STAMP

Deliverable D3.2

Initial Prototype Of a Log Optimization Tool

Project Number	:	731529
Project Title	:	STAMP
Deliverable Type	:	Report

Deliverable Number	:	D3.2
Title of Deliverable	:	Initial Prototype Of a Log Optimization Tool
Dissemination Level	:	Private
Version	:	1.0
Latest version	:	https://github.com/STAMP-project/docs-forum/tree/master/docs
Contractual Delivery Date	:	M12 November, 30 2017
Contributing WPs	:	WP 3
Editor(s)	:	Andy Zaidman, TU Delft
Author(s)	:	Arie van Deursen, TU Delft Pooria Derakhshanfar, TU Delft Xavier Devroey, TU Delft Vincent Massol, XWiki
Reviewer(s)	:	Benoit Baudry, KTH Caroline Landry, INRIA Yosu Gorronogoitia, ATOS

Abstract

This deliverable reports on the progress towards log data optimization for debugging. We address log data from web applications and the integration challenges that come with complex Web APIs. Such APIs can generate millions of log messages a day. We filter error logs and analyze the root causes of the underlying errors. Starting from over 2 million error logs, we identify 69 fault cases and 11 general underlying causes. Our findings pave the way for rigorous testing and analysis of web api integration problems, both at run time (monitoring log data) as well as at development time. We address debugging by focusing on automated fault localization techniques. Starting from an analysis published in 2011, we investigate the core impediments towards using automated fault localization techniques in practice. We draw lessons concerning practically relevant ways to measure the quality of automated debugging techniques. Furthermore, we observe that result comprehension is key, i.e. assisting the developer in making sense of the fault localization output. The underlying sub-deliverables appeared separately as:

- *An Exploratory Study on Faults in Web API Integration in a Large-Scale Payment Company. Joop Aué, Maurício Aniche, Maikel Lobbezoo, Arie van Deursen. Technical Report, Delft University of Technology.*
- *Revisiting the Practical Use of Automated Software Fault Localization Techniques. / Ang, Aaron; Perez, Alexandre; van Deursen, Arie; Abreu, Rui. Proceedings 8th International Workshop on Program Debugging (IWPD): International Symposium on Software Reliability Engineering Workshops. IEEE, 2017. p. 175-182.*

Chapter 1

Introduction

Service-oriented architectures are now more popular than ever. Companies and organizations increasingly offer their services through Web Application Programming Interfaces (Web APIs). Web APIs enable client developers to access third party services and data sources, and use them as building blocks for developing applications, e.g., Airbnb utilizes Google's Calendar API to automatically insert bookings into the renter's calendar, and Google Maps consumes Uber's Ride Request API to offer Uber's services as means of transportation in their maps application.

The capabilities and complexity of Web APIs inevitably differ from service to service. Retrieving a list of followers for a user on Twitter requires a GET request including a single parameter, and posting a Twitter status update using the Twitter API takes a single parameter POST request. As the complexity of the actions increases, so do the possibilities of failure. For instance, Github's Repo Merging API supports merging branches in a repository. In addition to the intended merge, other possible outcomes are a *merge conflict*, a *missing branch* error or a *nothing to merge* response.

Adyen[1], a multi-tenant Software as a Service (SaaS) platform that processes payments, offers an authorization request used to initiate a payment from a shopper, which takes up to 35 parameters. Multiple types of shopper interaction, and optional fields to optimize fraud detection and improve shopper experience lead to numerous failure scenarios. In addition to the happy path, the method can return at least 34 unique error messages to inform the API consumer that something has gone wrong. It seems that the more complex a Web API is, the more errors its facade can produce and hence, the larger the potential impact of these errors.

To make error handling for client developers easier, practioners have written a variety of best practice guides and blogposts on API design [15] [25] [17] [22]. Apigee [1], a platform offering API tools and services for developers and enterprises, discusses error handling in multiple ebooks. Apigee's error handling best practices focus on which HTTP status codes to use [6] and suggest to return detailed error messages for users and developers [7]. However, to our knowledge no research has been conducted on what type of errors occur when using Web APIs in practice and what causes them to happen. Not only can this knowledge complement existing API design best practices, it can help improve API documentation and help developers understand the common integration pitfalls.

The potential impact of API errors on API consumer applications is enormous. At the same time an understanding of API errors that occur in practice and their impact is missing. This gap of knowledge motivated us to investigate the domain of Web API errors. To this aim, we study the API error responses returned by Adyen webservices which handle millions of API requests on a daily basis. We analyze 2.43 million error responses, which we extract from the platform's production logs, discover the underlying faults, and group them into high-level causes. In addition, we survey API consumers about their perceptions on the impact and how often they observe such cases in their APIs, as well as practices and challenges they currently face when it comes to integrating to Web APIs. Finally,

[1] http://www.adyen.com

we provide API developers and consumers with recommendations that would help in reducing the number of existing integration errors.

Our results show that 1. faults in API integration can be grouped in 11 causes: invalid user input, missing user input, expired request data, invalid request data, missing request data, insufficient permissions, double processing, configuration, missing server data, internal and third party. Each cause can be contributed to one of the four API integration stakeholders: end user, API consumer, API provider, and third parties; 2. most faults can be attributed to the invalid or missing request data, and most API consumers seem to be impacted by faults caused by invalid request data and third party integration; and 3. API consumers most often use official API documentation to implement an API correctly, followed by code examples. The challenges of preventing problems from occurring are the lack of implementation details, insufficient guidance on certain aspects of the integration and insights in problems and changes, insufficient understanding of the impact of problems, missing guidance on how to recover and a lack of details on the origin of errors.

The main contributions of this work are as follows:

1. A classification of API faults, resulting in 11 causes of API faults, based on 2.43 million API error responses of a large industrial multi-tenant SaaS platform (Section 4.0.1).

2. An empirical understanding of the prevalence of API fault types in terms of the number of errors and impacted API consumers (Section 4.0.2).

3. An initial understanding of the impact of each cause as experienced by API consumers as well as their observations on current challenges during API integration (Section 4.0.3).

4. A set of recommendations for API providers and API consumers to reduce the impact of API related faults (Section 5.0.1).

Chapter 2

Background: Understanding the Web API environment

A web API integration can involve up to four different stakeholders, that all influence the interaction between the API and its consumer. As a result, each of these stakeholders can cause the API to return with an error that possibly leads to a failure in the consumer's application. In this section, we give an overview of the API environment, the parties involved and API error related terminology, to clarify the differences and nuances that could lead to confusion.

In a typical integration with web API, two stakeholders, or parties, are involved. On one side the *API provider*, offering their services by exposing an API, and one the other side the *API consumer*, utilizing the services offered by communicating with the API. The API provider may optionally itself be connected to *third party* services behind the scenes in order to provide the intended functionality. For instance, an API offering stock data may itself be connected to different stock exchanges to obtain the latest stock prices. We deliberately refer to API consumer, instead of API user, to leave room for the term *end user*. The API consumer may optionally provide an application used by its customers, who indirectly make use of the API's services. These end users supply information, while using the application, that is used as request data for the API. For example, Google Maps users are given the option to choose Uber as means of transportation when searching for directions. Indirectly they are supplying input to the Uber API, which locates nearby drivers and estimates the cost for the trip.

In the system under study, the API provider, Adyen, provides its payment services through its API. The API consumer is the merchant processing payments using the Adyen solution (*e.g.*, a store). The end users are shoppers, who use the merchant's services to buy goods or services. Finally, third parties connected to Adyen typically include banks, schemes, and issuers.

Each of the 4 stakeholders can, by introducing a fault, potentially cause an erroneous response to be returned by the API, which if unexpected can result in problems for the application. We provide an overview of the integration environment containing the involved stakeholders and the relation between faults, errors and failure in Figure 2.1. In case the API is indirectly in use by customers of the API consumer, the end user can, by supplying invalid information, cause the API to return an error. The API consumer, on the other hand, may have implemented the API incorrectly, which can result in requests with a specific input to be rejected by the API. The API provider may have a bug in its system, which could, for instance, cause requests to fail on a specific input. Lastly, when a third party service fails, the API provider may decide to return an error to the API consumer.

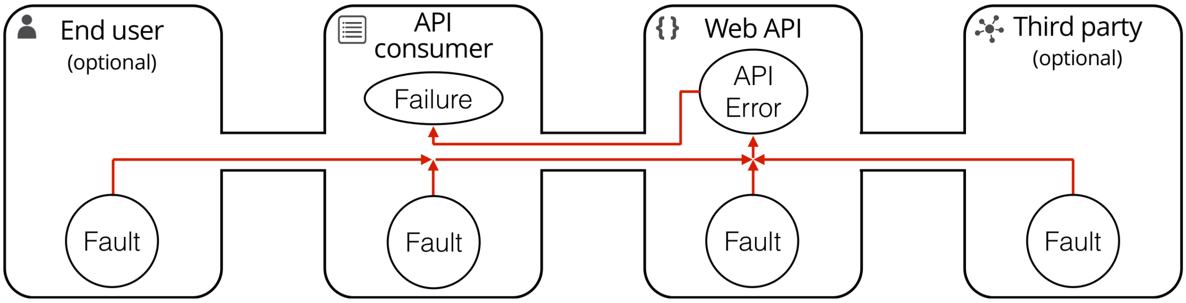

Figure 2.1: The API integration environment containing the involved stakeholders and the relation between faults, errors and failure.

Chapter 3

Research Methodology

The goal of this study is to understand the faults that occur in API integration that can potentially result in production problems for consumers of an API. To this aim, we propose the following research questions:

RQ$_1$. What type of faults are impacting API consumers? Translating the understanding of faults in one API integration to API integrations in general is difficult, because every API has its own use cases, specific methods and corresponding errors. To enable generalization of the results we identify general *causes* of faults that can occur in an API integration.

RQ$_2$. What is the prevalence of these fault types, and how many API consumers are impacted by them? The answer to this question provides insights into the frequency of each type of fault and the impact in terms of number of API consumers. API designers can leverage the information of what type of fault impacts the user the most when designing an API. This would allow to lower the probability that these faults take place and result into problems. In addition, the knowledge will help to identify what aspects of integration are more difficult to get right, which therefore require more attention in terms of, for instance, documentation.

RQ$_3$. What are the current practices and challenges to avoid and reduce the impact of problems caused by faults in API integration? Understanding what type of faults occur in API integrations, how often these faults occur and their impact is not enough to determine how the impact of production problems can be reduced. An understanding is needed of the current practices used to avoid and reduce problems, before recommendations for improvements can be made. Similarly, an understanding of the current challenges faced by API consumers is needed to identify areas of improvement.

To answer the RQs, we collect and analyze data from two different sources: over 2 million API error response logs from Adyen's production services, and a survey with API consumers. We use this first batch of data to answer the types of fault in Web API integration as well as their prevalence. After, we challenge our findings by means of a survey with 40 API consumers. In the following, we detail each data collection mechanism.

3.0.1 Analysis of the API error response logs

The data extraction approach can be depicted into four steps: (**1**) We extract API error responses from production log data to obtain unique API errors, (**2**) we manually analyze each unique error message in context of the web service and API method to identify unique faults, (**3**) the time span covered by the data set is verified to cover enough data, and (**4**) we derive a set of causes that explain the API integration failures.

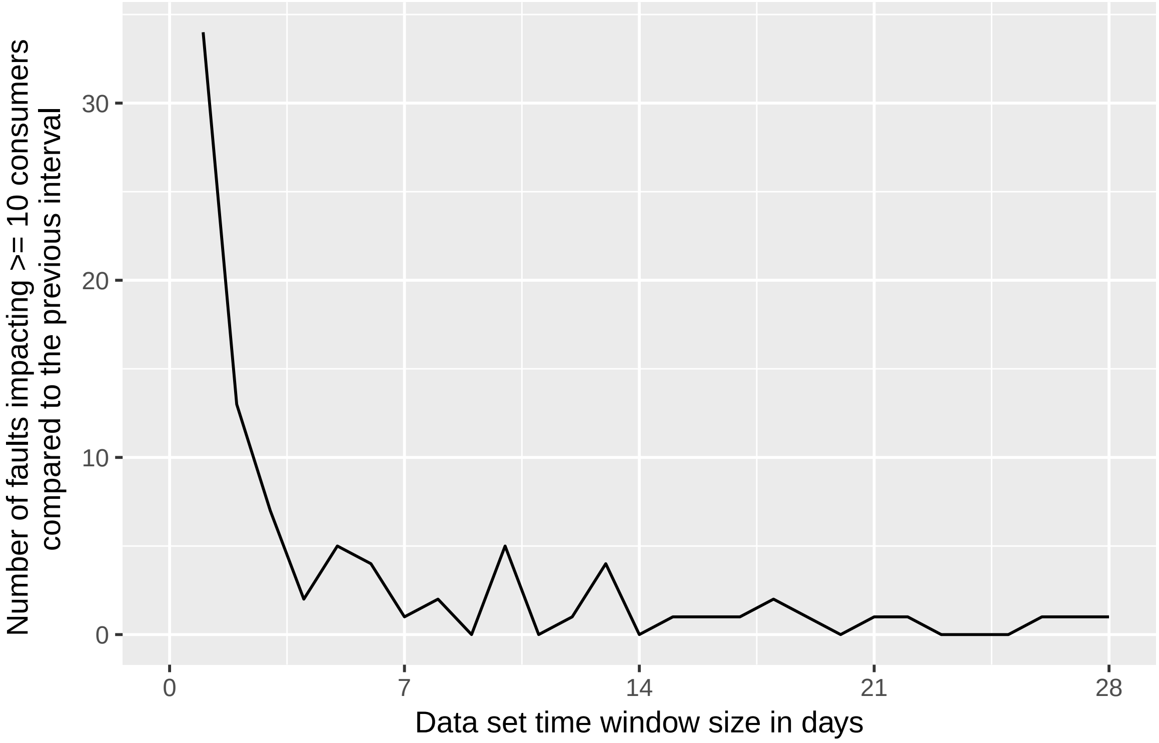

Figure 3.1: The number of newly identified faults for each interval for compared to the previous interval.

The logs of the system under study contain information about everything that happened in the production environment. Among the logs are the API requests and corresponding responses. Using domain knowledge of the system, we identified queries to capture the erroneous API response log messages from the entire set of log messages. The data set we use contains 28 days of data and 2.66 million API error responses.

To make sure 28 days would provide enough data for the analysis, we measured the amount of new information each new day was bringing to the dataset. We find that, after 14 days of data, at most 2 new faults are identified per day with the number decreasing as more days pass. For this reason we conclude that the most common and therefore impacting faults in our data set are discovered within 14 days and therefore consider the 28 day data set to cover a sufficient time span. In Figure 3.1, we show the amount of new information that each day aggregates to the dataset.

As the same error can happen more than once, we identify the unique errors (*i.e.*, unique erroneous API response messages) that have happened in the dataset. However, the error messages alone are not enough to explain the faults. There are messages that indicate one fault, but in practice have a different meaning. For instance, “Unsupported currency specified”, appears to be a configuration mistake or an invalid input fault. However, this specific error is caused by a missing value. Other messages, e.g., “Internal error”, are too ambiguous to categorize in the first place. We add context to the unique error messages to reduce the number of unique messages with multiple explanations. To do so, we use the corresponding API *method* and *web service*. For instance, the message “Invalid amount specified” has multiple explanations that depend on the API method used. Adding context allows for more granularity during analysis. We end up with 363 different errors to analyze.

In practice, we observed that the analysis of these errors are intensive manual tasks, *i.e.*, for each error, we had to comprehend the failure, inspect the source code, and talk to the developers of the system. Thus, we consider errors that impacted 10 API consumers or more. Following this approach, we analyzed 89 of the 363 errors, which covers approximately 2.44 (91.3%) of the 2.66 million erroneous API responses. After analyzing each of them, we found 69 different explanations for these 89 errors (we refer to them as fault cases *[FC01..FC69]* and they can be found in our online appendix [8]), which we use as an input to determine the high-level causes.

Identifying the causes and assigning each of them to a fault is an iterative process, based on a detailed qualitative analysis of the fault cases. Investigating a subset of faults gives the intuition needed to define initial causes, which can be assigned to most of the annotated faults. During further analysis, if a fault does not fit into one of the existing causes, we define a new cause. A cause that is too generic may have to be split up into two or more causes, while a cause that is too specific may be joined with another cause. Categorization can therefore not be described by a predefined set of steps, but is guided by our understanding of the problem domain, and the actual analysis of the cases at hand. After assigning a cause to each fault, we iterate once more over all faults to check that all causes are accurate. Following this procedure we obtain a set of causes that describe faults in API integration in general.

To verify the accuracy of the categorization, we asked a specialist in API integration from Adyen to check a subset of the cause assignments. In case the specialist does not agree with the cause, the difference is discussed until agreement is reached. Using this approach, we verify that the causes are understandable and reduce the possibilities of mistakes. At the end, we derived 11 causes that explain faults in Web API integration.

In practice, we observed that, due to unclear error messages, about half of the faults originating from error messages that have two or more different causes. Propagating these causes to the entire 2.43 error messages would require analyzing each case in isolation. Thus, we are not able to perfectly estimate the prevalence of each cause; rather, we provide lower and upper bounds.

3.0.2 Survey with API consumers

We challenge the 11 derived causes by testing them outside the scope of the system under study. To this end, we survey API consumers that have experience with problems related to API integration. We ask them for each cause whether they have experienced a problem. Furthermore, we allow them to name additional causes to capture the ones we may have missed.

The survey contains five main parts: 1) the process of integration to understand how the API consumer obtains the knowledge to implement the API correctly, 2) API fault prevention to identify areas of improvements for the API provider, which suggest current challenges faced by the API consumer, 3) API error handling practices employed by API consumers and the challenges they face while doing so, 4) the fault detection mechanisms in place, and the areas the API consumer seeks to improve, and 5) the API consumer's idea of what causes faults and related problems to occur.

The target audience of this survey is developers that have experience in API integration for an application that is used in production. To understand the participants in terms of experience we ask them about their years of experience in both software development and Web API integration. We exclude participants without API integration experience as they do not fit our target audience.

In order to learn the most about API problems we ask the participant to consider the API they worked with, which they consider to be the most complex and give them suggestions as to what kind of metrics they can use to determine this. These include: offered features and functionality, number of required or optional parameters and the number of possible error scenarios.

In addition, we ask them to answer the questions based on their experience, instead of what they believe is happening or is the ideal situation. For instance, we would like the participants to report their experience of what causes API errors to occur, and not what they think causes these errors in general.

We pre-tested the survey with five participants to make sure the questions are understandable and to remove possible ambiguities. The participants were asked to read the questions aloud as well as what they were thinking when answering the questions. This helped us understand the participants' reasoning and identify problematic situations.

We posted the survey on the following programming communities: Code Ranch's [2] *Web Services* forum, Hackernews [3] and Reddit's [4] subreddits *programming* (815,000 subscribers), *Webdev* (160,000 subscribers), *API* (600 subscribers) and *WebAPIs* (235 subscribers). Although the number of subscriber for the first two subreddits is high, the topics are very general, so the expected number of responses from these is relatively low compared to the more specific forums.

To increase the response rate, we additionally resorted to non-programming specific media and personal contacts. The survey was shared with the general public on Twitter by two colleagues; one with primarily academic followers (2500) and the other with a mix of academics and practitioners (4600 followers). In total, the posts were retweeted 25 times. On LinkedIn our post was viewed approximately 1000 times and was shared by two connections. Three companies in industry were contacted via personal contacts of which one was Adyen, the company under study. Lastly, the first author reached out to personal contacts that match the target audience.

The survey was online for three weeks. A total of 40 qualified participants answered at least 1 question. We decided to consider partial responses in the results as well, but only for those participants that answered questions that are not background related; 11 out of the 40 participantes provided partial responses. The survey can be found in our online appendix [8].

3.0.3 Characterization of Survey Participants

On average, the respondents have over 10 years of development experience and 5 years of API integration experience. 13 of the developers were individually responsible for the API integration and 27 worked in a team of two or more developers.

95% of the respondents answered the survey based on an application that they worked on in a professional setting. The remaining 5% used an API in a hobby project, which however was used in production. According to 28% of the participants, the API they consume can be considered complex; 22% of them consider the API not complex, and 50% were neutral. 13 APIs used by the participants were *data management* related. For instance, providing data about products and orders, and managing financial and account data. *Payment* related APIs were considered 6 times. Even though many respondents are from Adyen, a payments company, only 3 of the respondents considered an payment related API. Other APIs that the participants integrated with are used for *authentication*, *ecommerce*, *project management*, *geocoding* and *notifications*, such as SMS services.

Chapter 4

Results

4.0.1 RQ$_1$. What type of faults are impacting API consumers?

In Table 4.1, we show the 11 derived causes. Two causes, related to user input, can be contributed to the end user, who is also responsible for *expired request data* faults. The others are mostly caused by the API consumer, and the API provider and the third party stakeholder each match one cause. In the following, we detail each of the causes.

Invalid user input *Invalid user input* regards requests that fail because an end user supplied input that cannot be used to complete the intended action. The invalid information is forwarded by the API consumer to the API. There are multiple types of input invalidity. We observed inputs that 1. do not match a pre-defined list of expected values or ranges, *e.g.*, invalid country code, and month should be between 1 and 12 (FC-06, FC-22, FC-23, FC-35), 2. do not match the expected type or format, *e.g.*, year should be an integer value (FC-15, FC-33, FC-34, FC-37, FC-45, FC-63, FC-64), 3. do not contain the expected length, *e.g.*, CVC code should have three digits (FC-16, FC-36). In practice, *Invalid user input* can be caused by a user that is not aware that certain input is not allowed.

Missing user input *Missing user input* is strongly related to *invalid user input*. In this case however, the end user neglects to fill in required information, which causes the subsequent request to fail. We decided to distinguish between missing and *invalid user input*, because the nature of the mistake is different. An end user that does not fill out a field either forgets to or is unaware that the field is required. This is different from invalid input where the user supplies incorrect information. We observed cases such as missing payment details, such as bank information, card holder name, CVC, expiry month, IBAN, credit card (FC-03, FC-09, FC-17, FC-21, FC-38, FC-61), as well as billing information, such as city, state, country, and street of the buyer (FC-04, FC-07, FC-05, FC-08).

Expired request data *Expired request data* faults occur when the request is not handled in time. This occurs when the request contains a timestamp that defines a timeframe that the server has to handle the request (FC-19, FC-67). In Adyen's case, a timestamp is generated when a shopper starts a transaction. When the request comes in, the system checks whether the start of the transaction is not too far into the past. If a shopper takes too much time an *expired request data* fault translates to an error being returned.

Invalid request data *Invalid request data* faults are caused by input that cannot be handled by the API. There is a multitude of different reasons for such a fault to occur. We observe rounding problems, *e.g.*, passing value 72.20 instead of 72.21 (FC-01, FC-68), functionality not available for

Stakeholder	Cause	Explanation	Fault Cases (N=69)
End user	Invalid user input	A fault introduced by invalid input by the end user of the application	FC6, FC15, FC16, FC19, FC2 FC23, FC33, FC34, FC35, FC3 FC37, FC45, FC63, FC64
End user	Missing user input	A fault introduced by missing input by the end user of the application	FC3, FC4, FC5, FC7, FC8, FC FC17, FC21, FC38, FC61
End user	Expired request data	The input data was no longer valid at the moment of processing	FC-19, FC67
API consumer	Invalid request data	A fault introduced by invalid data caused by the API consumer	FC1, FC13, FC14, FC20, FC2 FC30, FC31, FC32, FC41, FC4 FC44, FC47, FC51, FC54, FC5 FC66, FC68
API consumer	Missing request data	A fault introduced by missing data caused by the API consumer	FC10, FC25, FC27, FC28, FC3 FC46, FC48, FC52, FC58, FC5 FC60, FC65
API consumer	Insufficient permissions	Not enough rights to perform the intended request	FC40, FC49, FC50
API consumer	Double processing	The request was already processed by the API	FC11, FC18, FC69
API consumer	Configuration	A fault caused by missing/incorrect API settings	FC26, FC53
API consumer	Missing server data	The API does not have the requested resource	FC56, FC57
API provider	Internal	An internal fault caused by the API	FC2, FC12, FC62
Third party	Third party	A fault caused by a third party	FC24, FC42

Table 4.1: The 11 causes of API faults, their related stakeholder, and the quantity of fault cases (FC) assigned to the cause.

that combination, *e.g.*, chosen bank does not support recurring payments (FC-13, FC-54, FC-55, FC-66), invalid information, *e.g.*, merchant does not exist (FC-41, FC-43, FC-47, FC-51), bad encoding, *e.g.*, wrong URL encoding (FC-14), bad format or data outside a list of acceptable values, *e.g.*, amount should be greater than zero (FC-20, FC-29, FC-30, FC-31, FC-32), and using test data into production environment, *e.g.*, use testing payment reference in production (FC-44). This is similar to the mistake made by the end user causing an *invalid user input* fault, however, in this case, caused by the API consumer.

Missing request data *Missing request data* faults are similar to *invalid request data* faults. However, in this case the API consumer neglects to send in information that is required for the intended action. Also in this case we decided to distinguish between invalid and *missing request data*. We reason that the mistake of not supplying required information is of a different nature than making a mistake by supplying incorrect input. We observe missing cryptographic data (FC-10, FC-25), and different business related data (FC-27, FC-28, FC-39, FC-46, FC-48, FC-52, FC-58, FC-59, FC-60, FC-65).

Insufficient permissions *Insufficient permissions* faults are caused by API consumers that attempt to use an endpoint or make use of a resource, while they are not allowed to do so. We find users making this mistake because they attempt to use the production services, while they are not yet through the process of obtaining the permissions for this (FC-40, FC-50), or waiting for the service to be properly configured (FC-49). We also see API consumers still interacting with the API, while their contract has been ended and therefore their permissions have been revoked.

Double processing *Double processing* faults are caused by API consumers that send in a request more than once. The API under study is designed to be idempotent; sending in the same call repeatedly will produce the same result. *Double processing* faults should therefore not be possible. However, in case of attempting to repeatedly delete the same remote object, a *double processing* faults occurs because the reference to this object can no longer be found, *e.g.*, contracts (FC-11) and payment related objects (FC-18, FC-69).

Configuration *Configuration* faults are caused by incorrect configuration of the API consumer account. The API consumer assumes certain that functionality is set up for their account, however in reality it is configured incorrectly or not set up at all, *e.g.*, configuration for installments (FC-26) or specific payment methods (FC-53).

Missing server data *Missing server data* faults happen when the API consumer asks for data that used to exist, but does not anymore, as it was updated, removed or disabled in the past (FC-56, FC-57).

Internal *Internal* faults occur when the API provider is unable to handle an incoming request for an unanticipated reason. This can be because of a bug, due to the system being unable to handle a specific input or unexpected API consumer interaction. Data related replication issues between internal components in the system can result in new data resources to not be available immediately on all servers in a distributed API server architecture (FC-02, FC-12). We also observe internal failures in cryptographic routines (FC-62).

Third party A *third party* fault can result in an API error when the API consumer makes a request that involves the API provider to make use of a third party (*e.g.*, a bank), which does not respond or returns an error (FC-24, FC-42). In this case the request failed and the consumer is notified by means of an error.

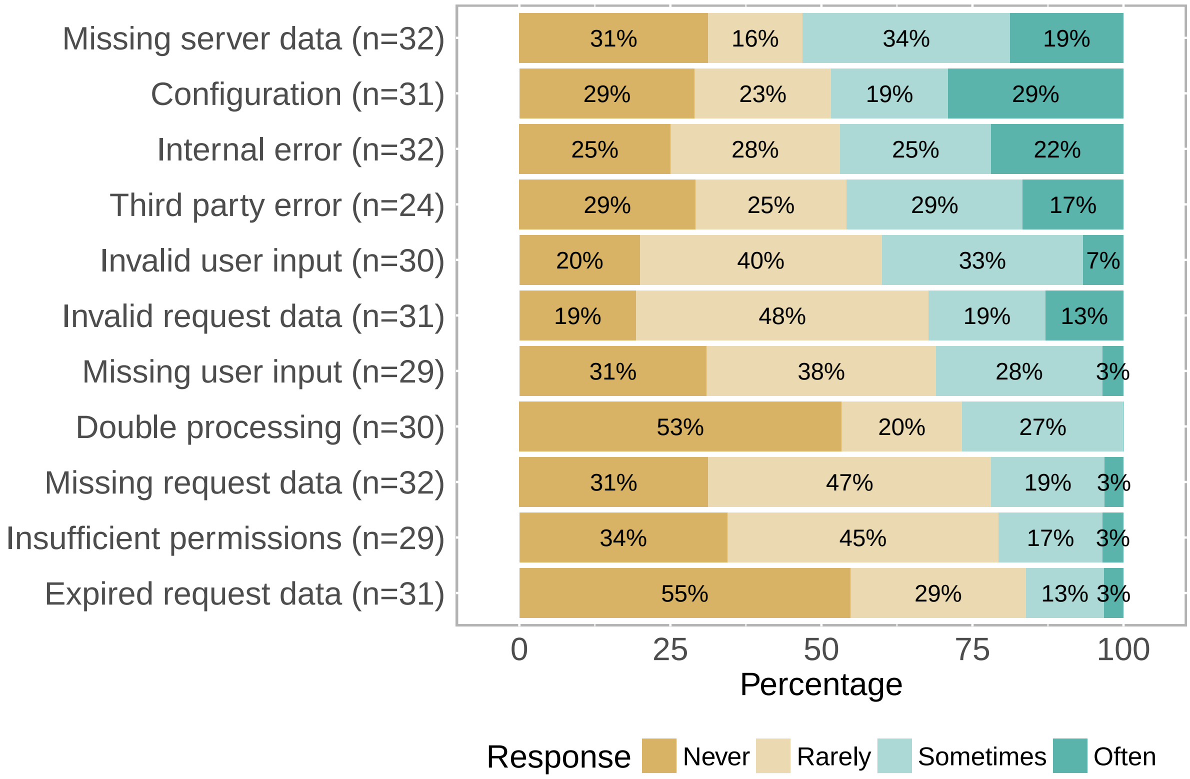

Figure 4.1: How often API related problems are experienced by the survey participants.

In Figure 4.1, we show how often survey participants experienced production problems with the API due to each of the 11 causes. *Missing server data* and *configuration* related problems were experienced relatively more often than other problems for the participants. Problems caused by the API provider and third parties, *internal* and *third party* fault related problems respectively, are relatively experienced more than other problems caused by the API consumer or end user. It is to be noted that for *third party* faults 10 out of 34 respondents did not know whether these problems occurred or regarded the cause as not applicable. *Missing request data* and *missing user input* faults both result into less problems than *invalid request data* and *invalid user input* faults. The latter two are experienced relatively by most participants. *Expired request data* and *double processing* related problems are not experienced by over half of the participants.

Several participants added additional causes to the 11 we propose. Four participants mentioned that they experienced errors because the API was not responding. We summarize these issues as API downtime, which we consider part of the *internal* cause. Furthermore, two participants experienced problems caused by hitting the API requests limits. We regard these to be related to faults in the *insufficient permissions* cause. Namely, the API consumer is not allowed to make more requests.

RQ$_1$: Faults in API integration can be grouped in 11 causes: *invalid user input*, *missing user input*, *expired request data*, *invalid request data*, *missing request data*, *insufficient permissions*, *double processing*, *configuration*, *missing server data*, *internal* and *third party*.

	Cases	Errors (%)		Consumers (%)	
Cause	(N=69)	*Lower bound*	*Upper bound*	*Lower bound*	*Upper bound*
Invalid user input	13	6.5	6.6	33.3	34.8
Missing user input	10	0.7	5.4	11.5	24.8
Expired request data	2	0.0	0.5	2.0	15.8
Invalid request data	17	3.2	10.5	23.9	62.3
Missing request data	12	23.0	28.7	20.2	24.0
Insufficient permissions	3	0.1	0.1	9.5	9.5
Double processing	3	36.0	36.0	12.3	12.3
Configuration	2	16.7	16.7	19.9	21.4
Missing server data	2	1.5	1.5	13.9	13.9
Internal	3	0.1	4.9	0.7	18.9
Third party	2	0.4	0.9	21.8	46.6

Table 4.2: The number of faults per cause grouped by stakeholder, and the estimated percentage range of errors and impacted API consumers. The percentages are based on 2.43 million API errors and 1,464 impacted API consumers. Note that the percentages do not add up 100%.

4.0.2 RQ$_2$. What is the prevalence of these fault types, and how many API consumers are impacted by them?

In Table 4.2, we show the number of unique faults that occur in each of the 11 causes found during manual analysis. As aforementioned, due to ambiguity we are not able to present the exact percentage of errors and impacted consumers. For this reason, we show the estimated percentage of corresponding API error responses and estimated percentage of impacted consumers for each cause. In four causes of faults no ambiguity was present, hence the exact percentages are given instead of a range. Note that the total percentages for the lower and upper bound do not add up to 100% due to the estimation in the total number of errors, and to the fact that the same consumer may generate different faults.

For the end user, we see the input to be the largest cause of faults. 23 of the 25 faults are caused by invalid or missing input, collectively causing 7.2% to 12% of the errors. *Invalid user input* does not only result in more unique faults than *missing user input*, it also impacts more API consumers, 33.3% to 34.8% compared to 11.5% to 24.8%. *Expired request data* sent in by the end user causes two different faults, causing an impact in 2.0% to 15.8% of our consumers.

The request data causes the most faults for the API consumer, similar to the end user. *Invalid request data* results in 17 of the 39 unique faults caused by the API consumer. Faults in this category impact the most consumers for this stakeholder, namely between 23.9% and 62.3%. This corresponds to between 350 and 912 API consumers. *Missing request data*, good for 12 faults, has an impact on fewer consumers (between 20.2% and 24.0%), but does however yield more erroneous API responses. Interestingly, only 0.1% *insufficient permissions* related errors caused by 3 faults impact 9.5% of the API consumers. *Double processing* related error in comparison are also caused by 3 faults, but occurred 36.0% of the time corresponding to 875,000 errors for 12.3% of the consumers. Two *configu-*

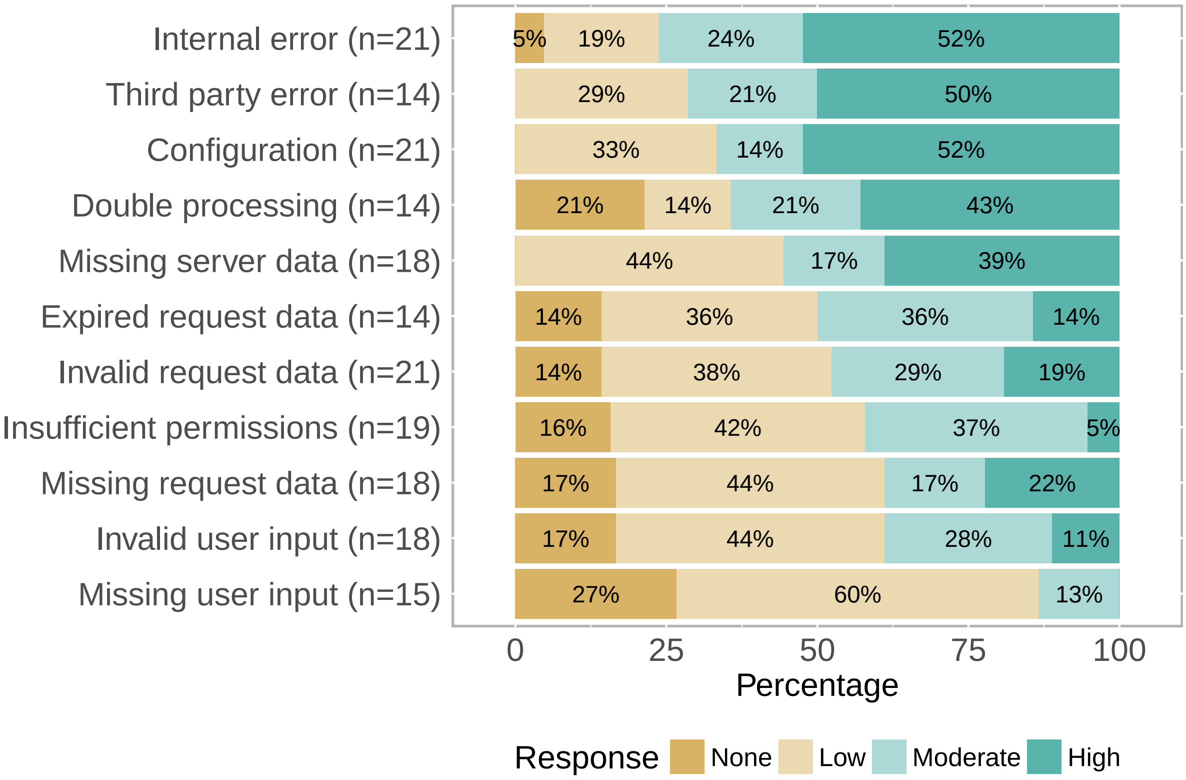

Figure 4.2: The impact of API related problems per cause as experienced by the survey participants.

ration faults cause more than 400,000 errors (16.7%) for 19.9% to 21.4% of the consumers. This is similar to the *missing request data*, with the difference that this cause has 12 unique faults. Finally, 1.5% *missing server data* fault related error responses are given back to the API consumer, which is a relatively small amount compared to the other causes for this stakeholder. The number of impacted consumers, however, is similar to other causes.

The API provider and third party stakeholder both experience errors in one category each, *internal* faults and *third party* faults respectively. The number of unique faults caused by these stakeholders is small compared to the end user and API consumer. *Third party* faults however impact more API consumers, which is estimated to be between 319 and 682, or 21.8% and 46.6%.

In addition, in Figure 4.2, we show the perceptions of survey participants' on the impact of each cause. We observe that: 1. *Internal* and *third party* related problems, caused by the API provider and third parties, are experienced as most impactful on production applications; 2. Problems originating from the end user, such as *invalid user input* and *missing user input*, have a relative small impact on the applications using the API; and 3. Interestingly, *double processing* related problems seem to have either no impact, or relatively much impact compared to the other causes.

RQ$_2$: Most faults are caused by *invalid request data*, *missing request data*, and *double processing*. Faults caused by the API provider and third parties are experienced most impactful according to API consumers. On the other hand, faults originating from the end user, although very frequent in our dataset, are regarded as having the least impact.

4.0.3 RQ$_3$. What are the current practices and challenges to avoid and reduce the impact of problems caused by faults in API integration?

To understand how API consumers obtain the knowledge necessary to integrate with an API we asked them how often they used different information sources. Official API documentation is by far used the most. 74% of the respondents indicated to be using this source of information *often* or *very often*. Only 10% did not use official API documentation when integrating with the API they selected during the survey. Code examples are second most used with 44% of the participants using them *often* or *very often*. About one-third of the participants uses them *sometimes*. Questions and answer websites are used *never* or *rarely* by 42% of the participants, while the number of participants that uses this information source *very often* is relatively low with 10%. The API provider support team is used the least with only 18% of the participants using this source *often* or *very often*.

In addition to the four proposed information sources the participants mentioned other sources of information. Four participants mentioned that they used a trial and error approach on the API to discover what is possible and what is not. Three respondents had access to the API's source code or used the schema definition of the web service to understand the workings of the API. Finally, two participants used the source code of existing external libraries that wrap the API to understand how to use the API.

When it comes to preventing problems that they experienced with the API, 13 (of a total of 18) mentioned the documentation should be improved. Common implementation scenarios could help prevent problems, instead of only stating the different options for API calls. The restrictions of calls and parameters should be more clearly documented. The API provider should identify the most common API mistakes and describe how to prevent them. In addition, more details on error codes should be given and the edge cases should be highlighted and better explained. Two participants mentioned the need of an API status page to inform the API consumer of any outages. On call support for any issues was a suggested improvement by two more participants. The participants suggested both more informative error messages as well as a categorization of errors based on their similarities. Furthermore, the respondents mentioned the importance of an upgrade policy of the API and the usefulness of more code examples to illustrate the different API calls. Lastly, one participant suggests the API provider to set up a testing environment that is capable of returning all possible API errors, which allows the API consumer to properly test and handle these responses.

A subset of the participants ($n = 15$) elaborated on the challenges they face in error handling. One of the main difficulties is understanding the impact of API errors. Three impact perspectives were mentioned: an implementation perspective, business perspective and end user perspective. Not knowing the details and impact of an error makes it difficult from an implementation perspective to know what the request did and did not do. A participant exemplified: "*You send a batch of 20 objects to be saved, but an error gets thrown. However, you don't know if none of them was saved or all of them but one.*" From a business perspective it is experienced as difficult to understand the business impact of the error. The error may explain that a parameter is invalid, but the consequences of this remain unclear. Finally, communicate errors to the end user is also experienced a challenge: "*Translating the messages to something actionable by the end user.*"

Another challenge faced in handling errors is the appropriate way to recover. Difficulties experienced include insufficient clarity and documentation about the right way to recover from a given error: "*Often errors have no clear recovery option or even worse, do not clearly indicate what's wrong.*" Handling errors is difficult when the different flows the application should take given the API response are not clear. This is even more difficult when multiple related API calls are subsequently made and can fail with different errors.

The survey participants ($n = 29$) also apply different strategies to detect problems in their integration. The end user detects problems in the application related to the API integration for 23 respondents. Log analysis is second most effective in detecting problems with 19 respondents. Monitoring dashboards have detected issues for 13 respondents and both alerts, such as SMS or email, and API integration tests worked for 9 respondents. Five respondents had additional mechanisms in place, among which are "continuous live smoketesting", "manual tests in production" and "Runscope Radar" [5], an API monitoring tool that can detect downtime and schedule API test cases.

RQ$_3$: API consumers most often use official API documentation to implement an API correctly, followed by code examples. The challenges of preventing problems from occurring are the lack of implementation details, insufficient guidance on certain aspects of the integration, insufficient understanding of the impact of problems, and missing guidance on how to recover from errors.

Chapter 5

Discussion

In this section, we provide developers with recommendations that we derive based on our findings (Section 5.0.1). Finally, we discuss the possible threats to the validity of this work and actions we took to mitigate them (Section 5.0.2).

5.0.1 Recommendations

We observed that a great challenge for both API providers and consumers is indeed the documentation; providers need to keep it up-to-date, while consumers need to understand it. The same challenge has been observed by Robillard et al [21], where authors propose API documentation to have clear intent, code examples, matching APIs with scenarios, discuss the penetrability of the API, and to have a clear format and presentation. Our findings suggest that an important feature of documentation is regarding the possible errors an API may return.

We therefore suggest documentation to clearly state which error codes can be returned by the API and in what circumstances it can happen. The API provider should enrich API error responses with actionable information. An error type allows for generic error handling for groups of errors, a handling action indicates the right action for the API consumer to take to deal with the error, and a user message can inform the end user of the system about the error and actions to proceed. In addition, API providers should also make explicit which errors are 'retriable', *i.e.*, , where users may try again, and which cases users have no way of recovering from the error.

As recovering from errors is a fundamental part of the API consumer's logic, we suggest API providers to offer easy-to-use test environments for integrations, where consumers can exercise not only the happy paths, but also the recovering paths. API consumers, on the other hand, should make sure that their application handles all error codes that are returned by the provider.

Finally, error messages are commonly logged by the API provider. Such logs are vital for future inspection and debugging. We suggest API developers to provide consumers with their logs. Toolmakers should step in and build dashboards that provide live insights about the consumers' API usage and, more specifically, about their errors.

5.0.2 Threats to validity

In this section, we discuss the possible limitations of this work and our approach to mitigate them. We distinguish between the internal and external validity of our results.

Internal validity Internal validity is concerned with how consistent the result is in itself. Factors that cannot be attributed to our technique, which can have an influence on the results, are a potential threat to validity.

1. The causes were derived manually and could therefore have been subject to bias or misinterpretation. To reduce this threat we worked together closely with the Adyen development and technical support team to avoid misunderstandings.
2. To discover possible multiple explanations of a fault, we analyzed the error messages for several API consumers. However, it is possible that a fault remained undiscovered because it occurred infrequently. This has a possible impact on our findings. Similarly, we filtered the data for analysis based on 10 impacted API consumers or more. The filtered data could be explained by faults that would alter the distribution of faults over the causes. For instance, internal faults could occur more often in the data that was filtered out, therefore posing a potential threat to validity.

External validity External validity is concerned with the representativeness of the results outside the scope of the research data.

1. We used API error log data from the Adyen's platform to determine the fault causes and provide insights into the frequency and impacted consumers of these faults. Since these results are applicable to Adyen only, we cannot generalize these results to faults in other APIs. To reduce this threat we verified the completeness of the fault causes by surveying API consumers.
2. An arbitrary window of 28 days of API error logs was selected for fault analysis and categorization. A different 28 day window could however have resulted in a different set of faults, and a different number of occurrences and impacted consumers. It would be useful to replicate the analysis based on a different time window to investigate the impact on the results.
3. We only obtained 40 survey responses, of which 11 were partial responses. This sample is insufficient to generalize results about integration, detection, handling and prevent practices, and future work would be required for such generalization.

Chapter 6

Related work

Wittern et al. [31] identified three challenges for developers calling Web APIs and argue in favor of corresponding research opportunities to support API consumers: 1) API consumers have no control over the API and the service behind it, both of which may change, in contrast to a traditional local library. 2) The validity of the Web API request, in terms of URL, payload and parameters, is unknown until runtime. When using a local library the compiler can check whether the call conforms to the library's API interface. Efforts to solve this challenge can help to reduce faults in the following categories we identified: *invalid user input* and *invalid request data*. 3) The distributed nature of the API connection comes with a set of issues concerning availability, latency and asynchrony. A different architecture or additional logic may be required to handle these issues.

Suter and Wittern [27] use API usage logs from 10 APIs to infer specifications based on API URLs and parameters using classification techniques to tag and detect parameters. The conclusion is that inferring web API descriptions is a difficult problem that is limited mostly by incomplete or noisy input data. Sohan et al. [24] apply a similar approach where API requests and responses are used to generate documentation. Using actual requests and responses the tool is able to include examples in addition to the documentation. The authors identified undocumented fields in 5 out of 25 API actions for which they generated documentation. However, in this work the precision of the generated documentation is not validated and compared to a ground truth making it difficult to see the usefulness of the proposed generator.

Bermbach and Wittern [9] performed a geo-distributed benchmark to assess the quality of Web APIs, in terms of performance and availability. The authors find a great variety in quality between different APIs. They make suggestions on how API providers can become aware of these problems by monitoring, and can mitigate them by suggesting architectural styles. This work discusses an angle of API related problems which we were not able to cover, as our work only considers API requests for which API consumers received an API response. We do however cover third party unavailability as this results into *third party* errors for the API consumer.

Wittern et al. [30] attempt to detect errors by statically checking API requests in JavaScript to overcome the fact that traditional compile-time errors are not available for developers consuming APIs. Their static checker aims to check whether the API requests in the code conform to the specifications that were made using the Open API specification. The authors report a 87.9% precision for payload data and a 99.9% precision on query parameter consistency checking.

A vast amount of research has been conducted in the field of traditional offline APIs, some of which can be relevant to the Web APIs as well. Robillard et al. [20] provide a survey on automated property inference for APIs. The authors state that using APIs can be challenging due to hidden assumptions and requirements, which is also found in this work. Robillard et al [21, 19] also investigated the obstacles of learning traditional offline APIs by surveying developers at Microsoft. Similar to our results, Robillard found most respondents use official documentation to learn APIs with code samples as the second most used source of information.

The learnability of an API can be affected by the overall usability of the API itself. Stylos et al. [26] find that if API providers take the effort to refactor their APIs to make them more usabe,this can help reduce the errors that could occur due to incorrect usage of the API. One such example shown by Ellis et al. [10] is the refactoring of the factory pattern in an API to the usage of the consutructor directly. More recently, Stylos and Myers [16] have suggested that API usability techniques are not limited to the world of offline APIs, but are applicable to the world of web APIs as well.

The evolution of an API has an impact on the API clients as well. Linares-Vásquez et al. [14] have shown that breaking changes in Android APIs can have a negative impact on the rating of an Android app. Sawant et al. [23] conducted a study of 25,567 Java projects to show how deprecation of an API feature can impact an API client. Robbes et al. [18] study the ripple effect of API evolution in the entire SmallTalk ecosystem and show that deprecation of a single API artifact can have a large ranging impact on the ecosystem.

In the case of web APIs, evolution of an API can have a major impact too. Espinha et al. [12] explored the state of Web API evolution practices and the impact on the software of the respective API consumers. The impact of API changes on the clients' source code was found to depend on the breadth of the API changes and the quality of the clients' architectural design. Suggestions for API providers include not changing too often, keeping usage data of different features and doing blackout tests, which involves disabling old versions for a short time to remind developers that changes in the API are coming. In other work, Espinha et al. [11] developed a tool to understand the runtime topology of APIs in terms of usage of different versions by different users. Wang et al. [29] study the specific case of the evolution of 11 REST APIs, where they collected questions and answers from Stack Overflow that concern the changing API elements and how API clients should deal with evolution. Li et al. [13] identified 6 new challenges when it comes to dealing with web API evolution as opposed to traditional API evolution. This understanding can be useful for maintenance purposes where the impact of changes can be evaluated and predicted.

Venkatesh et al. [28] mention that to help the integration process one should understand the challenges that are encountered by client developers. The authors base their analysis on developer forums and Stack Overflow by mining the questions and answers related to 32 Web APIs. They find that the top five topics per Web API category contribute to over 50% of the questions in that category. The findings imply that API providers can optimize their learning resources based on the dominant topics.

Chapter 7

Conclusion

API errors can indicate significant problems for API consumers. In the system under study over 60,000 API error responses are returned every day, causing the potential number of problems and their impact on API consumer applications to be enormous. Practitioners have written a variety of best practice guides and blog posts on API design and error handling, however to our knowledge no research had been conducted on what type of API errors occur in practice and what their impact is.

Our results show that 1. faults in API integration can be grouped in 11 causes: invalid user input, missing user input, expired request data, invalid request data, missing request data, insufficient permissions, double processing, configuration, missing server data, internal and third party. Each cause can be contributed to one of the four API integration stakeholders: end user, API consumer, API provider, and third parties; 2. most faults can be attributed to the invalid or missing request data, and most API consumers seem to be impacted by faults caused by invalid request data and third party integration; and 3. API consumers most often use official API documentation to implement an API correctly, followed by code examples. The challenges of preventing problems from occurring are the lack of implementation details, insufficient guidance on certain aspects of the integration and insights in problems and changes, insufficient understanding of the impact of problems, missing guidance on how to recover and a lack of details on the origin of errors.

Our findings indicate that the integration between API providers and consumers is still far from ideal. We hope this work motivates researchers to further explore the domain of faults in Web API integration. Furthermore, we hope that API providers use our findings to optimize their APIs to enable better integration, and that API consumers use our ideas to reduce the impact that API errors may have on their applications.

Chapter 8

Current and Future Work

We are currently working with STAMP partner XWiki to further finetune our EvoCrash approach. EvoCrash allows to replicate crashes of applications using stacktraces.

Bibliography

[1] Apigee: The API platform for digital business. https://apigee.com/. [Online; accessed 28-Jun-2017].

[2] CodeRanch. https://coderanch.com/. [Online; accessed 28-Jun-2017].

[3] Hackernews. https://news.ycombinator.com/. [Online; accessed 28-Jun-2017].

[4] Reddit. https://www.reddit.com/. [Online; accessed 28-Jun-2017].

[5] Runscope radar. https://www.runscope.com/. [Online; accessed 28-Jun-2017].

[6] APIGEE. Web API Design: Crafting Interfaces that Developers Love. https://pages.apigee.com/web-api-design-website-h-ebook-registration.html. [Online; accessed 28-Jun-2017].

[7] APIGEE. Web API Design: The Missing Link. http://bit.ly/2gtEhz6. [Online; accessed 28-Jun-2017].

[8] AUÉ, J., ANICHE, M., LOBBEZOO, M., AND VAN DEURSEN, A. An exploratory study on faults in web api integration in a large-scale payment company: Appendix. https://www.zenodo.org/record/1035151.

[9] BERMBACH, D., AND WITTERN, E. Benchmarking web api quality. In *International Conference on Web Engineering* (2016), Springer.

[10] ELLIS, B., STYLOS, J., AND MYERS, B. The factory pattern in api design: A usability evaluation. In *Proceedings of the 29th international conference on Software Engineering* (2007), IEEE Computer Society, pp. 302–312.

[11] ESPINHA, T., ZAIDMAN, A., AND GROSS, H.-G. Understanding the interactions between users and versions in multi-tenant systems. In *Proceedings of the 2013 International Workshop on Principles of Software Evolution* (2013), ACM.

[12] ESPINHA, T., ZAIDMAN, A., AND GROSS, H.-G. Web API growing pains: Stories from client developers and their code. In *Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)* (2014).

[13] LI, J., XIONG, Y., LIU, X., AND ZHANG, L. How does web service api evolution affect clients? In *Web Services (ICWS), 2013 IEEE 20th International Conference on* (2013), IEEE, pp. 300–307.

[14] LINARES-VÁSQUEZ, M., BAVOTA, G., BERNAL-CÁRDENAS, C., DI PENTA, M., OLIVETO, R., AND POSHYVANYK, D. Api change and fault proneness: A threat to the success of android apps. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering* (2013), ACM, pp. 477–487.

[15] Mulloy, B. RESTful API Design: what about errors? https://apigee.com/about/blog/technology/restful-api-design-what-about-errors. [Online; accessed 28-Jun-2017].

[16] Myers, B. A., and Stylos, J. Improving API usability. *Communications of the ACM 59*, 6 (2016).

[17] Patil, A. Get developer hugs with rich error handling in your API. http://bit.ly/2gxacyH. [Online; accessed 28-Jun-2017].

[18] Robbes, R., Lungu, M., and Röthlisberger, D. How do developers react to api deprecation?: the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (2012), ACM, p. 56.

[19] Robillard, M. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software 26*, 6 (2009).

[20] Robillard, M., Bodden, E., Kawrykow, D., Mezini, M., and Ratchford, T. Automated API property inference techniques. *IEEE Transactions on Software Engineering 39*, 5 (2013).

[21] Robillard, M. P., and DeLine, R. A field study of api learning obstacles. *Empirical Software Engineering 16*, 6 (Dec 2011), 703–732.

[22] Sandoval, K. Best Practices for API Error Handling. http://nordicapis.com/best-practices-api-error-handling/. [Online; accessed 20-Jul-2017].

[23] Sawant, A. A., Robbes, R., and Bacchelli, A. On the reaction to deprecation of 25,357 clients of 4+ 1 popular java apis. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on* (2016), IEEE, pp. 400–410.

[24] Sohan, S., Anslow, C., and Maurer, F. Spyrest: Automated restful API documentation using an HTTP proxy server (N). In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2015).

[25] Stowe, M. API Best Practices: Response Handling. https://blogs.mulesoft.com/dev/api-dev/api-best-practices-response-handling/. [Online; accessed 28-Jun-2017].

[26] Stylos, J., Graf, B., Busse, D. K., Ziegler, C., Ehret, R., and Karstens, J. A case study of api redesign for improved usability. In *Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008. IEEE Symposium on* (2008), IEEE, pp. 189–192.

[27] Suter, P., and Wittern, E. Inferring web API descriptions from usage data. In *Third IEEE Workshop on Hot Topics in Web Systems and Technologies* (2015).

[28] Venkatesh, P., Wang, S., Zhang, F., Zou, Y., and Hassan, A. What Do Client Developers Concern When Using Web APIs? An Empirical Study on Developer Forums and Stack Overflow. In *IEEE International Conference on Web Services (ICWS)* (2016).

[29] Wang, S., Keivanloo, I., and Zou, Y. How do developers react to restful api evolution? In *International Conference on Service-Oriented Computing* (2014), Springer, pp. 245–259.

[30] Wittern, E., Ying, A., Zheng, Y., Dolby, J., and Laredo, J. Statically checking web API requests in JavaScript. In *Proceedings of the 39th International Conference on Software Engineering* (2017), IEEE Press.

[31] WITTERN, E., YING, A., ZHENG, Y., LAREDO, J., DOLBY, J., YOUNG, C., AND SLOMINSKI, A. A. Opportunities in software engineering research for web API consumption. In *Proceedings of the 1st International Workshop on API Usage and Evolution* (2017), IEEE Press.