Online First
2019

Communication
Design
Quarterly

Published by the Association for Computing Machinery
Special Interest Group for Design of Communication
ISSN: 2166-1642

How Developers Use API Documentation: An Observation Study

Michael Meng
Merseburg University
of Applied Sciences
michael.meng@hs-merseburg.de

Stephanie Steinhardt
Merseburg University
of Applied Sciences
mail@steinhardt-dokumentation.de

Andreas Schubert
Merseburg University
of Applied Sciences
andreas.schubert83@gmail.com

Published Online January 29, 2019
CDQ 10.1145/3274995.3274999
This article will be compiled into the quarterly publication and archived in the ACM Digital Library.

Communication Design Quarterly, Online First
https://sigdoc.acm.org/publication/

How Developers Use API Documentation: An Observation Study

Michael Meng
Merseburg University
of Applied Sciences
michael.meng@hs-merseburg.de

Stephanie Steinhardt
Merseburg University
of Applied Sciences
mail@steinhardt-dokumentation.de

Andreas Schubert
Merseburg University
of Applied Sciences
andreas.schubert83@gmail.com

ABSTRACT
Application Programming Interfaces (APIs) play a crucial role in modern software engineering. However, learning to use a new API often is a challenge for developers. In order to support the learning process effectively, we need to understand how developers use documentation when starting to work with a new API. We report an exploratory study that observed developers while they solved programming tasks involving a simple API. The results reveal differences regarding developer activities and documentation usage that a successful design strategy for API documentation needs to accommodate. Several guidelines to optimize API documentation are discussed.

CCS Concepts
• Software and its engineering~Rapid application development

Keywords
API documentation, observation method, information design, usability

Manuscript received April 23, 2018; revised October 4, 2018; accepted November 16, 2018. Date of publication January 29, 2019.

CDQ 10.1145/3274995.3274999

INTRODUCTION
When developing software, engineers routinely make use of data and services provided by other applications via Application Programming Interfaces (APIs, Myers & Stylos, 2016). Stylos (2009) points out that many tasks require engineers to “stitch together” functionality that existing APIs provide instead of programming functionality from scratch (Stylos, 2009, p. 4). With businesses and organizations using the Internet to expose data and services, the importance of APIs has increased even more in recent years. Learning the features of an API, the elements it offers and how to combine these elements in order to bring a desired functionality about is a common task every software developer faces.

APIs are typically published along with API references, tutorials, example projects and other resources designed to facilitate the learning task (Mihaly, 2011). Still, getting into a new API often is a challenge and insufficient learning resources (including API documentation) have been described as a major factor contributing to this challenge (Robillard, 2009; Robillard & DeLine, 2011).

This article contributes the results of an empirical study that examined how developers use documentation when getting into a new API. Our work is driven by the hypothesis that problems with API documentation may in part reflect usability problems, and in particular, that content and structure of the documentation sometimes do not match the expectations and work habits of developers. Consequently, for API documentation to be an effective aid in learning an API, we need to know which general strategies software developers adopt when solving programming tasks, which information they need and which information resources they turn to.

To address these issues, we conducted a study using the observation method.[1] We asked developers to solve a series of programming tasks with an API that was unfamiliar to them. We then analyzed which strategies they adopted to solve the tasks, which parts of the API documentation they used, and which design features of the API documentation led to problems. Based on our findings, we propose several design guidelines that can help to make API documentation more effective.

BACKGROUND

Research on API Documentation

Trying to understand the general strategies developers adopt when solving programming tasks, their information needs and information resources they turn to, has been an area of active research in recent years (see Robillard & DeLine, 2011, and Meng, Steinhardt & Schubert, 2018, for overviews).

Regarding general strategies, there is evidence that developers follow different approaches when solving programming tasks with a new API (Stylos & Clarke, 2007; Meng et al., 2018). Clarke (2007) described these strategies in terms of three personas, referred to as systematic, opportunistic and pragmatic developers. While systematic developers approach an API top down and try to develop a more thorough understanding of the API before turning to the details of a task, opportunistic developers use a bottom-up approach. They try to start coding immediately and search for information, such as code examples, directly addressing the issue at hand. The pragmatic developer combines elements of the top-down and bottom-up approaches.

Regarding information needs of developers and information resources they use, studies demonstrated that developers expect standard quality criteria for technical documentation to apply to API documentation as well, such as accuracy, clarity and completeness (Uddin & Robillard, 2015; Watson, Stamnes, Jeannot-Schroeder & Spyridakis, 2013). Developers strongly rely on API reference information and code examples (McLellan, Roesler, Tempest & Spinuzzi, 1998; Nykaza et al., 2002; Meng et al., 2018). They are less willing to read documents that focus on conceptual information, although conceptual information, such as background information on the problem or domain addressed by the API, determines how efficiently API documentation can be used (Jeong et al., 2009; Ko & Riche, 2011). With respect to specific contents, it has been argued that API documentation should provide a concise overview of the overall purpose and the main features of the API to enable quick orientation (Watson et al., 2013; Inzunza, Juárez-Ramírez & Jiménez, 2018; Meng et al., 2018). Moreover, API documentation has to provide scenarios that illustrate entry points into the API, because identifying such entry points is a key problem in the initial learning process (Robillard & DeLine, 2011).

Methods in API Documentation Research

Most previous studies relied on interviews, questionnaires or other inquisitive methods such as diary studies (e.g. Inzunza et al., 2018; Lutters & Seaman, 2007; Robillard & DeLine, 2011; Meng et al., 2018; Uddin & Robillard, 2015; Sillito & Begel, 2013). While these methods have proven useful and helped to generate important findings, their dominance in research on API documentation poses several challenges (Lethbridge, Sim, & Singer, 2005). First, answers provided during interviews and in response to a questionnaire rely on self-report and the ability of developers to reflect on their work habits, the approach they take when solving a problem or the deficiencies they see in the documentation they use. This ability may vary greatly.

Moreover, answers in an interview or a questionnaire reflect what developers say they do, but not necessarily what they actually do. As discussed in Lethbridge et al. (2005), humans often do not recall events around them, and they tend to remember events that they find meaningful. In the same vein, it has been noted that what developers say they do and what they actually do does not necessarily coincide. For example, Lethbridge, Singer, & Forward (2003) noted that the software engineers who participated in their study claimed to spend approximately 40% of their time reading documentation. However, the observation results showed that only 3% of the logged events over the entire observation period were in fact related to documentation. We conclude that existing studies should be complemented by studies that do not rely on self-report and directly observe activities of software developers while working with an API.

THE OBSERVATION STUDY

Research Objectives

The goal of our study was to examine how developers use documentation when they learn a new API and start using it to solve programming tasks. To this end, we asked software developers to solve a set of pre-defined tasks using a public API unfamiliar to them on the basis of the documentation published by the API provider. The API documentation included different types of information resources, such as an API reference, a conceptual overview and examples illustrating specific usage scenarios (see below for more details). The research design was chosen to address the following objectives:

- To analyze which information resources offered by the API documentation developers use to which extent
- To characterize the strategies developers adopt when starting to work with a new API
- To identify aspects related to content, content design or content accessibility which hinder efficient task completion

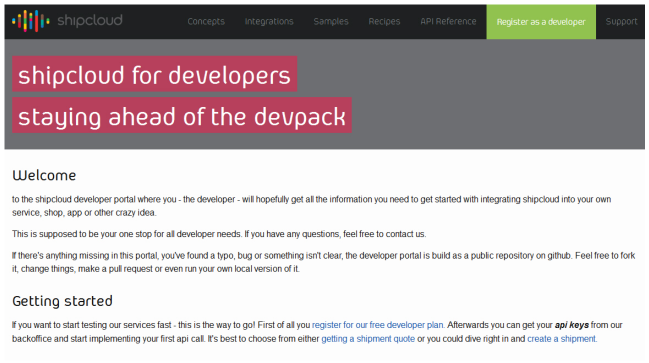
Our study was exploratory in nature and not designed to test specific hypotheses about documentation usage or programming strategies. We took a qualitative approach that focuses on individual observations and on patterns and trends that become apparent when comparing observations across participants.

Method

Materials

The purpose of the API selected for our test is to connect Web shops to shipping providers (http://www.shipcloud.io). The API serves as a proxy that enables simple and uniform access to a broad range of shipping providers and services. Using the API, shop owners can create shipping labels, place pickup requests, define drop authorizations, and much more. The API is based on the REST (REpresentational State Transfer) paradigm. A REST API exposes data through resources that are accessed using standard HTTP requests such as GET, POST or DELETE (see Johnson, n.d., for a nontechnical introduction to REST APIs). The shipcloud API uses resources to represent e.g. shipments, addresses or pickup requests. Via HTTP requests, these resources can be created, updated or deleted.

API documentation is provided online on a developer portal (Figure 1). The documentation is structured in several content categories, such as "Concepts" or "API reference," which are described in Table 1. Each content category is represented by a single page which can be accessed via a top-navigation link. Complementing the documentation, the developer portal contains additional links



shipcloud for developers
staying ahead of the devpack

Welcome

to the shipcloud developer portal where you - the developer - will hopefully get all the information you need to get started with integrating shipcloud into your own service, shop, app or other crazy idea.

This is supposed to be your one stop for all developer needs. If you have any questions, feel free to contact us.

If there's anything missing in this portal, you've found a typo, bug or something isn't clear, the developer portal is build as a public repository on github. Feel free to fork it, change things, make a pull request or even run your own local version of it.

Getting started

If you want to start testing our services fast - this is the way to go! First of all you register for our free developer plan. Afterwards you can get your ***api keys*** from our backoffice and start implementing your first api call. It's best to choose from either getting a shipment quote or you could dive right in and create a shipment.

Figure 1: Welcome page of the developer portal for the API used in our study. The screenshot illustrates layout and structure of the portal at the time of conducting the test (spring 2016). The different content categories (e.g. Concepts, API reference, Recipes) can be accessed using tabs in the top level navigation.

Table 1: Content categories of the API documentation used for the test

Category	Description
Welcome page	Entry point for developers; includes description of process to register as developer and access the sandbox environment, as well as pointer to some resources that provider recommends to access
Concepts	Brief overview of the API, authentication process, basic request structure and parameter handling, list of carriers and supported services; includes additional information on licensing, pricing as well as links to additional resources (integration guide, webhooks)
Integrations	Provides access to specific API integrations in different languages, such as Ruby. Link to GitHub repository containing integrations
Samples	Provides examples for basic use cases, such as creating a new shipment, updating a shipment or getting a shipment quote
Recipes	Provides examples for special use cases and carrier-specific services, such as weekend delivery, pickup requests or delivery to specific drop stations
API reference	List of available resources, parameter descriptions, description of payload; also includes description of how each resource can be accessed, updated or deleted

and resources, for example to register for a developer account or to access specific support services. The main shipcloud web site is available in German and in English. The developer portal and the API documentation are provided in English only which reflects standard practice developers in Germany are used to. During the test, participants had access only to the developer portal.

Participants

For the test, we recruited 11 developers (10 male, 1 female) as participants from three different organizations: six participants employed with a software company developing a standardized e-commerce software product, three participants working for a digital publishing company, and two participants from the IT department of our university. Most participants associated themselves with the job role “developer,” while two participants currently work as team lead, and one participant as database specialist (see Table 2). To recruit participants for our study, we contacted the respective organizations, asked whether they would be interested in supporting our study and requested volunteers.

The professional experience as developer varied from less than a year to 25 years (mean = 9 years). All participants were native speakers of German. Prior to solving the tasks, participants were asked to rate their practical experience with REST APIs on a scale ranging from 1 (= no experience) to 5 (= much experience). Only one developer indicated to have no prior practical experience with REST APIs, but confirmed that he knew the concepts and main architectural assumptions. Mean rating was 3.4 (median = 3). None of the participants was familiar with the API selected for the test.

Note that a sample of 11 participants is too small to warrant statistical analyses, but as we stated above, testing specific hypotheses was not the goal of this study. We believe that the sample size is sufficiently large to collect an interesting range of observations as part of a qualitative analysis, and is also sufficiently large to reveal trends and relations in the quantitative data. We will return to this issue when we discuss potential threats to validity below.

Procedure

Prior to the test, statement of consent was obtained from each participant. As part of the instruction starting each session, participants were told that the test was conducted to observe how developers would approach tasks with an API unfamiliar to them. Participants received a brief explanation of the purpose of the API. Afterward, a questionnaire was administered to obtain participant data on age, gender, professional experience and experience with REST APIs. Participants then had to solve the programming tasks developed for the test. When finished with the tasks, a second questionnaire was provided which asked participants to rate the quality of the API documentation in general, and the structure of the documentation in particular, both on a scale ranging from 1 (=good) to 5 (=bad). Also using a 5-point scale, participants were to indicate whether they had problems with the fact that documentation was provided in English (1=no problems, 5=many problems). Moreover, participants were asked to comment freely on aspects they liked about the documentation, weaknesses they noticed as well as suggestions for improvement. Participants were debriefed, which ended the session.

For the test, five tasks were developed in collaboration with the API provider. The tasks involved various parts of the API, including creating shipping labels for specific shipments (such as return shipments, shipment with same-day delivery), requesting pickup of shipments at pre-defined time slots, as well as using special addresses for delivery, such as dedicated drop stations. For all tasks, the API documentation provided a correct solution. According to the API provider, the tasks were of medium difficulty.

Solving the tasks did not involve actual programming, but rather the configuration of specific HTTP requests that had to be sent to the API service. To arrive at a correct solution, participants had to determine the type of request (POST or GET), the endpoints to use in order to get access to the correct API resource, and parameters to be submitted with the request. In addition, some tasks also required participants to identify and manipulate payload information to be transmitted with the request. This payload information could be provided in two different notations: using URL syntax or using JSON (JavaScript Object Notation). To assemble and submit the requests and to inspect the response returned by the API, participants had access to a simple Command Line Client for transferring URL data (cURL) and to a REST client (Postman), which they were free to use depending on their preferences. Both options were offered since the code examples in the API documentation sometimes used JSON and sometimes cURL commands to illustrate payload information.

Table 2: List of participants and participant properties

Participant	Professional experience as developer (years)	Experience with REST-APIs on a scale from 1 ("no exp.") to 5 (much exp.)	Current Field	Role
P01	2	3	General IT Services	Developer
P02	10	5	General IT Services	Developer
P03	20	4	E-Commerce	Team lead
P04	5	2	E-Commerce	Developer
P05	6	5	E-Commerce	Developer
P06	2	4	E-Commerce	Developer
P07	8	5	Publishing	Developer
P08	10	3	Publishing	Developer
P09	25	2	Publishing	DB specialist
P10	11	3	E-Commerce	Team lead
P11	1	1	E-Commerce	Developer

Tasks were given to the participants in written form on paper. All tasks were formulated in German. Participants were free to use the API documentation at any point they wanted, but they were not specifically encouraged to do so. For later analysis, screencasts (including audio) were recorded. In addition, the participants' eye movements were registered using an REDm eye tracker (SensoMotoric Instruments, SMI) that was attached to the laptop monitor. The REDm eye tracker operates in head-free mode. Hence, participants were allowed to move their heads when working on the tasks. We expected overall data quality to vary greatly from session to session, as we did not want to restrain the seating position of participants too much in order to enable them to work as naturally as possible. For this reason, no attempts were made to analyze the eye tracking data in detail, for example by using dedicated areas of interest. However, we expected the eye tracking data still to be useful as a tool to support the qualitative analysis.

The test sessions were run on-site at conference rooms provided by the respective partner companies or the university. Two researchers were present in each session. One researcher took care of the technical setup, started the screencast and eye tracking recordings, and monitored data recording during the session. The other researcher instructed the participants, answered questions and took notes. Test sessions lasted between 40 to 70 minutes. Due to organizational restrictions, test sessions had to be terminated after 70 minutes regardless of whether participants had already completed all tasks.

Data analysis

Two lines of data analysis were pursued. On the one hand, screencast videos were coded directly. Coding the screencast videos provided the basis for obtaining quantitative data, such as the total time required to complete a task and the time participants used a specific part of the documentation. In addition, verbal protocols of participant activities during the test were prepared which served as the basis for the qualitative analysis.

Coding of screencast videos was done using the INTERACT video analysis software (Mangold International, version 15). Two different coding schemas were applied, "task" and "active element."

- The coding schema "task" was used to mark the beginning and the end of each task. Hence, five different values were used in this schema for tasks 1 to 5. The coding schema "task" provided the basis to gather general information on task success and time needed to execute the tasks.
- The coding schema "active element" was used to assess which parts of the API documentation participants accessed during task execution. The codes marked intervals in which a specific page or window was active on the screen. The value "Editor & Client" marked all intervals in which participants worked in a window outside the documentation, such as Postman or the Command Line Client. Six additional values were used that captured the content categories offered by the API documentation: Welcome page, Integration, Concepts, Samples, Recipes, and API reference.

The coding schemas were applied in separate coding runs. Each video was coded once and all coding was done by a single researcher. Prior to coding, we jointly defined criteria for assigning the individual code values.

The verbal protocols for the test sessions were prepared using standard spreadsheet software. Protocol notes were based on post-hoc inspection of the screencast videos. The protocols included observable activities such as performing a search, scrolling up and down a page, copying a code example, writing code or submitting a request to the API. Protocol entries were added in chronological order and assigned a time stamp. The session protocols were also used to store comments made by the test participants during or after the test, points at which participants requested help as well as any additional observations noted during the test.

Results: Quantitative Findings

The quantitative analyses addressed two goals. First, we looked at measures that inform us about overall task execution (success on tasks, time on tasks) and appreciation of the API documentation. The second goal was to examine the time participants spent in the different parts of the documentation.

Success on tasks

Of the 11 developers participating in our study, eight developers solved all five tasks in the 70-minute period available for each session. The other three developers solved only tasks 1–3. While P3 solved only 3 tasks due to technical reasons, participants P7 and P8 were not finished after 70 minutes due to the difficulties they experienced with the tasks.

Questionnaire data

After completing the tasks, participants were asked to rate the overall quality of the documentation and the structure of the documentation on a 5-point scale (1=very poor, 5=very good). The mean rating for the overall quality of the documentation was 2.9 (median: 3), the mean rating for the structure of the documentation was 2.5 (median: 2). We infer that satisfaction with the overall quality of the documentation and its structure was only moderate.

Participants were also asked to indicate whether it was difficult for them to read the documentation in English although their first language was German. Again, a scale was used, ranging from 1 (no problems) to 5 (many problems). Mean on this question was 1.7 (median: 1). This indicates that most participants felt comfortable with the language of the documentation.

Time on tasks

In order to assess efficiency of task execution, we calculated mean times per task across all participants and for each participant individually. This approach was chosen in order to take care of the fact that three developers only solved three instead of all five tasks.

The overall mean time per task was 695 seconds (SD = 282). However, as shown in Figure 2, there was considerable variation between participants. As the pattern suggests, participants can be roughly divided in two groups: the "fast performers" (P3-6, P10) and the "slow performers" (P1-2, P7-9, P11).

The time needed by the developers to execute the test tasks does not seem to depend on general developer experience. For example, P3 and P10 have both more than 10 years of professional experience and solved the tasks rather quickly. On the other hand, P2, P8 and P9 who are similarly experienced required significantly more time to complete the tasks.

A factor that seems to affect the efficiency of task execution is e-commerce experience. All participants that solved the tasks rather quickly currently work in the field of e-commerce, but only one of the participants from the "slow performers" group. Given that the API we selected for our test is designed to be used in Web shops and related e-commerce applications, the participants of the fast group probably took advantage of background knowledge that they could transfer to the test API. The observation that relevant domain knowledge is an important factor that influences how easy it is for developers to get into a new API fits well with conclusions on the role of domain-related background knowledge reached in in other studies such as Jeong et al. (2009), Ko and Riche (2011) and Meng et al. (2018).

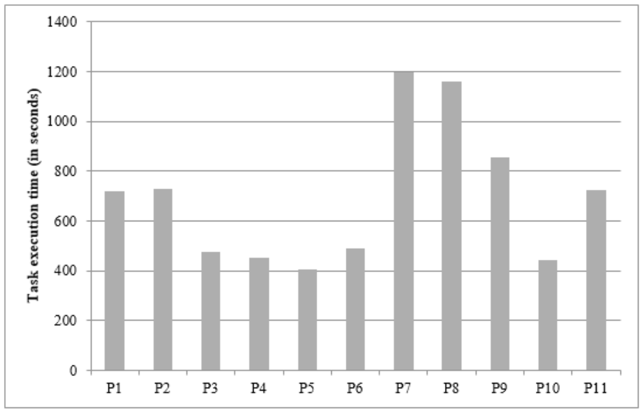


Figure 2: Mean time per task required by participants to execute the test tasks

Usage of documentation and content categories

On average, participants used API documentation about 49% of the time (Min: 31%, Max: 68%). A breakdown by participant revealed that there is only little individual variation, with the means for all but two participants ranging between 41% and 56%.

Table 3 shows the proportion of time participants spent in different content categories of the API documentation, such as "Concepts," "Samples" and "Recipes," and the proportion of time spent outside the documentation, for example in order to work on the assigned tasks using Postman or the Command Line Client.

The content category referred to most often is the API reference, followed by the Recipes page. When aggregating the times for Recipes and Samples, which both present code examples for basic use cases in a cook book-like fashion, both content categories together are head to head with the API reference and were active about 21% of the total time. On the other hand, the Concepts page is used as well, but less often compared to the API reference and the pages containing code examples. These findings show that the API reference is an important source of information, not only to solve specific programming issues when working with an API developer already have some experience with, but even in the initial stages of getting into a new API, in line with Meng et al. (2018). The findings also confirm the importance of code examples during the initial learning process, in line with results reported in McLellan et al. (1998), Shull, Lanubile, and Basili (2000), Nykaza et al. (2002) and Stylos, Faulring, Yang, and Myers (2009).

Table 3: Proportion of time spent outside the documentation (Editor & Command Line Client) and on different content categories within the documentation

Active window	% of total time
Welcome page	1.33
Concepts	7.91
Integrations	0.67
Samples	5.69
Recipes	14.99
API reference	18.35
Editor & Client	51.06

Usage of content categories by participant

As final part of the quantitative analysis, we calculated the proportion of time each participant spent on the different content categories such as "Concepts," "Samples" or "API reference," relative to the total time they spent on documentation-related page elements. To facilitate analysis, we reduced the number of categories. First, we aggregated the times for the content categories "Samples" and "Recipes," which are very similar in that they both contain code examples for basic use cases. Furthermore, times for "Welcome page" and "Integrations" were collapsed as well. As discussed above, both content categories do not contain information relevant to the test tasks and were therefore hardly ever used by the participants. Table 4 shows the mean values by participants for the resulting categories "API reference," "Concepts," "Examples" (Samples + Recipes) and "Other" (Welcome page + Integration).

As stated above, participants spent about 49% of the test time using documentation. However, as Table 4 reveals, there is considerable variation between participants with respect to the time they allocate to individual content categories. In particular, participants differ in whether they use information from the Concepts page or not. The values for P1, P2, P4, P5 and P6 range between 0% and 5.6%, whereas values for the other 6 participants range from 12.5% to 42.9%. Hence, while some participants, such as P1, P2 and P4, tend to completely ignore the Concepts page and rely more on API reference and content categories providing code examples, there is another group of participants, including for example P7, P8 and P10, that seems to use the Concepts page more extensively.

Whether the Concepts page is used does not seem to depend on overall developer experience or the availability of domain-related background knowledge. For example, P3, P7 and P8 are developers with 10 or more years of professional programming experience, and they all seem to resort to the Concepts page rather often. On the other hand, P2 and P9 are on a similar experience level, but belong to the group of participants that tends to ignore information on the Concepts page.

Table 4: Proportion of time (in %) spent on individual content categories of the API documentation by participant

Category	Participants										
	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11
API reference	43.5	50.0	45.5	68.3	24.8	61.9	28.7	7.8	34.9	48.6	21.3
Concepts	0.0	0.4	23.3	0.0	2.4	5.6	29.0	48.7	12.5	42.9	13.3
Examples	52.7	47.2	22.1	31.7	72.8	29.6	32.2	42.0	49.5	7.6	57.6
Other	3.7	2.5	9.2	0.0	0.0	3.0	10.1	1.5	3.1	1.0	7.8

Results: Qualitative Findings

Besides the quantitative findings discussed in the previous sections, we obtained a number of qualitative findings that were based on an analysis of the verbal protocols for the test sessions. The qualitative analysis focused on the general strategies adopted by the participants to solve the tasks and on barriers and obstacles that hindered efficient task completion.

Opportunistic versus systematic approach to programming

With respect to the general strategies adopted by the participants in order to solve the tasks, two groups emerged that seem to match the opportunistic and the systematic developer personas discussed in Clarke (2007). We found some developers (P2, P3, P9, P10) to develop the solutions for the test task in an exploratory fashion, which Clarke (2007) discusses as a characteristic feature of programmers taking an opportunistic approach. We note that these developers worked in a more intuitive manner and seemed to deliberately risk errors. They often tried solutions without double-checking in the documentation whether the solutions were correct. For example, P10 changed parameter values to values that seemed to match based on experience with similar problems, but he did not check in the documentation whether the values were actually correct or even existing. P2 inserted parameters that he had noticed at some point in the documentation before, but did not attempt to re-consult the relevant section of the documentation to make sure that the parameters were spelled correctly. In many cases, developers from this group did not follow the processes and suggestions described in the documentation.

We found that opportunistic developers in our test started the first task with some example code from the documentation which they then modified and extended. Once a task was completed, the piece of code that solved the task was used as starting point for the next task, which again was a potential source of error. Developers in this group worked in a highly task-driven manner, but also tried things that were not related to the task, but possibly helped them to build a broader understanding of the API in passing. For example, P9 submitted a request for a UPS service (United Parcel Service) which was not required by any of the tasks, simply in order to see what would happen.

We noted that developers which we assigned to the opportunistic group did not take time to get a general overview of the API before starting with the first task. They scrolled briefly through some pages of the documentation, checked the tools available and then started with the first task. Developers from the opportunistic group wanted fast and direct access to information. They did not systematically read larger sections of the documentation, but typically searched for a specific piece of information and then scanned the documentation in order to find it, sometimes in a very coarse-grained manner. For example, P2 jumped from page to page searching for a particular piece of information without some sort of search strategy becoming apparent. The exploratory, intuitive and active approach taken by the opportunistic developers bears many similarities with the exploratory and active approach described by John Carroll and colleagues in several studies observing how novice users learn to work with a computing system (see Carroll, 1990, for a detailed review).

In contrast to the opportunistic approach, another group of developers (P4, P5, P7, P8) seemed to follow a strategy that fits the systematic approach discussed by Clarke (2007). According to Clarke, systematic developers write code defensively and try to get a deeper understanding of a technology before using it. In our test, we note that these developers took some time to explore the API and to prepare the development environment before starting with the first task. Moreover, they took some time to get a general orientation. For example, P7 and P8 studied some sections in the documentation, then sent a GET request to the API and analyzed the response to check whether the request-response process worked as expected.

The systematic developers in our test started every task with a clean piece of code that was selected from the code examples in the documentation. They then used the examples on the Samples and Recipes pages as well as the API reference to modify the piece of code they started with in a systematic manner. Interestingly, they seemed to use a similar process to solve each task. Before starting a task, they would form hypotheses about the possible approach and (if necessary) clarify terms they did not fully understand. With respect to documentation usage, we also noted that they read sections of the documentation and code samples which were regarded relevant more carefully. In general, they attempted to follow the proposed processes and suggestions closely.

We also observed that the systematic developers apparently noticed parts of the documentation that were not directly relevant to the current task. However, when the information became relevant at a later point, they sometimes remembered that they had come across a section that was potentially relevant which they then tried to relocate.

Note that our classification of participants as opportunistic or systematic developer does not seem to predict general programming experience or the availability of domain-related background knowledge. For example, the developers we assume to have followed a systematic approach include experienced (P7, P8) as well as less experienced developers (P4, P5), and developers working within (P4, P5) or outside the e-commerce domain (P7, P8). Moreover, the strategy a developer follows does not seem to predict a tendency towards using information from the Concepts page in our test. We classified P4 and P5 as following a systematic approach, but both participants ignored the Concepts page almost completely. On the other hand, P3 and P10 which followed an opportunistic approach made extensive use of information presented on the Concepts page.

Barriers and obstacles

Based on the verbal protocols and comments made by participants after the test, a number of barriers and obstacles were identified that hindered efficient task completion.

- **Navigation.** Some developers mentioned that the API documentation used in our test lacked a consistent system of navigation aids, which sometimes led to the impression that the documentation was incomplete. A particular problem was that some pages, but not all, offered a side navigation including within-page links. We observed several times that developers wanted to go back to a certain piece of information which they had noticed in the context of another task, but had great difficulty doing so.

- **High-level structure of the API documentation.** Several problems were related to the high-level structure of the API documentation: the split of information in "Concepts," "Samples," "API reference" and so on. When searching for a particular piece of information, participants sometimes found it difficult to decide which content category to select. Typically, participants had no specific hypothesis whether the information they were looking for would be located in the Concepts, API reference, Samples section or somewhere else. The distinction between the type of content provided in "Samples" and "Recipes" was particularly unclear. Moreover, using "API reference" as label for a content category was mentioned to be misleading because this label suggests that the respective section contains the entire documentation.

- **Search.** The lack of a search function was identified as another barrier preventing more efficient task completion. Developers often wanted to use search when they were missing a particular piece of information, such as a term they did not know. Since the API documentation used in our test did not offer a dedicated search field, participants tried to use simple page search instead. Since the content was distributed over several pages, this strategy was often not successful.

- **Reuse of code examples.** Finally, we noted that participants developed their own solution starting from some sample code provided in the documentation, an observation in line with other reports in the literature (Maalej, Tjarks, Roehm, & Koschke, 2014; Kim, Bergman, Lau, & Notkin, 2005). Efficient reuse of code examples was sometimes hindered in our test due to the fact that the sample code contained placeholders referencing some other code example. Using placeholders eliminated redundancy across code examples, but made it impossible to simply reuse the code via copy and paste.

DISCUSSION AND IMPLICATIONS

Threats to Validity

When evaluating the results of our study, various aspects should be kept in mind that pose potential threats to validity. Several limitations result from the sample size of 11 developers tested. As already emphasized above, we believe that the sample size is sufficient to conduct a qualitative analysis and to reveal trends and relations in quantitative data. Note also that various other studies on API documentation that were performed in an industrial context relied on similar sample sizes, such as Jeong et al. (2009, N=8), Ko & Riche (2011, N=7), or Sillito & Begel (2013, N=10). Nevertheless, studies using larger samples should be carried out as follow-up in order to validate our findings.

A more general threat to validity relates to the observation method used in our study. If participants know they are observed, it is possible that they do not behave naturally anymore. For example, in our case this could mean that participants spend more time in the documentation than they would normally do. Again, follow-up studies using different methods are necessary to address this problem.

Note finally that our study focused on a single scenario for using API documentation (starting to work with a new API) and used a single API type (REST API). Whether our findings generalize to other scenarios, such as solving routine task once developers have become more experienced with an API or using the API documentation for specific activities such as bug fixing, and whether they also apply to other API types remains an open issue.

Implications for API Documentation Design

Despite these limitations, several general consequences for content and design of API documentation can be derived that may help to make documentation more efficient. We spell out these consequences in terms of guidelines designed to enable efficient access to relevant content, to facilitate initial entry into the API and to support different development strategies.

Enable efficient access to relevant content

Designing API documentation should include specific measures to facilitate effective access to content that is relevant to the task at hand. These measures have to respect the fact that developers differ with respect to the way they use documentation. We recommend the following guidelines to make relevant content more accessible:

- **Organize the content according to API functionality.** A first aspect concerns the high-level organization of the API documentation. From the results of our study, we conclude that API documentation should be structured according to categories that reflect the functionality or content domain of the API rather than using categories that signal the type of information provided. Instead of dividing documentation into "Samples," "Concepts," "API reference" and "Recipes," the API used in our study should be reorganized using categories such as "Shipment Handling," "Address Handling" and so on. If developers experience a problem while working with the API and turn to the API documentation to find information that solves the problem, they are likely to know the content domain of their problem (such as shipments or address handling), but it is more difficult for them to predict whether the information they are looking for is presented in the API reference, in a section dedicated to presenting code examples, or in a section discussing concepts. Note that this guideline can be viewed as an application of the principle of minimalist documentation according to which the components of the documentation should reflect task structure (van der Meij & Carroll, 1995).

- **Present conceptual information integrated with related tasks.** Another aspect relevant in this respect concerns the integration of conceptual information that developers need in order to use the API successfully. Confirming results reported in Meng et al. (2018), our study supports the conclusion that developers vary with respect to whether they use conceptual overviews that introduce important API concepts in a systematic way. While some developers use such offerings, others tend to ignore them. To reach both groups of developers, conceptual information should not be aggregated in a dedicated section or document that signals to focus on conceptual information. We recommend presenting conceptual information integrated with the description of tasks or usage scenarios where knowledge of these concepts is needed. To give an example from the API used in our test, information regarding the representation of a shipment should be introduced in the section describing how to create a new shipment, and specific features of a return shipment should be provided in the section describing how return shipments are handled.

- **Provide a transparent navigation and a powerful search function.** Our study emphasizes the need for providing appropriate means that enable efficient navigation through the API documentation. Navigation aids should enable developers to determine where in the documentation they currently are, and what the context of the current topic is. On several occasions, participants in our test remembered that they had already come across some piece of information that was now relevant, but often they were unable to return to the place in the documentation due to inconsistent and incomplete navigation options. Beyond transparent navigation, a powerful search function should be offered. If implementing a search function is not possible, e.g. for technical reasons, an alternative strategy is to facilitate simple search by presenting all information on a single page, instead of distributing the information across different (though possibly linked) pages (see Robillard & DeLine, 2011, for a similar proposal).

Facilitate initial entry into the API

The results of our study suggest that identifying appropriate entry points into the API and relating particular tasks or usage scenarios to specific elements of an API are key issues for successful API learning, confirming findings from earlier research based on interviews and questionnaires (see Robillard & DeLine, 2011, Meng et al., 2018). The following measures can be taken to support initial entry into a new API:

- **Provide clean and working code examples.** Special attention should to be paid to the code examples included with the API documentation. As we have observed, code examples are an important resource both for initial learning and when working towards the solution for a problem. Opportunistic developers heavily rely on code examples to understand how a specific API feature works. Moreover, both opportunistic and systematic developers use code examples as starting point when working toward the solution of a problem. In the API documentation used in our test, reusing code examples was hindered because the sample code often contained placeholders to avoid redundancy, which led to problems. Code example should be constructed very carefully. They should demonstrate the intended use of the API and they need to be complete and ready to be used via copy and paste.

- **Provide relevant background knowledge.** The quantitative results of our study provided hints that developers currently working in a company developing e-commerce software were able to solve the tasks more efficiently. It seems reasonable to assume that they took advantage of relevant background knowledge when starting to work with our test API which also addresses tasks related to e-commerce. As discussed above, other studies have confirmed the importance of background knowledge in the initial learning process as well (Jeong et al., 2009; Ko & Riche, 2011). API documentation should therefore provide background knowledge to facilitate entry into an API for developers without prior experience in the domain covered by the API. As with conceptual information in general, domain-related background knowledge should also be presented on-demand and integrated with the description of tasks and usage scenarios in which this knowledge becomes relevant.

- **Connect concepts to code.** A particular challenge for developers is to infer how certain concepts map to elements of the code. For example, the first task of our test required developers to create a label for a return shipment. Developers with background knowledge in e-commerce were more likely to form the hypothesis that a special parameter was used to mark a shipment as return shipment and not, for example, a dedicated resource. This hypothesis turned out to be correct, which greatly reduced efforts to identify the modifications that had to be made to the code. Hence, whenever introducing conceptual information, special efforts should be taken to signal to the developers how concepts are represented in the code by using appropriate code examples in which relevant elements are highlighted.

Support different development strategies

Our study suggests that API documentation has to respect the different strategies that developers adopt when approaching a new API. Both the content and the way the content is presented have to serve the needs of both opportunistic and systematic developers. The following guidelines can be used to support both types of developers:

- **Enable selective access to code.** Opportunistic developers focus on code. This reemphasizes the need to enrich API documentation with code examples that are complete and comprehensive. Moreover, proper design strategies should be used to clearly distinguish code examples from text, thereby making it easier for opportunistic developers to jump to relevant code examples directly. A design strategy that ensures such a clear distinction is to use a separate column for code examples that is aligned to the column containing the text blocks referring to the code examples, a technique which several popular API providers already use.

- **Signal text-to-code connections.** Specific efforts should be taken to support switches from text to code. Whenever the text refers to API elements such as methods or parameters, developers will want to identify those elements in the code example that accompanies the text. Therefore, signaling techniques should be applied, such as color coding, to highlight code elements both in the text and in the code example (Mautone & Mayer, 2001).

- **Provide important information redundantly.** The fact that opportunistic developers rely more heavily on code when learning a new API creates the risk that they miss sections in the API documentation that present critical pieces of conceptual information, including relevant domain-related background knowledge. This risk prevails even in case the conceptual information is presented in an integrated way, as part of describing tasks in which the concepts become relevant, because opportunistic developers are likely to skip the text and to focus on the code example that complements the text. This calls for an approach that presents critical pieces of conceptual information redundantly, e.g. integrated into the text describing how a certain task is handled with the API, but also (if possible) directly integrated into the source code by using code comments. Such an approach can help to make sure that opportunistic developers process this conceptual information even if they skip the text and focus on the code.

- **Enable fast use of the API.** Both opportunistic and systematic developers want to start using the API very soon after they begin dealing with a new API. In our study, the developers following an opportunistic strategy attempted to start with the first task almost immediately. In contrast, systematic developers took more time to get an overview of the API, but also got active early on, e.g. by trying sample API calls. Hence, for both types of developers, attempts should be made to enable fast use of the API, reemphasizing calls for an action-oriented approach to documentation advocated in the tradition of minimalism (van der Meij & Carroll, 1995). Possible strategies to enable fast use of the API are to provide code examples that can be used to generate sample API calls and to integrate try-out functions that developers can use to submit requests to the API directly and to inspect the response returned by the API.

CONCLUSIONS

Existing research on the information needs of software developers learning new APIs and the information resources they turn to relies to a large extent on inquisitive techniques such as interviews or questionnaires. The current study was undertaken to complement existing research with data collected through observing activities of developers while they attempt to solve first tasks with a new API. The results confirm and further substantiate findings from earlier research, thus contributing to a more solid empirical base on which design strategies to optimize API documentation can be based. Several guidelines proposing such design strategies have been discussed.

The results reported here and the design guidelines we proposed open several pathways for future research. A first step could be to examine whether API documentation that is optimized on the basis of our guidelines indeed leads to observable performance benefits when starting to work with a new API, such as shorter time on tasks and higher task accuracy. If such effects can be demonstrated, follow-up studies should address the contribution of individual design decisions and possible interactions of these design decisions with the different strategies developers can adopt. Moreover, research providing a more fine-grained analysis of the information units, such as text versus code examples, which developers attend to when using a certain section of the documentation also has the potential to further enhance our understanding of API documentation usage.

ENDNOTES

1. This research was supported by the German Federal Ministry of Education and Research, and anonymous reviewers agreed that the study's presentation of methods met expectations of ethical research design, clear communication with participants, and informed consent.

NOTES

Michael Meng, Stephanie Steinhardt, Andreas Schubert, Fachbereich Wirtschaftswissenschaften und Informationswissenschaften, Hochschule Merseburg.

Correspondence concerning this article should be addressed to Michael Meng, Hochschule Merseburg, Fachbereich Wirtschaftswissenschaften und Informationswissenschaften, Eberhard-Leibnitz-Straße 2, D-06217 Merseburg, Germany.

We would like to thank the two anonymous reviewers for helpful comments and suggestions for improvement.

The work reported here has been supported by the German Federal Ministry of Education and Research (BMBF), FHprofUnt, grant 13FH014PX4 to Michael Meng.

All images published with permission.

REFERENCES

Carroll, J. M. (1990). *The Nurnberg funnel: Designing minimalist instruction for practical computer skill.* Cambridge, MA: MIT Press.

Clarke, S. (2007). What is an end user software engineer? In *Dagstuhl Seminar Proceedings.* Schloss Dagstuhl-Leibniz-Zentrum für Informatik. Retrieved from http://drops.dagstuhl.de/opus/volltexte/2007/1080/, checked on 04/18/2018.

Inzunza, S., Juárez-Ramírez, R., & Jiménez, S. (2018). API documentation: A conceptual evaluation model. In *World Conference on Information Systems and Technologies*, 229–239. Springer, Cham.

Jeong, S. Y., Xie, Y., Beaton, J., Myers, B. A., Stylos, J., Ehret, R., & Busse, D. K. (2009). Improving documentation for eSOA APIs through user studies. In *International Symposium on End User Development,* 86–105. Berlin: Springer.

Johnson, T. (n.d.): What is a REST API? [blog post]. Retrieved from https://idratherbewriting.com/learnapidoc/docapis_what_is_a_rest_api.html, checked on 10/01/2018.

Kim, M., Bergman, L., Lau, T., & Notkin, D. (2004). An ethnographic study of copy and paste programming practices in OOPL. In *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium* , 83–92. IEEE.

Ko, A. J., & Riche, Y. (2011). The role of conceptual knowledge in API usability. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium*, 173–176. IEEE.

Lethbridge, T. C., Sim, S. E., & Singer, J. (2005). Studying software engineers: Data collection techniques for software field studies. *Empirical Software Engineering, 10*(3), 311–341.

Lethbridge, T. C., Singer, J., & Forward, A. (2003). How software engineers use documentation: The state of the practice. *IEEE Software, 20*(6), 35–39.

Lutters, W. G., & Seaman, C. B. (2007). Revealing actual documentation usage in software maintenance through war stories. *Information and Software Technology, 49*(6), 576–587.

Maalej, W., Tjarks, R., Roehm, T., & Koschke, R. (2014). On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM), 23*(4), 31.

Mautone, P. D., & Mayer, R. E. (2001). Signaling as a cognitive guide in multimedia learning. *Journal of Educational Psychology, 93*(2), 377.

McLellan, S. G., Roesler, A.W., Tempest, J.T., & Spinuzzi, C.I. (1998). Building more usable APIs. *IEEE Software, 15*(3), 78–86.

Meng, M., Steinhardt, S., & Schubert, A. (2018). Application Programming Interface Documentation: What Do Software Developers Want? *Journal of Technical Writing and Communication, 48*(3), 295–330.

Mihaly, F. (2011). Writing helpful API documentation [Blog post]. Retrieved from http://theamiableapi.com/2011/11/01/api-design-best-practice-write-helpful-documentation

Myers, B. A., & Stylos, J. (2016). Improving API usability. *Communications of the ACM, 59*(6), 62–69.

Nykaza, J., Messinger, R., Boehme, F., Norman, C. L., Mace, M., & Gordon, M. (2002). What programmers really want: results of a needs assessment for SDK documentation. In *Proceedings of the 20th annual international conference on Computer documentation,* 133–141. ACM.

Robillard, M. P. (2009). What makes APIs hard to learn? Answers from developers. *IEEE Software, 26*(6), 27–34.

Robillard, M. P., & DeLine, R. (2011). A field study of API learning obstacles. *Empirical Software Engineering, 16*(6), 703–732.

Shull, F., Lanubile, F., & Basili, V. R. (2000). Investigating reading techniques for object-oriented framework learning. *IEEE Transactions on Software Engineering, 26*(11), 1101–1118.

Sillito, J., & Begel, A. (2013). App-directed learning: An exploratory study. In *Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop,* 81–84. IEEE.

Stylos, J. (2009). Making APIs more usable with improved API design, documentation and tools (Doctoral dissertation, Carnegie Mellon University). Retrieved from http://www.cs.cmu.edu/~NatProg/papers/

Stylos, J., & Clarke, S. (2007). Usability implications of requiring parameters in objects' constructors. In *Software Engineering, 2007. ICSE 2007. 29th International Conference,* 529–539. IEEE.

Stylos, J., Faulring, A., Yang, Z., & Myers, B. A. (2009). Improving API documentation using API usage information. In *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium,* 119–126. IEEE.

Uddin, G., & Robillard, M. P. (2015). How API documentation fails. *IEEE Software, 32*(4), 68–75.

van der Meij, H., & Carroll, J. M. (1995). Principles and heuristics for designing minimalist instruction. *Technical Communication, 42*(2), 243–261.

Watson, R., Stamnes, M., Jeannot-Schroeder, J., & Spyridakis, J. H. (2013). API documentation and software community values: a survey of open-source API documentation. In *Proceedings of the 31st ACM International Conference on Design of Communication,* 165–174. ACM.

ABOUT THE AUTHORS

Michael Meng is professor of applied linguistics at Merseburg University of Applied Sciences where he teaches courses on text analysis, text production and research design. Before joining university, he worked for 12 years as a technical communicator for an international software company. His research focuses on using empirical methods to study the effects of linguistic and design variables on comprehension and the usability of information products in technical communication.

Stephanie Steinhardt earned a diploma in Technical Communication from Merseburg University of Applied Sciences. She teaches information design and works as a freelance consultant and e-learning specialist for industry clients.

Andreas Schubert received a Master's degree in Technical Communication from Merseburg University of Applied Sciences and worked as research assistant in a project on optimizing API documentation. He currently holds an industry position as technical communicator.