et al. (2013)				X	
Glassman et al. (2018)				X	
Meng et al. (2018)	X		X	X	X
Meng et al. (2019)	X			X	

Figure 4.1.: The concept matrix indicates which publication contributed to a concept.

rather special to this type of code examples. We claim that the findings of this field are hard to generalize. Also, the potentials of this topic such as validation and automation are not the focus of our study.

Code Examples: Integrate Concepts & Code

There is a consensus among researchers that API documentation must include code examples and the description of API concepts. However, it is not clear whether to include them closely interwoven with each other or in a dedicated documentation section. Nahesi & Maurer (2010) argue that the evolution of an API causes subjects to confuse high-level information with specifics of an API [33].

On the other hand, Meng et al. (2018) discovered that sections containing only conceptual information received less attention by developers [29]. Furthermore, an interwoven approach to integrate concepts and code aligns with previous work on conceptual knowledge [22, 39].

We argue that the evidence for an interwoven approach of concepts and code is stronger and widely accepted. Therefore, we adopt these findings for the design of the artifacts in this thesis.

Conceptual Knowledge

Conceptual knowledge in API documentation refers to any type of information required to use the API correctly that is not associated with specific API elements. Robillard & DeLine (2010) identified a conceptual gap coming from a lack of access to information on how to use the API [39]. Important conceptual or high-level documentation include the API's execution model, non-trivial samples, usage examples, and best practices. The evidence from large-scale surveys and interviews show the importance of high-level documentation for API learnability.

In further studies, Ko found that conceptual knowledge enables developers to effectively use and understand an API. The API consumers with access to conceptual knowledge were more successful in learning the API and judging the relevance of the content they used [22].

Another form of conceptual knowledge are best practices for API usage. The need for such an information was discovered in multiple studies [39, 50, 9]. According to these studies, developers expressed a demand for sections of the documentation showing the "most effective way to combine elements of the API" [39].

The previous findings were supported by Meng et al. (2018) with emphasis on the importance of conceptual knowledge on the initial learning phase. It is recommended to include conceptual information redundantly in the documentation and to integrate it into tutorials and code examples [29].

The high-level information plays an overarching role in API documentation and previous studies highly suggest to provide this in a redundant but coherent fashion. Understanding and reasoning about the API was also describes as being an important information

Frequent	Sometimes	Seldom	Never
13	1	0	0

Table 4.2.: Results of how often the presented problem occurs.

need for API developers [20].

Complexity

There has been thorough research on complexity of API examples. Robillard (2009) found that developers are actively seeking complex usage scenarios since trivial examples do not help them move beyond basic API usage [38]. This was confirmed by the field study of Robillard & DeLine (2010). However, more complex examples also seem to decrease the pedagogical power. In contrast, too simple scenarios are not perceived as useful by developers either [39].

Nasehi et al. (2012) confirmed these findings and added that complex scenarios might be added using hyperlinks [34]. In Meng et al. (2018) we find similar recommendations particularly stressing that complex examples must be presented as a organized series of small chunks [29].

From the presented evidence we conclude that complex usage scenarios are a crucial part of an example-driven API documentation. However, besides advanced scenarios we also need a set of simpler examples as an entry point.

4.3. Challenges

In the first part of our questionnaire, we asked our participants to assess the problem studied in this thesis. We explained the situation that API providers try to convey information to consumers using the documentation. This seemingly simple situation is complex because of the inherent information asymmetry. This section briefly presents the results of the first part of the questionnaire and points out the main challenges found in our sample.

How often do you encounter the presented problem in your role?

For the first question our participants had four options ranging from "Frequent" to "Never". The majority of our participants was very familiar with the problem which is obvious from Table 4.2.

How practically relevant is the presented problem?

By asking for practical relevance, we assessed whether participants come in touch with the problem with regard to their work. The options for this question were "Highly

Highly relevant	Moderately relevant	Irrelevant
11	3	0

Table 4.3.: Results of how relevant the presented problem was for our interviewees.

relevant", "Moderately relevant", and "Irrelevant". The Table 4.3 shows that the majority remarks that the presented problem is highly relevant for them. By drilling down into each group, we see that 100% of participants in the developer group rated the problem as highly relevant. Whereas, only 57% of the architects selected the highest relevance.

Problems & Causes

We asked our participants to talk about problems and causes for both sides, API consumers and providers. The challenges revealed in the interviews were categorized into either one of both sides or into "General" which indicates that the problem is rather overarching. We use squared brackets to show from which group of interviewees a particular statement came.

General

One of the general problems is that the communication and feedback between API providers and consumers happens over multiple channels [A,D]. Our participants mentioned that these channels are often not unified, resulting in a loss of information. An example is that some problems of an API are discussed on StackOverflow, a popular question and answer portal, rather than mailing the API providers. The knowledge on StackOverflow might be valuable for other API consumers but it might not be feasible for providers to bundle it into valuable feedback.

Further, the knowledge about an API is usually scattered across many locations or even hidden [D]. While the providers might know everything about the intended behavior and the internals, the consumers possibly have found bugs and corresponding workarounds. Apparently, it is challenging to exchange these two silos of knowledge. Accordingly, the consequences are an even more significant information asymmetry.

API providers

The first challenge limited to the API provider side is a lack of knowledge about the consumer needs [A]. Our participants stated that the use cases and goals of API consumers are often unknown. As a consequence, the providers do not know whether the API is constantly misused or not understood due to problems with the documentation.

Connected to the previous problem, we found that providers fail in providing solutions to common problems of consumers [A,D]. Since there is no exact knowledge about the most important use cases, the providers have trouble in offering the right information as part of an API documentation.

In addition, our participants generally agreed that documentation has a low priority for the providers [A,D]. There seem to exist several reasons for this problem such as lack of skills, time pressure, or simply the fact that writing documentation is less interesting for developers than coding.

Finally, the API provider is continuously confronted with feedback about their API. The feedback ranges from failing API requests to direct contact from API consumers. According to our interviews, the provider fails in including this feedback into the API lifecycle thus having difficulties to continuously improve [A]. This challenge must be interpreted together with the aforementioned issues of multiple feedback channels which make the integration into lifecycles even harder.

API consumers

An obstacle for API consumers is the underestimation of the API complexity [A]. Since the documentation is intended to give a simplified view on the API, some consumers might not realize the complexity until they have to work on a real problem. This results in misunderstandings and generally sets false expectations.

Another major problem is a lack of trust in the documentation and its helpfulness [D]. While developers often agree that the API documentation is in fact a useful learning resource, a "trial & error" approach is still popular. One reason is that API consumers might not think that the documentation solves their particular problem. Instead, it is often preferred to incrementally approach the problem instead of following a structured documentation. A low-quality documentation even further decreases the trust.

The third issue on API consumer side is the variety of developer strategies [D]. More precisely, this issue not totally limited to API consumers but we argue it originates from there. Previous research found differentiations in e.g. concepts-oriented or code-oriented developers which is problematic since the documentation should be useful for all developers. It is common that some developers immediately start working on examples while others thoroughly study the concepts. As a consequence, some API consumers might feel that the documentation does not address them properly.

In summary, we found that our studied problem is highly relevant and frequent in practice. Especially, developers confirmed the importance of the issues we address. It even seems like there is a higher awareness for the problems on the API consumers side. Further, our interviews serve as indication that the challenges in the field are not limited to one side or the other. We rather observed that a holistic study might be appropriate in the field of API documentation.

4.4. Knowledge in API documentation

In research question 2, we ask for knowledge types in API usage scenarios and examples. To answer the question, we analyze our interview data with particular focus on items

Table 4.4.: The occurrence of the knowledge types expressed as a fraction of participants and all codings.

Knowledge Type	% of Participants	% of all Codings
Quality Attributes and Internal Aspects (Q)	100	26,97
Functionality and Behavior (F)	100	26,67
Control-Flow (CF)	86	10,91
Concepts (C)	79	9,70
Directives (D)	79	8,79
References (R)	86	6,67
Structure (S)	79	5,76
Purpose & Rationale (P)	57	4,55

that indicate knowledge to be included in these resources. The results in this section are presented in descending order regarding the relative occurrence across all codings. An overview of the knowledge types is depicted in Table 4.4. For each knowledge type we describe our observations and implications in detail similar to [39]. Our implications are enumerated using one letter for the knowledge type, such as "F" for "Functionality and Behavior" and a number. We link to our interview data by referencing to the participant ID's as in this example:

"An executable example is required when entry barriers are high [A1; D1, D6, D7]."

4.4.1. Quality Attributes & Internal Attributes (Q)

Maalej & Robillard (2011) describe "Quality Attributes & Internal Attributes" as similar to "non-functional requirements" [25]. It also applies to information about the API's internals without being related to the behavior. In our case, we refer to characteristics that do not explicitly state a functionality of the API documentation but rather the quality in which we interact with it.

Copy & Paste

"At least as a starting point it is not uncommon at all to work using copy & paste [of examples]." [D1]

A common practice for API consumers is to copy an API example and use it for own use cases. We claim this is a quality attribute because simply offering a button for copying an example might not satisfy this requirement. In our interviews we have confirmed that this is generally a process applied by junior and senior developers [A1; D1, D2, D4, D5]. Often, an example is used as the foundation for own experiments which makes it desirable to copy and paste it. Moreover, developers agree that copying an example is better than starting from scratch [A1; D1, D2, D4]. As a main effect of

copyable examples, we found that the barriers to get started with an API are perceived as lower [D3, D5].

"I think that people don't really think about what they just copied. They often have a hard time thinking about how to adapt the copied code." [A1]

On the other hand, there are frequently expressed concerns with the copy & paste approach. First, there is a risk that consumers who only copy examples might not fully understand underlying concepts or even the snippet itself [A1; D2, D5]. This risk materializes when consumers try to adapt examples to their own needs which is a task that requires an understanding of the API.

Weighing drawbacks against benefits, we conclude that the ability to copy & paste examples and scenarios is a very important feature. Having the copy & paste feature in combination with descriptions addressing the drawbacks, might be useful as pointed out in an interview [A1].

Implication Q1: *API Scenarios and Examples should be copyable as a whole to lower the barriers to get started with an API.*

Implementation Details

"For each request that leads to an error, we should know what the root cause might be. [...] However, these are often internal information. Most of this information might also be confidential." [D4]

The internal implementation details of the API were a frequently mentioned point. Firstly, the participants brought up their opinion on whether textual descriptions should convey implementation details. There was no consensus among the interviewees. We found that some saw benefits in passing on information such as whether person data is saved in an API element or how an efficient algorithm was programmed [A1; D2]. On the other hand, we noticed concerns that valuable details are often secret and therefore not meant to be published [A7; D1, D2]. Also, most of the internals are regarded irrelevant for API consumers [D3, D5, D7].

Secondly, we revealed disparities regarding implementation details in error messages. While some participants see that error messages must contain the root cause in the implementation [A6; D4] others argue that this information is of no interest at all [D3, D7]. Even though there is a general need for the root cause in case of an error, none of the interviewees asked for technical details within the API.

Based on these findings we are not able to make clear implications. However, it is important to point out that there is an observable disagreement in both groups concerning the implementation details.

Readability

"Conveying good content in a comprehensible language without distorting facts ... that's quite hard." [A1]

Mostly, there seem to be issues with writing good and attractive content in a language that is easy to read but does not omit important facts [A1; D1, D6]. This is challenging because most of the documentation is written by engineers who do not necessarily have the skills for writing professional documentation. As a consequence, some consumers might lose trust in the API due to a bad first impression of the documentation [A1; D1]. Some interviewees explained how they worked together with technical writers to deliver high-quality text documentation [A1; D6]. This is regarded as highly effective but we argue it is rather specific to the company we conducted the study at. It is also not clear whether this is directly related to usage scenarios and examples. Therefore, we do not conclude direct implications from our findings in this category.

Consumability

For the sake of clarity, we have further divided consumability into the following three sub-categories which will be discussed separately:

- discoverability
- detailedness
- applicability

"Sometimes tutorials are hard to find. That really bothers me ... if I need to register to access important learning material. The product doesn't have to be for free but at least the documentation should be public." [D1]

The **discoverability** of scenarios and examples refers to how convenient it is to find and access these learning resources within the API documentation. A frequently mentioned concern was that the most important learning resources are not available without a registration or other steps [D1, D7]. Closed access to some resources is perceived as a major impediment especially among developers. Further, the available resources should be placed visibly to show that they are intended as an entry point [D2, D7].

Implication Q2: *API usage scenarios and examples should be as accessible as possible to allow for high discoverability.*

Additionally, we observed a dissent with regards to whether usage scenarios and examples should be accessible in one central place. While one developer claimed that there should be a central knowledge base to avoid redundancies [D1], others suggest

that distributing big pieces of information to several places with redundancies might be even more effective [A5; D4]. Lacking a clearly observable pattern, we do not make any implications for this point.

"Often, nobody has the time and inclination to see an API in every detail. [...] The value of an API is not only the functionality but also the easy access to its functionality." [D1]

The degree of **detailedness** in the scenarios and examples reflects the information density. The prevalent opinion on the degree of detail is "less is more" [A7; D4]. Especially, from a developers side it is argued that full detail is not often sought for [A5; D4]. Further, an increased detailedness of scenarios and examples comes at a cost of readability and consumability in general [A7; D1, D4]. In case the descriptions are perceived as too detailed, developers seems to start scanning the text to save time [D4]. Also, if the scenario offers a too detailed variety of options or alternatives, the ability to transfer the tutorial for own purposes might be impeded (we call this "applicability") [A5; D7].

Implication Q3: *API usage scenarios and examples should be described in limited detail with a "less is more" philosophy to minimize the impeding effect on applicability and consumability.*

"It's optimal if you have code in front of you. It definitely helps translating from the abstract descriptions into code if there is already a practical example." [D1]

The last aspect of consumability is **applicability**, the transition from explanations and examples to the own use case. To make examples "digestable", it might be useful to rather create a collection of smaller examples instead of some bigger, more complicated examples [A5; D5]. This does not only help developers understand examples quickly but also facilitates transfer into the own implementation. Smaller pieces of functionality are arguably easier to incorporate into an existing implementation than bigger examples [D1, D5].

Furthermore, we found that developers see tutorials as a way to transfer examples and arrange them in order to further develop their mental model [D7]. From our observations we imply that the applicability is increased by offering smaller pieces of functionality:

Implication Q4: *API examples should be offered in pieces of functionality supported by usage scenarios which arrange these to ensure high applicability for developers.*

Coverage

"It's important to consider how many consumers can be reached with [usage scenarios]. The 'sweet spot' is probably somewhere at 80%. So, if you can reach 80% of your consumers with the scenarios, that should be sufficient." [A7]

Another recurrent topic in our interviews was the coverage of usage scenarios and examples. First, we have seen indications that consumers assume usage scenarios to provide end-to-end coverage [A2, A3, A4]. Apparently, the coverage of clearly defined cases fosters understanding of how the elements of an API relate to each other.

Implication Q5: *API usage scenarios should cover their use cases from end-to-end to illustrate relations between all included endpoints.*

Second, the participants agreed that covering all cases possible with an API is neither required nor feasible. Instead, there is a sweet spot as a trade off between effort and consumers reached. Some participants mentioned 80% of the common use cases as a good compromise [A7; D2]. There should be sufficient usage scenarios to help the majority of the consumers with common problems. Documenting only unusual or corner cases might even confuse consumers since there is only a minor fraction of developers with these special use cases [A7; D1].

Implication Q6: *API usage scenarios should cover the most common use cases to a large extent (e.g. 80%) to serve the majority of API consumers.*

Complexity

"Usually you have tutorials starting from zero and they are very helpful to learn features if you increase the complexity step-by-step." [D6]

We follow the informal definition of complexity by [39] for our work: "[...] combination of length of the example and amount of interaction with the API". Some participants pointed out where the complexity in API scenarios and examples comes from. As soon as an example contains multiple requests in a sequence with relations between them, it is perceived as complex [A2; D1, D7]. The complexity resulting from such sequences is often underestimated by API consumers, leading to misunderstandings or wrong assumptions [A4; D1].

However, from our interviews we discovered a consensus regarding the degree of complexity within usage scenarios. It appears that a step-by-step increase in complexity within a tutorial is perceived as helpful in learning the features and concepts of an API [D2, D6, D7]. Further, we have seen that a consistent level of complexity is important to see a beneficial effect [D1, D7]. A tutorial should neither be so easy that "you are just scrolling through" [D7] nor too hard to understand.

Implication Q7: *API usage scenarios and examples should increase the complexity step-by-step to be helpful as a learning resource.*

Visualization

"[...] sequence diagrams would work for me. But that highly depends on the individual, I'm rather a visual type so I learn best with UML diagrams." [A6]

A moderately discussed topic was the visualization of API usage scenarios in the documentation. The most dominant finding is that the preferred form of visualization depends on personal preferences. There is no clear consensus on what might be the best way to illustrate usage scenarios. For instance, there are advocates and opposers of Unified Modeling Language (UML) diagrams for usage scenarios [A6; D6]. Also, there are interviewees who prefer no diagrams at all but rather textual descriptions only [D2, D4]. The only form of implication for this category is to note that this knowledge highly depends on personal preferences. We could not observe a dominant pattern in our limited data set hence we do not state an explicit implication.

4.4.2. Functionality & Behavior (F)

The knowledge about "Functionality & Behavior" describes what functionalities are part of the API documentation [25]. The implications from this category can be perceived as features of usage scenarios and examples relevant for the implementation.

Executable examples

"I really think [executing the examples] makes sense. Basically, it saves you so much effort. You can try the API without having to set up anything." [D5]

Executable examples of an API element are pieces of code that can be run in a real runtime environment without additional effort. There seem to be several reasons for the perceived usefulness of executability. The execution or "try-out" of an example is regarded as very "helpful for the process of learning" [A1; D2, D7]. Another valuable benefit of the immediate execution of examples is the decreased effort for setting up an environment [D5, D6, D7]. The interviewees also refer to "lower barriers" as a result of executability [D6, D7]. This enables them to easily deal with some complexity e.g. endpoints with many parameters [D6]. Furthermore, executability is regarded as highly valuable for developing a "feeling for what is happening in the API" [D6, D7].

"If I'm trying to learn an API and the entry barriers to try out something are extremely high, then I would need [an environment] where I can just easily execute an example." [A1]

Across all interviews, executable examples were commonly seen as useful. Selecting which elements need to be executable seems to be an important point, too. First, an executable example is often required when the barriers to execute API examples are very high [A1; D1, D6, D7]. This can be related to a complex local setup or to additional

required steps such as a registration. Moreover, the need for executability seems to depend on the kind of service. A quite simple service should be able to provide executability whereas a complex system with lots of dependencies might not be able to provide such a runtime [A1, A3]. These seemingly contradictory points can be explained considering the expectations of API consumers. For a simple service they expect that it is easy to provide a runtime environment for examples.

"It definitely helps understanding the API. But I have seen cases where this also might be confusing when e.g. the examples are slightly incorrect." [D3]

In the interviews, participants also pointed out some drawbacks. Participant D3 mentioned in his aforementioned quote that incorrect examples result in confusion. The correctness of API examples was a necessary requirement mentioned multiple times [A5; D1, D2, D4, D5].

Implication F1: *API Scenarios and Examples should be executable to foster the process of learning, decrease set up effort, and lower the initial entry barriers.*

Adaptable examples

"If you can play around and run the example ... change something here and there. That gives you at least a good feeling of what is happening." [D2]

With adaptable examples we refer to exemplary pieces of API functionality that offer convenient ways for variations. First and foremost, an adaptable example and scenario commonly serves as a foundation for own implementations as mentioned by most of the developers [D1, D2, D4, D5]. By changing an executable example according to the users needs, it seems easier to have a working foundation for further implementations. This implies that adaptable examples necessarily need to be executable, too. Besides that, we found that developers use adaptable examples to "just see what happens" [D2, D4, D5]. This indicates that small changes to an API example can serve as a way to test the mental model of an API which is important for the learning process. Lastly, the adaption of examples can be a convenient way to explore the alternative flows within the API [D2, D4].

Implication F2: *API Scenarios and Examples should be adaptable to serve as a foundation for own development and help developers explore alternative flows.*

Correct examples

"The worst thing that could happen is when examples don't work. This happens quite frequently. I suppose they were written for a certain version and never got updated after that." [D2]

Incorrect examples were frequently mentioned by developers as a common problem. Understandably, the wish for correct examples has been equally expressed in the interviews. For most developers, it was a major concern if the examples are outdated and don't work [A1; D1, D2, D4, D7]. Apparently, this seems to be among the most frustrating quality issues among all participants. According to our observations, the reasons are that documentation has a low priority and it is generally very hard to maintain an API together with its documentation in an automated manner. Further, the participants suggested versioning and change management as the most important remedies on the API provider side [A4; D6, D7]. An easier and more immediate countermeasure is supposedly to indicate the version for which an example or tutorial was created [D6, D7]. This would at least help mitigate the irritation faced by API consumers when they assume that an example works. This also leads us to an implication for our API usage scenarios and examples:

Implication F3: *API usage scenarios and examples should be version-controlled and labeled with their intended version to warn developers of potentially broken examples.*

Feedback

"As a user we should just be able to give feedback for particular parts. For example, if something is not clear or not working. This feedback can be used by providers to improve their tutorials." [D7]

The purpose of a feedback feature is to increase maturity and quality of the API documentation in general through continuous communication with API consumers [A3, A5, A7]. Some architects referred to their own projects, in which feedback loops led to a significant increase in maturity. Early feedback might not only benefit examples but also lower-level documentation such as technical specifications. Further, we found that feedback functionalities might help developers with incorrect examples which was mentioned as one of the biggest frustrations [D2, D4, D5]. The feedback might be given for the specific code example or for a bigger section of text and code [A7; D7].

One concern with a feedback functionality was that it might result in additional manual effort for providers [D7]. The feedback given by consumers must be processed and incorporated into the API lifecycle which could result in significant effort depending on current processes.

The increased effort is negligible regarding the high potentials for API scenarios and examples through continuous feedback loops.

Implication F4: *It should be possible to give instant and specific feedback for API Scenarios and Examples to allow for a continuous feedback loop between consumer and provider.*

Entry Points & Barriers

"It's way more helpful to first have a high-level entry into the API. The technical specification and details can be studied later [...]." [D1]

A major topic in our interviews were entry points and barriers. First, our participants made a clear distinction between high-level and low-level elements of the API documentation. The entry point to an API is rather expected to be a high-level documentation element such as a scenario, example, or description of some concept [A1; D1, D2, D4, D5, D6]. The deep-dive into low-level documentations like technical specifications or reference documentations are commonly regarded as unsuited for an entry point because of the higher complexity and barriers [D1, D3, D6, D7]. However, some developers suggest to have clear references to the low-level documentations since these are studied after the initial learning [A1; D1, D2, D3, D4]. From these findings we derive the following implication:

Implication F5: *API scenarios and examples should reference to lower-level documentations (e.g. specification, API reference) to serve as an entry point with the option to dive into details.*

Further, we found that some developers search for a "main scenario" which serves as an entry point. This could be a very typical usage scenario of the API in form of a tutorial starting from scratch and increasing the complexity up to a certain point [D1, D4, D6, D7].

"I would say the main scenario is the most useful. It doesn't have to be a lot of them. Just the basics with a "getting started" are enough." [D4].

We did not observe the need for multiple entry points but rather found that "less is more" when it comes to entry examples [D4, D6]. From this we draw another implication for API usage scenarios and examples:

Implication F6: *There should be one main API usage scenario to serve as entry points.*

Textual Descriptions

"I don't think the purpose of a tutorial is to be copied but rather to explain the software. That's why it should include a healthy mix of text descriptions, too." [A1]

In a majority of interviews we found that textual explanations and descriptions are a topic highly associated with API usage scenarios and examples. One of the reasons for textual descriptions is to mitigate the risk of using examples and scenarios without the correct context [A5, A7; D1, D4]. By explaining what examples and scenarios were made for and what concepts and flows they implement, misunderstandings can be avoided or

mitigated. Another finding is that the detail in descriptions depends on the complexity of examples or scenarios. For non-trivial parts of the API there should be sufficient explanations [D1, D4, D6].

"Not every attribute needs its own description. There should be something like a compromise between the actual example and its description." [D6]

Moreover, we observed that the balance between examples and its describing texts is crucial for the perceived usefulness. A complex usage scenario with only one sentence of text is seen as insufficient as a so-called "text desert" [D2, D3, D4, D6]. The interviews also indicate that there are various opinions on what to include in the textual descriptions. On a high-level we found that concepts, flows, and errors are some of the most demanded knowledge types [A5, A7; D1, D4, D6]. We investigate this topic thoroughly in Section 4.4.

From the findings in this chapter we imply the following:

Implication F7: *API usage scenarios and examples should contain balanced textual descriptions explaining concepts, flows, errors and more to mitigate the risk of misunderstandings.*

Tooling

"A really good API documentation also has a Swagger console. It should include lots of examples there or even end-to-end use cases. Or help with doing the same in ." [A2]

Throughout the interviews, we have discovered a set of tooling that is already familiar to most of the participants. The first class of tooling frequently mentioned was API consoles and sandboxes. Especially, the Swagger UI ¹ was among the most frequently noted tools in the area of API examples. It is often seen as a place with "all the information about the API" and executable examples [A4; D3, D5, D6]. Further, this tool can improve the communication between teams of developers [A4]. The limits of the Swagger UI become obvious when we try to model sequences of API examples [A2, A4]. An API sandbox to test requests against a test environment is generally perceived as effective but it may not always be necessary [A3, A4; D1].

Another class of tools comprises local development tools such as Postman ² which was frequently mentioned. With Postman it is possible to test out an API in a manual or automated kind. Postman collections are used to group individual requests together and reuse or modify them. On the other hand, Postman scripts are used as a runtime for sequences of API requests with a built-in script language to specify validations and conditions. In our interviews, both tools were mentioned as a way to work with

¹See <https://swagger.io/tools/swagger-ui/>

²See <https://www.getpostman.com/>

examples and usage scenarios [A2, A6; D1, D3, D6].

We take these two classes of tools due to their high occurrence as a direct implication for API usage scenarios and examples:

Implication F8: *API usage scenarios and examples should be supported by familiar tools for the execution of examples and local API development.*

4.4.3. Control-Flow (CF)

The "Control-Flow" knowledge type describes the flows of control through an API including events, callbacks, and ordering requests.

Happy & Unhappy Flows

A common categorization made independently by interviewees was to split paths into happy and unhappy flows. A happy flow can be seen as an usual flow through the API without any exceptions. Contrarily, unhappy flows refer to paths where unexpected behavior occurs.

"You always just implement the happy paths and see in the specification that an error might occur. More important are the unhappy flows. But if you never encounter them, you don't think about a fallback solution for them." [A1]

Our first observation was that the unhappy paths are rarely presented in an API documentation [A1; D1, D6]. However, there is a consensus that both, happy and unhappy flows, should be covered [A7]. Especially during the learning phase it might be helpful to see "the most common errors" and demonstrate how they "crash" [A1, A3; D6]. Further, participants mentioned that API examples mostly just cover happy paths in the simplest way which makes them "not production-ready" [A6].

Implication CF1: Besides happy paths, the API usage examples should also involve unhappy paths and include them as natural part of the learning process.

Alternatives & Corner Cases

In contrast to happy and unhappy flows, the alternative and corner cases describe different branches of a flow. An alternative case might be a completely distinct path which achieves the same. A corner case can be explained as a quite unusual path to achieve something.

"First, the provider should illustrate the 'normal' flow. If needed, there can be references to alternative flows. 'Junctions' so to say. Without further complicating the normal flows you can offer consumers an alternative." [A1]

Depending on what developers are implementing, they are interested in the alternative flows through the API. Generally, the common flows have a higher priority but the alternatives are interesting in complex APIs, too [A2; D1, D6]. Seemingly, some developers naturally search for alternatives when they walk through a tutorial by "stopping and seeing what else could happen" [D2]. For this, we observed that developers like to see the "junctions" in a scenario clearly. The only drawback mentioned by one of our participants was that many alternatives might confuse consumers. The demonstration of the "best flow" could be beneficial in some cases [D4].

The corner cases were mentioned less frequently than alternative cases. However, interviewees agree that the coverage of corner cases is not good enough in most of the API documentation [A2; D6].

Implication CF2: If appropriate, the API usage scenarios and examples should show alternative implementations of the same use case. However, by default the best possible implementation should be demonstrated.

Order

The last control-flow category concerns orderings within the API. The ordering might be easy to find in case of simple API's. However, with multiple API elements and more complex interrelations, the composition is less obvious.

"I might know the REST interface very well. But if I don't know how to nest them or in which order to compose them, I will probably have a hard time solving my problems." [D4]

We found that the order becomes especially important if there is a variety of options to choose from. If developers only see the technical specifications, the implicit orderings are often not evident for them. Tutorials are perceived as useful to uncover these implicit characteristics [A7; D1, D6]. Moreover, we observed that this contradicts with the demand for alternative and corner cases. By presenting more possibilities, we also create a stronger demand for ordering.

Implication CF3: The API usage scenarios and examples should explain the order of interaction in the accompanying text if it is not obvious.

4.4.4. Concepts (C)

The knowledge type "Concepts" describes design or domain concepts used by the API or explain the meaning of terms and information used in the context of the API. Based on our findings, we subdivided this category to give a more detailed insight into the interview statements.

Design & Domain

Some interviewees explained the importance of domain understanding when working with tutorials and examples.

"If you are not familiar enough with everything ... or don't have the domain knowledge, then it is very hard to solve problems by yourself." [D5]

First, it is essential for developers and architects to understand how elements of an API relate to each other e.g. as part of a tutorial [A4, A5]. We observed that explaining overarching concepts of the domain is highly valued [A5]. Ideally, the domain concepts are explained very early in the process of learning an API [D4, D6]. Apparently, a thorough domain understanding helps API consumers not only in the learning phase but also later during development [D1, D6]. Further, we have found that participants agreed on the difficulty of problem solving without proper domain knowledge [D4, D5, D6]. API usage scenarios were seen as one way to get insights into the domain and the API itself [D6].

Implication C1: The concepts of the domain in which the API is applied should be part of the usage scenarios. They might be included in the textual descriptions accompanying the examples.

Context

One frequently mentioned concern was the misuse of the API due to misconceptions. For some participants, this problem comes from the missing usage context while working with tutorials and examples [A4, A5].

"The risk is that users use the examples without understanding the context. That's why you need descriptions for each [example]." [A5]

Adding context information to examples does help with using the example itself but also facilitates the developer's ability to solve problems [A4; D4]. Further, the context is necessary as part of an example to build a story to guide consumers [D2, D5].

Implication C2: The context of each example and scenario should be explained as a textual description.

Namings

A common source of misunderstandings in an API documentation are formulations and namings of concepts. This issue was reflected by our interview participants, too [A2, D3]. Either the namings are ambiguous and inaccurate or the different stakeholders have inherently contrasting interpretations of the same terms. Both cases might lead to misunderstandings regarding concepts. A commonly mentioned way to mitigate

misunderstandings are glossaries [A2].

Implication C3: An API glossary can help clarifying namings and terms contained in the API documentation.

4.4.5. Directives (D)

A directive is knowledge provided by the API provider to indicate to consumers what they should do or not do with API elements [25].

Best Practices

"They (API providers) should document their best practices. You just don't know what the ideal way is. If somebody has found it, I would think it is cool if you communicate that to your consumers." [D2]

Some developers asked for best practices as one kind of directive. This does not only include a recommendation how to accomplish something but rather gives a sophisticated and optimal solution [D1, D2]. Even though it is regarded as an invaluable resource, it is supposedly rare in API documentations [D1].

Implication D1: The examples and scenarios should demonstrate the best practices intended by the providers.

Important Use Cases

"There need to be examples showing use cases the API was built for. You would kind of have that with usage scenarios." [A5]

Regarding directives in the API documentation, we have found the demand for use cases as a way to tell users what to do with the API. The most important use cases might serve as an entry point for the API consumers thus demonstrating intended use of the software [A2, A5; D1]. This could be presented using a standard or "main" scenario for the API and guiding the consumers through it in a "getting started" fashion [D1, D4]. Besides the intended use cases, the documentation could provide descriptions of use cases the API was not designed for [A5].

We discovered mainly two reasons why participants proposed this idea. First, the concrete use cases as directives should effectively prevent misuse, a common problem to most of the interviewees [A4, A5; D2, D4]. Second, the mere technical details of the API might not convey the intended use to consumers. The description of use case in combination with technical examples could according to our participants communicate a clear directive [A2; D2].

Finally, we observed a central problem to this approach we already mentioned in Section 4.3. The API provider might not necessarily know the consumer's use cases. Thus,

describing some important use cases as part of the documentation might not meet the set of use cases consumers have in mind [A7; D1]. The API provider must make sure to understand the common usage of their API to deliver clear directives.

Implication D2: The API usage scenarios should describe the most important use cases and demonstrate some unintended usages. Moreover, there should be one main scenario which can serve as an entry point.

4.4.6. References (R)

The "References" type refers to knowledge that links to other (external) documents, such as API specification, in the form of hyperlinks or tags. With an occurrence of 86 % in the interviews, it was a frequently discussed topic.

"And especially there should be a cross-reference to the documentation on a lower-level where the details are explained." [A1]

As already discussed in Section 4.4.2 on "Entry Points & Barriers", we observed a pattern regarding references between high-level documentation (usage scenarios, concepts, ...) and low-level documentation (specification, API reference, ...). A further investigation revealed that this might be implemented in various ways. First, participants proposed to reference low-level documentation from within tutorials [A3; D2, D4]. Allegedly, this makes the high-level documentation more useful by enabling developers to make quick lookups. Second, the consoles containing executable examples could reference to the specification directly [A1, A4; D4]. Both ways satisfy the demand of API consumers to dive into the more technical parts of the documentation where necessary.

Implication R1: The connection between high-level and low-level documentation should be established through links in tutorials (for scenarios) or consoles (for examples).

"I don't know how we as a team would publish these [usage scenarios]. If there is a standard in the industry we would of course adapt to it." [A4]

Another recurring topic was concerning standards in the API documentation. As quoted above, the emerge of a standard for how to document usage scenarios is a topic of particular interest. We found that the use of standards is especially important for architects [A4, A5, A6, A7]. Some of the frequently mentioned standards were OpenAPI³ (formerly Swagger; de-facto standard for RESTful API specification) and AsyncAPI⁴ (standard for web API specifications in message and event-driven systems). In our interpretation, there is a demand for standardization in the space of API documentation. While there are de-facto standards for API specifications, there is nothing comparable

³See <https://swagger.io/specification/>

⁴See <https://www.asyncapi.com/docs/specifications/latest>

for the specification and presentation of usage scenarios or examples.

Implication R2: The adherence to standards in the API space should be stressed and referenced where appropriate.

4.4.7. Structure (S)

The meaning of structure in the context of API documentation is how the API is internally organized and how the elements are related to each other. In our interpretation, this also applies to the structure of the usage scenarios and examples in an API.

Relations between Entities

A significant challenge faced by stakeholders mainly from the architects group is that an API contains implicit relations between entities.

"Scenarios and orchestration are a major problem. [...] An example from the commerce domain is that for an order the product must be available. Descriptions of such situations are often missing." [A4]

An example mentioned by A4 (see quote) was referring to the e-commerce domain, where complex relations between entities such as carts, orders, and products are common. The API documentations apparently fail in conveying this knowledge properly [A1, A4, A7]. This problem is known to some of our participants but at the same time there does not seem to be an obvious solution to it [A4]. For this reason, we do not formulate an implication.

Common Thread

Another structural knowledge type of the API documentation is a common thread that runs through usage scenarios. It is intended to guide the developer in working on the tutorial.

"[Tutorials] irritate me a lot if they are a total mess. So, often they just miss a common thread. Of course you always have some references that help you find your way but I'm often missing the story." [D2]

Without a coherent thread and focus in the tutorials, we found that participants are irritated. This irritation can materialize in developers feeling "lost in the scenario" [D4] or missed expectations [D2, D6]. This characteristic of tutorials was often described as the "story" and refers to a scenario with defined context and outcome [D2, D5]. We assume that these mentioned properties might be able to help establish a story in a tutorial: context, outcomes, and focus.

Implication S1: API usage scenarios should have a focused story that defines the context and outcomes.

4.4.8. Purpose & Rationale (P)

The knowledge about the purpose of certain API elements or rationale of design decisions is another type of semantic content. This knowledge is often used to answer the important question "Why would we want to use this?".

"Basically, in the beginning you want to know what the API does and whether you want to use. This way you can decide if the tool fits your problem." [A1]

Our findings regarding the purpose and rationale mainly focus on a pattern frequently described by our interviewees, the problem-solution fit. As a crucial step, stakeholders decide whether they want to use the API or not. To facilitate this decision, the consumers require basic information on what problem the API solves [A5; D1]. This should be presented in a way that is completely separated from "the usual marketing texts" [D1] because these conceal the useful information. A long list of functionalities is cumbersome to use, too. The majority of participants regard a comprehensive list of capabilities as valuable to decide whether to use an API or not [A1; D6]. If the API fails to convey information about its capabilities, some may assume that it simply does not solve their specific problem[D1].

Implication P1: The documentation should include a concise list of capabilities describing the solution.

4.5. Discussion

In this section, we show how our findings from the interviews can be interpreted in consideration of the existing literature. We have analyzed previous research in Section 4.2 and found that there are certain topics common to most of the work in this field. Similar patterns were found in our interview analysis. We will separately discuss how our findings support or oppose previous work. Table 4.5 shows an overview of all stated implications in an abbreviated form.

Quality Attributes and Internal Aspects

One of the formulated implications was that consumers expect examples to be "copy-friendly". While the copy-paste approach of developers is diversely discussed in literature, recent studies recommend a high copy-friendliness since it is a common approach for developers [28]. We support this position with **Q1**.

Moreover, we found in **Q2** that accessibility of the examples in an API documentation is perceived as important, too. This is also supported by [28]. They propose the following

4. Challenges and Knowledge Types

Table 4.5.: All knowledge types with the corresponding implications stated in the previous section.

ID		Implication
	1	Usage scenarios and examples should be version-controlled and labeled with their intended version to warn developers of potentially broken examples.
	2	Usage scenarios and examples should be publicly accessible to be more discoverable.
	3	The descriptions should follow a "less is more" philosophy to minimize the impeding effect on applicability and consumability.
G	4	Examples should be offered in pieces of functionality ready to be copied and reused.
	5	Usage scenarios should cover their use cases from end-to-end to illustrate relations between all included endpoints.
	6	Usage scenarios should cover the most common use cases to a large extent (e.g. 80%) to serve the majority of API consumers.
	7	Usage scenarios should increase the complexity step-by-step to be more helpful as a learning resource.
	1	Executable examples foster the process of learning, decrease set up effort, and lower the initial entry barriers.
	2	Adaptable examples serve as a foundation for own development and help developers explore alternative flows.
	3	Copyable examples help to lower the barriers to get started with an API.
F	4	Instant and specific feedback for API scenarios and examples allows for a continuous feedback loop between consumer and provider.
	5	References to lower-level documentations (e.g. specification, reference) can serve as an entry point with the option to dive into details.
	6	A main API usage scenario could serve as the main entry point.
	7	Balanced textual descriptions should explain concepts, flows, errors as well as conceptual information to mitigate the risk of misunderstandings.
	8	Support examples and usage scenarios by familiar tools for the execution of examples and local API development.
	1	Besides happy paths , the API usage examples should also involve unhappy paths and include them as natural part of the learning process.
	2	Usage scenarios and examples should show alternative implementations of the same use case.
	3	Usage scenarios should explain the order of interaction in the accompanying text if it is not obvious.
C	1	Concepts of the domain in which the API is applied should be part of the usage scenarios included in the textual descriptions accompanying the examples.
	2	Contexts of each example and scenario should be explained as a textual description.
	3	An API glossary can help clarifying namings and terms contained in the API documentation.
D	1	Best practices should be presented and explained in the examples.
	2	The most important use cases should be described and the unintended usages should be demonstrated.
R	1	Connections between high-level and low-level documentation should be established through links in tutorials (for scenarios) or consoles (for examples).
	2	Adherence to standards in the API space should be stressed and referenced where appropriate.
S	1	Usage scenarios should have a focused story that defines the context and outcomes .
P	1	The documentation should include a concise list of capabilities .

measure for accessibility: organization of content, integrate conceptual information, and a powerful navigation. We complement that usage scenarios and examples should be available without requiring users to register or pay.

A further implication is that some of our participants propose a "less is more" strategy to textual descriptions (**Q3**). The effect of the information overload in software documentation was mentioned as an "information barrier" in [21]. In direct relation to API documentation, [38] found that intent documentation should be included on an as-needed basis instead of covering all possible cases. We can support Robillard's findings in the sense that we also found descriptions to be perceived as more useful if it is short and concise. In contrast to our observation, Meng et al. (2019) found that specific content should be presented redundantly across the different parts of the documentation. Our finding stands in contrast to this since we consistently observed a "less is more" principle. In our judgment this requires a thorough trade off decision to meet both requirements to a satisfactory degree.

In **Q4** we argue that examples should be offered in cohesive pieces since developers tend to copy only parts of examples into their own implementation. In previous studies, [34] found that the presentation of a solution in a step-by-step manner is preferred by most developers. Further, in [28] it was observed how opportunistic developers, i.e. developers who start coding immediately, want to have selective access to code. In our study, we provide further evidence for the necessity of dividing examples into "chunks". With **Q5** we found that API usage scenarios should provide an end-to-end view of the use case. This supports similar findings in [38] and [29]. These two studies found that completeness of examples is highly value.

Moreover, in **Q6** we observed that participants require the coverage of the most important cases. This finding occurs frequently in previous research, e.g. in [34]. We support this in our findings and add that participants mentioned 80 % coverage as a rough estimate.

Regarding the complexity of usage scenarios and examples (**Q7**), we find different opinions. On one hand, we have reviewed studies that recommend simplicity over complexity. Even though there is a danger of "too simple examples" [34], previous research states that complexity must be kept at a low level to ease the learning process for developers. Robillard & DeLine observed that increasing complexity might decrease the pedagogical power of examples [39]. These mentioned points were outlined in our literature review in Section 4.2. However, we found that 57% of our participants regard increasing complexity in tutorials as necessary. This insight was not stated in the previous research and rather contradicts with some findings. However, we note that an intentional increase of complexity limited to usage scenarios and examples might be beneficial. Still, we agree that the overall complexity of the documentation must be kept low.

Functionality and Behavior

Our implication **F1** is referring to the executability of examples in the API documentation. This topic was covered in [13] with regards to executable test cases which served as API examples. In a different context [42] report that the automatic generation of examples which consist of executable code was rated as beneficial by most of their study participants. [42] also propose a sandbox as execution runtime which we also found across various interviews. We support these findings and add that this feature seems to be more valued when the set up of a runtime requires much effort if done by the developers.

The benefits of adaptable examples (**F2**) were most recently confirmed in [28]. They found that developers seek to create a mental model of the API and correct it by "playing around with the API". We have seen very similar statements in our interviews. Participants even literally mentioned the term "mental model".

Another frequently observed problem in the API documentation area is that examples are outdated and thus do not work. This was reflected in our implication **F3** and serves as further empirical evidence for the findings in [24] and confirmations of this problem e.g. by [46] or [29]. However, in contrast to previous studies we propose labeling and versioning as explicit suggestions. Our observations indicate that the disappointing effect of defect API examples might be decreased by labels. We interpret this feature as a way to show developers whether they can expect the examples to work.

The need for feedback in API documentation (**F4**) was already pointed out by [19]. They found that the incorporation of user feedback might improve examples. However, there were no indications on how to collect feedback. In this thesis, we found that participants explicitly asked for "instant feedback". In our understanding this refers to the ability to rate specific parts of the documentation, such as an example, description or the whole tutorial, with a simple interaction. Additionally, we note that this feature only occurred in 29% of our interviews, which is very rare compared to other statements.

The most recent work on explicit connections to low-level documentation comes from [28]. They found that connecting "concepts to code" is crucial for developers to explain the concepts. More details about this pattern can be found in Section 4.2. We support this observation with implication **F5**. An explicit and explained connection to the low-level documents from within usage scenarios might be helpful for developers. Further, [28] explicitly call for studies evaluating phenomena like these which emphasizes the importance.

In a field study on learning obstacles, [39] found that developers look for point-of-entry overviews. Furthermore, in [29] the lack of a coherent entry point was found to be causing difficulties for developers. We can support these results with our own observation. In **F6** we state that developers especially look for a "main scenario" while starting to use examples and tutorials.

The need for descriptions accompanying examples (**F7**) is recognized in most of the research. [39] found that these textual descriptions should contain information about the error codes and parameters. We support this and complement that knowledge about

concepts and flows through the API should be explained, too.

In implication **F8** we state that developers seek support for their familiar tools in the API documentation. Similar statements were observed by [29]. They found that developers expect an integration of the API in their development environment to ease the process of set up. We discovered that this integration might work better if the developers are supported in using their familiar tools such as Postman or SwaggerUI. Additionally, we propose that tool support might especially benefit the executability of examples. [39] argue that developers rather use tools for example discovery instead of execution. We claim that our observation is not contradicting but complementary to the findings in that study.

Control-Flow

Our findings for the knowledge type "Control-Flow" were categorized in three parts: happy & unhappy flows, alternative & corner cases, and order. Firstly, we observed that it might be beneficial to demonstrate both successful and error cases through examples in the API documentation (**CF1**). The existing body of knowledge does not stress the need for unhappy flows through the API. A study by [42] shows that according to developers, the unhappy paths are seldom part of an API documentation. Our study explicitly presents this need based on multiple developer's and architect's statements. Next, we found that developers are interested in alternative implementations besides those explained in examples (**CF2**). This results is strongly supported by literature. In [38], it was revealed that particularly the high-level documentation (i.e. tutorials, conceptual explanations) is perceived useful since it conveys information about alternative ways of solving the same problem. Previous studies do not clearly point out what part of the usage scenarios and examples is perceived as interesting. We found that the junction points, i.e. the part of an example where an alternative implementation could begin, are important.

Lastly, our interview analysis indicate a significance of the interaction order (**CF3**), in particular the explanation of it. This topic is rarely discussed in the literature. In some studies, the order of interaction is discussed in the context of API libraries and the parameters that are passed to elements of the API [32]. We add that the importance of interaction order might depend on the complexity of the domain. Further, the order is often perceived as not obvious to API consumers and therefore needs explanation. Potentially, a more complex domain might require detailed explanations of the interaction order where necessary.

Concepts

We found that regarding the concepts of an API especially the domain understanding, context of use, and terminology are important to stakeholders. The important role of domain understanding (**C1**) was observed in multiple studies [22, 39, 28]. A frequently researched aspect is that developers form a mental model of the API. We can confirm

this observation with our own findings. More information on conceptual knowledge is given in Section 4.2. Furthermore, [34] found that the context of examples could decrease the "cognitive distance" to the API consumers. The relationship between examples and their context was also observed in [39] and we provide further evidence for the potential beneficial effect of a defined context of use (**C2**). Finally, the need for a glossary in API documentation was frequently mentioned in the literature especially with regards to API libraries or SDKs [35]. We confirm the need for a glossary to prevent misunderstandings resulting from ambiguities in terminologies specific to the API domain (**C3**).

Directives

We sub divided directives into best practices and intended or unintended use cases. With regards to API best practices (**D1**), we found that the developer's interest in best practices was also observed in [38] and [39]. Here, the best practices were defined as "best way to combine the API" [39]. Our results are in line with the literature and we add that best practices should not only be shown but also explained. A common topic in research is how API providers convey the intended use of their API to the consumers. This is referred to as "intent documentation" [39]. Our findings cover not only the intended way of using the API but also the unintended way or misuses of the API (**D2**). Previous studies do not propose to also present the known misuse scenarios to consumers. Even if we only observed this in two interviews, it might be effective in preventing very common misuses. Obviously, this requires the API providers to have good knowledge of the way their API is used. As explained in Section 4.3, this is rarely the case.

References

The "Reference" knowledge refers to connections within an API documentation or to external documents. We covered this approach in our literature analysis. The main categories in our findings are the connections between high-level and low-level resources. Previous studies have shown that a reference from low-level documentation to concepts and design decisions might help developers understanding API behavior [38, 39]. Our interviews indicate that some developers also need links from high-level to low-level documentation (**R1**). This feature could serve as a way to look up details while working through a usage scenario.

The issues around standardization are discussed frequently, mostly remarking the lack of standards [22, 41]. In [22], it is pointed out that examples in API documentation should follow a standard. This problem is reflected in most of our interviews (**R2**). We also observed that examples should indicate whether they follow any convention or standard to establish visible connections.

Structure

The structural knowledge types in our results mainly focus on the relations within the domain of an API and the story that connects elements of an API usage scenarios. First, the knowledge of inherent relations in the domain were mentioned in the literature, e.g. in [29]. Researchers found a lack of background knowledge was mentioned as a major challenge in learning and integrating the API into an implementation. The existing studies, however, do not explain in detail what part of the domain causes the learning barriers. Our interview participants mentioned the implicit relations between entities in the domain as a cause for the lack of domain understanding. An example we have seen was that in the e-commerce domain, the API consumers should know that shopping carts, products, and orders have implicit relations (stated by participant A4, see Table 4.1).

Further, the need for a story in examples was once referenced by [39] as "story-telling" (**S1**). The explanation of an usage scenario as a story was perceived as helpful. We confirm this finding frequently in our interviews. Moreover, we observed that the story should define a context of use and explicitly state the outcomes for the scenario. This helps API consumers in adjusting their expectations for an API usage scenario.

Purpose & Rationale

Previous studies discovered that developers try to find the capabilities of an API to check whether it fits their needs. Especially, the studies [35] and [28] point out that it is a part of the learning process. Additionally, we found that this need could be satisfied with a concise list containing the API functionalities (**P1**). Some of our interviewees pointed out that this resource should be free of any other content, such as marketing-related information. This further supports previous findings and adds hints for the implementation of this feature.

5. Implementation

This chapter presents the artifacts we designed based on our findings from the existing literature and our insights from interviews. This can be regarded as the "Design" step in the Design Science approach (see Figure 1.1). We created a *basic* and *advanced* API documentation. While the basic API documentation contains only necessary and common features, the advanced API documentation is supposed to serve as an optimization using the implications identified in Chapter 4. We evaluate these artifacts in our case study following in Chapter 6. The goal of these artifacts is to research the effect of differences between the two versions of the documentation. Even though we develop two versions, our goal is to make both documentation useful for developers. First, we present how the implemented features and the API were selected. Next, we describe the design approach, i.e. how we worked together with the API provider to create the API documentation and provide an overview of the used tools. Finally, we demonstrate how the features were realized in the documentation.

5.1. Selecting the Features & API

To conduct our implementation we selected a subset of the discovered features and a suitable API project. The features were chosen based on insights from interviews and our comparison with existing research. We especially studied the aspects which have less empirical evidence in the literature or seem to contradict with previous findings. The project selection was based on criteria that support our implementation and provide high representativeness for our study.

5.1.1. Selected Features

From our findings in Chapter 4, we selected implications that should be part of our implementation. This sub set was divided into two artifacts. In the *basic documentation* we included all features that are necessary to understand and get started with an API. The *advanced documentation* additionally contains features that according to our interviews would potentially benefit devX. With special focus on the examples and usage scenarios in the documentation, we designed significant differences into these two artifacts to research the effect of added features.

Table 5.1.: The criteria to find basic and advanced features for the prototypes.

Feature Selection Criteria	
Preselection	Selection "Advanced"
<ul style="list-style-type: none">- Potentially useful- Feasible within constraints	<ul style="list-style-type: none">- Low occurrence in previous research- Contradictions with previous research- Controversial among interviewees

Criteria

We selected the features in two steps. First, we pre-selected features which we would include in both artifacts. Second, we selected features exclusively for the advanced artifact. The criteria for these steps are listed in Table 5.1.

To be included in our implementation, features must be useful for API consumers and also feasible within the thesis constraints. Given the limited time, we excluded some features from the implementation. Features were also excluded if they rarely occurred in the literature and our interviews. Further, we decided whether a feature belongs into the advanced implementation. The advanced documentation was supposed to contain features that might improve the developer experience for API consumers. Also, we wanted to evaluate our controversial findings to discern whether they would be an improvement.

Selection

Based on the outlined criteria we selected a set of features based on our implications we wanted to integrate. Moreover, we split our selection into the basic and advanced artifact. This is depicted in Table 5.2.

The realization of these elements are the focus of this chapter. We note that the implementation was not always satisfiable to the full extent. The exact implementation is presented in Section 5.3.

5.1.2. Project "Compass"

In our design phase, we want to create a documentation for an existing API. This implies that we need a project where the API forms a central element but is not sufficiently documented yet. Before searching for a project in particular, we defined a set of criteria which will be presented here. Further, we will give some insights into the selected project named "Compass" itself.

Criteria

Our criteria concerning the project selection are categorized into organizational and technical constraints as shown in Table 5.3. Since we planned to design our artifacts in collaboration with API providers, we had some organizational constraints. Further,

Table 5.2.: The implications that we integrate in the basic and advanced artifacts.

Implication	Basic	Advanced
Q1 – Copy-friendly	X	X
Q2 – Accessibility	X	X
Q3 – „Less is more“		X
Q6 – Main use case coverage		X
Q7 – Increasing complexity		X
F1 – Executable	X	X
F2 – Adaptable	X	X
F5 – References to low-level		X
F6 – „Main scenario“		X
F7 – Text descriptions	X	X
F8 – Familiar tools		X
CF2 – Alternative implementations		X
C3 – API glossary	X	X
D1 – Best practices		X
R1 – High-level to low-level connections		X
S1 – Focused story in tutorial		X
P1 – Capabilities list	X	X

Table 5.3.: The organizational and technical criteria applied to select an API for our implementation.

Project Selection Criteria	
Organizational	Technical
- Team availability	- Sufficient complexity
- Collaboration	- Multiple scenarios
- Knowledge availability	- Familiarity

the artifacts were intended to be used in a case study with developers, which led us to define technical constraints as well.

First, an easily available team is crucial for iterative collaboration in fast cycles. This collaboration was crucial to get valuable feedback on our prototypes and discuss what features are important from an API provider point of view. Further, we required material that would support us in designing our artifacts, ideally provided by the team building the product.

From a technical perspective, we were interested in a sufficiently complex product. We argue that trivial software might intuitively be understood by developers and require less learning which is less suitable for our purposes. Also, there should be enough possibilities to use the API which would allow for multiple usage scenarios. Lastly, the concepts of the API should be rather familiar to most developers but still require learning to productively work with it.

Further, we searched for a relatively young project to ensure there is not a mature API documentation already in place. We finally selected an open-source project called "Compass" ¹. We will shortly explain what Compass is and what it does. However, it is not necessary to grasp Compass in full detail to understand our design artifacts.

What is Compass?

On a high level, Compass is a software tool to manage computing clusters and external applications. Through Compass, the applications can be connected to these clusters and consume services securely. These clusters are also called runtimes. It serves as a management plane for these runtimes and applications and is used as a central place to control and connect an entire application landscape. The different components of Compass expose APIs which can be used by an administrator to interact with the landscape (see Figure 5.1). One of the main use cases of Compass is to pair the external applications to the runtimes to enable a secure business data flow. Another important use case of Compass is to propagate changes from the applications to the runtimes and ensure continuous synchronization. Since most application landscapes become very heterogeneous over time, dealing with changes in the system is a common challenge for most companies.

¹See <https://github.com/kyma-incubator/compass>

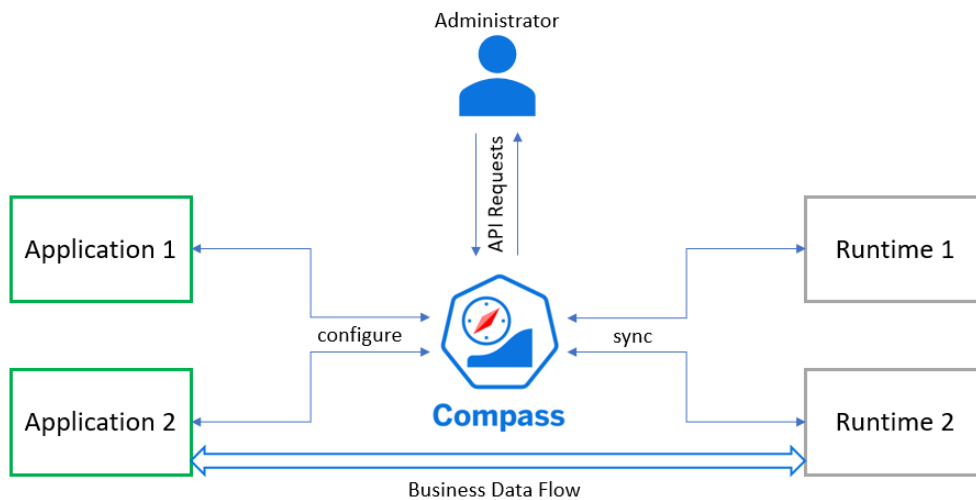


Figure 5.1.: An overview of how Compass is used to manage applications and runtimes.

Why did we choose Compass?

The decision to choose Compass as a project was made considering how it fulfills the defined criteria. Firstly, Compass is developed and maintained by a project team at the large software enterprise. This circumstance made it easy to fulfill all organizational criteria. The team was easy to collaborate with via online discussions and ad-hoc meetings. Further, it was possible to get deeper insights into the idea behind the software and what the documentation should express by directly talking to the API provider team. During our design phase, we also had access to additional non-public material which helped us understanding what concepts to convey in the documentation. Secondly, Compass also satisfies all technical criteria. The complexity of Compass is rather low at first sight because it mainly manages two entities, applications, and runtimes. However, interactions with Compass APIs can reach a high level of detail. It is possible to include webhooks, events, health checks, and a labeling system, which increases the maximum possible complexity significantly. Also, there are many ways to interact with Compass which makes it easy to model different usage scenarios and flows. Often, there is an alternative implementation for a certain flow. As the project is still quite young, there was no documentation at the time of writing this thesis. The mentioned characteristics of Compass make it suitable for our study and allow us to create realistic artifacts.

5.2. Process & Technologies

The basic and advanced API documentation were developed in collaboration with the API providers. We had access to background and technical documentation. In a collaborative approach, created the API documentation for Compass from scratch. The

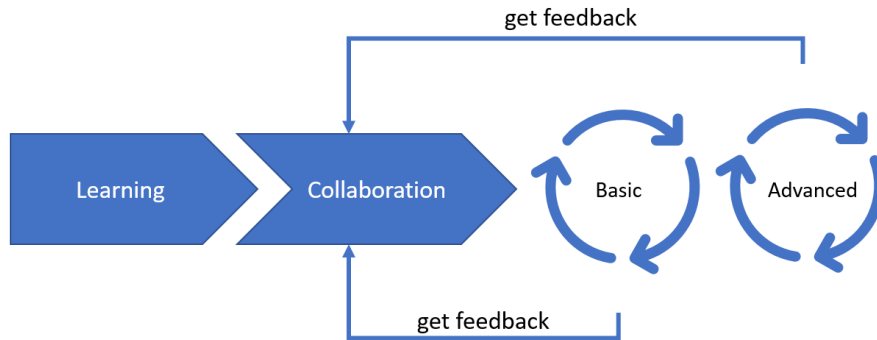


Figure 5.2.: The process of defining our artifacts included a learning phase and an iterative collaboration and design phase.

process of creating the artifacts and supporting tools and technologies are outlined in this section.

5.2.1. Designing the API documentation

We designed our artifacts in collaboration with the API providers using an iterative and incremental approach. This included a learning phase, multiple collaboration sessions, and iterations of the artifacts. Figure 5.2 illustrates an overview of the process.

Learning

Before documenting the software, we aimed to thoroughly understand all aspects of Compass. This included conceptual and technical topics. The conceptual topics consist of the components, flows, and capabilities of Compass. The API provider had material explaining most of the conceptual questions which we received for our implementation. Further, we needed to understand the technical topics of Compass. The Compass API uses GraphQL, which we were not familiar with. Also, we had to understand how our actions with the API trigger side effects and flows. This phase of the implementation included mainly self-study of the material about Compass and the technological prerequisites.

Collaboration

The collaboration was realized in the form of remote sessions with a software developer or sometimes with the product owner of the API provider team. In these sessions, we discussed what the most important information about the API is and what developers will need to know. Especially, we asked for advice concerning the examples and usage scenarios we should cover. The developers of the API provider team helped us

by explaining the common intended use of the software which we transformed into examples and scenarios. To prevent a one-sided view on the usage, we also worked with Compass on our own and proposed our usage scenarios which were assessed by the API provider. We developed increments of our artifacts and reached out to the team in an on-demand manner to get feedback. This form of collaboration turned out to be very convenient given that the provider team could be interviewed. Sharing internal information about the project was possible, too. The collaboration allowed us to create documentation for Compass with the provider's intentions in mind.

Implementation

The basic version of our artifact served as a foundation for the advanced documentation. After finishing our work on the basic version, we started with the iteration for the advanced version. The features included in both documentation were synchronized by copying the changes to the basic version and over to the advanced documentation. However, the parts of the documentation where our two artifacts differed were not synchronized since these differences were intended. After a complete iteration for both versions, we started to actively seek feedback from the provider's if possible. This is where our process includes another round of collaboration.

5.2.2. Architecture of the API documentation

We used a combination of tools to build artifacts that help developers understand and get started with Compass. In total, we used four components to offer holistic documentation of Compass. The architecture is depicted in Figure 5.3. On the bottom, we show an instance of Compass with its exposed GraphQL API and schema file. The three components on top form the documentation for API consumers. The "API Documentation" component presents an overview, concepts, samples, and references to the GraphQL Playground and Specification. The "GraphQL Playground" can be used to test requests against a real Compass API. The "GraphQL Specification" is generated from the schema file and consists of an interactive page to explore the objects and actions of the API.

Compass

The important parts of the Compass component in Figure 5.3 are the exposed API and the schema file describing the API's structure. Compass uses GraphQL as an underlying technology to describe, run, and query its API ². The API is specified using a schema file which is written in a specific schema description language (SDL). The GraphQL schema contains definitions of types, objects, and operations like queries and mutations. Usually, this file is exposed with the API endpoint. In contrast to REST APIs, the client of a GraphQL API can specify what data to receive. Also, GraphQL uses, as the name

²See <https://graphql.org/learn/>

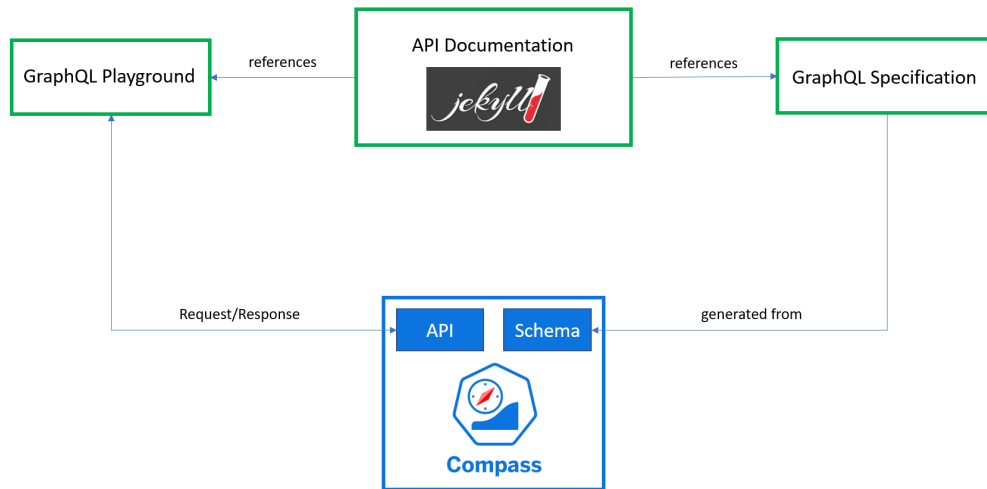


Figure 5.3.: The overall architecture of the documentation we built.

indicates, its own query language which mainly consists of queries (read data) and mutations (manipulate data). An API request in GraphQL is issued as a HTTP POST request with the GraphQL query in the request body. The API response only contains data requested by the client.

Further, Compass is connected to a relational database where it saves created entities and other data structures. This is not depicted in Figure 5.3 for reasons of comprehensibility.

API Documentation

The API documentation contains all the necessary information to learn Compass and get started with samples and tutorials. This component is focused rather on content than technical features. Therefore, we selected a framework that takes a simple input format such as Markdown ³ and generates a static page, respecting all specified references and formatting details. For our purposes we used the open-source project Jekyll ⁴ which is an extensible, easy-to-use static page generator. Jekyll is adopted by many important platforms such as Github ⁵ making it a de-facto standard for static page generation. In Figure 5.4, we present an overview of the inputs and outputs of Jekyll. The Jekyll community has created pre-configured themes with fundamental elements such as navigation bars or layouts. By using a Jekyll theme, we could focus on creating the structure and content of the documentation in Markdown. If needed, we adapted the CSS of the theme to adapt some design elements. Jekyll takes the Markdown and theme as an input and creates a static page consisting of HTML and CSS out of it. This step must be executed on each change in the Markdown files.

³See <https://www.markdownguide.org/>

⁴See <https://jekyllrb.com/>

⁵See <https://github.com/>

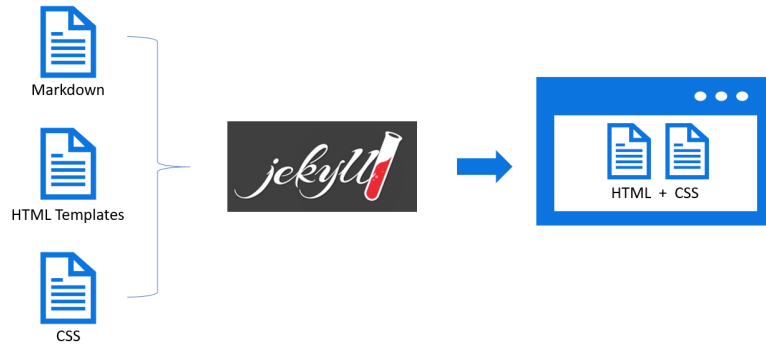


Figure 5.4.: Jekyll takes HTML, Markdown and CSS as an input and create a static web page.

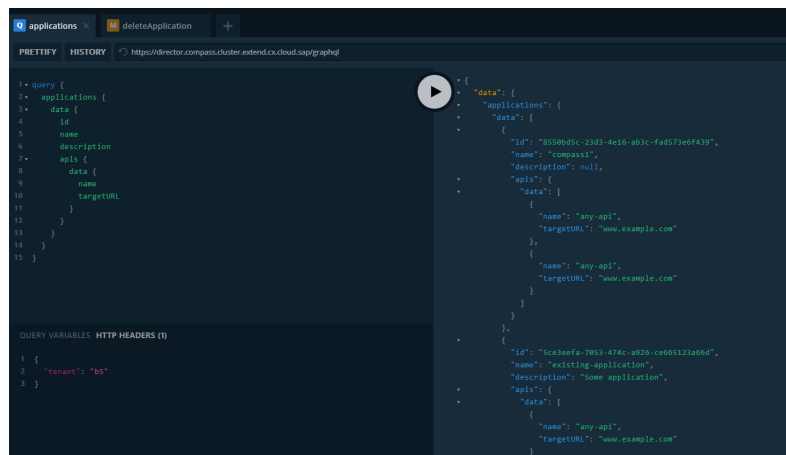


Figure 5.5.: GraphQL Playground supports API consumers in creating requests for a given API endpoint.

GraphQL Playground

The playground (or "sandbox") is a central element for learning the API. We set up the playground to give API consumers a way to test requests against the real API. We integrated the popular GraphQL Playground⁶ into our documentation. The playground consists of a web interface that takes the URL of a GraphQL API endpoint and enables developers to formulate requests (see Figure 5.5). A valuable feature of the playground is that it fetches the GraphQL schema from the endpoint and supports developers with auto complete and syntax highlighting. We set up the playground as an independent component of our documentation.

⁶See <https://github.com/prisma/graphql-playground>

GraphQL Specification

The aforementioned schema file written in the specific SDL contains all information about types, objects, and operations of the GraphQL API. Since this information is necessary for every API consumers, we dedicated an independent component for the specification. Similar to the API documentation, we used a software to create a static web page from the schema file. The project *graphdoc*⁷ generates a searchable page with an overview of all important API elements. A useful feature of the generated pages is that the elements are connected via hyperlinks thus making it possible to click the whole specification. Since we did not expect any changes to the GraphQL schema, we conducted the generation once at the beginning of our implementation and left it unchanged.

5.3. Features & Views

In this section we present the results of our implementations. We demonstrate the key features and UI views of the basic and advanced API documentation that we built for Compass. After giving an overview of the features and sections contained in each artifact and how exactly they differ, we show all aspects of the basic artifact. For the advanced artifact, we only show the features that are substantially different from the basic version.

5.3.1. Overview

We briefly present how we structured the documentation and what the sections contain. Also, we show what the differences between our artifacts regarding the sections are.

Sections of the documentation

There is no consensus in literature regarding a unified structure for API documentation. Our documentation was guided by some of the API software "leaders" according to Gartner (2018) [8], such as MuleSoft⁸ or Apigee (subsidiary of Google)⁹. Another main resource is the blog "Documenting APIs - A guide for technical writers" which explains and references the documentation of leading companies¹⁰. We adopted the following sections which are common in API documentation: overview, getting started, tutorial, and samples.

In **overview**, we explain Compass, the problem it solves, and how it works. This includes architectural diagrams, explanations for each component, and descriptions of possible use cases. The overview in Apigee covers these points for most products¹¹.

⁷See <https://github.com/2fd/graphdoc>

⁸See <https://docs.mulesoft.com/general/>

⁹See <https://docs.apigee.com/api-platform/get-started/get-started>

¹⁰See <https://idratherbewriting.com/learnapidoc/>

¹¹See <https://docs.apigee.com/api-platform/get-started/what-apigee-edge>

The section **getting started**, involves explanations on the first steps towards working with an API. This might include authentication, set up, and learning resources. As an example, IBM Cloud directly references the necessary resources and also directs users to their sandbox application ¹².

A **tutorial** section, is used to demonstrate an exemplary usage scenario of the API. Besides just showing some API functionality, the tutorial has a context and defined outcome. MuleSoft, uses API tutorials to build useful tools, such as instant messaging bots ¹³.

Finally, **samples** are a collection of examples for the various functions of an API. They differ from a tutorial by having no context of use. Instead, the examples are often presented independent of other content with references to the specification or to similar examples. Microsoft presents samples for the Azure REST API using a collection of independent usage examples ¹⁴.

In conclusion, the structure we used for our API documentation is common for many leading API products and covers conceptual topics, usage scenarios, and examples.

Core features

In Table 5.4, we list which artifact contains what information grouped by the sections of our documentation. We note that the sections **Best Practices** and **Glossary** are findings mentioned by our interview participants.

Our two versions do not differ regarding the *Overview*, *Getting Started*, and *Glossary*. For our *Tutorial* section, our basic documentation describes a simple and short scenario whereas the advanced version has a more sophisticated scenario with additional features indicated by the IDs in the table. Further, we designed differences in the *Samples* section by adding more examples in the advanced version alongside important references and tool support. Finally, the advanced artifact contains a *Best Practices* section with hints on using the playground and GraphQL, which is not part of the basic artifact.

5.3.2. Basic

The basic documentation contains all necessary features to help API consumers learn and understand the application. In this section we present what this version contains.

Overview

The overview is supposed to answer the following questions:

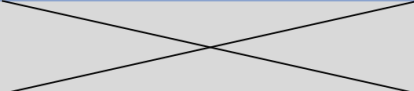
- What is Compass?

¹²See <https://cloud.ibm.com/docs/services/watson?topic=watson-about>

¹³See <https://developer.mulesoft.com/guides/quick-start/muletranslate-bot-using-microsoft-translator-slack>

¹⁴See <https://docs.microsoft.com/en-us/rest/api/apimanagement/>

Table 5.4.: The differences between the basic and advanced artifacts listed per section in the documentation.

Section	Basic	Advanced
Overview	Concepts Components Flows	No additions
Getting Started	Playground Specification	No additions
Tutorial	First steps Simple scenarios	+ Story + Increased complexity + Tool support + Concise descriptions
Samples	Queries Basic I/O behavior	+ Complex requests + References + Increased coverage
Best Practices		GraphQL hints Working with Playground
Glossary	All entities	No additions

- What problem is solved by Compass?
- How does Compass internally work?
- What could I do with Compass?

The first two questions are answered on the landing page of the overview as depicted in Figure 5.6. We included a bullet point list of the main features to highlight these characteristics. Further, we used a simple diagram to illustrate the situation in which Compass is useful.

To answer how Compass works and what possibilities it offers, we further divided the section into *Components* and *Flows*. In *Components* we cover an architectural view of the internal components and their relations. Besides the diagram depicted in Figure 5.7, we provide thorough explanations of each component. The *Flows* are described using sequence diagrams as for example in Figure 5.8.

Getting Started

The Getting Started guide introduces the most important learning resources and tools of the documentation. In our case, the playground and specification are crucial tools to learn Compass. We decided to explain these in detail because we expected developers to use it intensively. Figure 5.9 shows the explanation we created for the playground.

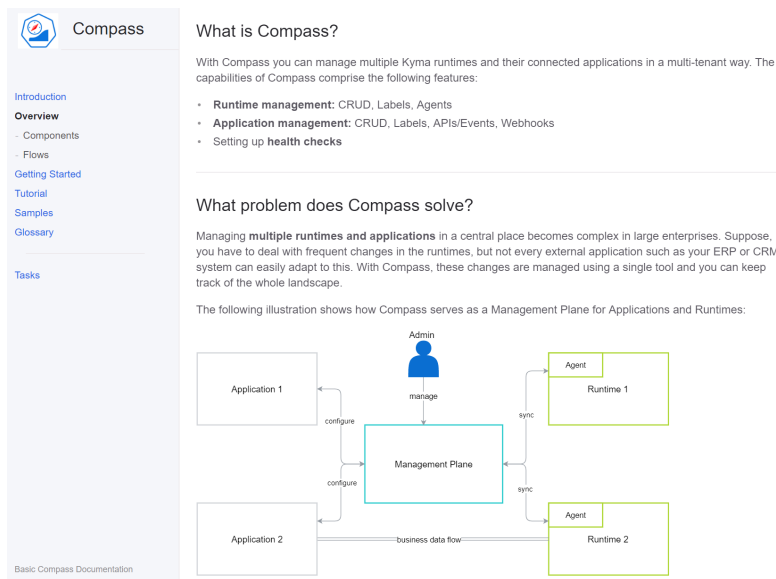


Figure 5.6.: The overview landing page answers what Compass is on a high-level and what problem it solves.

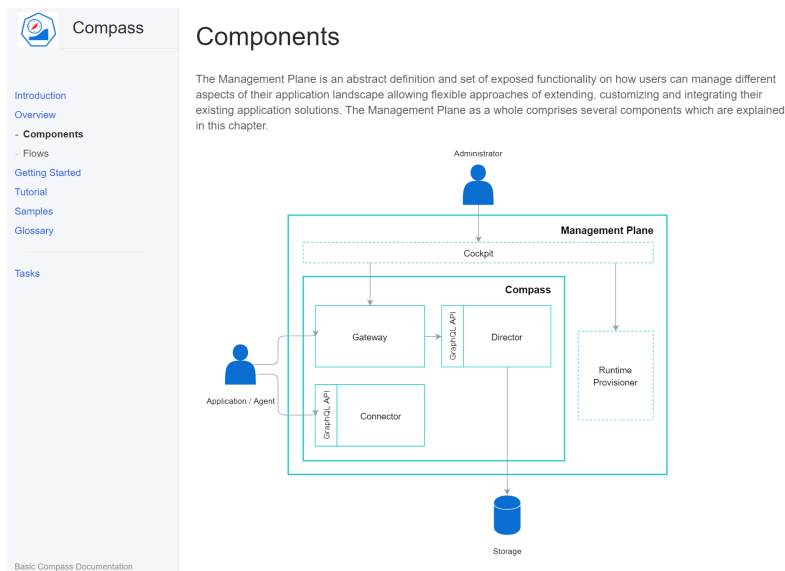


Figure 5.7.: The components are explained using a high-level diagram with each component.

5. Implementation

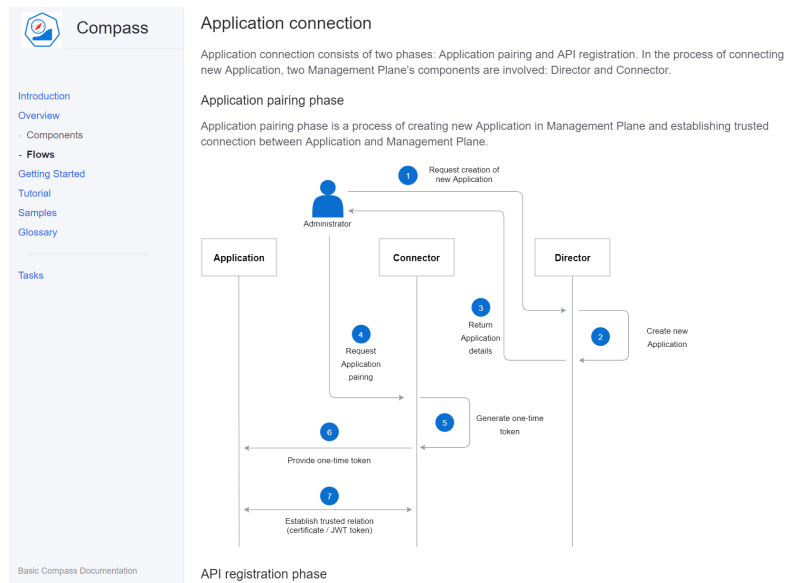


Figure 5.8.: The most common flows in Compass are illustrated using sequence diagrams.

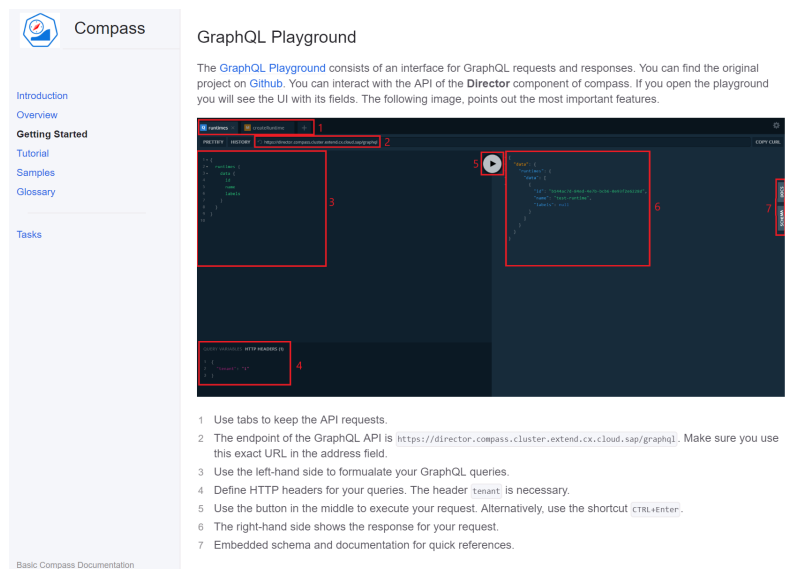


Figure 5.9.: In getting started, we explain important tools, such as the playground, in detail.

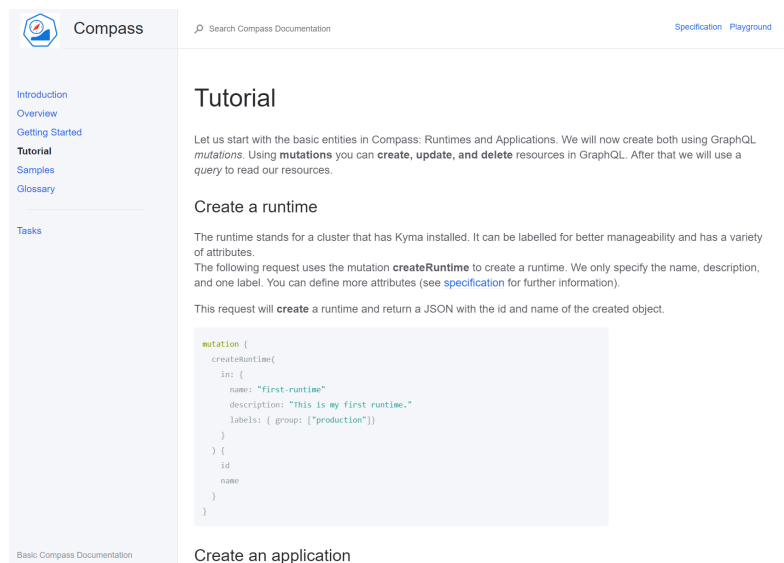


Figure 5.10.: The tutorial contains text segments and simple example requests.

Tutorial

The tutorial is a section where the two artifacts differ significantly. The basic documentation does not have an explicit story but rather demonstrates how to create and query the basic entities of Compass. Also, it is by design not clear what the tutorial wants to achieve. In total, there are three examples in the tutorial with a relatively low level of complexity. The text descriptions are rather long and rarely contain hyperlinks to other supporting resources. The reason why we kept this tutorial simple is because we want to implement a meaningful contrast between the two artifacts. Figure 5.10 shows that the examples are preceded by unstructured text segments and that the samples are rather simple. We demonstrate GraphQL mutations in the tutorial, but we do not demonstrate how API consumers could create complex objects using the samples. Still, the tutorial might help users because it covers the important first steps with Compass.

Samples

The samples section represents a collection of examples for different elements of the API. In our case, the samples only cover the most important ways to query Compass entities. We did not include any descriptions for the examples and also do not reference additional resources. This section contains four examples of API usage. Again, this artifact has a quite basic samples section in order to create a solid difference to the advanced version. The queries are more complex than in the tutorial and therefore provide additional value for API consumers.

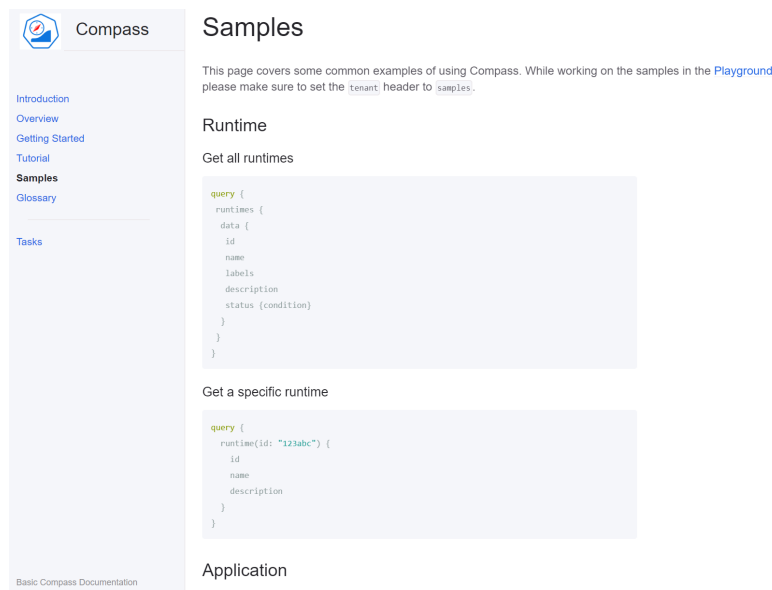


Figure 5.11.: The samples are presented as a list with short titles and the code itself. We only cover the most important queries.

Glossary

The glossary contains definitions and details on special terminology used in the documentation. According to our interview findings, this feature is quite rare in practice but helps to prevent misunderstandings. We included it in our basic version to generally test whether it is perceived as useful. This section consists of an overview table and a list of terms and definitions (see Figure 5.12).

5.3.3. Advanced

The advanced version of our artifact contains additional features and improvements based on our findings from Chapter 4. The *overview*, *getting started*, and *glossary* are the same as in the basic version. We improved the tutorial and samples sections by implementing the previously selected features. Additionally, we added a best practices section containing recommendations from an API provider view.

Tutorial

The tutorial section of the advanced artifact adds functionality mentioned by interview participants to the basic version. The following implications were implemented in the tutorial section:

- F6 - Main scenario as entry point
- F8 - Support familiar tools

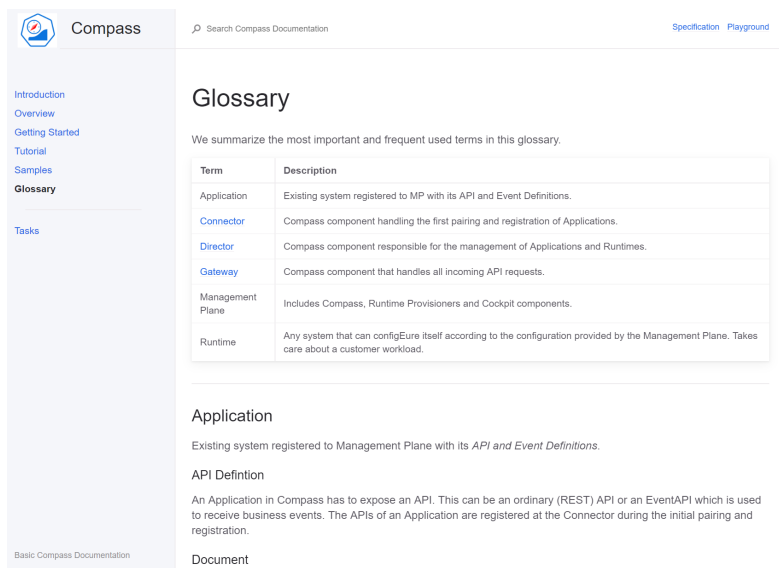


Figure 5.12.: The glossary explains each special term in its context and references further resources in the documentation.

- Q3 - "Less is more" for textual descriptions
- Q7 - Increasing complexity in usage scenarios
- S1 - Story in tutorials
- CF2 - Junctions in paths

In our interviews, we found that a "main scenario" covering a typical usage of the API might be perceived as useful (F6). We implemented this by creating a tutorial using very common steps of working with Compass, such as creating, querying, and updating entities. Further, we included support for the popular API development environment Postman¹⁵. This feature was frequently mentioned, mainly by our developer group (F8). As illustrated in the screenshot 5.13, we embedded a *Run in Postman* button which gives developers the option to import the complete tutorial into their Postman application. Regarding the accompanying text in the tutorial, we focused on a "less is more" philosophy (Q3). In contrast to the basic version, this tutorial contains only very concise text, e.g. in bullet points. A major difference to the basic artifact is the overall complexity. This section contains seven examples, which is more than twice the examples in the basic version. Also, the samples in the tutorial cover complex update mutations that are only present in this version. Additionally, we define our outcomes and a story that was frequently mentioned in our interviews (S1). In the screenshot, this is shortly described as a bullet point list for better readability. Furthermore, we offer

¹⁵<https://www.getpostman.com/>

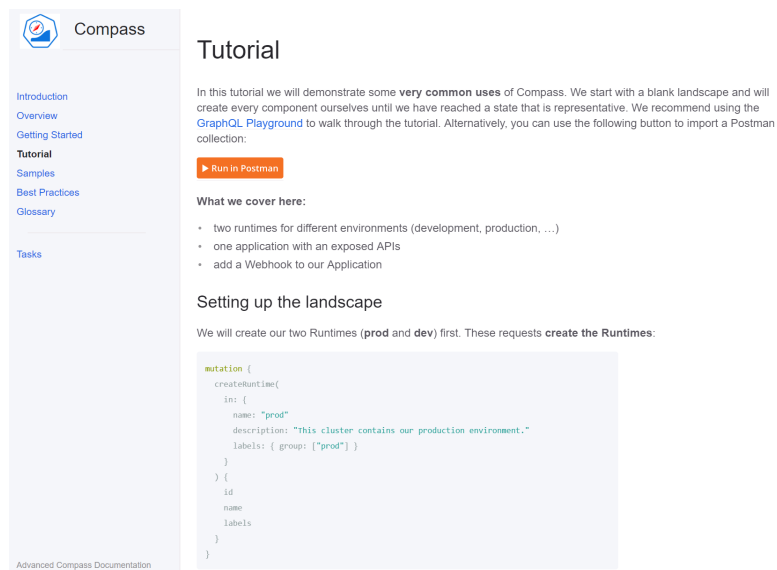


Figure 5.13.: The tutorial starts with a definition of the outcome and explanation of the story.

alternatives between examples (CF2). For example, before the last mutation, we mention what other requests are possible at that point to reach a similar outcome.

Samples

The samples in the advanced documentation are enhanced with frequent references to the specification (F5), tool support for Postman (F8), and a higher coverage of API usages (Q6). This section contains eight examples which is exactly twice as much as in the basic artifact. We mainly added more complex examples for some of the important GraphQL mutations. The descriptions for each example help with looking up type structures and relations to other object in the specification.

- F5 - References to low-level documentation
- F8 - Support familiar tools
- Q6 - Coverage of main cases
- R1 - High-level to low-level

Figure 5.14 shows the upper segment of the samples. Again, we embedded a Postman button to import the samples. Further, each example is accompanied by a short description with a reference to further resources such as the specification. This ensures that developers can instantly open the corresponding definition of the mutation or query.

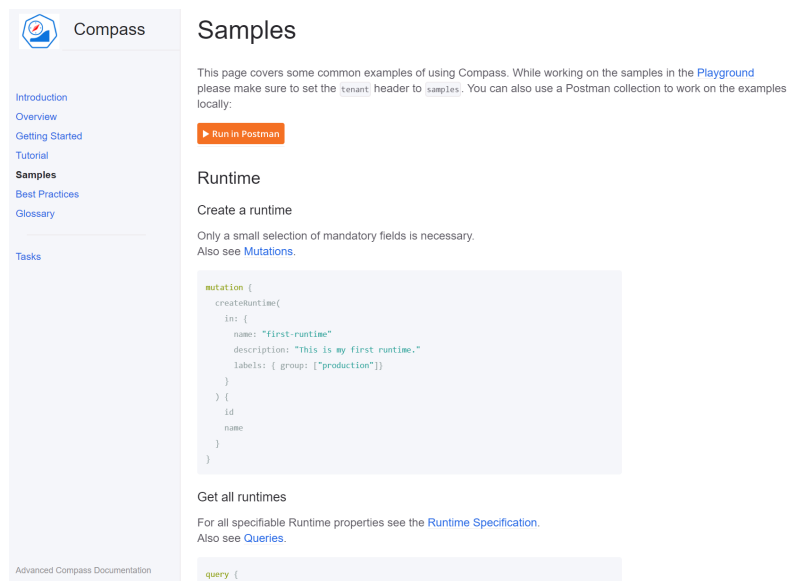


Figure 5.14.: The samples are guided by textual descriptions with references to the type and object definitions in the specification.

Best Practices

The last difference between the basic and advanced version is the best practices section. Some interview participants explicitly mentioned that this knowledge might be useful for working with examples and usage scenarios. It is intended to show API consumers how to work in a highly effective way with the given resources. Together with the API provider we found some practices that might help with learning Compass. In Figure 5.15, we show that best practices regarding GraphQL are instantly demonstrated using an example. We also included tips for working with the playground which were supported by annotated screenshots.

5.3.4. Possible extensions

We are aware that there are missing features which are common in many API documentation. Due to our time constraints, we could not implement all important features that occur in practice. Also, this would not be beneficial for our purposes, since we aim to test a selected set of knowledge in a case study. In this section we list some other features that might have been included in our artifacts.

"Helper Buttons"

A common feature in many API documentation are buttons to facilitate repetitive tasks such as copying, pasting, executing, or changing the programming language of examples. We acknowledge the usefulness of these buttons and also see that large

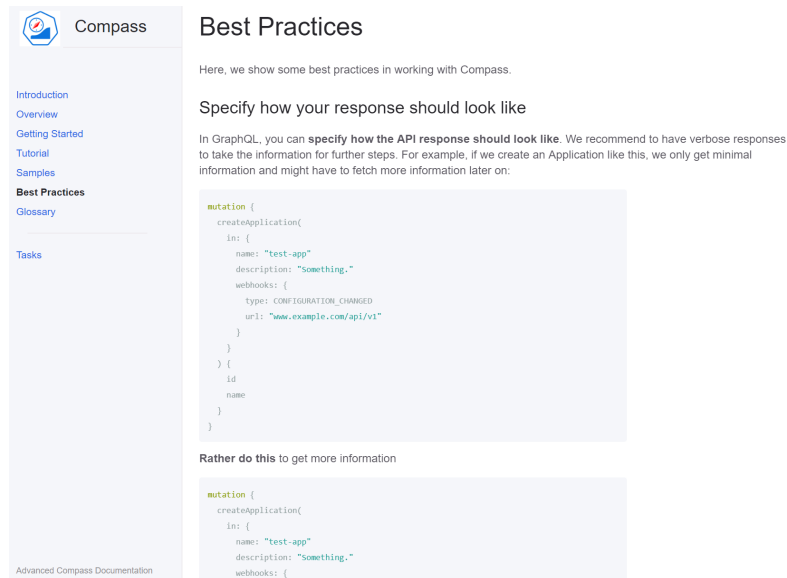


Figure 5.15.: The best practices are specific to GraphQL and the playground. They consist of an explanation and an example to demonstrate the practice.

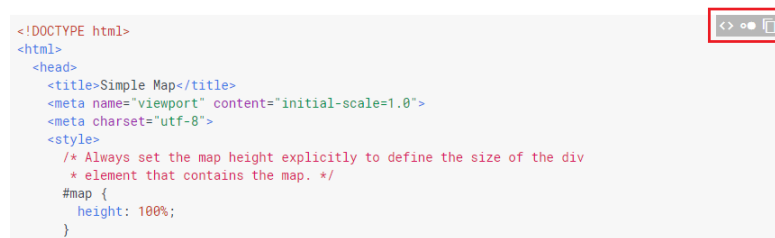


Figure 5.16.: Helper buttons embedded in an examples can support developers in copying and executing of code.

software companies use them in their documentation. For example, the Google Maps API ¹⁶ equips each example in a tutorial with a variety of buttons. In Figure 5.16, we depict an example that has three helper buttons. The first one from left is used to transfer the example to a playground where it can be executed. The second switches between light and dark theme, and the third simply copies the whole example to the clipboard. The first and last one would also be highly suitable for our own implementation.

Instant feedback

Even though we name "instant feedback" as one of our implications (Q4), we did not implement it in our artifacts. However, progressive API documentation such as the one from Twilio ¹⁷ use the feedback feature for each page in their documentation. A click

¹⁶<https://developers.google.com/maps/documentation/maps-static/ntro>

¹⁷See <https://www.twilio.com/docs/usage/api>

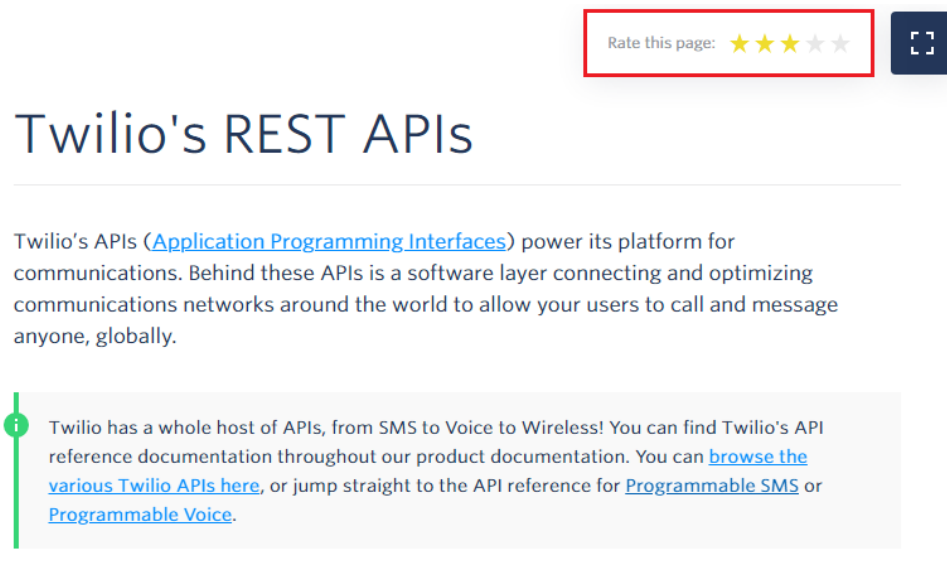


Figure 5.17.: The Twilio API documentation offers an option to instantly rate the current page (accessed on 13.08.19).

on the field highlighted in Figure 5.17 opens a dialog providing the user with options to further specify the feedback. While we believe that this is a valuable feature for a documentation, we chose to not implement it in our artifact due to the low occurrence in the interviews and our constrained time.

Integrating Playground and Specification

The popular API development tool Swagger UI¹⁸ integrates an execution environment with the specification of the API as highlighted in Figure 5.18. This brings the two resources close together and enables developers to work in one central place. One potential drawback of our documentation is that specification and playground are physically separated. However, considering the tools we have chosen the integration would be complicated and therefore not feasible within the scope of this thesis.

¹⁸<https://swagger.io/tools/swagger-ui/>

5. Implementation

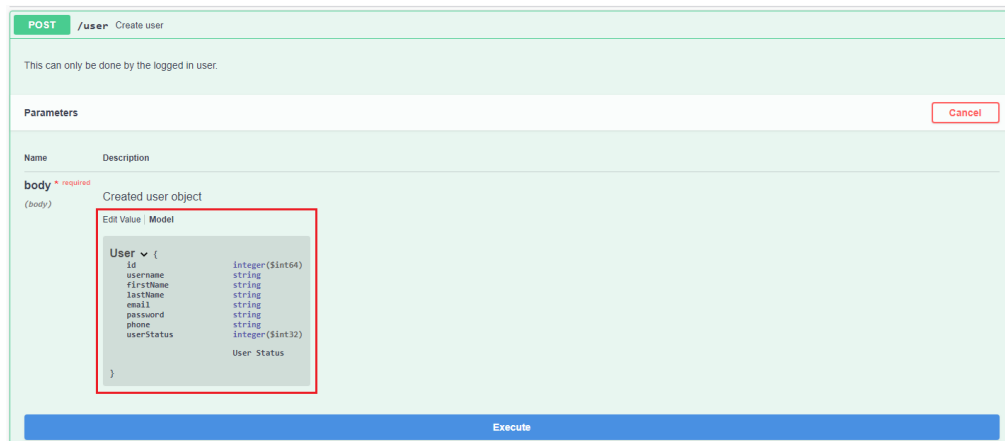


Figure 5.18.: Swagger UI offers a playground that includes information about the model and types (accessed on 13.08.19).

6. Case Study

This chapter presents a case study to research the effect of differences between our two designed artifacts. We presented our two API documentation in two groups of developers and asked them to solve tasks within a given time. Our design of this study is explained by answering some guiding questions in section 6.1. Further, we provide a detailed definition of the case we researched in 6.2. In section 6.3, we demonstrate the results of our quantitative and qualitative analysis. We put all the findings together by giving concrete recommendations in Section 6.4.

6.1. Case Study Design

Our study design is guided by the research of Runeson & Höst (2008) on case studies in software engineering [40]. In their paper, Runeson & Höst posit a set of central questions that any rigorous case study plan should answer. To give the reader a holistic view of our case study, we follow these six questions and explain our answers for each of them.

Objective - what to achieve?

Our case study aims to observe and measure the effect of our design differences in the artifacts on devX. In an explanatory manner, we plan to find evidence for potential differences in developer productivity, perceived usefulness, and satisfaction. To reach this goal we designed two significantly contrasting artifacts that would be used by software developers to solve the same problems. This approach intends to find indicators for features that have a causal relationship to improvements or decreases in devX.

The Case - what is studied?

We conducted our case study with a group of professional software developers from a large software company. We characterize our study as an embedded single-case study [52]. The case is a small sample drawn from the company which we further divide into two groups, each with one version of the artifact. The groups with the basic and advanced artifacts form our units of analysis, hence making it an embedded case study. Since we do not have a comparative sample from a different context or company, we consider this a single-case study. The large software enterprise is well suited for our case study because there is an increased awareness for the importance of developer experience. In Section 6.2 we lay out the details of the case.

Theory - frame of reference

The underlying theories of our case study are mainly concerned with API documentation, the role of examples, and developer experience. We have analyzed the body of knowledge in Chapter 2. Further, we note that there is related work that builds on similar foundations as our study, which is described in Chapter 3.

Research questions - what to know?

This study is supposed to answer our third and last research question:

RQ 3: How can usage scenarios and examples help increase Developer Experience for API consumers?

The question consists of an action and an outcome. The action we apply is providing software developers with API documentation to solve problems. The documentations we provide have differences regarding the usage scenarios and examples which we explained in Chapter 5. The outcome of the research depends on whether the differences in the resources also affect the developer experience. We answer whether our optimizations help with devX by analyzing quantitative and qualitative data collected during the study.

Methods - how to collect data?

We apply multiple data collection methods to gain a holistic view of the studied phenomenon. We do not simply rely on metrics to measure devX but also interview and observe our participants. Overall, the following measures are applied [52]:

- direct observations
- surveys
- semi-structured interviews

Our **direct observations** were conducted by the author and included formal and casual instruments. First, we observed and measured the following metrics using a formal field protocol: duration of phases and tasks, number of API requests, and usage of documentation sections. Using a stopwatch during the study, we could measure the times and note them in our protocol. We observed the laptop screen of the participant to see whether a feature is used or a request is issued. This information was also recorded using tally sheets. If there was any notable behavior, such as a trial and error approach, we noted that particularly. We encouraged participants to express think-aloud comments whenever they find something useful, cumbersome, or just unusual. The utterances of our participants were noted separately as qualitative data.

After participants solved the tasks, they were asked to fill out a **survey** which was used to calculate the System Usability Scale (SUS) score [3]. As recommended in Brooke (1996), we handed out the questionnaire without a discussion or prior briefing about the used API documentation. The questionnaire consists of ten questions with a five-point Likert-scale ranging from "Strongly disagree" to "Strongly agree" (also see Appendix B). Using the SUS, we compared the perceived usability of the participants for the basic and advanced artifact.

Finally, we conducted **semi-structured interviews** with each participant to talk about the usefulness and features of the artifacts. We made notes of the answers but also created audio recordings after a confidentiality agreement. The guiding questions are listed in the Appendix B and were used as a baseline. Often, we asked further questions to clarify the statements of the participants. During the interview, interviewees were allowed to show us features of the artifact on their screen which helped us explicitly note what they perceived as useful.

Selection strategy - where to seek data?

For our study, we had to select a sample of software developers and a suitable set of tasks to be solved by each participant. To select a representative group of software developers, we defined the following requirements which every developer had to fulfill:

- familiar with APIs but NOT with GraphQL
- familiar with domain but NEVER worked with Compass
- at least four years of professional experience in software development

First, it is a prerequisite to know what web APIs are and how they work. Most software developers have likely worked with REST APIs or software libraries before. However, we require that the subjects do not have substantial experience with GraphQL to eliminate side effects that might originate from variations in GraphQL skills. Second, the participants should know the domain of Compass, especially the concepts around runtimes and applications. This is a requirement to prevent elemental misunderstandings. Even though they are familiar with the concepts, the subjects should not have worked with Compass before the study. Finally, we expect the participants to have at least three years of professional software development experience. Previous studies mostly recruit students for their research, which allows for bigger samples but is not desired in our case [33, 22, 19]. We aim to study a phenomenon in a real-world setting and decided to work with professional, experienced software developers instead. We decided to set four years of experience as a minimum similar to previous studies working with participants of "moderate to extensive programming experience" [31, 53]. In contrast to existing research, we do not count experience in university projects or student jobs to ensure we include only participants with four years of professional experience. Using these conditions, we selected 12 participants from the large software vendor for

our case study.

The second selection strategy we defined was concerning the tasks for our study. Before designing a suitable set of tasks to present to our participants, we specified requirements to be satisfied:

- resemble real scenarios of using the API
- varying levels of difficulty
- supported through API documentation without disclosing the solution
- suitable to be solved in a limited amount of time

We aimed to design our study as realistic as possible. The baseline for researching a realistic phenomenon is to provide tasks that emulate real usage scenarios of the API. This is supposed to prevent a strong feeling of fiction during the study. Further, the tasks should get increasingly complex and difficult. Again, this is a requirement to ensure we conduct our study in a realistic setting. By challenging the software developers, we encouraged them to apply problem solving techniques they also use during their day-to-day activities. The API documentation serves as a support to solve the tasks and by abstracting from the tutorial and samples, it should be possible to solve the tasks. However, the tasks should not easily be solvable by copying and running one of the examples. Lastly, the study has fixed time constraints which should also be reflected in the task design. Each task should be solvable in a limited amount of time depending on the difficulty of the task.

In summary, we answered the most important questions around our study design. We aim to research the effect of differences in the API documentation, especially regarding usage scenarios and examples, on the developer experience of professional software developers. The findings of the study are intended to answer our third research question. We collect our data using observations, a SUS survey, and semi-structured interviews. Prior to selecting participants and suitable tasks, we defined selection criteria which we just outlined.

6.2. Detailed Case Description

In this section, we present more details on the case and units of analysis. As part of a complete case description we present the study participants, assigned tasks, and the procedure used to conduct the case study.

Participants

As previously mentioned, we study a case with two embedded units of analysis. Here, the case is defined as a small group of professional software developers from a large

Table 6.1.: An overview of our participants and their professional experience as software developers.

ID	API Experience	Total Experience
A1	10	14
A2	3	4
A3	10	15
A4	4	4
A5	3	9
A6	9	10
B1	7	9
B2	7	7
B3	10	15
B4	2	4
B5	6	10
B6	3	4
Mean	6,17	8,75
Mean Group A	6,50	9,33
Mean Group B	5,83	8,17

multinational software vendor which were selected using a selection strategy described in the previous section. We divided the group into two equally sized, balanced subsets which we call group A (advanced) and group B (basic). The years of professional experience were used as a metric to balance the groups. The two resulting groups served as our units of analysis within the case. In total, we recruited 12 participants for our study which are listed in Table 6.1. Regarding the mean values of the experience, the two groups only differ slightly.

Tasks

All participants were provided with the same set of tasks. Based on the previously specified requirements, we created three tasks. Table 6.2 presents each task with its key facts. The covered topics are an indicator for the boundaries of a task. For example, task two is concerned with two topics which requires the participants to know the structure and especially the relations between "Applications" and "Documents". This task expects the subjects to have deeper knowledge of the documentation. Further, we list the minimum required steps to solve the task, i.e. the number of API requests to get the desired output. We emphasize that this not a direct indicator for the complexity of the task but rather demonstrates how many resources are queried or created during the task.

Table 6.2.: The three tasks differ in difficulty, topic, and required solution steps.

ID	1	2	3
Task	Runtime Status	Application with Documents	Adding API to Application
Topic	Runtimes	Applications, Documents	Application, APIs, Specifications
Description	Find status information about an existing runtime named "production".	Create an application with an embedded document. The document has a certain format and content.	An existing application has an API with a particular spec and version. Add an API to the previously created Application with the same spec and version.
Minimum required steps	2	2	3
Difficulty	Easy	Moderate	Hard



Figure 6.1.: The procedure of our case study that was executed with each participant.

Procedure

Before executing the case study, we designed a procedure to ensure we answer the most important questions and give all participants the same circumstances. A session with one participant took approximately 60 minutes. The steps of the procedure are depicted in Figure 6.1.

The **preparation** of our study consisted of starting the Compass documentation application. All sessions with participants were conducted using the author's personal computer. Further, we had to set up the prerequisites in the GraphQL playground which was required for the tasks. This technical preparation always happened before the meeting. Next, we met the study subject and discussed the **organizational questions**. This included the confidentiality statement for our audio recording and the other data collection methods. The professional experience of the participants was also inquired in this step. Before we started the first phase, we explained the whole study procedure and the time limitations. The **learning phase** marked the beginning of our time measurement. The participants were allowed to read through the documentation and give comments via a think-aloud protocol which we noted. During the learning, they were restricted from opening the task descriptions even if they were interested in the tasks. However, it was possible to end this phase even before the full ten minutes. If the participant was still in the middle of trying an example or finishing the tutorial, we granted extra time but verbally expressed that the time is restricted. The **task phase** gives the subjects around 35 minutes to work on the tasks. We adjusted the time depending on whether participants were very close to the solution or not. We gave a full point for each solved

task, half a point for unfinished tasks with the right solution approach, and zero points for wrong solutions. Our survey to calculate the SUS was conducted immediately after the task completion without prior discussions. The final step of our study were **semi-structured interviews**. Besides the questions listed in Appendix B, the participants were free to comment on other topics as well, e.g. how realistic the study is.

The presented procedure was conducted with each of the 12 participants. The next section presents our results from analyzing the quantitative and qualitative data.

6.3. Results

The findings of our case study can be categorized into quantitative and qualitative results. The quantitative data comprises all metrics collected during the observation and the survey results of our SUS questionnaire. The qualitative data includes the think-aloud protocol and semi-structured interviews with participants.

6.3.1. Quantitative Analysis

We analyzed the quantitative data using mainly descriptive statistics. More precisely, we inspected the distributions, mean values, and variances or deviations in our data. Each subsection corresponds to one type of data we collected during the case study. The size of our samples (N) can be assumed to be 12 for all the following results.

Phases & Success

The main goal of the participants was solving the tasks correctly in minimal time. An exception is the learning phase, in which participants were encouraged to read through the material. In Figure 6.2, we illustrate the mean duration per phase for the groups A and B. On average, the learning phase across all participants took 9,92 minutes which is close to the ten minutes we set as time constraint. Regarding each group separately, we see that group A had a 42% longer learning phase than group B. For each task, we consistently observe that group A is faster in solving the tasks. For task 1 the difference is 18%, for task two it is 31% and finally for task three group A is 14% faster than group B. In total, group A solved tasks 20% faster than group B. The standard deviations, depicted as error bars in the figure, indicate a rather low variation in the estimation. Steadily, the standard deviation corresponds to around 10 to 30% of the mean.

Observation (Time for tasks): Participants in group A took 42% longer to learn and solved the tasks 20% faster than participants in group B.

Further, we investigated the success of our participants measured in points per task. As shown in Figure 6.3, the participants were very successful with a mean score of 2,88. Only three out of 12 subjects did not get full points for task three. Considering both groups, we see that group A scored around 3% better than group B. Drilling down on

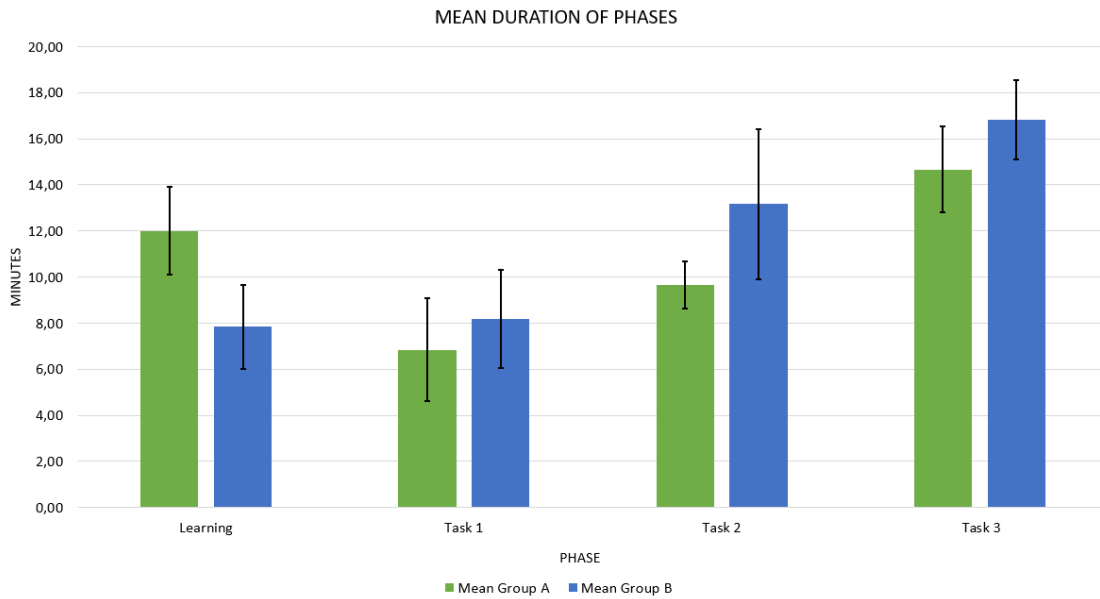


Figure 6.2.: Mean durations for each phase of the study. The error bars stand for one standard deviation into both directions.

task 3 only, which was the most complex exercise, the difference between group A and B amounts to 10%. In summary, we observed clear differences in our time measurements but comparatively small differences in terms of success.

Observation (Correct solutions): Participants in group A scored 10% better in solving the most complex task. There are no differences for the other two tasks.

API Requests

The participants could issue API requests using the GraphQL playground. These requests either failed or successfully returned the requested data. We translated this activity into request success and failure rates depicted in Figure 6.4. The green bars show what ratio of the requests were successful. Respectively, the blue bar shows the corresponding failure rate. At a first glance, we observe that group A has steadily higher success rates than group B. In total, we find that group A makes 44 % more successful API requests than group B. Another result is that the variance among subjects in group B is higher than in group A. Precisely, the standard deviation for group B is 125 % higher than for group A.

Observation (API requests): Participants in group A issues 44% more successful API requests than in group B.

We further discovered a strong positive correlation between API request success rates

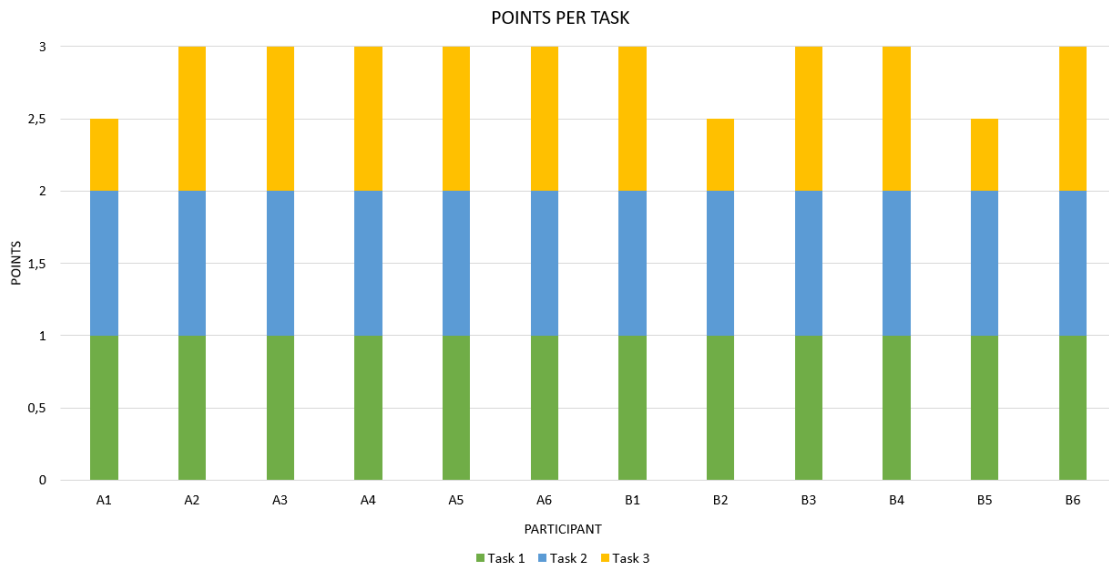


Figure 6.3.: Total points of participants in the three tasks.

and the duration of the learning phase in group A (correlation coefficient $r=0,79$). This indicates that the time investment into learning and trying the API resulted in higher general success rates. Contrary, we found that this does not apply to group B. We even see a moderate negative correlation ($r=-0,47$). The participants in group B issued more successful API requests.

Documentation Usage

Further, we observed what sections of the artifact were used frequently by our participants by noting the "clicks". For each subject, we estimated based on our tally sheet what fraction of their time was spent on the particular section. Again, we analyzed the groups in separation to observe differences. Figure 6.5 depicts the usage of each section as a fraction of the total time. Concerning the mean values, we found three major differences between group A and B. First, the overview part was visited for 22 % longer in group B than in group A. Second, the getting started section was used around 70 % longer in group B. And third, the participants in group A spent circa 29 % more time in the tutorial. Generally, the groups also differ in the standard deviation as pointed out by the error bars. In group A, we consistently notice higher variations than in group B.

Observation (Feature usage): Participants in group A spent less time in the overview and getting started sections but more time in the tutorial.

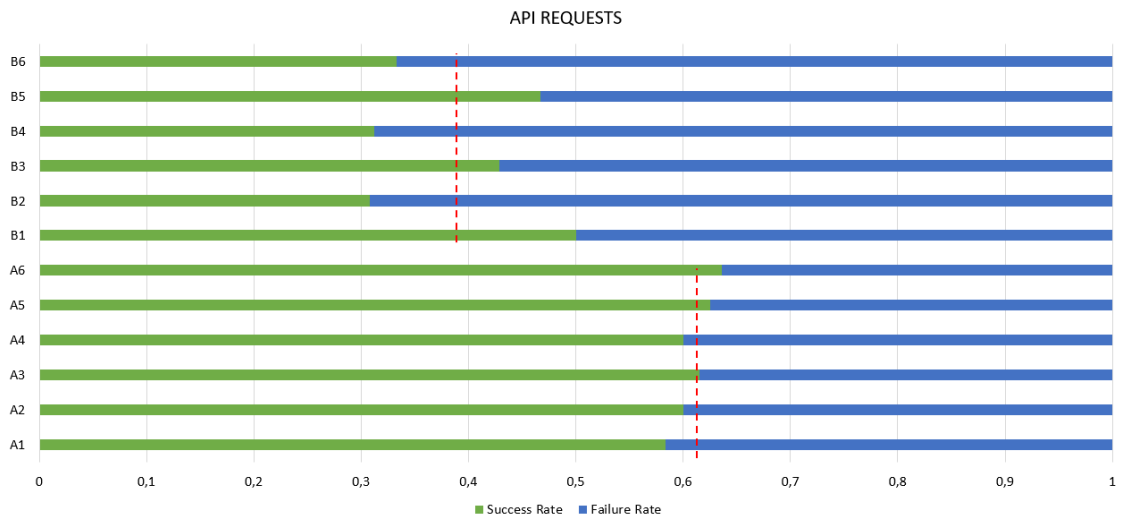


Figure 6.4.: The bars depict success rates of API requests. The dotted red lines represent the mean success rates per group.

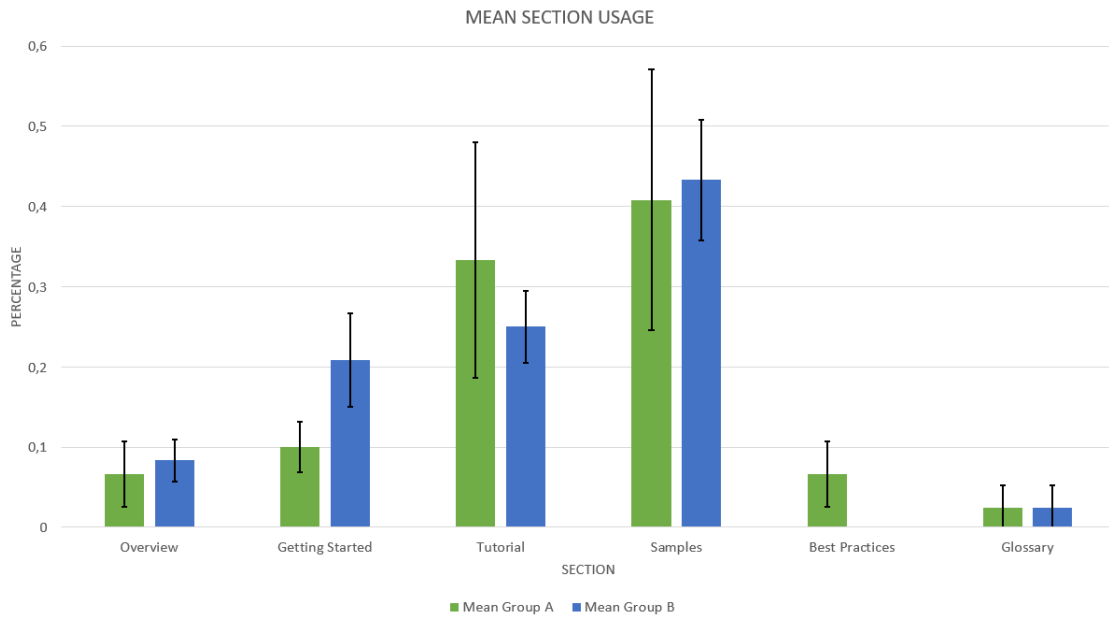


Figure 6.5.: Mean relative usage of the documentation sections. The error bars symbolize one standard deviation into both directions.

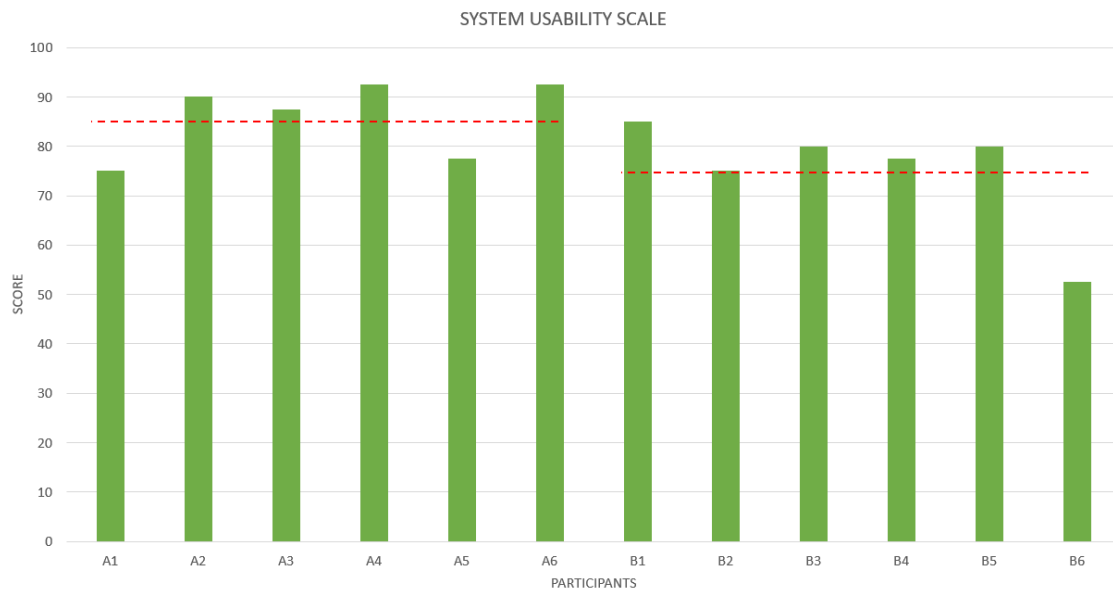


Figure 6.6.: The SUS score for each participant. The dotted red line symbolizes the mean values of group A and B.

System Usability Scale

The final part of our quantitative analysis concerns the System Usability Scale (SUS). We conducted the standardized survey with 12 participants and show the scores in Figure 6.6. The horizontal red line represents the mean SUS score per group. Our advanced artifact was rated at around 85,8 points. The basic API documentation scored 75 points. The difference in SUS scores is around 13 % for group A over B.

Observation (SUS): Participants in group A responded with a 13% higher SUS score to their artifact than participants in group B.

Relating SUS to other metrics, we found that there is a moderate positive correlation between the SUS score and the API request success rate ($r=0,66$). This hints at a higher perceived usability, if the participant's API requests were more successful. This potential relation is also visible in individual cases. If we take B1 and B6 as examples, we observe that B1 has the highest SUS score and API request success rate in group B. B6 on the other hand, has the lowest SUS score among all participants and at the same time one of the lowest API request success rates.

Following our insights from the quantitative data, we present our results from the semi-structured interviews and other notes we made during the study.

6.3.2. Qualitative Analysis

We transferred the notes from our sessions with participants into a qualitative data analysis software where we coded interview using an inductive approach. We reviewed the data with a focus on the frequently mentioned concepts. In this section, we present our findings structured similar to the questionnaire listed in A. For each question, we explain the similarities and differences between the statements of each group. Analogous to chapter 4, we indicate the links to our evidence with the participant ID in squared brackets.

Describe the usefulness of the API documentation for solving the tasks.

We will first explain the statements where we observed consensus among the participants. Regarding the overview section, most of the subjects responded that the information was not useful for the tasks but might become convenient in very complex cases or for integration scenarios with 3rd applications [A2, A4; B3, B4, B5]. The specification of the API was generally regarded as useful, especially for lookup purposes and very specific problem solving [A2; B5, B6]. Also, the fact that there was an execution environment was considered "vital" [B5] and helped participants gain confidence and experience [A2, A3].

The tutorial section was rarely commented on by participants of group B. In group A, however, the tutorial was seen as helpful because with small adjustments to it, the participants were able to solve problems [A1, A2]. Further, the tutorial put samples in a context and therefore helped with real implementation tasks [A4, A5]. The samples were used as a lookup point in both groups, but in group B it was also seen as the main entry point [B1, B2, B5, B6]. Finally, our subjects in group A commented that the tool support for Postman might only be useful if the API consumers are Postman "power users" [A3, A5].

What parts of the documentation did you like the most? And why?

Among all respondents, the specification and playground were the favorite parts of the documentation. The specification was mostly liked for the clickable elements [A1, A2], search function [B3, B4], and the structured presentation [B5]. The playground was regarded as essential for working on the examples and tutorial [A1, A2, A4; B4, B6]. The overview was considered useful to solve more complex tasks which were not part of our study [A1; B2].

The tutorial is often favored in group A, especially regarding the references to other resources that we included in the advanced artifact [A1, A2, A5]. Furthermore, the "good coverage with samples" was liked by some subjects [A3, A4]. Also, the best practices section, which was only available to group A, was appreciated by two participants because "this information is always missing" [A4, A5].

The subjects in group B mainly mentioned the samples as the best part of the documen-

tation [B1, B2, B4]. The positive parts about the samples were that they provide a good entry point [B1, B2].

What parts of the documentation did you not like at all? And why?

There is a general consent about the overview and glossary section. Both were perceived as rather unnecessary and "too much" for solving the tasks by many participants [A2, A3; B2]. However, the interviewees pointed out that the conceptual information might become important for more complex scenarios or non-developers [A4; B4]. Also, there is an agreement that the important text in the documentation was not highlighted enough [A5; B2, B5].

In addition to the mentioned points, group B stated that the descriptions for examples and the tutorial were too long [B1, B6]. Further, the documentation as a whole was perceived as not "developer-centric" enough [B3, B4].

What information or features were missing in the documentation?

In both groups, we found that there was a lack of "helper buttons". This includes small functions e.g. to conveniently copy a code example or transfer examples directly into the playground and execute there. We did not include these features which were found cumbersome to use by some participants [A5; B2, B5]. Further information on this topic can be found in Section 5.3.4.

The participants of group A found that further information on prerequisites was missing [A4].

More than 75% of the statements about missing features came from group B. First, many subjects found that the artifact requires more examples with higher complexity [B1, B2, B6]. Additionally, some participants noticed a lack of links between the resources which required them to conduct more searches [B2, B6]. Finally, the descriptions for examples and tutorials should have been "better" in terms of conciseness and usefulness [B4].

6.4. Recommendations

As a conclusion, we evaluate our case study findings and link them back to the features of our artifacts. We collected evidence about the potential effects of these features and present recommendations based on quantitative and qualitative data. We note that not every feature in our documentation was converted into a recommendation. Features that are not covered in this section provide insufficient evidence for a grounded recommendation. First, we will examine the elements included in both artifacts and second, we cover the differences between the artifacts in detail. An overview of all recommendations can be found in Appendix C.

Executable & Adaptable Examples

We provided both documentations with a way to execute and modify usage examples. This was our implementation of the implications of F1 and F2. Our evidence shows that this feature is "vital" [B5] and often among the favorite parts of the documentation. From our observations, we can confirm how essential it was for the developer experience. We assume that developers would still find a way to execute our presented examples using local applications or their browser. However, we claim it makes a difference for the devX if the API provider maintains the playground for developers.

Recommendation 1: API providers should make an API playground available as an easy way to execute and modify the examples in the documentation.

Misunderstanding Terms

As part of both artifacts, we included a glossary explaining the most important terminology. This is based on the knowledge we found during our first interview round. Our study illustrates that this conceptual knowledge was disliked by many participants and rarely used as our feature usage data shows. The interviews revealed that the API was perceived as too simple to have a glossary. We take this valuable insight to propose the usage of glossaries for highly complex applications. Otherwise, the terminology might also be explained upon the first usage without a separate glossary section.

Recommendation 2: A glossary should be part of the documentation for high-complexity APIs but not for simple APIs. For simple APIs, the glossary is perceived as unnecessary.

References between high-level and low-level documentation

A substantial difference between the documentation was the frequent references we included in the samples and the tutorial of the advanced artifact. The implication F5 served as a foundation for this feature. Our evidence reveals that these references were liked in group A and the tutorial usage was significantly higher than in group B. Further, group B mentioned that references between the specification and tutorial were missing in their version. We take this consistent need as an indicator for the high value of references between the high-level and low-level documentation.

Recommendation 3: The high-level documentation (usage scenarios, examples) should reference the low-level documentation (specification) to facilitate lookups and searches.

Text descriptions & "Less is more"

The textual descriptions in the advanced artifact were concise and focused on crucial information. We implemented the features F7 and Q3 in our tutorial and samples to a large extent. The findings demonstrate that group B disliked the textual descriptions which were less concise and did not highlight much information. Further, we realized that participants in group A executed more examples of their tutorial. We assume that less text in front of examples lowers the barriers to execution. This, in turn, resulted in much higher API request success rates in group A. Even though we highlighted parts of the advanced documentation, the participants responded in interviews that it is much more highlighting necessary. We recommend keeping textual descriptions especially in combination with examples very short and concise.

Recommendation 4: The textual description accompanying examples and usage scenarios should highlight crucial information, be concise, and focused.

Use Case Coverage

An intended major difference between the documentations was the higher number of examples in the advanced version. Our implication Q6 indicates that the coverage of use cases is an important element of usage scenarios. The study results show that participants in group B regard the number of examples in their version as too low. At the same time, group A did not complain about missing examples and the quantitative results of how their attempts at executing samples were much more successful. We discovered a moderate correlation between higher request success rates and perceived usability which is in line with the statements from our participants in group A concerning usage examples. We recommend that increasing the coverage of use cases might positively influence usability. However, it is obvious that more examples might at some point reverse the effect and cause a worse developer experience.

Recommendation 5: Increasing the coverage of the most important cases with API examples might improve the developer experience with regards to developer performance and perceived usability.

Increasing complexity

The usage scenarios in the advanced documentation covered more cases and had an increasing complexity. We based this feature on the implication Q7. Interestingly, the interviews with group B showed that the basic artifact was missing complex examples according to participants. For task two which required the subject to issue the first complex API requests, we observed a significant performance variation between the groups. There is also a substantial difference in the most complex task. We assume that group A had more success with the more complex tasks because they had seen

examples with increasing complexity in the tutorial. Our evidence might be an indicator of the positive effect on developer experience by providing challenging tutorials.

Recommendation 6: The complexity of examples and usage scenarios presented in the documentation should get increasingly higher.

Tool support

The idea to support a popular API development tool in our advanced artifact originated from F8. We implemented a button that can be used to execute examples in Postman. In our study, we observed that this feature was not always noticed and never productively used for the tasks. The interviews disclosed that the tool was seen as unnecessary because of the GraphQL playground. The participants that clicked the button had a positive impression of this feature but think it only makes sense for "power users" [A1, A6]. We also saw an appreciation for the fact that the API provider offers the consumers the possibility to pick a tool. We think this would be a useful feature if the provider offers the right tool support.

Recommendation 7: The examples and usage scenarios should be supported by the right tools. Ideally, the addressed API consumers should be power users of the supported tools.

Helper buttons

A feature we did not include in our artifacts because of time constraints were "helper buttons" (see Section 5.3.4). Unsurprisingly, participants in both groups confirmed that this small feature is standard in other documentations and therefore expected by many developers. A missing "copy to clipboard" button might decrease the developer experience while using a tutorial if copying and testing become cumbersome. We claim that these "helper buttons" might be beneficial for the developer's experience through usage scenarios and examples. There is a variety of helper buttons in practice, some of which are presented in our Section 5.3.4.

Recommendation 8: The examples and usage scenarios should be supported "helper buttons". These help API consumers to work efficiently with the resources by automating repetitive tasks.

7. Discussion

In this chapter, we briefly summarize the key findings of this master's thesis and critically discuss the limitations.

7.1. Key Findings

In the following, we describe the most important findings from this thesis.

Problems with API documentation are impacting API consumers and providers

As expected, the challenges in working with API documentation are not limited to API consumers only. Interestingly, our participants even had valuable insights about the "other" side, i.e., product owners commented on problems API consumers have. This indicates that it is necessary to conduct studies on API documentation from multiple perspectives including all stakeholders.

"Less is more"

A significant effect was observed for our "less is more" implication (Q3). In a quantitative and qualitative analysis, we found that concise descriptions with less cluttered knowledge help developers in extracting the relevant knowledge.

High expectations for executability of examples

Software developers expect that examples work correctly and are executable "out-of-the-box". The demands are higher in the sense that an execution environment and integration to the specification are also assumed to be provided. Missing features such as "helper buttons" are noticed quickly and lower the devX.

Textual descriptions have a significant impact

As previous research pointed out, the textual descriptions for usage scenarios and examples significantly affect the impact on devX. We confirm with multiple observations that this assumption is justified. The descriptions are necessary to create references between high-level and low-level documentation. Further, descriptions help to put examples in a context to lower the "cognitive distance" to the API consumer. In our study, we also found the direct effects on the perceived usability of the documentation. Further, a considerable lack of meaningful descriptions was noticed by participants working with our "basic" documentation.

Complex usage scenarios challenge developers but improve devX

Previous research stressed the need for simplicity over complexity for the API documentation in general. We claim there are resources where it makes sense to increase the complexity. In our interviews, we found that according to participant's usage scenarios should increase their complexity step-by-step (Q7). The case study confirmed that a low complexity is perceived as a disadvantage regarding a tutorial. In reality, developers are challenged with complex tasks. This justifies the need for complex usage scenarios in the API documentation as well (Recommendation 6).

Long learning phases result in high overall API success rates

The more extended learning phase was caused by complex and numerous examples and usage scenarios. Spending more time working on the examples obviously increased the API usage success rates. Besides, we observed that the results during the task phase had been even better. The participants seemed to be more willing to try out examples and had a better idea of what working examples look like. In recommendation 5 we encourage to increase the coverage with examples to foster this effect.

Playground and specifications are useful and a highlight for developers

Even though we did not test the effect of the GraphQL playground and specification, our participants clearly stated that they favored those tools. Our intention behind the playground was to provide a platform for executability and adaptability of examples which was perceived as very useful and a vital feature. On the other hand, the specification was merely meant to present the schema of the API. However, we noticed that the static page provided a much better developer experience in exploring the API schema than a raw file. Previous studies have frequently found the need for a sandbox or playground but the benefits might be even more significant than anticipated [42]. Therefore, we state recommendation 7 which calls for tool support.

Conceptual information is perceived as necessary but rarely used intensively

The role of conceptual knowledge in API documentation is an important topic in research [22, 25]. In theory, there is a consensus that the concepts and context must be understood deeply to use the API as efficient as possible. We confirm most of the findings with our interviews and also formulate implications emphasizing the importance of conceptual knowledge. However, in our practical study, we found that the real usage of these resources is rather low, even during the learning phase. The popularity is even lower, as most of our participants rated the glossary and overview of the documentation as the least favorite sections. At the same time, software developers agree that this knowledge is necessary for efficient usage of complex APIs or the execution of integration scenarios.

Features requested by developers should be evaluated in practical studies

Generally, we found that features requested by developers might not be per se useful. We made this experience with the glossary and overview section of the documenta-

tion. Although we observed a need for these conceptual knowledge resources in our interviews, our evaluation showed that there are many more dependencies we did not have in mind in the first place. For example, the glossary was not perceived useful for simple APIs (Recommendation 2) but rather for complex APIs. The implication is that the realization of beneficial effects heavily depends on the API itself.

7.2. Limitations

The limitations of our study can be explained separately for the three methods we applied: semi-structured interviews, implementations, and the case study.

The findings of our semi-structured interviews must be seen with mainly two limitations. Firstly, generalizability is limited by the fact that we selected a small sample from only one organization. Even though this organization is part of the software industry, the findings might not easily be generalized across domains or even within the same domain. To mitigate this threat, we thoroughly included previous research into our findings. The knowledge types we found were compared to literature and analyzed using an existing taxonomy. Secondly, the questions of our interviews might have been understood differently depending on the participant's vocational background. Our interviewees had different experiences and worked in various projects which has an inevitable effect on their understanding of our questionnaire. This limits the reliability of our collected data. As a countermeasure, we conducted a pre-study with three of the interview participants to review the questions and test the understanding. Our main interview round was the second iteration for our questionnaire.

The two types of API documentation designed during our study are subject to limitations as well. As noted in Chapter 5, we did not implement all common features due to feasibility and relevance reasons. These missing functionalities limit the representativeness of these proof-of-concept implementations. Modern API documentation includes more sophisticated features which certainly affect the devX. This might have also affected our overall usability of the artifacts and therefore influenced the behavior of our participants. However, we minimized this threat by preventing strong disadvantages through missing features. For example, the playground and specification were not integrated but we still provided two separate powerful tools enabling developers to search through the specification easily. Also, we pointed out the possible extensions in this thesis, so that future studies can include them in their API documentation.

Lastly, we conducted a case study which comprises multiple data collection methods including their limitations. We assessed our threats to validity based on the four assessment criteria presented in [40]:

Construct validity: The construct validity indicates to what extent the measures that are studied reflect what the researchers had in mind. This occurs, for example when the discussed constructs are interpreted differently by interviewer and interviewee [40]. In our study, this threat was mitigated by developing artifacts together with a team of API

providers. This ensured that the message conveyed through the API documentation was in line with the provider's view of the API. Also, we gave participants the chance to ask questions during the case study if they were not related to the task solution, leaving enough room for clarification during the conduct of the study.

Internal validity: The threat to internal validity is important when the researcher investigates causal relations within the observed factors. When the observed factors are additionally influenced by a third factor, the internal validity is limited. During a study with developers there might be multiple variables originating from the participant's background knowledge or current projects. We did not have an influence on any of these and they might have affected the relations we observed. However, we designed the case study to be resistant against some significant threats. For example, we used multiple data sources to base our observations on more evidence. Finally, our selection of participants followed criteria that eliminate significant knowledge differences. For example, none of our participants had professional experience with the API technology GraphQL.

External validity: The external validity refers to the generalizability of findings outside of the studied case [40]. We selected participants from only one organization which limits the generalizability of the case study. As already noted, this small environment only has a limited representativeness regarding software developers. On the other hand, our selected participants mainly work in open-source projects which might be closer to the population than highly specialized developers. Also, the selected project "Compass" is an open-source project as well, which mitigates the threats coming from studying a domain-specific closed API.

Reliability: The reliability of the case study reflects what extent the data and analysis depend on the specific researcher [40]. The data from our case study analyzed by only one researcher. This results in a strong influence of the researcher's experiences and biases on the findings. Therefore, the same study by a different researcher might yield other outcomes. The countermeasure for this threat was to include not only qualitative data that is interpreted by the coder, but also collect quantitative data. Our quantitative findings are less dependent on the researcher and there is a decreased room for interpretation.

8. Conclusion

In this chapter we summarize the thesis and provide suggestions for future research.

8.1. Summary

As a summary of our thesis, we answer the research questions presented in the introduction.

RQ1: What are the approaches and concepts to create and publish usage scenarios and examples?

The research on usage scenarios and examples covers the concrete implementation, conceptual information, and quality characteristics. Some approaches aim to provide advanced usage scenarios and examples by focusing on the integration of conceptual knowledge such as references, explanations, and best practices. Other research directions are the relation between examples and the API, complexity, and the usage of test cases as an example. In our study, most previous findings could be confirmed or complemented. The confirmations give further empirical evidence for the relevance of the phenomena observed by other researchers. However, we also found contradictions and novel discoveries that are not yet backed by much literature. This indicates that there may be more approaches and concepts for usage scenarios and examples that are not evaluated for potential usefulness yet.

RQ 2: What are the knowledge types that must be included in usage scenarios and examples?

The semi-structured interviews proved to be a powerful tool to analyze the knowledge that must be included in usage scenarios and examples. Especially, interviewing API providers and consumers was valuable since we had very different perspectives on the same problem. The knowledge can be categorized in a structured way using the knowledge type taxonomy by [25]. In our judgment, the taxonomy covered more types of knowledge that we could find in the interviews and therefore is suitable for holistic considerations of API documentation. The knowledge types to be included in usage scenarios and examples mainly refer to concrete API functionality, quality attributes, and control flow. Conceptual knowledge and directives from API providers are also relevant for usage scenarios and examples. We made 27 implications based on our observations of frequently mentioned statements.

RQ 3: How can usage scenarios and examples be leveraged to improve the developer experience for API consumers?

The improvements in developer experience through optimized usage scenarios and examples are observable in multiple ways. In our case study, we were able to see improvements in performance, perceived usability, task success, and API request success. Besides, our qualitative evaluation shows higher satisfaction with the optimized artifact provided by us. The "How?" question can be answered by closely studying the differences between our two API documentation. For some improvements, we roughly know that they had a significant effect on devX e.g., the increased complexity and coverage of our examples. However, other features may have contributed with which we can not build cause-effect relationships. We also saw that the effects through usage scenarios and examples heavily depends on API characteristics. We concretized these findings in eight recommendations that might help practitioners in improving their documentation.

In the introduction, we outlined the research gaps addressed by this thesis: lack of holistic investigation of problems around API documentation, the fine-grained contribution of different elements within usage scenarios and examples, and practical evaluations with observable effects on devX. Our thesis provides novel contributions to these gaps. By interviewing API providers and consumers, we were able to give more perspectives on challenges and knowledge types in the API documentation. Our two API documentation were designed to show the fine-grained contribution of selected features. Finally, we were able to measure significant differences in the overall devX.

8.2. Future Work

The time constraints of this thesis made it only possible to study contemporary phenomena around the topic of API documentation and developer experience. Our work only provides a snapshot of the effects on devX by our optimizations. In practice, the documentation of an API is consulted frequently over a longer period. It might be valuable to conduct longitudinal studies to measure lasting effects on devX. This could even lead to observable effects on API adoption on the consumer side.

Another potential topic for future research is the detailed study of developer characters and the relation to perceived usefulness of API resources. To make progress in an endeavor to provide optimized API documentation targeting all software developers, more research should take the developer personalities into account.

The artifacts presented in this thesis only investigated a selected set of features ignoring the potential effects of elements such as the playground or specification. At the same time, we found that most developers regard these tools as crucial. As a consequence, we suggest conducting more fine-grained studies to measure the individual contribution of other API resources. Future work could elicit the requirements for such an API playground from developers and conduct a case study to test the effect of an "optimized" API playground.

Finally, we note that most of our work in this thesis was manual. This includes the creation of the examples and usage scenarios but also the references in textual descriptions. There is research on API documentation that occupies itself with an automatic generation of examples or documentation. A future study could try to incorporate findings from our and previous research into automatic workflows. Also, it might be valuable to know which parts of the documentation are suitable for automation at all.

A. Interview Guide - Challenges & Knowledge Types

This interview guide was used for an interview round including 14 IT professionals.

Advance Letter

"Modern business models heavily depend on the adoption of the API they expose to consumers. At [company name] there are several examples for the crucial role of API adoption in business success. An increased API adoption is fostered by enabling users to quickly learn and use a product in an insightful manner with the least number of obstacles. This thesis is supposed to find challenges faced by API providers and consumers in their communication. Based on this information, we propose an approach to increase adoption with usage scenarios and examples and test it in a case study here at [company name]."

Introduction

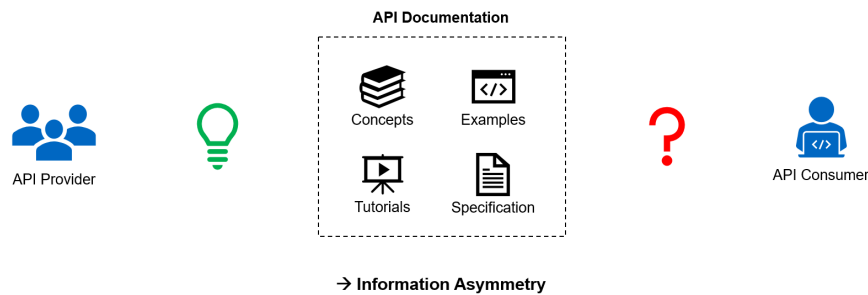
Goals

The high-level goals of this interview round are the following:

- Get empirical evidence for the problem itself
- Elicit existing and new requirements for usage scenarios and code examples as part of an API documentation
- Derive information needs of API consumers and providers in the context of usage scenarios and code examples

Format

This interview comprises 3 closed and 8 open questions. This means that you are encouraged to answer from your own perspective and experience. The interview will approximately take 30 to 45 minutes.



Confidentiality Statement & Documentation

For the confidentiality of your answers in this interview, we guarantee you the following guidelines:

- Your answers are anonymized and will not be traced back to you. The only personal data we will collect are your years of professional experience and role.
- Your answers are used for this thesis only and not for any other non-research related activities.
- Any confidential statements made during the interview, will not be included in our transcript and qualitative analysis.

Questionnaire

Important note: The questionnaire presented here only forms the template with the most important questions. Each interview usually had a different focus topic which resulted in additional questions by the interviewer. However, this set of questions was part of almost every interview.

Personal

1. Which of the following roles describes you the best?
Software Architect, Solution Architect, Enterprise Architect, Developer, Project Manager, Product Owner
2. How many years of professional experience do you have in this or related roles?
1 - 2 years, 3 - 5 years, 6 - 10 years, 11 - 15 years, 16 - 20 years, > 20 years

About the Problem

1. How often do you encounter the presented problem situation in your role?
Frequently, Sometimes, Seldom, Never

-
2. How relevant is the presented problem in practice?
Very relevant, Moderately relevant, Irrelevant
 3. Which kind of problems did you encounter yourself in this context?
 4. What are possible root causes for this problem?

Knowledge in usage scenarios and examples

1. How do API usage scenarios and examples contribute to learning and understanding an API?
2. Do you encounter problems when you use API usage scenarios and examples in your work?
(If yes) Could you name and explain the problems you have?
(If no) What make the working with usage scenarios and examples so seamless for you?
3. How do you solve problems with inaccurate or incomplete examples? And which of the problems **MUST** be solved as opposed to problems that just **SHOULD** be solved?
4. How and for what do you use usage scenarios generally? Please describe your ideal usage scenarios and examples.
5. How would you estimate the effect API usage scenarios and examples have on the API adoption?
Significant effect, Moderate effect, Insignificant effect

Knowledge needs

1. What kind of information do you expect to find when you are using usage scenarios and examples?
2. What kind of information is hardly accessible for API consumers? (such as implicit knowledge, complex domain knowledge, etc.)?

B. Case Study - Material

This appendix contains all materials used during the case study by the participants or author.

Advance Letter

"Dear Participant,
as announced, I would like to invite you to my case study on "Improving the Developer Experience of API Consumers". As part of my thesis, I want to evaluate an API documentation from a developer's perspective. In this study, I will provide you with an API documentation and ask you to solve three tasks only using the available resources. After the tasks, we will have a short evaluation interview. In total, we will need around 45 to 60 minutes.

Additional information:

- You do not need to prepare anything
- We will use a GraphQL API but you do not need to be familiar with this
- The context of the API is Kyma and Compass (<https://github.com/kyma-project> and <https://github.com/kyma-incubator/compass>)

Please let me know if the proposed time works for you! And feel free to ask for more details on the case study.

Thank you and Best regards,
Arif"

Field Notes

Figure B.1 shows the template we used for our field notes to simplify documenting our observations.

Times

Task	Learning	Task 1 - done	Task 2 - done	Task 3 - done
Time				

Requests

Response	Count
Success	
Failed	

Feature Usage

Feature	Usage Count

Figure B.1.: We prepared tables and tally lists to document the times, requests, and features usages.

System Usability Scale

After the tasks, we provided a survey to calculate the System Usability Scale (SUS). To clarify the purpose of the survey, we added a small paragraph with explanations as depicted in Figure B.2.

The System Usability Scale is used to evaluate your experience with the API documentation. Please do only consider the components specific to the documentation. Please answer as intuitively as possible and don't think too long about specific features. If you don't have an answer to a question, just select the middle column.

Question	Strongly disagree 1	2	3	4	Strongly agree 5
1) I think that I would like to use this system frequently.					
2) I found the system unnecessarily complex.					
3) I thought the system was easy to use.					
4) I think that I would need the support of a technical person to be able to use this system.					
5) I found the various function in this system were well integrated.					
6) I thought there was too much inconsistency in this system.					
7) I would imagine that most people would learn to use this system very quickly.					
8) I found the system very cumbersome to use.					
9) I felt very confident using the system.					
10) I needed to learn a lot of things before I could get going with this system.					

Figure B.2.: The SUS consists of ten questions with a Likert scale.

Interview

Finally, we interviewed the participants using the following questions as a guidance:

- Describe the usefulness of the API documentation in solving the tasks.
- What parts of the API documentation did you like the most? Why?
- What parts of the API documentation did you not like at all? Why?
- What information or features were missing in the documentation?

C. Case Study - Results

This appendix contains all results of the case study in detail with data points per participant.

Recommendations

Based on our case study findings we formulate the eight recommendations depicted in Table C.1.

Table C.1.: The summary of the recommendations we stated based on our case study.

ID	Topic	Recommendation
1	Examples	API providers should make an API playground available as an easy way to execute and modify the examples in the documentation.
2	Terminology	A glossary should be part of the documentation for high-complexity APIs but not for simple API. For simple APIs, the glossary is perceived as unnecessary
3	References	The high-level documentation (usage scenarios, examples) should reference the low-level documentation (specification) to facilitate lookups and searches.
4	Descriptions	The textual description accompanying examples and usage scenarios should highlight crucial information, be concise, and focused.
5	Coverage	Increasing the coverage of the most important cases with API examples might improve the developer experience with regards to developer performance and perceived usability.
6	Complexity	The complexity of examples and usage scenarios presented in the documentation should get increasingly higher.
7	Tool support	The examples and usage scenarios should be supported by the right tools. Ideally, the addressed API consumers should be power users of the supported tools.
8	Helper Buttons	The examples and usage scenarios should be supported "helper buttons". These help API consumers to work efficiently with the resources by automating repetitive tasks.

Bibliography

- [1] W. C. Adams. "Conducting Semi-Structured Interviews." In: *Handbook of practical program evaluation*. Vol. 35. Jossey-Bass & Pfeiffer Imprints, Wiley, 2015, pp. 492–505.
- [2] A. Anthony. *Tracking the Growth of the API Economy*. <https://nordicapis.com/tracking-the-growth-of-the-api-economy/>. (Accessed on: August 29, 2019).
- [3] J. Brooke. *SUS-A quick and dirty usability scale*. 1996.
- [4] D. S. Cruzes and T. Dyba. "Recommended Steps for Thematic Synthesis in Software Engineering." In: *International Symposium on Empirical Software Engineering and Measurement (ESEM), 2011*. IEEE, 2011, pp. 275–284.
- [5] D. S. Cruzes and T. Dybå. *Research synthesis in software engineering: A tertiary study*. 2011.
- [6] B. De. *API management: An architect's guide to developing and managing APIs for your organization*. First edition. New York: Apress, 2017.
- [7] F. Fagerholm and J. Munch. "Developer experience: Concept and definition." In: *2012 International Conference on Software and System Process (ICSSP)*. Ed. by R. Jeffery. IEEE, 2012, pp. 73–77.
- [8] Gartner. *Magic Quadrant for Full Life Cycle API Management*. 2018.
- [9] E. L. Glassman, T. Zhang, B. Hartmann, and M. Kim. "Visualizing API Usage Examples at Scale." In: *Engage with CHI*. The Association for Computing Machinery, 2018, pp. 1–12.
- [10] L. Guy. *An API-First Development Approach*. <https://dzone.com/articles/an-api-first-developmentapproach->. (Accessed on: August 29, 2019).
- [11] A. Hevner. *A Three Cycle View of Design Science Research*. 2007.
- [12] A. Hevner, S. March, S. T. Park, J. Park, Ram, and Sudha. *Design Science in Information Systems Research*. 2004.
- [13] D. Hoffman and P. Strooper. *API documentation with executable examples*. 2003.
- [14] D. Hoffman and P. Strooper. "Prose+test cases=specifications." In: *Proceedings*. IEEE Computer Society, 2000, pp. 239–250.
- [15] S. E. Hove and B. Anda. "Experiences from Conducting Semi-structured Interviews in Empirical Software Engineering Research." In: *11th IEEE International Software Metrics Symposium*. IEEE Computer Society, 2005, p. 23.

- [16] ISO 9241-210:2010. *Ergonomics of human system interaction - Part 210: Human-centered design for interactive systems*. Switzerland, 2010.
- [17] J. Jiang, J. Koskinen, A. Ruokonen, and T. Systs. "Constructing Usage Scenarios for API Redocumentation." In: *15th IEEE International Conference on Program Comprehension, 2007*. IEEE, 2007, pp. 259–264.
- [18] T. Johnson. *Documenting APIs - A guide for technical writers*. <https://idratherbewriting.com/learnapidoc/>. (Accessed on: August 29, 2019).
- [19] J. Kim, S. Lee, S.-W. Hwang, and S. Kim. *Enriching Documents with Examples: A Corpus Mining Approach*. 2013.
- [20] A. J. Ko, R. DeLine, and G. Venolia. "Information Needs in Collocated Software Development Teams." In: *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 344–353.
- [21] A. J. Ko, B. A. Myers, and H. H. Aung. "Six Learning Barriers in End-User Programming Systems." In: *2004 IEEE Symposium on Visual Language and Human Centric Computing*. IEEE, 2004, pp. 199–206.
- [22] A. J. Ko and Y. Riche. "The role of conceptual knowledge in API usability." In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2011*. IEEE, 2011, pp. 173–176.
- [23] A. Kuhn and R. DeLine. "On designing better tools for learning APIs." In: *2012 ICSE Workshop on Search-Driven Development - Users, Infrastructure, Tools and Evaluation*. IEEE, 2012, pp. 27–30.
- [24] T. C. Lethbridge, J. Singer, and A. Forward. *How software engineers use documentation: the state of the practice*. 2003.
- [25] W. Maalej and M. P. Robillard. *Patterns of Knowledge in API Reference Documentation*. 2013.
- [26] S. Maddox. *API Technical Writing*. <https://de.slideshare.net/sarahmaddox/api-technical-writing>. (Accessed on: September 01, 2019). TechComm, 2014.
- [27] S. G. McLellan, A. W. Roesler, J. T. Tempest, and C. I. Spinuzzi. *Building more usable APIs*. 1998.
- [28] M. Meng, S. Steinhardt, and A. Schubert. *How Developers Use API Documentation: An Observation Study*. 2019.
- [29] M. Meng, S. Steinhardt, and A. Schubert. *Application Programming Interface Documentation: What Do Software Developers Want?* 2018.
- [30] P. Morville and P. Sullenger. *Ambient Findability: Libraries, Serials, and the Internet of Things*. 2010.
- [31] B. A. Myers, S. Y. Jeong, Y. Xie, J. Beaton, J. Stylos, R. Ehret, J. Karstens, A. Efeoglu, and D. K. Busse. *Studying the Documentation of an API for Enterprise Service-Oriented Architecture*. 2010.

-
- [32] B. A. Myers and J. Stylos. *Improving API usability*. 2016.
- [33] S. M. Nasehi and F. Maurer. "Unit tests as API usage examples." In: *IEEE International Conference on Software Maintenance (ICSM), 2010*. IEEE, 2010, pp. 1–10.
- [34] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns. "What makes a good code example?: A study of programming Q&A in StackOverflow." In: *ICSM 2012*. IEEE, 2012, pp. 25–34.
- [35] J. Nykaza, R. Messinger, F. Boehme, C. L. Norman, M. Mace, and M. Gordon. "What programmers really want." In: *Proceedings of the 20th Annual International Conference on Documentation*. ACM Press, 2002, pp. 133–141.
- [36] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee. *A Design Science Research Methodology for Information Systems Research*. 2007.
- [37] P. Rantanen. *REST API example generation using Javadoc*. 2017.
- [38] M. P. Robillard. *What Makes APIs Hard to Learn? Answers from Developers*. 2009.
- [39] M. P. Robillard and R. DeLine. *A field study of API learning obstacles*. 2011.
- [40] P. Runeson and M. Höst. *Guidelines for conducting and reporting case study research in software engineering*. 2008.
- [41] S. M. Sohan, C. Anslow, and F. Maurer. "A Case Study of Web API Evolution." In: *Proceedings, 2015 IEEE World Congress on Services*. Conference Publishing Services, IEEE Computer Society, 2015, pp. 245–252.
- [42] S. M. Sohan, C. Anslow, and F. Maurer. "Automated example oriented REST API documentation at Cisco." In: *ICSE-SEIP 2017*. IEEE Computer Society, Conference Publishing Services, 2017, pp. 213–222.
- [43] S. M. Sohan, F. Maurer, C. Anslow, and M. P. Robillard. "A study of the effectiveness of usage examples in REST API documentation." In: *VL/HCC 2017*. IEEE, 2017, pp. 53–61.
- [44] J. Stylos. *Making apis more usable with improved api designs, documentation and tools*. 2009.
- [45] J. Stylos, B. Graf, D. K. Busse, C. Ziegler, R. Ehret, and J. Karstens. "A case study of API redesign for improved usability." In: *IEEE Symposium on Visual Languages and Human-Centric Computing, 2008, VL/HCC 2008*. IEEE, 2008, pp. 189–192.
- [46] G. Uddin and M. P. Robillard. *How API Documentation Fails*. 2015.
- [47] K. Vasudevan. *Best Practices in API Documentation*. <https://swagger.io/blog/api-documentation/best-practices-in-api-documentation/>. (Accessed on September 01, 2019). SwaggerBlog, 2017.
- [48] K. Vasudevan. *Design First or Code First: What's the Best Approach to API Development?* <https://swagger.io/blog/api-design/design-first-or-code-first-api-development/>. (Accessed on September 01, 2019). SwaggerBlog, 2017.

- [49] S. Wang, I. Keivanloo, and Y. Zou. “How Do Developers React to RESTful API Evolution?” In: *Service-Oriented Computing*. Vol. 8831. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 245–259. ISBN: 978-3-662-45390-2.
- [50] R. Watson, M. Stamnes, J. Jeannot-Schroeder, and J. H. Spyridakis. “API documentation and software community values.” In: *SIGDOC '13*. ACM, 2013, p. 165.
- [51] J. Webster and R. T. Watson. *Analyzing the past to prepare for the future: writing a literature review*. 2002.
- [52] R. K. Yin. *Case study research: Design and methods*. 5. edition. SAGE, 2014.
- [53] J. Zhang, H. Jiang, Z. Ren, T. Zhang, and Z. Huang. *Enriching API Documentation with Code Samples and Usage Scenarios from Crowd Knowledge*. 2019.