An Exploratory Study on Faults in Web API Integration

Master's Thesis

Joop Aué

An Exploratory Study on Faults in Web API Integration

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Joop Aué
born in Zwolle, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Adyen B.V.
Simon Carmiggeltstraat 6-50, 1011DJ
Amsterdam, the Netherlands
www.adyen.com

©2017 Joop Aué. All rights reserved.

An Exploratory Study on Faults in Web API Integration

Author: Joop Aué
Student id: 4139534
Email: joopaue@gmail.com

Abstract

Nowadays, service-oriented architectures are more popular than ever, and more and more companies and organizations depend on services offered through Web APIs. The capabilities and complexity of Web APIs differ from service to service, and therefore the impact of API errors varies. API problem cases related to Adyen's payment service were found to have direct considerable impact on API consumer applications. With more than 60 thousand daily API errors the potential impact is enormous. Similarly, API consumers of any API can experience errors, and depending on the application the impact can be costly.

In an effort to reduce the impact of API related problems, we analyze 2.43 million API error responses to identify the underlying faults and derive 11 generic categories that describe them. We quantify the occurrence of faults in terms of the frequency and impacted API consumers. We investigate the impact of API faults on API consumer applications and illustrate this with 3 case studies. Furthermore, an overview is given of the current practices and challenges to avoid and reduce the impact of API errors by API consumers. Using the results, we introduce 16 recommendations for API providers and API consumers to reduce the impact of API related faults.

Thesis Committee:

Chair and university supervisor:	Prof.dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. M.F. Aniche, Faculty EEMCS, TU Delft
Committee Member:	Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
Committee Member:	Dr. C. Hauff, Faculty EEMCS, TU Delft
Company supervisor:	Ir. L.W.M. Lobbezoo, Adyen B.V.

Preface

Someone once told me that great opportunities always seem to come my way. Perhaps I have repeatedly had the lucky chance to make, what I genuinely think to be, "the best choice of my life". I believe, at this moment, I have made three of those choices.

The best choice of my life was to start my education in Computer Science and move to Delft in 2011. The infinite possibilities to create anything using software fascinates me, and my study taught me how to be effective in this. I got the chance to develop myself in what I am passionate about: understanding and solving complex problems, and creating simple and effective solutions.

Around the same time, I made the best choice of my life to participate in the rich social and extracurricular activities that the Delft student life has to offer. I met the most amazing people and made friendships that I am sure will last a lifetime. Together, we shared so many fantastic experiences, and took part in so many incredible activities, that I look back on with great joy.

I am very happy that I decided to delay my master's, apply for an internship at Adyen and move to the city of Amsterdam, which I regard to be the best choice of my life. Not only did I learn how my studies in Computer Science can be applied in industry, I also identified an interesting and challenging research topic for my master's thesis, the result of which lies in front of you. The combination of the practical implications and the scientific aspect made this thesis a joy for me to work on.

I would like to thank Arie van Deursen for providing me with invaluable advice and feedback over the last two years that we have worked together. Also, thank you for the introduction to Adyen, which has shaped my past year and a half. This allowed me to develop myself, both socially and technically, in an amazing environment.

Maurício, it has been a great experience working with you during the Software Architecture course and my thesis, and I am sure our future collaboration will not be any different. You are an amazing person, and probably the most Dutch non-Dutch guy I know. I am forever grateful for your invaluable feedback and your 24/7 availability for all my questions. Thank you for dealing with my occasional stubbornness and tendency to overthink and for taking the time to explain your point of view to me.

Maikel, many thanks for all of your input and feedback over the course of the past year and a half. Our opinions sometimes led us to agree to disagree, but I mostly felt that your

thoughts on the project complemented mine, making for fruitful conversations. You take thinking outside the box, seeing the bigger picture and reasoning on an abstract level to a whole new level. This made me realize that the impossible is most certainly possible. Thank you for continuously challenging me to take things to the next level, and providing me with countless opportunities to develop myself.

Lisanne, Marco, Peter, Tjerk and Willem, thank you for the walks and talks. Not only did you help me organize and structure my thoughts, you are one of the primary reasons I enjoy my time at Adyen as much as I do.

I would like to thank all of my colleagues for taking the time to help me understand, obtain information and bring me in contact with customers. Without your support I could not have completed my thesis.

Finally, I want to thank my friends and family for their unlimited support and belief in me. Although my technical jargon probably did not make much sense to you, your outsider input was extremely useful nonetheless.

If I look back at the incredible time I have had and all that I have learned in the past 6 years, I cannot begin to imagine what amazing experiences and opportunities the next 6 years will bring. My Delft adventure is ending, but a new adventure at Adyen is about to begin and I am quite certain that I am on the verge of making the best choice of my life once again.

Enjoy reading!

Joop
Amsterdam, September 2017

Contents

Preface iii

Contents v

1 Introduction 1

2 Background 5
- 2.1 Web APIs . . . 5
  - 2.1.1 Web API history . . . 5
  - 2.1.2 SOAP and REST . . . 6
- 2.2 Industry partner: Adyen . . . 8
  - 2.2.1 Adyen API . . . 9
  - 2.2.2 Error responses . . . 9
  - 2.2.3 Logging . . . 10

3 Understanding the API environment 11
- 3.1 API stakeholder overview . . . 11
- 3.2 The cause-failure chain . . . 12
- 3.3 API integration environment . . . 13

4 Research Questions & Methodology 15
- 4.1 Research questions . . . 15
- 4.2 Research methodology . . . 17

5 API Fault Data Extraction Approach 19
- 5.1 Extracting unique error messages from API logs . . . 20
  - 5.1.1 Log data set . . . 20
  - 5.1.2 Data requirements . . . 20
  - 5.1.3 Unique error extraction . . . 21
- 5.2 Identifying unique faults . . . 22
  - 5.2.1 Removing ambiguity . . . 22

5.2.2 Prioritization . . . 22
5.2.3 Annotation . . . 23
5.3 Data set time window justification . . . 23
5.4 Data set for quantitative analysis . . . 24

6 Faults in API integration **27**
6.1 Methodology . . . 27
6.1.1 Data set . . . 28
6.1.2 Categorization of unique faults . . . 28
6.2 Fault types in API integration . . . 28
6.2.1 Invalid user input . . . 29
6.2.2 Missing user input . . . 30
6.2.3 Expired request data . . . 30
6.2.4 Invalid request data . . . 30
6.2.5 Missing request data . . . 30
6.2.6 Insufficient permissions . . . 31
6.2.7 Double processing . . . 31
6.2.8 Configuration . . . 31
6.2.9 Missing server data . . . 32
6.2.10 Internal . . . 32
6.2.11 Third party . . . 32
6.3 API integration fault prevalence . . . 33
6.3.1 Stakeholder perspective . . . 33
6.3.2 Category perspective . . . 34

7 Illustrative API Integration Problem Cases **37**
7.1 Methodology . . . 37
7.1.1 Interviewee selection . . . 38
7.1.2 Interview design . . . 39
7.1.3 Conducting the interviews . . . 40
7.2 Case 1: Unhandled contract update . . . 40
7.2.1 API consumer description . . . 41
7.2.2 Problem description . . . 41
7.2.3 The integration process . . . 42
7.2.4 Error handling . . . 42
7.2.5 Verifying integration correctness . . . 42
7.2.6 Monitoring . . . 43
7.2.7 Suggestions . . . 43
7.3 Case 2: Insufficient permissions in chained API calls . . . 43
7.3.1 API consumer description . . . 43
7.3.2 Problem description . . . 44
7.3.3 The integration process . . . 44
7.3.4 Error handling . . . 45
7.3.5 Verifying integration correctness . . . 45

7.3.6 Monitoring . . . 45
7.3.7 Suggestions . . . 45
7.4 Case 3: Invalid encryption key . . . 45
7.4.1 API consumer description . . . 46
7.4.2 Problem description . . . 46
7.4.3 The integration process . . . 47
7.4.4 Error handling and monitoring . . . 47
7.4.5 Suggestions . . . 47

8 API Consumer Perspective 49
8.1 Methodology . . . 49
8.1.1 Target audience . . . 50
8.1.2 Overall design . . . 51
8.1.3 Question design . . . 51
8.1.4 Evaluation and improvements . . . 52
8.1.5 Sampling and respondents . . . 53
8.2 Fault types and their impact . . . 55
8.2.1 Fault types experienced by API consumers . . . 55
8.2.2 Fault type impact experienced by API consumers . . . 57
8.3 API integration practices and challenges . . . 58
8.3.1 API integration by API consumers . . . 58
8.3.2 API fault prevention . . . 59
8.3.3 API error handling . . . 59
8.3.4 API fault detection . . . 61
8.3.5 Underlying causes . . . 62

9 Recommendations 65
9.1 Fault type detection . . . 66
9.1.1 Fault detection action . . . 66
9.1.2 API error detection dashboard proposal . . . 67
9.2 Fault type prevention . . . 68
9.2.1 Prevent by validation . . . 69
9.2.2 Prevent by fix . . . 70
9.3 Fault type handling . . . 72
9.3.1 Feedback and retry . . . 72
9.3.2 Recover . . . 73
9.3.3 Retry or recover . . . 73
9.3.4 API error response proposal . . . 74
9.4 Problem priority . . . 76

10 Discussion 79
10.1 Related work . . . 79
10.2 Threats to validity . . . 81
10.3 Future work . . . 82

10.4 Lessons learned . . . 84
10.4.1 Understanding the data . . . 84
10.4.2 R testing . . . 84

11 Conclusion 87

Bibliography 91

A Adyen Fault Cases 95

B Interview Guide 105

C API Integration Survey 107

Chapter 1

Introduction

Nowadays, service-oriented architectures are more popular than ever. More and more companies and organizations 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 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, a multi-tenant Software as a Service (SaaS) platform that processes payments, offers an authorize request used to initiate a payment which takes up to 35 parameters. Multiple types of shopper interaction, and optional fields to optimize fraud detection and improve shopper experience make for numerous failure scenarios. In addition to the happy flow, 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 it can produce and hence, the larger the potential impact of these errors. It may be interesting to investigate the errors in API integration and the impact that they have.

We describe several problems related to Web API integration resulting in API errors for multiple API consumers to motivate the need to investigate Web API errors and their impact. The cases were discovered for Adyen merchants that use the payment service provider's Web API to process transactions.

We describe a configuration problem that caused 5000 payments to be rejected on a weekly basis for a merchant. Two payment methods were not configured for the merchant account that was used to process them, causing these payment API requests to return with an error. The merchant, processing hundreds of thousands payments on a monthly basis, was unaware of this problem, which caused it to go unnoticed for 9 months.

The next problem is caused by an unexpected indirect effect that one API call has on another call. Due to this issue, 50 subscription payments were found to fail on a weekly basis and the problem went unnoticed for 8 months. This number has increased to 200 in the course of three months and has not been resolved due to the prioritization of other projects. The merchant in this case expected one contract to be used, whereas actually a new contract had been issued. This caused invalid information to be supplied, which generated an API error response.

Two-and-a-half million euros is the amount of money not processed via Adyen's platform due to a bug in the system of a merchant. Approximately 1000 payments failed on a weekly basis for a period of 1 year due to outdated references, resulting in API error responses that were not acted upon by the merchant.

Similarly, each month 20.000 shoppers were unable to pay, because a merchant was using an incorrect merchant identifier. This problem, persistent for 9 months, resulted in 35 thousand euros worth of failed payment authorizations.

With over 60 thousand API error responses returned by Adyen's Web API every day, the potential number of problems and their impact on API consumer applications is enormous. For this reason, it will be valuable to understand how the impact of API error responses can be reduced.

To make error handling for client developers easier, practioners have written a variety of best practice guides and blogposts on API design [24] [36] [26] [32]. 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 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, a multi-tenant SaaS platform for thousands of businesses, that handles millions of API requests on a daily basis. We analyze the error responses, which we extract from the platform's production logs, discover the underlying faults, and group them into generic categories. Next we develop an intuition of the practices and challenges of dealing with API related problems by interviewing API consumers. The initial understanding acts as input to a survey for API consumers that teaches us about API faults, practices and challenges.

The main contributions of this work are as follows:

1. A classification of API faults, resulting in 11 generic categories of API faults, based on the API error responses of a large industrial multi-tenant SaaS platform.

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

3. An understanding of the fault type impact as experienced by API consumers.
4. An illustrative and reproducible approach to obtain API fault categories from API error log data.
5. A description and analysis of 3 illustrative cases of API related problems and their impact.
6. An overview of the current practices to avoid and reduce the impact of API related problems.
7. Insights into the challenges to avoid and reduce the impact of API related problems.
8. 16 recommendations for API providers and API consumers to reduce the impact of API related faults.

This work is structured as follows: We introduce the concept of Web APIs and give a background of our industry partner in Chapter 2. In Chapter 3, we elaborate on the API environment, which is at the basis of this work. Chapter 4 motivates the research questions and outlines the methodology to answer them. Next, we illustrate the approach used to extract API fault data from production API error logs in Chapter 5. We answer the first two research questions using the API fault data in Chapter 6. In Chapter 7, we illustrate three problems cases and develop an intuition of the practices and challenges concerning API problems, after which we answer the remaining questions in Chapter 8 based on an API consumer survey, and strengthen the answers to the first two research questions. Based on the results we propose recommendations for API providers and API consumers in Chapter 9. Chapter 10 discusses related work, the threats to validity of our results, the possibilities for future work and some of the lessons learned. Finally, we conclude this work in Chapter 11.

Chapter 2

Background

In this chapter we provide background knowledge on Web APIs and our industry partner. Section 2.1 elaborates on the definition, history and common terms of Web APIs. In Section 2.2 we introduce our industry partner and describe the practices of error handling and logging.

2.1 Web APIs

An Application Programming Interface (API) is a clearly defined protocol that specifies how two software components can communicate. It allows developers to create software using existing building blocks by using the API for communication. An API can act as an interface for software libraries and frameworks, e.g., the Java API [25], or as an interface for remote resources accessible through the web, such as a database or service. The latter type of API is referred to as a Web API, signifying the remoteness of the interface. Nowadays, due to its popularity, Web APIs are commonly referred to as APIs, without the prefix 'web'. In this work the two terms are used interchangeably.

Section 2.1.1 gives a short overview of the history of Web APIs. In Section 2.1.2 two terms commonly associated with Web APIs are explained to avoid the misconceptions that exist.

2.1.1 Web API history

On February 7th, 2000, when Salesforce[1] launched its services, the first commercially available Web API was introduced [21]. Salesforce's XML based API delivered what today is called Software as a Service (SaaS). In the same year eBay[2] rolled out their first API to a selected number of partners and developers with the aim of standardizing how applications integrated with eBay. Other significant API releases include those of Twitter and Google Maps, which both were created in response to a large number of people exploiting the original applications in order to simulate an API. Developers were scraping Twitter's web pages,

[1] https://www.salesforce.com/

[2] https://www.ebay.com/

and applications such as HousingMaps[3] exploited Google Maps' JavaScript library to use its map functionality. Today in 2017, over 17,500 API's are indexed by ProgrammableWeb's API directory [5] with this number growing every day.

2.1.2 SOAP and REST

Simple Object Access Protocol (SOAP) [9] and Representational State Transfer (REST) [15] are often used to describe Web APIs. The two terms can however not be compared, as the former is a protocol and the latter is an architectural style. Both are explained to remove any misconceptions.

SOAP is an XML based protocol designed for the exchange of information in a decentralized, distributed environment [9]. The protocol consists of three parts: an envelope that defines the message structure and how to process it, encoding rules to express application-defined data types, and a convention for representing procedure calls and responses. SOAP tightly couples the client and the server. A change on one side is expected to break the existing implementation requiring a contract update. An example SOAP payment authorize request is given below.

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema- instance">
  <soap:Body>
    <ns1:authorise xmlns:ns1="http://payment.services.adyen.com">
      <ns1:paymentRequest>
        <amount xmlns="http://payment.services.adyen.com">
          <currency xmlns="http://common.services.adyen.com">EUR</currency>
          <value xmlns="http://common.services.adyen.com">100</value>
        </amount>
        <card xmlns="http://payment.services.adyen.com">
          <cvc>737</cvc>
          <expiryMonth>08</expiryMonth>
          <expiryYear>2018</expiryYear>
          <holderName>Simon Hopper</holderName>
          <number>4111111111111111</number>
        </card>
        <reference xmlns="http://payment.services.adyen.com">123</reference>
      </ns1:paymentRequest>
    </ns1:authorise>
  </soap:Body>
</soap:Envelope>
```

REST is an architectural style described by Roy Fielding in 2000 [15]. Unlike SOAP, REST is not a standard, but a set of desired properties, architectural elements, and con-

[3] http://housingmaps.com/

straints on those elements inducing the required properties. With as goal to meet a number of desirable properties, such as scalability. RESTful APIs are stateless, meaning that any request from the client contains all information for the server to process the request, i.e., session state is kept in the client. REST services are uniform, which decouples the architecture by enabling the client and server to evolve independently. To this end, representations of the resources themselves are independent of each other and can be sent in XML, JSON or any other format. The representation of a resource held by a client is sufficient to modify or delete the remote resource. Finally, Fielding states that REST APIs must be hypertext-driven, commonly referred to as Hypermedia As The Engine Of Application State (HATEOAS) [14]. A HATEOAS API allows the client's application to navigate a REST API without prior knowledge except the initial URI. From that moment, all application state transitions are driven by choices returned by the server given the client's manipulation of the represented resources. For instance, the JSON response to a GET request to display the balance of a bank account displays the account number and balance, and in addition the possible choices the client can make: depositing, withdrawing or transferring money.

```
{
  "account_number": 123456,
  "balance": {
    "amount": 1000,
    "currency": "USD"
  },
  "links": [
    {
      "rel": "deposit",
      "href": "https://example.com/account/123456/deposit"
    },
    {
      "rel": "withdraw",
      "href": "https://example.com/account/123456/withdraw"
    },
    {
      "rel": "transfer",
      "href": "https://example.com/account/123456/transfer"
    }
  ]
}
```

In case the amount is negative, the server does not return the choice of withdraw and transfer, limiting the possible state transitions for the client.

APIs are often dubbed RESTful, even when they do not comply with all constraints and properties proposed by Fielding. Instead the term is (mis)used to indicate an HTTP API, that accepts JSON and uses the HTTP methods to retrieve, create, update and delete resources on a server.

2.2 Industry partner: Adyen

Research on API integration requires real world API data. We use an industrial data set provided by our industry partner Adyen. Adyen is a Payment Service Provider (PSP) offering a global payment solution for thousands of businesses, referred to as merchants. Using a single connection to the Adyen platform, merchants are able to process transactions globally online, in-app and in-store, offering over 250 different payment methods in more than 150 currencies. An overview of Adyen's solution is depicted in Figure 2.1 which shows the stakeholders involved in the process: the shopper, merchant, Adyen and third parties, such as schemes and issuers. Schemes are payment networks that set rules and provide infrastructure that allows payments to be processed. An issuer is a bank or financial institution that provides branded payment cards to consumers.

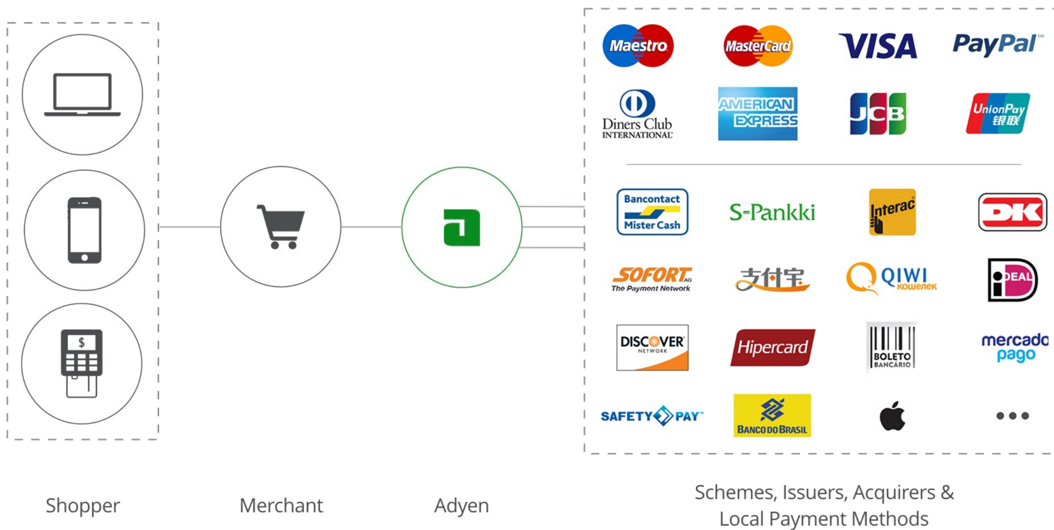


Figure 2.1: The stakeholders in the Adyen value chain

An example interaction with the Adyen system is outlined below. A shopper that wants to purchase goods from a merchant selects one of the available payment methods offered through, for example, the merchant's website. The available payment methods are those configured on the Adyen platform. The shopper submits their payment details and the merchant proceeds to send a payment request to the Adyen platform. Adyen in turn assesses the transaction risk, optimizes the request for higher authorization rates and processes the payment using the applicable third parties. The merchant receives a response, and in case of success the shopper can be notified and the goods can be shipped. Using the connection with Adyen, the merchant is able to perform additional actions, such as refunding payments, submitting payouts and making recurring payments as part of a subscription model.

In Section 2.2.1 we describe the Adyen API that can be used by merchants to connect to the Adyen platform. Error response handling in the Adyen API is explained in Section 2.2.2. Section 2.2.3 describes the logging practices at Adyen.

2.2.1 Adyen API

Adyen's customers have multiple ways of integrating with the Adyen platform. Merchants can use plug-ins offered by third parties like Magento[4], an ecommerce platform. The second option is to use an Adyen SDK or library that allows quick and basic integration. The third option, offering most functionality and flexibility is the Adyen API.

Communicating with the Adyen API takes place by submitting requests to a set of endpoints which allow specific actions, such as authorizing and capturing a payment, initializing recurring contracts and disputing chargebacks. The Adyen API supports XML, JSON, and URL query strings as input formats. For XML, the API offers Web Service Definition Language (WSDL) files that describe the endpoints, which can be used to construct SOAP messages. JSON and URL query strings do not have a strict communication contract, and can be constructed using the Adyen API reference[5]. Although the Adyen API exhibits some of the constraints and properties proposed by Fielding [15] it is not truly RESTful. It meets the stateless constraint as no client context is being stored and incoming merchant requests contain all information necessary to process that request. The HATEOAS constraint is however not met as the API does not support dynamic navigation of the interfaces, but rather requires merchants to use the documentation to understand the specification.

We consider the Adyen API to be complex due to the platform's elaborate set of features: supporting hundreds of payment methods, allowing for payouts and recurring payments, and offering multiple possibilities to reduce risk, detecting fraud and optimizing shopper experience. As a consequence, API methods such as *authorise* take up to 35 parameters of which 32 are optional. Similarly the *submit* request to make a payout accepts 15 parameters of which 11 are required.

2.2.2 Error responses

HTTP responses, e.g., to an API request, contain a status code indicating one of five standard HTTP response classes: *Informational*, *Successful*, *Redirection*, *Client Error* and *Server Error* [28]. Adyen utilizes the following three to inform the merchant about a processed request:

- Successful: status code 200 to denote that a request has been successfully processed.
- Client Error: status code 4XX, such as "404 resource not found" and "422 unprocessable entity" to indicate that the server could not understand the request.
- Server Error: status code 500 to indicate that the server encountered unexpected conditions that prevented it from processing the request successfully.

In an Adyen API integration, the number of possible error scenarios is high due to the large number of parameter combinations, as well as the input constraints of the individual parameters. An Adyen API error response, i.e., a non-200 response, contains an HTTP

[4] https://magento.com/

[5] https://docs.adyen.com/developers/api-reference

status code and specific error information to add more detail. This extra information comprises an error type, error code and error message. The error type is one of four categories: *validation*, *security*, *configuration* or *internal*. The error code is a number that uniquely describes the corresponding error message, which is Adyen specific. Adyen documents over 150 different error codes and messages [2]. An example JSON error response returned by the Adyen API is given below.


```
{
    "errorCode" : "137",
    "errorType" : "validation",
    "message" : "Invalid amount specified",
    "status" : "422"
}
```


2.2.3 Logging

Logging in the field of computer science is the practice of recording events that provide information about the execution of an application. Adyen logs approximately 700 million log messages on a daily basis to record detailed information about the platform's execution. These log messages are used for a variety of goals:

- Providing high level platform and server stability metrics, such as latency and the number of errors.
- Enabling technical support to investigate merchant inquiries.
- Facilitate development and operations to do root cause analysis of failures and errors.
- Clustering error logs to prioritize and uncover hidden issues in the massive amount of log data [13].
- Identifying anomolies in Point of Sale transactions, and identifying test scenarios based on production log data using passive learning [39].
- Feeding log metadata back to the developer to give insights into the production behavior of their code.

The Adyen logs also contain the incoming merchant API requests and outgoing API responses. We utilize this part of the logs in our work to investigate API errors.

Adyen uses Log4J to log *debug*, *info*, *warning* and *error* messages of events that occur in the system [17]. These log messages are made available for use through the Elastic Stack consisting of Elasticsearch, Logstash and Kibana[6]. Logstash parses data from different data sources and transforms it into input for Elasticsearch. Elasticsearch is a distributed full-text search engine designed to work with large volumes of data. Adyen visualizes Elasticsearch data using Kibana, a dashboard that provides visualizations.

[6]https://www.elastic.co/

Chapter 3

Understanding the API environment

An API integration can involve 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 chapter 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. This information serves as illustration and basis for the rest of this work.

Section 3.1 describes the different stakeholders that are, and can be, involved in an API integration. Next, in Section 3.2 we explain API error related terminology and explain how the different terms are related. Finally, the relation between the API error related terms and the API stakeholder overview is elaborated on in Section 3.3.

3.1 API stakeholder overview

In a typical integration with an 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. Figure 3.1 depicts the four stakeholders in the API integration environment.

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

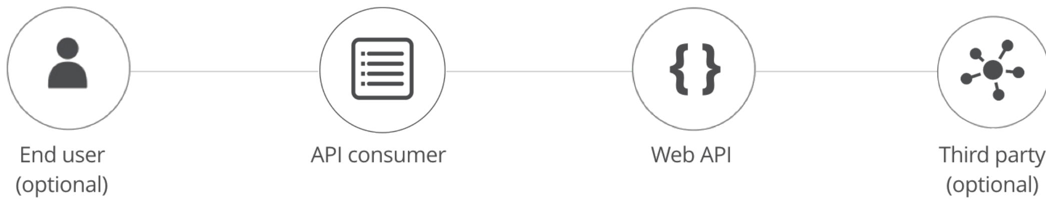


Figure 3.1: The stakeholders in an API.

3.2 The cause-failure chain

In this section we describe API error related the terminology. Furthermore, the relationship between the different terms in the context of this work is elaborated upon.

The terms *error*, *problem*, *failure* and *fault* seem similar in meaning. However, the nuances of these terms in the scope of this work are important to explain. Furthermore, in different contexts the terms can have different meanings, potentially causing confusion.

We refer to an *API error response*, *API error* or simply *error* when describing an API response that falls into the HTTP response classes *Client Error* and *Server Error* [28]. These classes correspond to responses with HTTP status codes 4XX or 5XX.

We denote a *failure*, or *problem*, by the inability of a system or component to perform its required function [27], resulting in a direct impact on the user of that system or application. A failure, in the scope of this work, can be the result of an API response with a 4XX or 5XX HTTP status code and has an impact on the API consumer's service or application. For instance, a user that attempts to make a payment, which does not succeed because system cannot process the payment, experiences a failure.

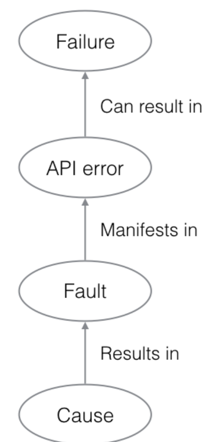


Figure 3.2: The relationship between a cause, fault, API error and failure.

We define *fault* as the *mistake* that is being made or the incorrect action being performed. A fault is not a problem or failure by itself. However, a fault can result in a problem or failure. In context of this work, the fault manifests in an API error, which in turn can result in a failure. Consider the example of a payment request that is rejected with an error, because the transaction amount if too high. In this case, setting the amount too high is the fault. The potential failure is that the transaction cannot be processed.

The *cause* is the underlying reason for a fault to happen. Continuing the previous example, the amount may be set too high because the documentation does not report a transaction limit, or the amount is set dynamically and no validation is in place.

Figure 3.2 illustrates the different terms and their relation. Note that the terminology in this work is different from the IEEE Standard which is commonly used in the field of software engineering. The IEEE Standard Glossary on Software Engineering Terminology (IEEE Standard 610.12-1990) [27] defines a fault as an incorrect step, process or data defi-

nition in a program, an error as the difference between the observed and expected outcome and a failure as the inability of a system or component to perform its required function. The term *failure* and *fault* as we explain them are very similar to the IEEE Standard. However, the term *error* is defined as the deviation between the actual and expected value, while we use it to describe the message that communicates a fault to API consumer.

3.3 API integration environment

Each of the stakeholders in the API stakeholder overview can potentially cause an erroneous response to be returned by the API, which if unexpected can result in problems for the application. 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.

We combine the stakeholder overview and the *fault* to *failure* part of the cause-failure chain in Figure 3.3. The faults that each of the stakeholders can cause result in an API error that is returned by means of an API error response to the API consumer. This error can potentially result in a failure for the API consumer. We left the *cause*, as described in the cause-failure chain, out of the diagram. This because the cause is not dependent on the origin of the fault. A fault at the API consumer could for instance be caused by a bug introduced by an developer or incorrect documentation of the API. What caused the fault also depends on how far one wants to go with determining the cause. An incorrectly implemented API call may be caused by a developer that did not read the documentation properly. However, one could also argue that the documentation was not clear enough, which therefore caused the fault. Likewise, unclear documentation may be caused by improper API design, which makes explaining the functionality difficult.

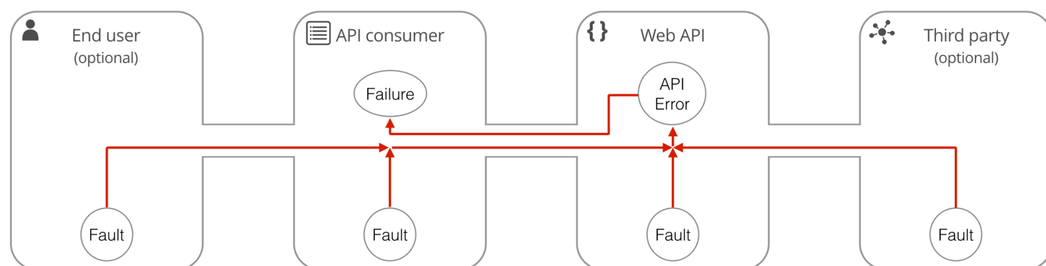


Figure 3.3: The API integration environment containing the involved stakeholders and the relation between faults, error and failure.

Chapter 4

Research Questions & Methodology

This chapter outlines the research questions and methodology used to understand the faults that occur in API integration that can potentially result in production problems for consumers of an API. We consider API integrations that involve four stakeholders: the *API provider*, *API consumer*, *end user* and *third parties*, explained in Section 3.3. Although the latter two stakeholders are optional for an API integration, this work is applicable to the two-stakeholder scenario with only the API provider and consumer as well.

Some APIs are more prone to return errors than others, and some API errors have more impact than others. This work attempts to cover any Web API integration, ranging from simple APIs to more complex ones. One could describe the complexity of an API in terms of the number endpoints or resources, the number of methods, the number of parameters (required and/or optional), parameter formatting constraints, the number of possible error scenarios or whether API requests depend on other requests. For instance, listing the number of followers of a Twitter user or using sending direct messages to other users on Twitter are simple operations that require at most two parameters. The formatting constraints are limited to numbers and strings, and the number of possible errors is limited. In contrast, the Salesforce Marketing Cloud SOAP API[1] offers more than 20 methods and can return up to 570 different error messages [3]. Intuitively, more functionality, more parameters, more constraints and more possible API errors increases the chance of problems and their impact on API consumer applications.

In Section 4.1 we motivate and pose the research questions answered in this work. Section 4.2 elaborates on the high level methodology to answer these questions.

4.1 Research questions

In order to reduce the impact of production problems that are caused by mistakes in API integration two things can be done: 1) Reduce the number of problem occurrences and 2) reduce the impact of individual problem occurrences. Before one can effectively do this, an understanding is needed of the underlying faults that occur in API integration that can result

[1] https://developer.salesforce.com/docs/atlas.en-us.noversion.mc-apis.meta/mc-apis/index-api.htm

in production problems. To this end we aim to answer the following research questions:

***RQ 1**: What type of faults, resulting in API errors, 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 *types* 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 impact the user the most when designing an API to lower the probability that these faults take place and result into problems. In addition, the knowledge will help 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 type of faults do API consumers consider the most impactful?*

The fault types and their frequency of occurring give insights into API related faults in general, However, this does not say anything about the impact that they have on the API consumer's application. By answering this question, fault types can be prioritized based on what is considered important by the API consumer.

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. Hence, we ask the following questions:

***RQ 4**: What are the current practices to avoid and reduce the impact of production problems caused by faults in API integration?*

The answer to this question helps to understand the current efforts of reducing the impact of faults in API integration. This information can be taken as starting point when developing recommendations.

***RQ 5**: What are the current challenges to avoid and reduce the impact of production problems caused by faults in API integration?*

The challenges of reducing the impact of API faults gives us an understanding of what is experienced as difficult. This allows recommendations to be made that take these challenges

into account.

4.2 Research methodology

To identify API fault types and understand their frequency of occurrence we need a data set that captures this information. To this end we use API error response logs from Adyen's production server. In Chapter 5, we describe the process of obtaining a data set from the API error logs that contains the unique faults that result in API errors and their explanation, the number of individual occurrences of these faults and the number of impacted API consumers.

Next, to answer ***RQ 1***, we use the annotated API fault data set to obtain a set of general categories that describe the faults. Using the frequency of errors and the information about the impacted API consumers we answer ***RQ 2*** about the quantitative impact of these faults. Chapter 6 describes the methodology and results of the API error data analysis.

To develop an intuition and better understanding of API integration problems, their impact and challenges we conduct three interviews with API consumers. The interviews, of which the methodology and results are given in Chapter 7, are used to illustrate API related problems experienced by API consumers. Furthermore, the developed intuition acts as basis for the questions of the API consumer survey described next.

We obtain insights from API consumers to learn what type of faults are experienced the most impactful to answer ***RQ 3***. Furthermore, we acquire information on the API integration process, API monitoring, testing and fault detection by API consumers. With this understanding we formulate an answer to ***RQ 4*** and ***RQ 5***. Finally, using API consumer insights we verify the results of the data analysis and thereby strengthen the answers to ***RQ 1*** and ***RQ 2***. Chapter 8 elaborates on the methodology and results of the API consumer insights.

Chapter 5

API Fault Data Extraction Approach

To understand the impact of API errors and identify possibilities to reduce this impact, data is needed that captures erroneous API interaction. The data set should contain an explanation of the fault for each API error response under analysis. This chapter acts as illustration as to how this data can be obtained, and describes how we use API request log data of the system under study to this end. The resulting data set is used as input for further analysis, and its properties are explained in Chapter 6 in more detail.

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 combine the manual annotations with the API error response data to allow for analysis per category. Figure 5.1 illustrates the four steps, which are elaborated on in Sections 5.1-5.4

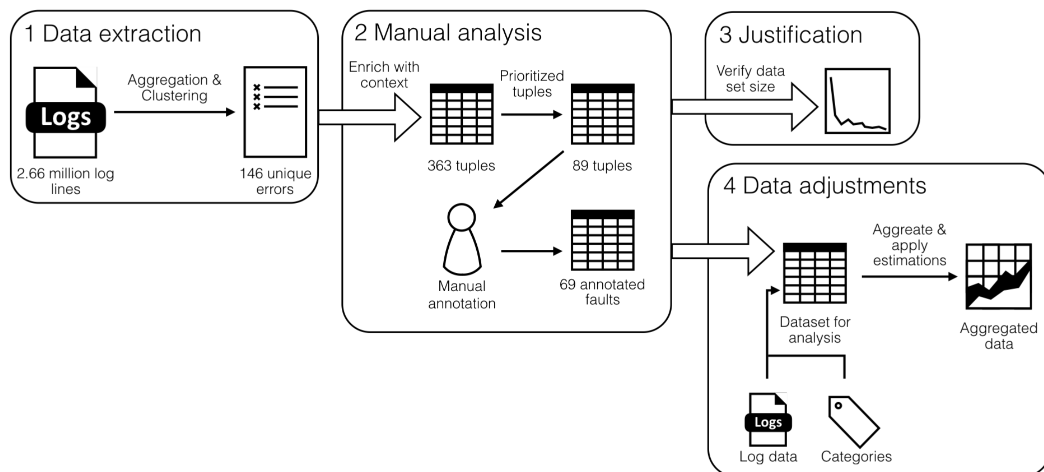


Figure 5.1: The four steps depicting the data extraction approach

5.1 Extracting unique error messages from API logs

We use the production logs from the system under study to obtain API error responses. Here, we describe our approach to obtain these responses and preprocess them to obtain the unique errors.

5.1.1 Log data set

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. Querying these logs is difficult as the logs that contain the API responses cannot be uniquely identified by a specific field or property. Using domain knowledge of the system, we identified queries to capture the erroneous API response log messages from the entire set of log messages.

5.1.2 Data requirements

To allow for the next steps we require the data to have the following response information:

Timestamp to denote the moment in time the request was made.

HTTP status code to indicate whether the request was successful or not.

API username that specifies who initiated the request.

Web service that indicates which endpoint was called.

Method to show what method was called.

Error code that identifies the specific error that occurred.

Error message that explains the error in more detail.

The following fields are optional, but helpful in our case in the manual analysis step:

Thread id to find log messages related to an erroneous request.

Reference which identifies a (failed) payment.

Host to identify the server that processed the request.

Company to help identify related problems for the same merchant.

We created a Java tool to execute the queries on Logsearch, a searchable dashboard which runs on top of Elasticsearch[1], parse the result and store the processed log messages in a database. The queries also captured API logs that are outside the scope of this work, which we therefore removed. Furthermore, internal API calls made by the system under

[1] https://www.elastic.co/products/elasticsearch

study itself were captured. Such internal errors can provide useful and interesting insights into internal API problems. However we are only interested in errors that have an effect on external API consumers, hence we remove this group of errors. The final data set contains 2.66 million API error responses and corresponds to 28 days of data.

5.1.3 Unique error extraction

In the manual analysis step we investigate the unique errors that occur in the system. To identify these, we use the error code, which is linked to an error message template in the API's source code. This template is a blueprint for error messages, e.g., "Currency {0} is not supported", which results in populated error messages such as "Currency EUR is not supported". Despite this mapping, we identify messages without error code, and messages with a (meaningless) "0" error code. In these cases, the developer has not specified a unique error code in the designated exception class that contains a list of error enums, but rather hard-coded the error at the place the error occurs. These messages cannot be mapped to a message template from the source code. To approximate the message templates, we cluster the messages using hierarchical clustering [31]. Next, we consider one message in each cluster as a representative for that cluster, which we refer to as the message template.

Before we resort to hierarchical clustering, we apply two heuristics to the error messages, which we empirically found to improve the precision of our results. One heuristic is to replace all digits by asterisks (*). By manual inspection of the error messages, we found digits to be rarely used in the constant part of the message. In cases where it is, it does not influence the later steps in the clustering process. An example of this heuristic is given below:

validation expired, pspReference=4513836233034604 $\longrightarrow$
*validation expired, pspReference=*****************

By manual inspection we find relatively long messages to have a variable part at the end, causing similar messages to be clustered separately. Hence, the second heuristic is to limit the number of characters for each message to 50, which we empirically found to yield the best results in our situation, to avoid long and similar messages to be identified as separate clusters.

We do not consider the prevalence of messages, hence after applying the heuristics we take the unique set of messages, reducing the number of message to consider from 350.000 to 150.

We apply hierarchical clustering [31] with the complete linkage method, and Levenshtein distance [22] as the distance metric, on the 150 messages to obtain a hierarchy tree. We empirically found a Levenshtein distance of 7 to be the right cut-off point to create the clusters. We verified the correctness of the clustering by manually checking the message clusters.

Next, we selected the first message of each cluster as the message template. For example, instead of the actual template "Currency {0} is not supported" the message template found using clustering could be "Currency EUR is not supported". This does not pose a

problem, as the human aspect of the analysis can abstract away the specifics by using the context and semantics of the message. In case of doubt the source code can be consulted to remove discrepancies.

Using the combination of messages templates from the source code and hierarchical clustering, we obtain 146 unique error messages from 2.66 million API error responses. Next, we manually analyze the messages to enable categorization.

5.2 Identifying unique faults

The next step is to place each of the error messages, or rather the underlying faults, in a category. To enable accurate categorization we first add context to remove ambiguity. Second, we prioritize on impact to optimize the use of resources. Next, we gather information to support the analysis. Finally, using this knowledge we annotate the error messages within the given context.

5.2.1 Removing ambiguity

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 refer to the combination of an error message, API method and web service as an *error tuple*.

Note that the additional context reduces ambiguity, but does eliminate it. A unique message given the API context can have different interpretations depending on the situation. In Section 5.2.3 we explain how we try to mitigate the problems that could arise because of this. Using the approach of adding context to 146 unique error messages, we end up with 363 error tuples to annotate. In other words, the average error message occurs in at least two distinct relevant contexts.

5.2.2 Prioritization

We prioritize based on impact, as manual analysis is time intensive, by selecting a subset of the error tuples. We count the number of errors that occur per error tuple as well as the number of API consumers that is experiencing the error in the given context. The number of errors is not a reliable metric to base the prioritization on as we found that some consumers retry failed API requests on a highly frequent basis distorting the view of what is important. For this reason we use the number of API consumers that is experiencing a specific error as

a way to prioritize the error tuples. We consider every error tuple that impacts 10 API consumers or more. Using this approach we select 47 of the 146 unique errors, while covering approximately 91.3% of the 2.66 million erroneous API responses. This corresponds to 89 of the 363 error tuples. The prioritization is visualized in Figure 5.2.

Due to the filtering, 99 unique errors are not analyzed. As illustration we give four messages that were therefore ignored.

1. *"197 - This bank does not accept SEPA Direct Debits"*
2. *"198 - Reference may not exceed 79 characters"*
3. *"0 - java.net.SocketTimeoutException: Read timed out"*
4. *"171 - Unable to parse Generation Date"*

These errors may indicate a problem for the API consumer, however we do not pursue them further.

5.2.3 Annotation

In the annotation process we consider the unique error messages one by one. For each error message we select all related error tuples. Because these error tuples are related this speeds up the analysis process as we consider them together. Per error tuple we prioritize on the three consumers that are experiencing the most errors. For these consumers we dive into the request data to identify the underlying fault. We do this using the following information sources: request data, logs of the API request, API source code, technical support and platform developers.

For each error tuple we record one or more explanations of the error, given the context, which describe one or more faults. In addition, we add the stakeholder that caused the erroneous response to be returned by the API. The possible stakeholders we distinguish are the *end user*, *API consumer*, *API provider* and *third parties*, as explained in Section 3.3. The stakeholder information will give additional context to the faults and the corresponding categories. We identified 69 unique faults by annotating the 89 error tuples, illustrated by Figure 5.2. For each of the 69 faults, we wrote one or two sentences capturing the essence of the fault. These fault case descriptions are listed in Appendix A.

5.3 Data set time window justification

We selected error tuples for analysis based on the number of impacted API consumers being equal to or higher than 10. To show that the 28 day data set used for analysis covers a sufficient time window we compute the number of error tuples that satisfies the prioritization constraint for different time intervals. To this end we take 28 subsets of the entire data set, each corresponding to a different time window. The first subset contains 1 day of API errors logs, the second subset contains 2 days of API error logs and the nth subset contains n days of API error logs. For each subset we extract the unique error messages, remove the ambiguity, prioritize based on the number of API consumers and count the number of

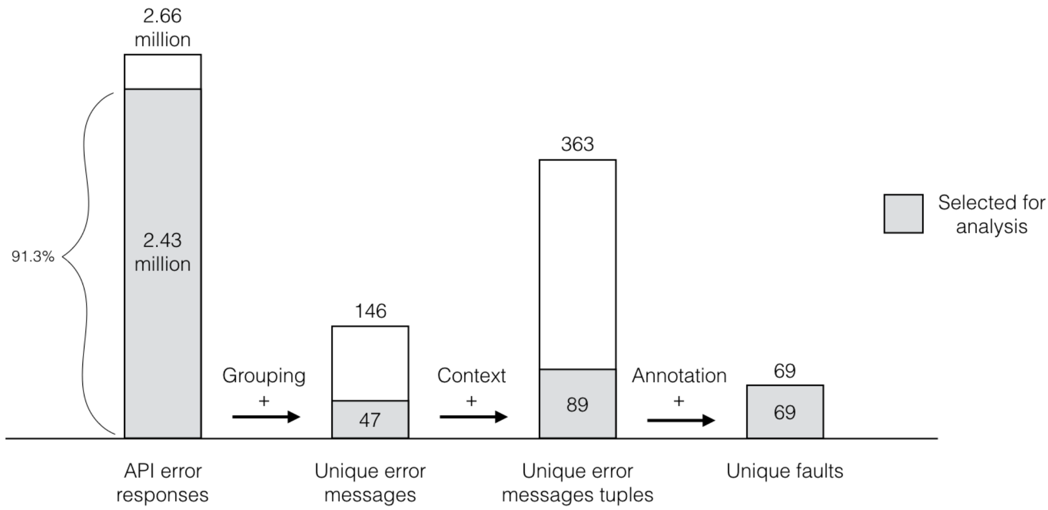


Figure 5.2: The stages of identifying unique API faults from the set of API error responses. That what is analyzed after prioritization is highlighted in grey.

error tuples that meet the constraints. For one day of API error data this resulted in 34 error tuples. Next, for each subset, we calculate the number of new error tuples identified compared to the previous subset. The resulting differences between the subsets are shown in Figure 5.3.

We find that after 14 days of data at most 2 new error tuples are identified per day with the number decreasing as more days pass. For this reason we conclude that the most common and therefore impacting error tuples in our data set are discovered within 14 days and therefore consider the 28 day data set to cover a sufficient time span. We used the entire data set to benefit from all available information.

5.4 Data set for quantitative analysis

In Section 6.1.2, we categorize the 69 faults. To this end, the error tuples are extended with a category for each fault. Although the error tuples help to avoid ambiguity, it cannot be completely eliminated. For this reason we apply estimations to obtain a data set for analysis. There are two situations on an error tuple level that introduce ambiguity by having two or more explanations for the error tuple. In the first case there are multiple categories assigned to an error tuple, caused by two or more different faults. In the second situation, there are multiple stakeholders that belong to two or more different faults. For these error tuples there is no effective way to say what error messages belong to what fault and hence to which category and stakeholder they belong. The only way to be sure is to analyze all errors manually, which is unfeasible considering the millions of API responses. Instead of a single number we provide a range of the number of API errors for the aggregations. For the number of errors per category for instance, in case of error tuple ambiguity, the

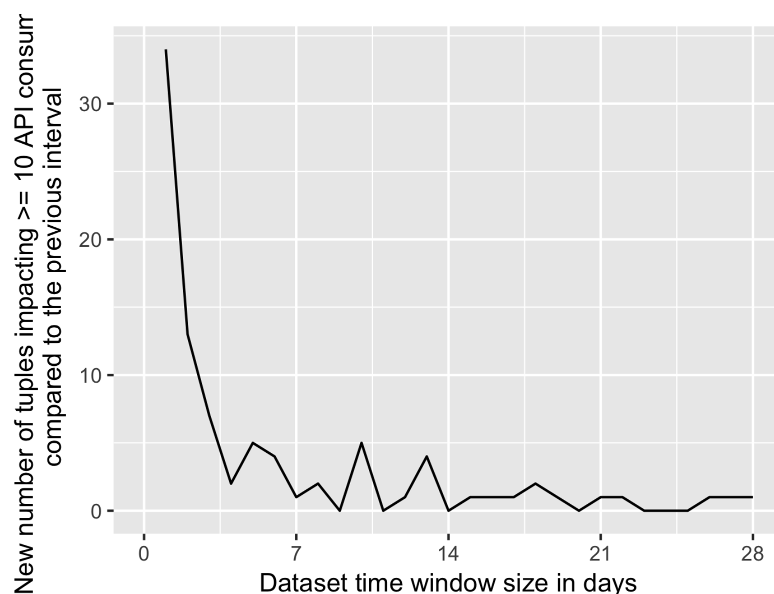


Figure 5.3: The number of newly identified error tuples for each interval for compared to the previous interval.

lower bound of this range is based on the assumption that zero of the errors belong to that category. Similarly, the upper bound assumes that all errors belong to that category and not to the other categories of that error tuple.

The range for the number of impacted API consumers is created in a similar manner. For example, an error tuple contains category X and category Y, and we are estimating the number of impacted consumers for category X. Furthermore, assume that n API consumers are impacted by category X in other error tuples. The lower bound assumes that no new API consumers in the error tuple, with category X and Y, are impacted by category X faults. In other words, after including the error tuple there are still n API consumers impacted by category X faults. The upper bound assumes that all new API consumers in the error tuple, with category X and Y, are impacted by category X faults. In other words, after including the error tuple there are n plus the number of new API consumers impacted by category X faults.

Chapter 6

Faults in API integration

In this chapter we analyze the faults in API integration that occur in the system under study to understand what types of faults occur, based on the data previously extracted. Furthermore we investigate the frequency of occurrence for these fault types and the impact in terms of number of API consumers.

In Section 6.1 we describe the data set used for analysis and the categorization approach. Next, Section 6.2 lists and explains the identified categories. Finally, in Section 6.3 quantitative measures in terms of errors and impacted API consumers are given for each stakeholder and category.

6.1 Methodology

RQ 1: *What type of faults, resulting in API errors, are impacting API consumers?*

To answer this research question we select a data set of API fault data from Adyen's API. Next we categorize the API faults to obtain a set of general fault types that describe the specific faults.

Consumers of a relatively simple API will probably not experience the same variety of problems that consumers of a more complex API experience. Analysis on a more complex API will therefore more likely result in more complete results. For this reason we decide to base the analysis on Adyen's Web API.

The Adyen API provides over 20 different methods, with the number of parameters going up to 35 for the *authorise* method. Some parameters are always required, some are sometimes required, and some are optional. Parameter formatting constraints include matching ISO standards, regular expressions for email and telephone numbers, predetermined string values or Adyen specific objects with their own parameters. The number of documented API errors is over 150 [2]. Furthermore the API is used by thousands of API consumers, collectively responsible for millions of API requests on a daily basis, among which are over 60 thousand errors. The described properties and usage statistics make the Adyen API in our opinion a useful subject for API integration problem analysis.

In Section 6.1.1 we describe the data set used as basis for the analysis. The process of fault categorization and category verification is explained in Section 6.1.2.

6.1.1 Data set

To obtain the categories we use a data set containing 69 unique faults, explained in detail in Appendix A, which we identified in Chapter 5. The data set corresponds to 4 weeks of API error responses from the system under study. For each fault the number of occurrences and the number of impacted API consumers is available. In total the data set represents approximately 2.43 million API errors impacting a total of 1464 unique API consumers. Chapter 5 describes the process of obtaining this data set from the API errors of the system under study in detail.

6.1.2 Categorization of unique faults

Identifying categories and assigning them to each 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 categories, which can be assigned to most of the annotated faults. During further analysis, if fault does not fit into one of the existing categories we define a new category. A category that is too generic may have to be split up into two or more categories, while a category that is too specific may be joined with another category. 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 category to each fault we iterate once more over all faults to check that all categories are accurate. Following this procedure we obtain a set of categories that describe faults in API integration in general.

To verify the accuracy of the categorization an expert is asked to check a subset of the category assignments. In case the expert does not agree with the category assignment the difference is discussed until agreement is reached. Using this approach we verify that the categories are understandable and reduce the possibilities of mistakes.

6.2 Fault types in API integration

By categorizing the API faults, 11 categories were identified. These API fault categories are shown in Table 6.1, each of which can be contributed to one of the stakeholders in the API integration environment. Two categories, related to user input, can be contributed to the end user, who is also responsible for *expired request data* faults. Most categories are caused by the API consumer, and the API provider and the third party stakeholder each match one category. Appendix A describes the fault cases used to determine the categories. Most categories have 2 to 3 fault cases, one has just 1, and four have between 12 and 17 faults each. Table 6.2 lists the fault cases per category. Next, we describe the relevance of each category and illustrate each category with faults in integration in the system under study.

Stakeholder	Category	Explanation
End user	Invalid user input	A fault introduced by invalid input by the end user of the application
End user	Missing user input	A fault introduced by missing input by the end user of the application
End user	Expired request data	The input data was no longer valid at the moment of processing
API consumer	Invalid request data	A fault introduced by invalid data caused by the API consumer
API consumer	Missing request data	A fault introduced by missing data caused by the API consumer
API consumer	Insufficient permissions	Not enough rights to perform the intended request
API consumer	Double processing	The request was already processed by the API
API consumer	Configuration	A fault caused by missing/incorrect API settings
API consumer	Missing server data	The API does not have the requested resource (e.g., document or object)
API provider	Internal	A fault caused by the API
Third party	Third party	A fault caused by a party integrated with the API

Table 6.1: The categories of API faults including the related stakeholder.

Category	Fault cases
Invalid user input	FC6, FC15, FC16, FC19, FC22, FC23, FC33, FC34, FC35, FC36, FC37, FC45, FC63, FC64
Missing user input	FC3, FC4, FC5, FC7, FC8, FC9, FC17, FC21, FC38, FC61
Expired request data	FC67
Invalid request data	FC1, FC13, FC14, FC20, FC29, FC30, FC31, FC32, FC41, FC43, FC44, FC47, FC51, FC54, FC55, FC66, FC68
Missing request data	FC10, FC25, FC27, FC28, FC39, FC46, FC48, FC52, FC58, FC59, FC60, FC65
Insufficient permissions	FC40, FC49, FC50
Double processing	FC11, FC18, FC69
Configuration	FC26, FC53
Missing server data	FC56, FC57
Internal	FC2, FC12, FC62
Third party	FC24, FC42

Table 6.2: The fault cases described in Appendix A that fall in each category.

6.2.1 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. Making a typographical error can cause the input to be of invalid length. Within the Adyen API we see that shoppers accidentally type an extra number for their credit card number. Using an incorrect input format where for instance letters are used instead of numbers can also result in an error. *Invalid user input* can also be caused by a user that is not aware that certain input is not allowed. A user may want to ship something to their house, while there is no delivery possible in their area, causing an error. Note that this category is not applicable when there

is no end user that supplies input via the API consumer.

6.2.2 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. An example of an error caused by *missing user input* is when the shopper neglects to fill in their CVC code or part of their billing address. Note that this category is not applicable when there is no end user that supplies input via the API consumer.

6.2.3 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. 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. *Expired request data* faults can be attributed to the end user.

6.2.4 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. One reason is an incorrect input format, such as supplying a floating point number, while an integer is expected. We discovered authorize requests where the amount was 14297.9999999 instead of the likely intended 14298. This is similar to the mistake made by the shopper causing an *invalid user input* fault, however in this case caused by the API consumer.

Another common mistake made is not using proper encoding before sending the request. Forgetting to URL encode parameters that contain the &-symbol will cause API parsing faults.

The previous mistakes can be contributed to mistakes made on the API consumers side because of a bug or forgetting to implement a detail. However, there are also mistakes caused by incorrect assumptions, where the API consumer assumes a certain action works with the given input, while the API does not support the functionality or cannot handle the input. For instance, we found a merchant attempting to collect recurring payments using SEPA Direct Debit using Swiss Francs, which is not supported.

6.2.5 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. *Missing request data* be attributed to unawareness of the API specifications or a bug. For instance, we identified faults caused by payment requests that were missing the amount of the transaction or did not specifying the merchant account to be used. Similarly, a merchant attempted to create recurring contracts for its shoppers, but forgot the required shopper reference field, causing the request to fail with an error.

6.2.6 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. In other cases, users have part of their system still interacting with the API, while their contract has ended and therefore their permissions have been revoked.

In the previous examples the API consumers is not allowed to use any services. At Adyen, companies can have multiple API user accounts, each with different permissions. Accounts configured for Point of Sale transactions that are used for Ecommerce will result in *insufficient permissions* faults.

6.2.7 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. In the Adyen system recurring shopper contracts can be disabled using the disable method. When a merchant attempts to disable a contract that is already disabled an error is returned related to this fault type.

6.2.8 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. The Adyen API, for instance, returns an error indicating a *configuration* fault in case a merchant attempts to make a transaction for a combination of a payment method and currency for which there is no configuration. This configuration is needed to determine which acquiring bank will acquire the funds from a shopper for the given payment method and currency.

6.2.9 Missing server data

Missing server data faults are the result of incorrect management of data that exists in two places. The API consumer in this case is managing identifiers of resources accessible via the API. Inconsistency is introduced when the API consumer does not update changes to resources correctly. For instance when a resource is removed or updated.

An example fault that arises in the Adyen API is a recurring contract that cannot be found. In this case the merchant often manages the references to contracts of shoppers locally, instead of requesting the latest contract details for a shopper by making an API call. In certain situations Adyen updates the contract and updates the reference. If the merchant is unaware of this situation and does not update the reference, the next request will fail. An example situation occurs when the shopper makes a change to their billing address.

A similar fault occurs when the API consumer sends in an identifier, which never existed in the first place. This can be contributed to a bug in the consumer's system.

6.2.10 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. In addition, we regard API downtime as an *internal* fault, which causes API error responses for all incoming requests.

In the Adyen system an unexpected situation arises when a new recurring contract is created in one call and within a few minutes a recurring transaction is attempted with that contract. Due to a delay in database replication, the contract is not found at the moment the recurring transaction is sent in, which we attributed to an *internal* fault.

6.2.11 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, which does not respond or returns an error. In this case the request failed and the consumer is notified by means of an error. The API requests that result in this type of error may yield a different result in case of a retry, in contrast to for example errors corresponding to *invalid request data* faults. The latter request will fail every time simply because the format is not correct.

As an example, for an Adyen merchant a *third party* fault situation arises when the iDEAL service is used and one of the underlying banks does not respond. Another situation that causes a *third party* fault is when the system is unable to handle an encrypted response from a third party, which is not formatted according to the implemented specification.

Note that this category is not applicable when there is no third party involved.

***RQ 1**: What type of faults, resulting in API errors, are impacting API consumers?*

Based on the API fault data, faults in API integration can be grouped in 11 categories: *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 category can be contributed to one of the four API integration stakeholders: *end user*, *API consumer*, *API provider*, and *third parties*.

6.3 API integration fault prevalence

We analyzed the frequency of faults from two perspectives: a stakeholder perspective and a category perspective. The stakeholder perspective explains the origin of the fault in the integration chain, and the category perspective helps to understand the prevalence of faults on a category level.

6.3.1 Stakeholder perspective

Table 6.3 shows the number of unique faults resulting in an API error caused by each stakeholder. In addition the range of the estimated percentage of errors and impacted consumers is given. It is not possible to compute an exact number due to the ambiguity explained in Section 5.4. In some cases the range is extreme. For example the percentage of errors that we contribute to the API provider ranges from 0.1% to 4.9%. Although we are unable to provide an exact number, we do have an intuition about the true percentage of errors from annotating the errors by hand. To communicate this we embolded the bound which most closely matches our intuition of the exact percentage of errors and impacted consumers.

Note that the total percentage for the lower and upper bound for the errors does not add up to 100% due to estimation. We therefore omit the total from the table. The percentage of consumers is based on 1464; the total number of impacted consumers. As consumers can be impacted by errors related to multiple stakeholders they can be counted multiple times, which explains why the sum of the percentages is well over 100%. A total percentage would therefore not make sense and is hence omitted.

We considered 2.43 million API errors and identified 69 unique faults. Approximately 86.3% - 87.7% of these errors, resembling more than 2 million occurrences, corresponding to 39 (56%) of the faults, are caused by the API consumers themselves. In other words, the majority of the errors is caused by a little more than half of the faults. The number of API consumers that is experiencing faults caused by themselves is high compared to the faults that the other stakeholders cause. Out of 1464 consumers experiencing faults 67.5% to 84.6% are impacted by their own mistakes.

The end user is responsible for between 7.4% - 12.3% of the errors, which relates to 25 (36%) different faults. The large portion of unique faults in this case relates to a relatively small number of errors, in contrast to the faults caused by the API consumer. After the

Stakeholder	**# of faults**	**Errors (%)**		**Consumers (%)**	
		Lower bound	*Upper bound*	*Lower bound*	*Upper bound*
End user	25	7.3	**12.4**	39.8	**50.7**
API consumer	39	86.3	**87.7**	67.5	**84.6**
API provider	3	**0.1**	4.9	**0.7**	18.9
Third party	2	0.4	**0.9**	21.8	**46.6**
Total	69				

Table 6.3: The number of faults per stakeholder, the estimated percentage range of errors and impacted API consumers. The percentages are based on 2.43 million API errors and 1464 impacted API consumers. Note that the percentages do not add up 100%.

API consumer, the end user is causing the most errors and impacting the most consumers: between 39.8% and 50.7%.

The API provider in the system under study is responsible for 3 (4.3%) of the faults. The number of errors related to these faults ranges between 0.1% and 4.9%. This large range can be contributed to an ambiguous error tuple that is explained by two faults; one caused by the end user and the other by the API provider. The large number of errors accompanying this error tuple causes the upper bound to be relatively high compared to the lower bound. Similarly the number of impacted API consumers varies greatly due to the ambiguity, ranging from 0.7% to 18.9%.

Finally, 2 (2.9%) faults are caused by third parties, which make up 0.4% to 0.9% of the API errors. Interestingly the two issues and the small number of errors are impacting between 21.8% and 46.6% of the consumers. Compared to the other stakeholders, the number of impacted consumers for these faults is relatively high.

6.3.2 Category perspective

In Table 6.4 we show the number of unique faults that occur in each of the 11 categories found during manual analysis. Similar to Section 6.3.1, due to ambiguity we are not able to present the exact percentage of errors and impacted consumers. For this reason for each category we show the estimated percentage of corresponding API error responses and estimated percentage of impacted consumers. In four categories of faults no ambiguity was present, hence the exact percentages are given instead of a range. In addition we embolded the bound that is closest to the exact number according to our intuition. Note that the total percentages for the lower and upper bound for the errors do not add up to 100% due to estimation, and are therefore omitted.

For the end user we see the input to be the largest cause of faults. 24 of the 25 faults are caused by invalid or missing input, collectively causing 7.2% to 12.5% 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 40.4% compared to 11.5% to 24.8%. *Expired request data* sent in by the end user causes one fault. The number of errors, 65 (0.003%), is relatively small compared to the number of impacted API consumers, which is 2.0% of the total.

Category	Stakeholder	# of faults	Errors (%)		Consumers (%)	
			Lower bound	*Upper bound*	*Lower bound*	*Upper bound*
Invalid user input	End user	14	6.5	**7.1**	33.3	**40.4**
Missing user input	End user	10	0.7	**5.4**	11.5	**24.8**
Expired request data	End user	1	0.0	0.0	2.0	2.0
Invalid request data	API consumer	17	**3.2**	10.5	**23.9**	62.3
Missing request data	API consumer	12	23.0	**28.7**	20.2	**24.0**
Insufficient permissions	API consumer	3	0.1	0.1	9.5	9.5
Double processing	API consumer	3	36.0	36.0	12.3	12.3
Configuration	API consumer	2	**16.7**	16.7	**19.9**	21.4
Missing server data	API consumer	2	1.5	1.5	13.9	13.9
Internal	API provider	3	**0.1**	4.9	**0.7**	18.9
Third party	Third party	2	0.4	**0.9**	21.8	**46.6**
Total		69				

Table 6.4: The number of faults per category 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 1464 impacted API consumers. Note that the percentages do not add up 100%.

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 *configuration* 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* category, with the difference that this category 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 categories for this stakeholder. The number of impacted consumers, however, compares to the categories.

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%.

> ***RQ 2**: What is the prevalence of these fault types, and how many API consumers are impacted by them?*

Based on the API faults data, from a stakeholder perspective most faults, 39 out of 69, can be contributed to the API consumer. This compares to 86.3% to 87.7% of 2.43 million API errors and between 67.5% and 84.6% of the 1464 impacted API consumers.

From a category perspective, most faults, 17 out of 69, can be contributed to the *invalid request data* category. However most errors, 36.0%, are related to *double processing* faults. Most API consumers seem to be impacted by faults in the *invalid request data* and *third party* categories.

Chapter 7

Illustrative API Integration Problem Cases

To develop an intuition and initial understanding of the current practices and challenges that API consumers experience we conduct three API consumer interviews based on three identified problem cases. The initial understanding is used as input for the API consumer survey described in Chapter 8. Furthermore, the three cases act as practical illustrations of API related problems experienced by API consumers and the accompanying challenges that are experienced.

To this end, we discuss an identified API related problem for the API consumer and elaborate on API integration in terms of the integration process, error handling and monitoring. Furthermore we detail the API consumers' suggestions for the API provider to help avoid API problems from occurring.

In Section 7.1 we describe the interview methodology that provides a structured way to extract information from API consumers. Sections 7.2, 7.3, and 7.4 elaborate on three problem cases based on interviews conducted with the respective API consumers.

7.1 Methodology

Analysis of the API error response data gives us a quantitative understanding of the faults that occur. To illustrate the need to understand API errors and reduce their impact we interview API consumers to learn about problem cases they experience related to API errors. Obtaining a qualitative understanding of these faults allows us to dive into the complexity of the problem, rather than abstract it away, making the results more informative.

The goals of these interviews are as follows:

1. Develop an intuition of why API problems occur. Specifically we are interested in what causes API problems, what their impact is, why they go undetected if this is the case, and what could have been done to prevent them.

2. Understand how API consumers work with APIs in terms of the integration process, testing, monitoring and error handling.

3. Identify the API consumer's opinion about what the API provider can do to help prevent problems.

4. Provide an illustration of API problems that API consumers experience.

The interviews are designed based on the work by Seaman [33], and Hove and Anda [18]. Seaman presents several qualitative methods for data collection and analysis for empirical studies of software engineering. Hove and Anda combine experiences from 12 software engineering studies, identify four areas of challenges and share advice and suggestions on how the quality of software engineering interviews can be improved.

In Section 7.1.1 we explain the process of selecting interviewees and describe them. Section 7.1.2 elaborates on the design of the interview and the questions. Next, in Section 7.1.3 the logistics and the process of conducting are explained. The interview questions are given in Appendix B.

7.1.1 Interviewee selection

Arranging interviews is a difficult process, especially in a business environment where time and money are important factors. Our interview goals also provide no direct reward for the API consumer, making it even harder to arrange them. For this reason our approach is to identify a previously unknown problem for the API consumer and help to solve it, so that the consumer is more inclined to speak with us after resolving the issue. While this is a time consuming approach for us, it makes reaching the first goal easier as this is focused specifically on API problems. After resolving the issue, the merchant will have the details of the problem fresh in mind and we will have the detailed knowledge about the problem which helps us ask the right questions.

Using the API error responses we identify API consumers that are experiencing errors. We select the occurrences that seem the most interesting in terms of possible impact and frequency. We reason that this will improve the response rate and likelihood of arranging interviews. Using the request data, request logs, source code of the API and the help of the technical support team we analyzed the errors to identify faults for API consumers. We selected 21 faults that had an understandable reason for occurring and likely were of significance to the API consumer. This turned out to be time consuming as it was often not possible to estimate the impact and because we did not want to contact Adyen merchants without a strong intuition of the problem and its impact. Next, we communicated these faults and related potential problems, impacting 18 different API consumers, to the respective Adyen account managers. We received a response from the account managers in 19 out of 21 fault cases. After discussing the faults and possible problems, 11 of them were further communicated to the relevant API consumers. Of the remaining faults, 2 were regarded as not an issue, for 1 the timing to communicate the issue was not right and in the other cases we are awaiting further response from the account managers. After informing the API consumers of the 11 faults and potential problems 6 were eventually resolved. We proceeded to explain our research and asked them to take part in an interview focused on the identified problem and their integration in general. Finally, 3 of the API consumers agreed to talk to us, resulting in 3 interviews.

As the goals of the interview are of technical nature we require the interviewee to have development knowledge or to be closely involved in the integration process. For the three interviews we describe the interview and interviewee below.

Interview 1
The interviewee in interview 1 represents a global e-commerce fashion website processing hundreds of thousands of payments every month. The interviewee is the payment and fraud manager in a six person team responsible for accepting payments and preventing fraud, and oversees the overall integration in terms of key performance indicators and anomalies. Although not a developer himself, the interviewee has in-depth technical knowledge about the integration and processes around it.

Interview 2
In interview 2 we spoke to two developers representing a transporting business processing hundreds of thousands of payments on a monthly basis. One of the developers had worked on most of the integration with the API. The other was relatively new and did not yet have a lot of experience with the integration.

Interview 3
The interviewee in interview 3 is a third party integrator for a merchant that processes tens of thousands of payments every month in the transport sector. The third party is responsible for the entire payment integration of the merchant.

7.1.2 Interview design

There are different types of interviews, each with their own characteristics and costs. In *structured interviews* the questions are asked by the interviewer and the response is given by the interviewee [33]. The interviewer has specific objectives which are translated into questions. In the extreme, no qualitative information is gathered and all responses can be quantified directly. *Unstructured interviews* on the other hand have as objective to obtain as much information as possible on a topic. The questions asked are open-ended and the conversation is an unstructured discussion.

We use a combination of the two types and conduct *semi-structured interviews*. Structured specific questions to adhere to the goals of the interview and open-ended questions that open up discussion to capture unexpected information. We start with background questions to learn about the characteristics of the interviewee and the payments team. Knowledge questions are asked to learn about the API consumer's testing process, error handling and monitoring. Follow-up questions aim to discover why certain processes are in place, or why they are not. We use reflexive questions to learn about the interviewee's opinion on how, for instance, the API provider can help to prevent problems.

The questions are designed not to be leading to allow the interviewee to answer based on their own thoughts. If the question was phrased such that it suggests an answer, the interviewee may be inclined to go with that answer and not think about other options. If we, however, during the interview feel that the answer to the question is insufficient or feel

the interviewee does not clearly understand we rephrase the question to include examples to get the conversation going. For instance, when we ask about the monitoring processes in place, we could follow-up by giving examples, e.g., logging and alerts, to get the thought process of the interviewee going.

The interview was designed in multiple iterations and altered based on discussions with colleagues to remove unclarities and to structure the interview better.

7.1.3 Conducting the interviews

Before conducting the interviews, the participants were sent an email with a short introduction to the topic and what the interview would be about.

The duration of the interviews varied from 30 to 45 minutes, depending how much the interviewee had to share. At the beginning of the interview, held over Skype and taken by one interviewee, we gave another short explanation about the topic and the usefulness of the interview. We explained that both the interviewee and the company will remain anonymous to make them feel comfortable to speak freely and share information. In addition we explained that the interview is split into one part focused on the identified issue and one part designated to the API integration in general.

We used an interview guide during the interviews to structure the conversation. We included several conditional follow-up questions to be asked based on the answers of the interviewee. If, for instance, the API consumer has testing in place we asked about the reasons for this to be integrated, or when this is not in place we asked why this is not the case. The interview guide is included in Appendix B.

The interviews were recorded to avoid loss of information. In addition it allowed the interviewer to be more focused on the answers and to identify possible follow-up questions, instead of spending time taking notes.

As expected, during the interviews we would wander of and end up discussing other parts of the interview. We would not cut the interviewee off to revert to the original question, which could result in information to be missed. Instead we would let them finish talking and kept track of which questions were answered and which were not, and steered back appropriately to the subject of the original question.

After conducting the interviews we summarized the conversation in writing. Interesting phrases were transcribed to be used in the results as quotes. We noted the time at which the interviewee started to answer a question. In case more information was required we could easily navigate to the appropriate time in the recording.

7.2 Case 1: Unhandled contract update

The first case we discuss is about incorrect management of references on the API consumer side. The API could in this case not locate the contracts that the references point to, resulting in a *missing server data* problem.

We first describe the merchant experiencing this problem, after which we explain the problem, its impact and the cause. Next we explain how the merchant experienced the integration process, handles API errors, verifies the correctness of the integration and how

they monitor the integration. Finally, we elaborate on the suggestions that this merchant has for the API provider to reduce the impact of API problems.

7.2.1 API consumer description

The API consumer in this case is an e-commerce fashion website selling clothes and footwear for men and women. They offer more than 100,000 items through their webshop and processes hundreds of thousands of payments every month. Their payment processing is handled in-house by a team of six people, among which are four developers, one quality assurance engineer and one person overseeing the overall integration in terms of key performance indicators and anomalies.

7.2.2 Problem description

The merchant in this case makes use of Adyen's one-click payment solution. This allows shoppers to make transactions with a single click, without having to re-enter credit card details for every purchase, using a recurring contract set up in advance. Instead of supplying the credit card details, the merchant can send in the reference to the contract to be used for that shopper. Such a contract can be set up in two ways. One is to store the credit card details after the shopper makes an initial payment and the other is to import credit card contracts in bulk during a migration to Adyen. Recurring contracts are updated when the recurring payment request contains details which are different from the details stored in the contract. For instance, when the billing address of the shopper changes, the contract is updated and a new reference is returned. Subsequent one-click payments should be made using this new reference.

What happened is that this merchant was not updating the references as they changed. Hence subsequent one-click payments failed with API error message "PaymentDetail not found", corresponding to FC57 in Appendix A. Updates occurred to contracts that were imported, but did not have a billing address stored. After the first successful one-click payment, which included a billing address in the request, the contract was updated and the second one-click payment would fail. Another reason was a change in the existing billing address, which triggered the contract update as well, resulting in the same problem.

This problem resulted in about 1000 one-click payments failing every week around the time of detecting the problem. In the 13 months the problem existed over 40,000 payments failed, causing Adyen to not process 2.5 million euros. Compared to the total number of payments processed for this merchant, between 1% and 1.5% of them failed due to this problem. Fortunately, the merchant had a backup mechanism in place that retries every failed payment with another party providing services similar to Adyen's. However, the merchant mentioned that this situation was not ideal as Adyen's services result in higher revenue.

The cause of this problem was that the merchant was internally duplicating the credit card profiles inside the system for some time around transactions. They would then update one profile, while using the other. The updated profile would later be discarded, keeping the outdated profile in the system. A system was built that tokenizes credit cards that could

be used by two different gateways. The merchant mentioned this may have opened up the window for the details to be duplicated.

The merchant is unsure how this issue could have been prevented. The complex scenario was something that they could not reasonably test or they would have to create thousands of tests for these types of complex scenarios. The issue was not detected, because it did not impact the authorization rates, as they were retrying the payment using another gateway.

7.2.3 The integration process

During the integration process the merchant experienced problems with the documentation. "*What the documentation was showing, was not exactly what had to be used.*" The support team was able to help them out quickly, but outdated documentation cost them a lot of time. The merchant mentions accurate documentation has a lot to do with credibility: "*The bad thing is, at the end of the day, if the developer thinks the documentation is not up to date they will more often think things are wrong because the documentation is out of date, instead of something being their own mistake.*"

7.2.4 Error handling

Most of the unique error responses that Adyen returns are not directly mapped to specific error handling logic. Rather, the merchant's system understands that something went wrong and they automatically resort to their backup gateway. Only when payments also fail using the backup gateway they investigate further. "*If at the end of the day it is not impacting customers we tend not to investigate specific errors.*" The merchant acknowledges this is not an optimal scenario: "*I think we took the easiest path, maybe not the smartest path.*" However, they mention that trying to map all errors requires development time which will not be prioritized due to a huge backlog.

The merchant prefers the API to return many specific errors instead of fewer general errors. The specific errors allow for a better understanding of what is going wrong, while a generic error would be confusing. One way to improve would be to make sure that the errors that are returned describe the right problem. The merchant mentioned spending a great deal of time debugging an error stating that no expiry month was set, while in reality they were trying a transaction without recurring token. This issue corresponds to FC21 in Appendix A.

7.2.5 Verifying integration correctness

To verify the correctness of the integration this merchant makes automated transactions for every payment method they support. This poses some issues, as the cards are blocked from time to time by the issuer, because of the suspiciousness of small repetitive payments. Also, the payments team closely collaborates with the customer service in case of payment problems for customers. The merchant however mentions that this situation is not ideal, because in essence the customers are doing quality assurance.

7.2.6 Monitoring

Monitoring is focused on transactions that are not completed. "*That is what worries us the most.*" The merchant compares their authorization rates to the rates of the previous weeks and investigates if there are major differences. In this case alert texts are sent to the people on duty. Furthermore, using Adyen's reporting of transactions filtered by fraud the merchant detects fraud attacks or risk settings that are not functioning properly.

By periodically extracting error data from their database and using this data to build dashboards, the merchant looks for strange behavior that was missed in the big picture reports. They then do root cause analysis on the errors and patterns they found. They mention that the extraction and analysis is a manual process, influenced by the person that does the assessment, and therefore is not optimal.

7.2.7 Suggestions

The merchant suggests that the API provider, Adyen in this case, can help prevent integration problems by alerting the API consumer when detecting strange patterns. When the updated contract problem was communicated to them it drove them to take action: "*We were postponing this. We knew that this was happening. Let's try to understand exactly what is happening here.*"

The merchant mentions that people interested in investigating issues often do not have the knowledge to go into the server log files and discover what is going on. To investigate an issue the payment expert has to ask a developer to go into the logs and retrieve the information. If the API provider would make the errors they return accessible, including the corresponding requests data, this would enable non-technical people to more easily look into strange patterns and check whether the behavior is expected. "*There is a cost involved in where you have to stop what you are doing and access all of these different servers and all these different logs.*" Insights would make the investigation of issues less costly.

7.3 Case 2: Insufficient permissions in chained API calls

The second case is about a permission problem in API calls that are related, where the merchant was using different credentials for the separate requests resulting in an *insufficient permissions* problem.

We first describe the merchant experiencing this problem, after which we explain the problem, its impact and the cause. Next we explain how the merchant experienced the integration process, handles API errors, verifies the correctness of the integration and how they monitor the integration. Finally, we elaborate on the suggestions that this merchant has for the API provider to reduce the impact of API problems.

7.3.1 API consumer description

The merchant in this case offers transport services in multiple international markets. On a daily basis the company facilitates tens of thousands of transport connections and on a

monthly basis the merchant processes hundreds of thousands of payments using the Adyen platform. The payments team consists of four developer focus on the API integration and one product manager. There is another team of developers that works on their mobile payment solution.

7.3.2 Problem description

Certain API calls may be dependent on previous calls. Such a situation arises when merchants want to process payments using an extra layer of security by authenticating the shopper's identity. This fraud prevention scheme, called 3D Secure[1], requires an initial payment authorization request to the Adyen platform. Adyen returns a URL pointing to the issuer's website which the shopper should be redirected to for authentication. Additional request data and a payment session identifier are included in the response. After the shopper has authenticated themselves, the merchant is to forward the response data from the issuer to Adyen in the 3D Secure authorization request. Adyen then matches the session identifier and processes the 3D Secure payment.

Adyen allows for multiple accounts to be set up per company, with separate API consumer credentials. A problem occurred for this merchant because they were sending in the linked 3D Secure requests using different credentials, resulting in API error message "Invalid Merchant Account". This problem did not take place in the time period corresponding to our API response data set on which our analysis is based. During this time no similar problem occurred and therefore this problem is not on the list of problems described in Appendix A. This poses a threat to validity which we discuss in Section 10.2.

Around 300 transactions a week were failing because of this problem. For this merchant the impact is relatively small, however could impact the customer satisfaction and increase the load on the merchant's support team. Due to infrastructure limitations we were unable to identify how long this problem has been taking place.

The merchant is using different shops, which all use different credentials to do transactions. In some cases however, the shopper would change websites to complete the 3D Secure payment. Because the different websites use different credentials the subsequent request failed, causing this problem to occur.

As the number of occurrences of this problem was low it did not influence the merchant's monitoring dashboards. Therefore the merchant was not aware of this problem until we notified them about it. The merchant mentioned that they were unaware that this situation would be a problem as the Adyen documentation did not mention this explicitly.

The issue was resolved by making sure the accounts used in the chained requests are the same. Adyen has updated the documentation to clarify the need for the credentials to match in case of the 3D Secure payments.

7.3.3 The integration process

During the integration process the merchant made primarily use of the API documentation. However, the biggest bottleneck for them was missing documentation. As an example

[1] http://www.mastercard.com/gateway/implementation_guides/3D-Secure.html

the merchant explained that there was no information about the Ratepay payment method. Therefore they had to integrate directly into their test system to be able to verify that the logic was working correctly, while contacting our support to find out how the integration was supposed to go. The merchant stressed, that if the documentation was more complete, problems like the one described could be prevented.

7.3.4 Error handling

The merchant is using a combination of handling specific errors and handling using fallback logic for unmapped responses. The errors that are mapped are translated to responses for shoppers to notify them of what happened. When the CVC of the credit card is wrong the shopper gets a message saying the payment information is incorrect.

7.3.5 Verifying integration correctness

Besides manually testing scenarios during development the only API related tests in place are related to the structure of messages. The merchant has unit tests in place to verify the structure of the messages that are sent to the API. If there are anomalies in the dashboards, described next, or they receive feedback from customer support they start a process of identifying what is going on.

7.3.6 Monitoring

The merchant monitors their integration using dashboards that show the number of successful transactions. In addition there is a system that compares the number of transactions to previous days to check for anomalies. There are multiple dashboards on a detailed level, which show the transactions that have been completed, those that are refunded and those that are rejected. Specific errors are however not monitored. The merchant keeps a log of the requests and errors, but only use this for root cause analysis. "*If we don't notice the problem, we don't know what we are looking for.*"

7.3.7 Suggestions

The merchant mentions that the error responses from the API are not enough to help prevent problems with the integration. They suggest notifications in case anomalies in requests happen by means of email notifications or a dashboard. What they would like to see is the reason of the anomaly and a list of payments that are failing because of the errors.

7.4 Case 3: Invalid encryption key

The case described in this section is about an invalid encryption key problem. The merchant in this case seems to encrypt the credit card credentials of a subset of the transactions using an incorrect encryption key, causing these payments to fail.

We were unable to record the complete story behind this case, because at the moment of interviewing the problem was not yet investigated and at present it has not been resolved.

We first describe the merchant that is experiencing the problem. Next, we explain the problem, the suspected cause and impact. Finally, we describe how integration was experienced, how errors are detected and monitored and we name the suggestions for improvements on the API provider's side.

7.4.1 API consumer description

The merchant in this case is active in the transport sector in a single domestic market. On a monthly basis the company transports hundreds of thousands of passengers and processes tens of thousands of payments using the Adyen platform. The integration for this merchant was outsourced to a third party integrator that is responsible for the implementation and verification of the API integration.

7.4.2 Problem description

To handle credit card information a merchant has to be PCI DSS[2] (Payment Card Industry Data Security Standard) Level 2 compliant. For merchants that are not compliant on this level Adyen offers a client-side encryption library that encrypts credit card information locally on the shopper's device using a merchant specific public key. The encrypted data is then sent in encrypted form through the merchant's server to the Adyen platform. Adyen decrypts the data and processes the payment using the supplied card data. The public-private key pair can be regenerated by the merchant if necessary. Adyen will use the latest private key to decrypt the data.

The merchant in this case is experiencing errors indicated by the "Unable to decrypt data" message. Adyen is unable to decrypt the supplied credit card information and this causes the respective payments to fail. Although the actual problem has not been identified yet it seems that the merchant is using an outdated public key for a subset of the transactions, which corresponds to FC61 in Appendix A. Namely, only part of the transaction using the encryption functionality failed with this error. Also the portion of failing transactions is too large to be contributed to a possible internal problem of the API that also causes this error (FC62 in Appendix A).

The problem results in several thousands of failed authorization requests each week which corresponds to about 12.5% of the total number of requests. Because the data related to this error is not readily available we cannot determine for how long this issue has been taking place. We do know that this is at least 6 months, which means that tens of thousands of payments have failed over this period for this merchant. Since we were unable to follow-up on this problem the actual impact of this problem cannot be determined. If the merchant has a backup payment mechanism in place it could be that the transactions were eventually processed, but if this is not the case it could be that these failing transactions have a significant impact on the revenue of this merchant.

The integrator mentioned that the merchant has different checkout channels. This causes us to suspect that in one of these checkout channels an incorrect public key is being used. We were however not able to verify this as we received no reply to our followup inquiries.

[2] https://www.pcisecuritystandards.org/

7.4.3 The integration process

The third party integrator mainly used the API documentation provided by Adyen. They did not resort to GitHub or other sources to learn from code samples, but when in doubt contacted the technical support team. Integration was experienced as very easy. The integrator liked the documentation and needed only two days to complete all the required work. Compared to competitors the Adyen integration was experienced as easier to do.

7.4.4 Error handling and monitoring

The integrator did not go into details on the error handling mechanism, but explained that they are responsible for the handling of any errors that the API returns. Right after explaining this the integrator said that they do not have any statistics on the error rate and do not monitor the integration.

7.4.5 Suggestions

The integrator was overall pleased. "*Adyen is already helping by reaching out and pointing out the errors.*" When we asked whether insights provided by the API provider into these errors would help them, the integrator mentioned that since they are not monitoring it would be very helpful if the API provider implements this on their side.

Chapter 8

API Consumer Perspective

The API consumer interviews provide useful case studies that illustrate the faults and problems API consumers face. However, the limited number of interviews is insufficient to say much about faults and problems in API integration in general. Hence, we conduct a survey among API consumers to strengthen the fault category results, and to test the findings outside the scope of the system under study. In addition, we aim to understand the current practices and challenges of reducing the impact of problems caused by faults in API integration.

Section 8.1 explains the methodology used to verify the categories and understand the practices and challenges by means of a survey. In Section 8.2, the findings related to the category verification are given as well as the impact of these fault types, and Section 8.3 elaborates on the practices and challenges in reducing API problem impact.

8.1 Methodology

In Chapter 6 we identified 11 fault categories to answer ***RQ 1***. We challenge these categories 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 category whether they have experienced a problem. Furthermore, we allow them to name additional categories to capture the ones we may have missed.

To complement the quantitative findings of the frequency of occurrence found in Chapter 6 to answer ***RQ 2***, the respondents are asked to rate the frequency of occurrence for problems in each category. This provides an angle of insights that is different from the system under study.

RQ 3*: What type of faults do API consumers consider the most impactful?*

To answer this question, we ask API consumers for each fault category how the impact of related problems on their API integration is experienced.

RQ 4*: What are the current practices to avoid and reduce the impact of production problems caused by faults in API integration?*

RQ 5*: What are the current challenges to avoid and reduce the impact of production problems caused by faults in API integration?*

We answer these questions by asking the participants about five topics. 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 face 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 sees to improve and why this is not in place, and 5) the API consumer's idea of what causes faults and related problems to occur.

In the remainder of this section we elaborate on the survey design and the sampling process, which can be used as a structured way to survey API consumers. Surveys are commonly used to gather information about processes, people or products. Although constructing a survey by putting together interesting questions seems straightforward and easy, the process of setting up a survey requires careful thinking in terms of objectives, design, participants and result processing [10, 16, 20]. For our survey we use the principles of survey research as proposed by Kitchenham and Pfleeger [20] in six parts.

Section 8.1.1 describes the target audience. Section 8.1.2 explains the overall design to conform to best practices, but also to make sure we obtain relevant results. Section 8.1.3 elaborates on the question design to target the intended audience and obtain the relevant information. The survey evaluation is discussed in Section 8.1.4 and finally, the sampling process and respondents are described in Section 8.1.5. Finally, Appendix C contains the complete survey as sent to the participants.

8.1.1 Target audience

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.

8.1.2 Overall design

Next, we aim to understand the API that the participants integrated with. We expect developers to handle and detect errors in hobby projects differently compared to professional applications. For this reason ask the participant to categorize the application using the API as a *professional*, *hobby*, *research* and/or *open source* application. Furthermore we ask the participant what the API does and what the number of developers is that worked on the integration. The participants are asked to rate the API's complexity based on their opinion in terms *different feature and functionality*, *required/optional parameters and value constraints*, *the number of possible error scenarios* and finally the *overall* complexity. The response is captured on a five-point Likert scale [23] with symmetric equidistant categories and a midpoint. This allows us to treat the information on an ordinal scale. Note that with these questions we do not exclude any participants based on how complex they consider the API to be. Developers that only have experience with simple APIs can still answer the questions. The contextual information however allows us to distinguish between responses during analysis based on these metrics.

We are interested in the current faults that occur in API integration and the practices in this area. To this end we ask our participants whether they have experience with API integration in the last 3 years. If this is not the case they are thanked for their participation and disqualified from taking the survey.

To motivate people to answer the survey we supply three key pieces of information as suggested by Kitchenham and Pfleeger [20]: the purpose of the study, the relevance to the participant and how confidentiality will be preserved. Furthermore, we offer the option to be notified of the results of the study via email to encourage people to take the survey as there is a reward for them.

To avoid bias in the survey we tried to use wording that does not influence the way the respondent thinks about questions and the corresponding answers. For instance asking "Do you think this idea is good?" introduces bias as it suggests the idea to be good. Also, we thought about the order of the questions, such that the answer to one question does not influence subsequent responses to other questions.

To capture any comment on the survey itself or the topic the participant has the option to include feedback at the end of the survey. This allows us to identify possible problems with the survey or can point us to thoughts or ideas we have not come up with.

8.1.3 Question design

The questions are designed to directly relate to the survey objectives, while making the objectives measurable. Kitchenham and Pfleeger [20] recommend questions to be purposeful, meaning the respondent can see the relationship between the survey objectives and the intention of the question. They argue that if the purpose of the question is unclear, the participant may ignore the question or provide a less thoughtful response. To this end we describe the intentions and the goals of the survey at the beginning. Also, the questions are to be concrete; precise and unambiguous. To conform to these recommendations we include additional explanation when the terminology we use can be interpreted in multiple ways.

Section 8.1.4 explains how we further limit ambiguity and unclarity by means of evaluation with test participants.

We formulated two types of questions: *open* and *closed* [20]. Closed questions are easier to analyze as the choice of reply is restricted. The restriction on the other hand may force the participant to make a choice even though they are unsure about the answer. For this reason we include an "I don't know" option when this is a logical choice, e.g., the participant may not know the impact of a particular problem. However, this option is not applicable when asking the participant how often they made use of code examples.

One of the objectives is to verify the completeness of the fault categories. We account for this objective by providing the option "other" to the list of choices in our question about the categories. The participant is given the chance to complement the list with other categories that they feel are missing.

For the answer options we use a five-point Likert scale with symmetric equidistant categories and a midpoint where possible. Not in all cases we found a five-point Likert scale with a neutral midpoint to be an adequate set of answers. For instance, the question about the impact of faults in API integration in our opinion is best answered using the four-point Likert scale with options: *none*, *low*, *moderate* and *high*. Using for instance *very high* as fifth option will introduce uncertainty and undermine the equidistant property. The participant may have trouble determining whether the impact of the fault was high or very high as the difference is not clear.

We complement the closed questions with open questions as we are interested in the API integration practices of the participants and their ideas on how to reduce the number of problems. To reduce misinterpretation and confusing answers, the open questions are mostly followup questions to closed questions, such that the participant has more knowledge and context about the question.

8.1.4 Evaluation and improvements

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.

Participant 1

1) Additional explanation was added to the survey introduction to clarify the notion of Web API integration. 2) We emphasized that the questions are to be answered in the context of an API that is used in production. 3) The order of the questions was changed to reflect a more natural order in which we start with integration questions.

Participant 2

1) More information was added to some of the answer options to remove ambiguities. E.g., the term *resource* is ambiguous in the context of APIs as it can refer to an endpoint or a remote object on the server. 2) To explain the relationship between the different stakeholders

involved an image was added to illustrate this. 3) We emphasized that the questions are to be answered based on the participant's experience with an application they worked on.

Participant 3
1) In all questions related to problems in production we emphasized the production aspect. 2) The exhaustiveness of the answers was improved by adding additional options. 3) The coherence between the questions was improved and alternative scenarios were covered, such as the situations in which participants have not experienced any errors in production.

Participant 4
1) An open question was clarified as participants were inclined to answer the question based on information given in preceding questions. 2) Participant 4 mentioned that the question asking for the years of experience in API integration may be confusing to answer, because one does not work on API integrations continuously. However, after considering other options, e.g., using a Likert scale, we decided to leave the question as is.

Participant 5
1) The key aspects of the explanations are highlighted to indicate the important aspects.

Besides a few minor improvements, such as spelling mistakes, participant 5 did not experience any unclarities, ambiguities or incomplete answer possibilities.

8.1.5 Sampling and respondents

The target audience for this survey, developers that have integrated with a Web API that was used in a production environment, is difficult to reach. To our knowledge there are no dedicated communities for this topic and therefore we resort to communities that target developers in general or have a subsection that is related. We posted on the following programming communities: Code Ranch's[1] *Web Services* forum, Hackernews[2] and Reddit's[3] 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 author reached out to personal contacts that match the target audience as described in Section 8.1.1.

[1] https://coderanch.com/
[2] https://news.ycombinator.com/
[3] https://www.reddit.com/

The survey has been online for three weeks and a total of 29 participants completed the survey out of 70 who answered at least 1 question. 7 people were disqualified, because they did not meet the requirements to participate as explained in Section 8.1.2. We decided to consider partial responses in the results as well, but only those participants that answered questions that are not background related, resulting in an 11 partial responses. Table 8.1 shows for the separate ways of sharing the number of complete and partial responses.

Shared via	**Complete**	**Partial**	**Total**
Adyen	10	0	10
Code Ranch	0	1	1
Hackernews	1	0	1
LinkedIn	2	0	2
Reddit	5	4	9
Twitter	9	6	15
Other companies and contacts	2	0	2
Total	29	11	40

Table 8.1: The number of complete and partial survey responses per means of sharing the survey.

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. 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 tough 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.

The participants were asked to rate the API's complexity in terms *different feature and functionality*, *required/optional parameters and value constraints*, *the number of possible error scenarios* and finally the *overall* complexity. Figure 8.1 shows the responses on a Likert scale distribution for the four complexity metrics. The participants consider the number of possible errors returned by the API to add the most complexity. The number of different features on the other hand seems to contribute less to the complexity of the API. Overall half of the participants consider the API as moderately complex. One participant could not recall the complexity of the API in terms of the number possible errors returned by the API.

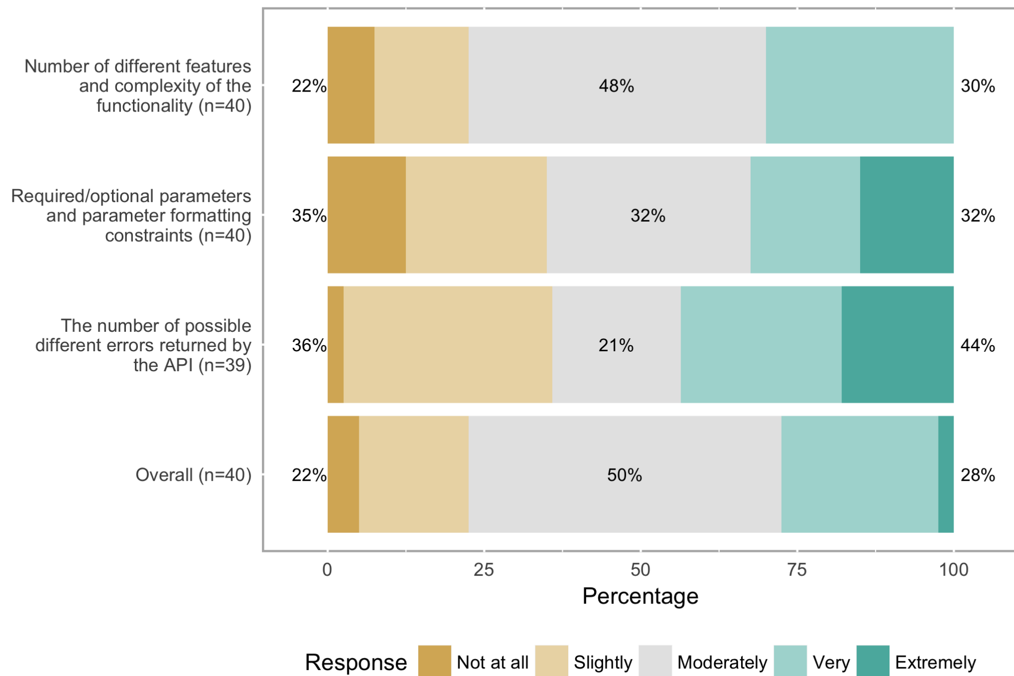


Figure 8.1: The complexity of the API considered by the survey participants based on different properties.

8.2 Fault types and their impact

In this section we describe the results of the survey related to the 11 identified fault categories. Section 8.2.1 reports on what fault types are encountered and how often they are experienced by API consumers. Section 8.2.2 elaborates on the experienced impact of these fault types by API consumers.

8.2.1 Fault types experienced by API consumers

The survey participants were asked to indicate how often they experienced production problems with the API in each of the 11 categories. Figure 8.2 illustrates the response distribution on 4-point Likert scale *Never*, *Rarely*, *Sometimes* and *Often*.

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 category 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*

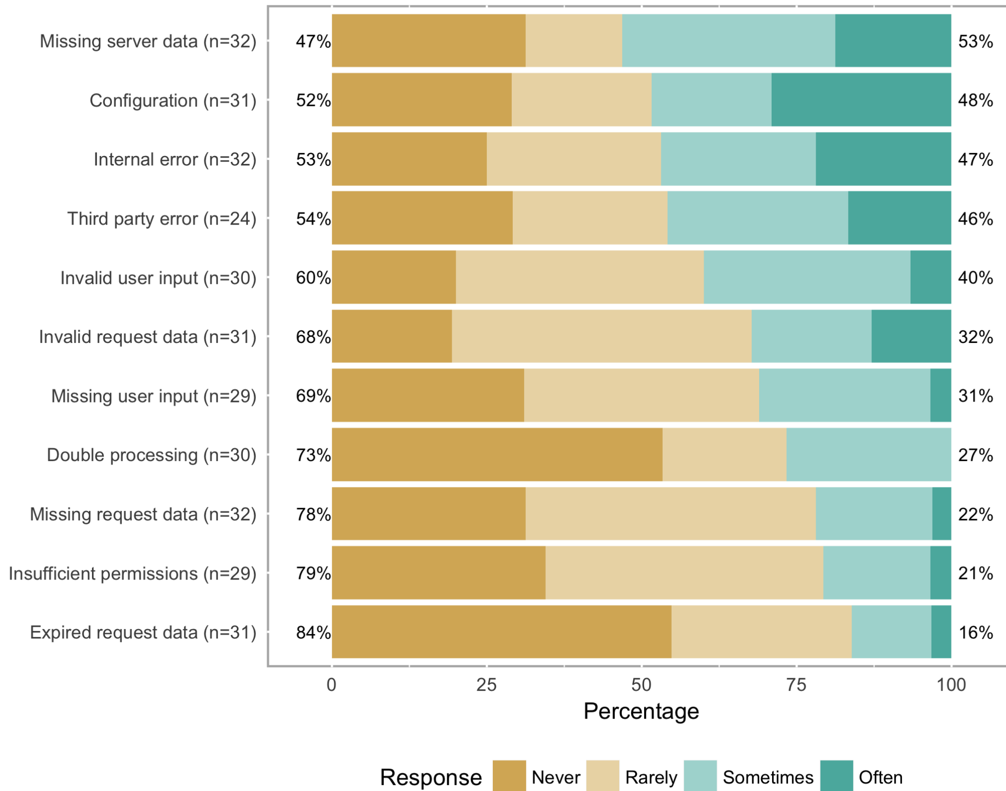


Figure 8.2: How often API related problems are experienced in a production environment per fault category by the survey participants.

request data and *double processing* related problems are not experienced by over half of the participants.

Several participants added additional categories 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* category. Furthermore, two participants experienced problems caused by hitting the API requests limits. We regard these to be related to faults in the *insufficient permissions* category. Namely, the API consumer is not allowed to make more requests.

> ***RQ 1**: What type of faults, resulting in API errors, are impacting API consumers?*
>
> The survey results verify the completeness of the 11 categories identified in Section 6.2. Although the participants propose additional categories we regard them as part of previously identified categories.

***RQ 2**: What is the prevalence of these fault types, and how many API consumers are impacted by them?*

Based on the survey results, *missing server data* and *configuration* faults were experienced the most by API consumers. Faults caused by the API provider and third parties, categories *internal* and *third party* respectively, were also found to impact relatively many API consumers. *Double processing* and *expired request data* faults were not experienced by over half of the participants.

8.2.2 Fault type impact experienced by API consumers

To understand the impact of problems in API integration we asked the survey participants what the impact of the problems they experienced in each fault category was. Figure 8.3 shows the distribution of problem impact on 4-point Likert scale *None*, *Low*, *Moderate* and *High*.

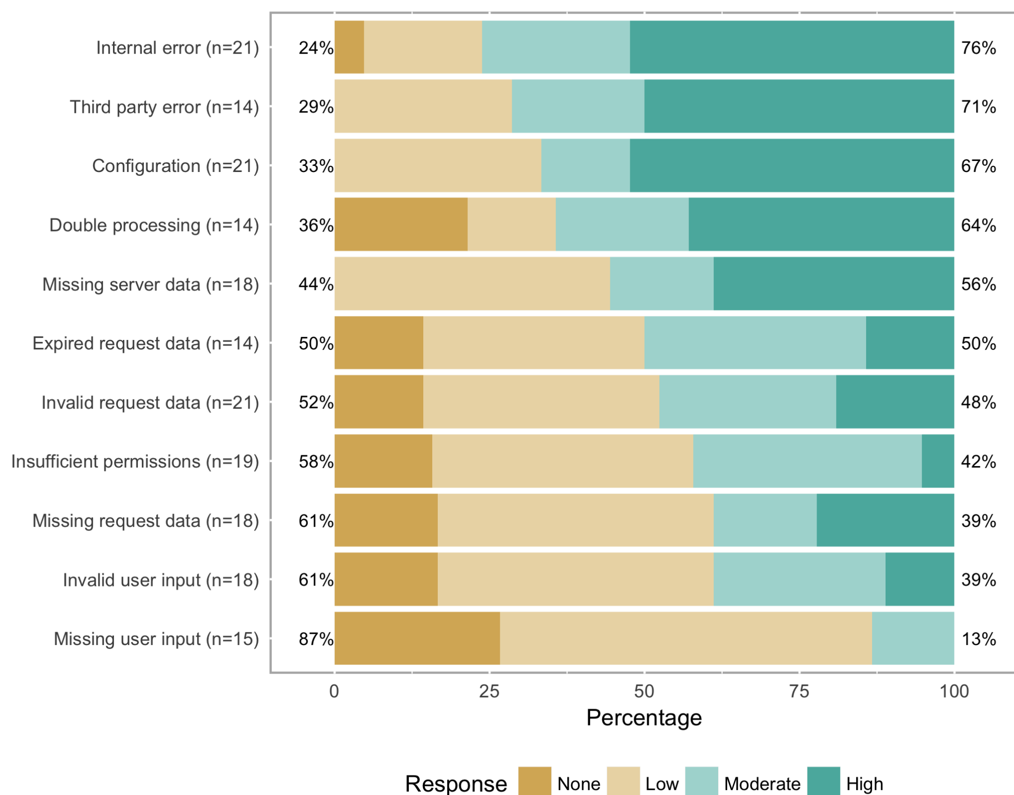


Figure 8.3: The impact of API related problems per fault category as experienced by the survey participants.

High impact *Internal* and *third party* related problems, caused by the API provider and third parties, are experienced as most impactful on production applications.

Low impact 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.

Notable Interestingly, *double processing* related problems seem to have either no impact, or relatively much impact compared to the other categories.

> ***RQ 3****: What type of faults do API consumers consider the most impactful?*
>
> 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 are regarded as having the least impact.

8.3 API integration practices and challenges

The current practices of API integration were investigated in terms of the process of integration, error handling and fault detection, and are reported on in Sections 8.3.1, 8.3.3 and 8.3.4 respectively. Similarly, the challenges of these aspects were identified. We report on the fault detection challenges and problem causes experienced API consumers in Sections 8.3.2 and 8.3.5.

8.3.1 API integration by API consumers

To understand how API consumers obtain the knowledge necessary to integrate with an API we asked them how often they used different information sources. Figure 8.4 shows the usage of four information sources by the survey participants on a 5-point Likert scale.

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.

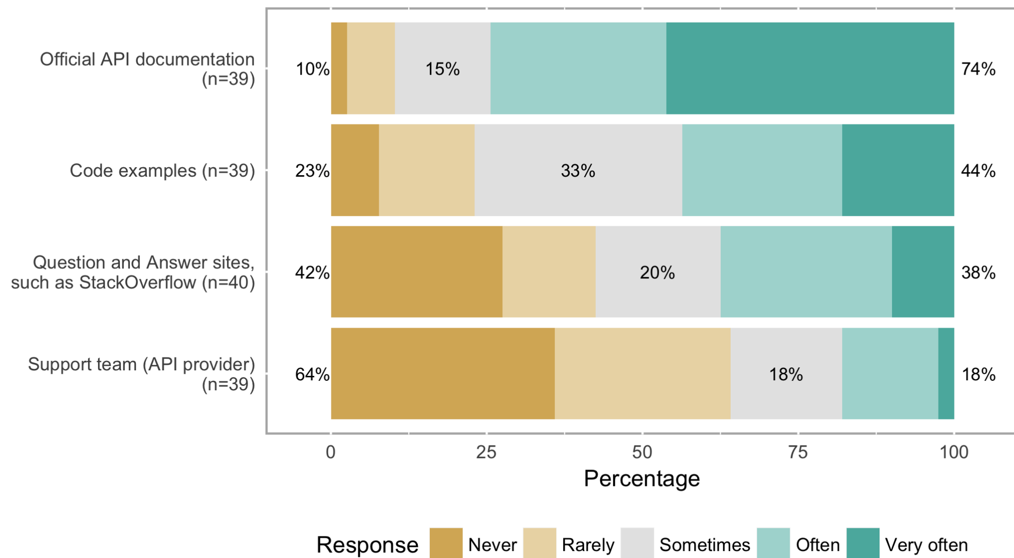


Figure 8.4: The usage of different API integration information sources.

8.3.2 API fault prevention

The API provider can help prevent failures in API integration. The survey participants were asked what the API provider can do to prevent problems experienced with the API according to them.

Of the respondents ($n = 18$) 13 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.

8.3.3 API error handling

Not properly handling API error messages can result in application failures. It is therefore interesting to know how API consumers handle API errors returned by the API. We asked the participants to indicate how they handle API errors by asking how often they used dif-

ferent types of error handling. The results, given on a 5-point Likert scale, are given in Figure 8.5. We distinguish between *specific logic*, *general logic* and *no logic*.

Specific logic Custom error handling for unique error codes or messages, such as error code and message *"137 - Invalid amount specified"*.

Generic logic Error handling is the same for a group of error messages, such as all error messages returned with HTTP status code 500.

No logic No error handling in which case errors are not acted upon.

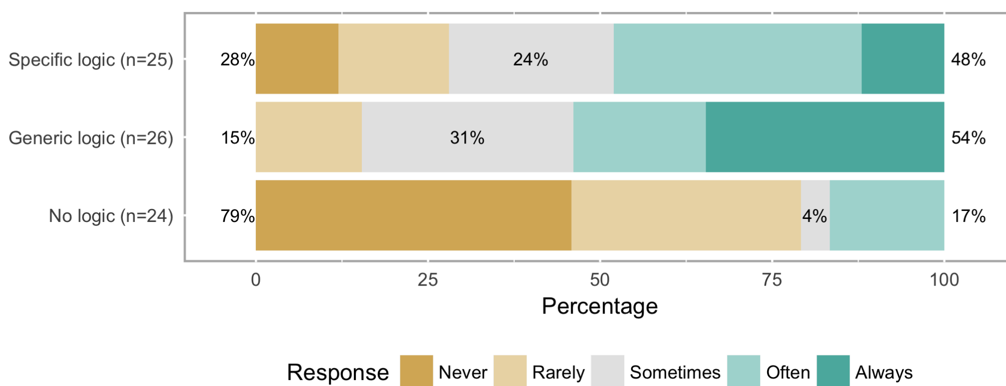


Figure 8.5: The usage of different types of API error handling logic.

Note that this figure is not to be interpreted as a regular Likert distribution in which the different items are compared by studying the relative response for each answer type. Namely, for *specific logic* error handling the *always* answer is the most positive, while for *no logic* error handling *never* is the most positive. For this reason we discuss them separately.

Specific logic, used to handle distinct error responses, is used *often* or *very often* 48% of the time. Only 12% of the participants does not use specific logic to handle errors. Generic logic to handle errors is used most often compared to the other methods of handling. 35% of the participants always has generic logic in place to handle a group of API errors. All participants use generic logic to some extent as none answered *never*. 45% of the participants handle all errors in some way; they do not use no logic in which case errors would not be acted upon. In total, 79% of the participants *rarely* or *never* has no logic to handle errors and only 17% indicates to *often* not handle all errors.

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. *"E.g., 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, it is experienced a challenge to communicate errors to the end user. "*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.

In the development process it is experienced as difficult to know what edge cases will result in errors and which should be handled and which are not likely to occur. "*The large number of edge cases, which can lead to more errors if not all handled well.*" The participants experience difficulties because not all error responses are documented or there are insufficient error details to allow for proper error handling.

8.3.4 API fault detection

The survey participants ($n = 29$) were asked how problems in their application with the API were detected. Figure 8.6 displays the number of responses for each detection mechanism.

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. 5 respondents had additional mechanisms in place, among which are "continuous live smoketesting", "manual tests in production" and "Runscope Radar[4]", which is an API monitoring tool that can detect downtime and schedule API test cases".

To understand what type of detection mechanisms the participants do not have in place, but would be useful we asked them how they could improve the detection of problems ($n = 19$).

Testing related improvements include testing the API with random input to better cover client use-cases. Another approach would be to test every API error response, but the respondent noted that this would be an unfeasible amount of work. Daily running API test cases was another suggestion to detect changes to the API's behavior. Another code related improvement is to add more descriptive exceptions, such that they can be used to detect problems.

Several log related improvements include enriching current log messages with more details about the error and daily analyzing the logs for anomalies.

In terms of monitoring the respondents mentioned that using monitoring system Sentry[5] could improve the problem detection. Other improvements include verifying the API re-

[4] https://www.runscope.com/

[5] https://sentry.io/

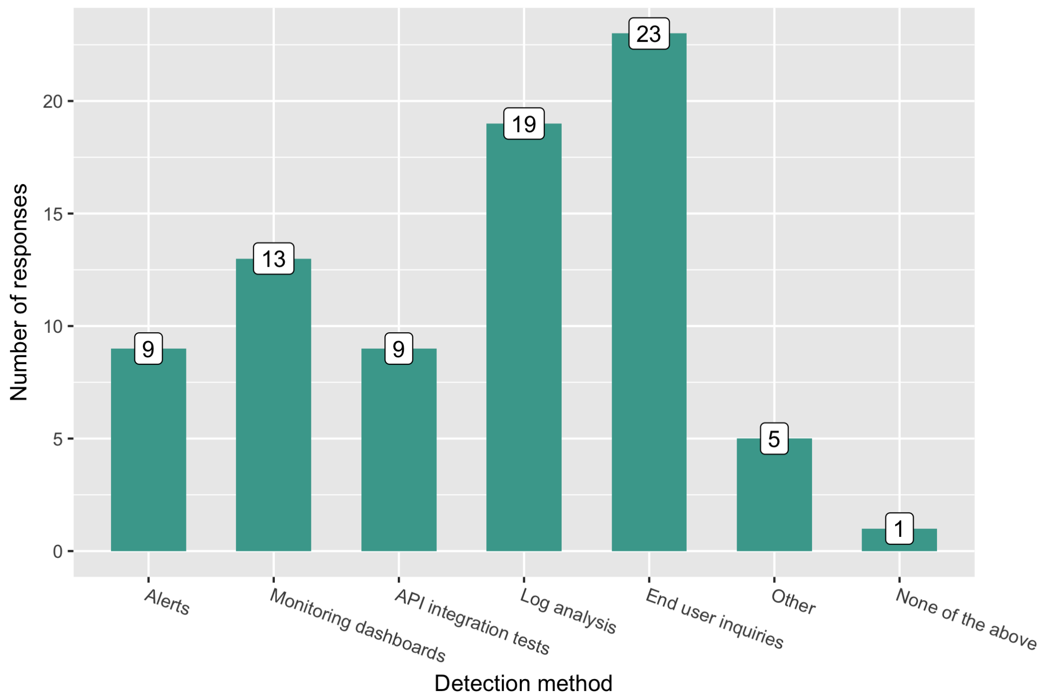


Figure 8.6: The usage of different means of detecting problems in API integration.

sponses. "*Strict specification and validation of responses and trigger alerts if the responses differ.*"

Two participants noted that there was no difficulty in detecting problems in their system. "*Pretty happy with our detection rate at the moment.*"

We further asked the participants ($n = 16$) why this way of problem detection is not in place. The most prominent reason being the amount of work required to get such a mechanism into place. It is a large time and cost investment, or the priority is with other projects. "*There is a steep time investment and it would only deliver in edge cases. Also it has a questionable return on investment.*"

8.3.5 Underlying causes

To understand why API related problems are introduced in the first place we asked the participants ($n = 22$) about their idea of the underlying causes. The respondents where asked to answer based on their experience with a production application integrated with an API in mind.

The biggest cause of API related problems, according to the respondents, is the API documentation. Problems arose due to missing documentation, incomplete documentation, incorrect documentation or documentation that is not specific enough to cover the details. Furthermore, an insufficient number of examples makes it more difficult to understand the API as well as the lack of an architectural explanation of how to implement the API. These

documentation specific issues make integration more difficult and force the API consumer to make assumptions, which in some cases turn out to be incorrect.

The API itself was named as cause of problems as well. The API should be fault tolerant in that it captures and handles third party errors. Furthermore, it should do validation of the input to avoid duplicate requests and should not crash on input that it cannot handle. Configuration and security policy changes were named as causes of API problems. "*Unexpected caching or gateway changes.*" Improper versioning combined with changes in the API were found to cause compatibility issues often leading to API errors. Lastly, inconsistencies between different APIs from the same provider can cause confusion and lead to incorrect assumptions followed by potential API errors.

Three respondents denoted themselves as origin of API problems. "*I think it's almost always the fault of the developer of the software that is using the API. Bad APIs and more importantly bad documentation make it very hard to interact with them without something going wrong, but at the end of the day, unless the API unpredictably failed, it is something I could have caught in testing.*" Further causes include hitting the API limit and not having error handling in place to deal with this scenario, and miscommunication with the API support team resulting into misinterpretation that subsequently results in incorrect implementation.

RQ 4*: What are the current practices to avoid and reduce the impact of production problems caused by faults in API integration?*

API consumers most often use official API documentation to implement an API correctly, followed by code examples. The impact of faults and potential problems is reduced via specific error handling. However, not as often as via generic error handling. Most often, API related production problems are detected by the end user of the application, followed by log analysis and monitoring dashboards.

RQ 5*: What are the current challenges to avoid and reduce the impact of production problems caused by faults in API integration?*

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. Handling is experienced challenging due to an insufficient understanding of the impact of problems, missing guidance on how to recover and a lack of details on the origin of errors. Detection is challenging because of the unfeasibility of testing all scenarios, a lack of detailed exceptions and the amount of work required to analyze logs regularly. In addition, missing, incomplete, incorrect and unspecific documentation makes avoiding errors more difficult as well as APIs that cannot handle specific input, or do not do versioning properly.

Chapter 9

Recommendations

We learned about the type of API faults that occur, their occurrence and impact. Furthermore, we obtained an understanding of the current practices and challenges in reducing the impact of API fault related problems in terms of integration, error detection, error prevention and error handling. Based on this knowledge, we formulate recommendations for API consumers and API providers to reduce the impact of API faults.

The ultimate scenario would be a situation where there are no failures. Although debatable whether this is practically achievable there are two ways to reduce the impact of production problems related to API faults: 1) Reduce the number of problem occurrences and 2) reduce the impact of individual problem occurrences. This starts with *detecting* faults related to API integration that manifest in errors. Once a fault has been detected, the options are to *prevent* the fault in the future or to *handle* it appropriately. Figure 9.1 gives an overview of this. Newly detected faults can already be handled by the application using generic or specific logic. *Handling* API faults can help to reduce the impact that these faults would have without handling. *Preventing* the API faults from occurring again naturally eliminates the impact that the faults could have.

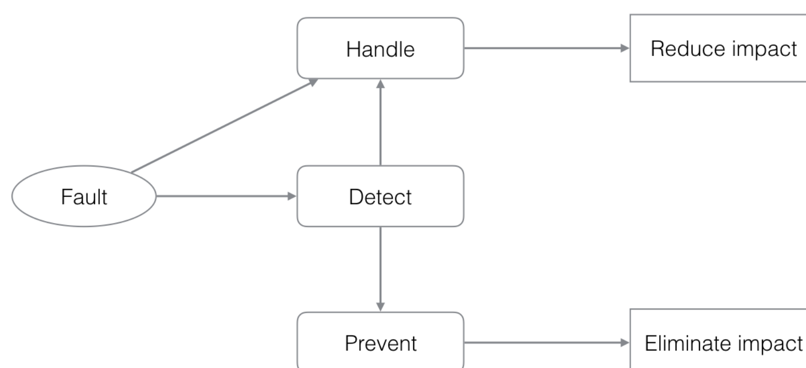


Figure 9.1: Reducing the impact of API related problems.

In Section 9.1 we elaborate on the appropriate detection actions to take for faults in each

category and make a suggestion for the API provider to help the API consumer identify API related faults. Section 9.2 describes what fault types can be prevented and explains the different approaches to do so. Next, Section 9.3 discusses the methods of handling faults of the different types. Table 9.1 gives an overview of the detection action, prevent action and handle action for each category. Finally, in Section 9.4 we discuss the priority of different problems based on their impact.

Stakeholder	Category	Detection action	Prevent action	Handle action
End user	Invalid user input	Prevent + Handle	Validation	Feedback + Retry
End user	Missing user input	Prevent + Handle	Validation	Feedback + Retry
End user	Expired request data	Handle	-	Feedback + Retry
API consumer	Invalid request data	Prevent + Handle	Fix	Recover
API consumer	Missing request data	Prevent + Handle	Fix	Recover
API consumer	Insufficient permissions	Prevent + Handle	Fix	Recover
API consumer	Double processing	Prevent + Handle	Fix	Recover
API consumer	Configuration	Prevent + Handle	Fix	Recover
API consumer	Missing server data	Prevent + Handle	Fix	Recover
API provider	Internal	Handle	-	Retry/Recover
Third party	Third party	Handle	-	Retry/Recover

Table 9.1: The different actions to take as an API consumer when detecting, handling or preventing types of API faults.

9.1 Fault type detection

The goal of detecting faults is to be aware of what happens in the system, and to understand the potential consequences of detected faults and to act appropriately. In Section 9.1.1 we describe the actions to be taken for each category once a fault has been detected. Section 9.1.2 outlines a proposal for the API provider that can help the API consumer identify API related faults.

9.1.1 Fault detection action

After identifying a fault, the API consumer can decide to prevent the fault from occurring again, to handle the fault to reduce the harm it may cause, or to ignore the fault. We argue that the categories of faults, identified in Chapter 6, cannot be acted upon the same and elaborate on this per category. Table 9.1 displays for each category the action to be taken.

Expired request data faults are caused by the end user and indicate that the input data was no longer valid at the moment of processing. When dealing with input data that can expire, this fault cannot be 100% prevented from occurring. Thus the way to reduce the impact of these faults is to reduce the impact of individual occurrences by handling appropriately. *Internal* and *third party* faults, caused by the API provider and third parties respectively, are outside the reach of the API consumer. The API consumer may work together with both of these stakeholders to resolve the origin of the faults, however can not do this on their own.

Reducing the impact of individual fault occurrences by handling the faults is therefore the best option.

Faults in the other categories on the other hand can be prevented by the API consumer. Section 9.2 elaborates on this. Logic to handle these faults is however still recommended in case new faults are introduced or previously known faults reoccur.

Recommendation 1*: Be aware of the detection action per fault category*

API consumers should resort to error handling to reduce the impact of the errors in categories that are outside the control of the API consumer. Errors in these fault categories cannot be prevented.

The appropriate action for errors related to other fault categories that can be controlled is to prevent the error from occurring again to eliminate the impact. Handling logic is still recommended for these faults types to limit the impact once they reoccur.

9.1.2 API error detection dashboard proposal

In Section 8.3.4 we saw that application problems related to API integration were most often detected via end user inquiries. Survey participants explained that they could improve fault detection using tests, better logging, daily analysis and monitoring. The amount of work, time constraints and cost investments were the primary reason for this to be not in place. Furthermore, the interviewees reported they had no automatic monitoring system in place to identify errors and that manual analysis was required to attempt to do so.

Instead of having all API consumers monitor their API integration to identify issues and anomalies, the API provider can do this for them. The API provider receives all incoming requests, processes them, determines that a request cannot be processed, and returns an API response. In other words, the API provider has all necessary information to detect potential problems for the API consumer.

An implementation of this idea would be an API consumer facing dashboard that shows, for a configurable time period, the number of API errors returned per unique error message. In addition, the error type and handling action as proposed in Section 9.3.4 can be included. This overview can be especially useful for API consumers that do not have the technical capabilities or development resources to identify these errors themselves. Undocumented explanations of the error message can be added over time. The API consumers can contribute in this respect making the documentation partially a community effort.

The interviewee in Section 7.2 mentioned that a non-technical person interested in the correctness of the API integration often does not have the skills to go through the server log files to discover what is going on. By including the API requests and responses in the dashboard, both technical and non-technical people can quickly look into the request data and get an intuition of the problem.

To help the API consumer identify the cause of the problem, each of the error messages displayed in the dashboard should link to detailed documentation about that message. In addition to an explanation of the error message the underlying fault explanations should be

given, including the steps necessary to solve the problem. For instance, the documentation on error message “Invalid amount specified” of the system under study should include the four fault case descriptions described in FC29-F32 in Appendix A and additional mitigation actions. The information will allow the API consumer to more quickly identify, understand and fix API related problems. API support will be proactive, instead of reactive. In other words the consumer is contacted and notified of a potential issue they may have, instead of the consumer finding an issue and contacting the API support team to identify the problem. The workload of the API provider’s support team can therefore be reduced, since consumers are able to troubleshoot issues on their own. Not only will this be applicable to existing API integrations, new API consumers can benefit from this information during integration to help them identify problems more effectively.

The API consumer should be able to configure alerts such as email or SMS to be notified of new API errors. This way API consumers will become aware of new or recurring issues immediately. Problems that went undetected for months such as the case described in Section 7.2 this way would have been identified right after its first occurrence.

In some cases the API consumer may be aware that certain errors occur. They may regard the errors as non-critical or may deliberately use the API’s validation and handle the accompanying errors. The API consumer should therefore be able to annotate certain error messages as *acknowledged* or *non-critical*, such that they are not alerted upon. Perhaps a non-critical error is not a problem when it occurs only a couple of times per month, but needs further action when the error rate increases to several thousand occurrences. A threshold per error message should prevent any alert from being triggered as long as the error rate remains below the threshold.

This solution is applicable to all consumers of an API, which can become a worthwhile investment when the number of consumers is large. In addition it may make the API provider aware of any internal issues it may have which are experienced by their consumers.

***Recommendation 2**: Provide API error insights*

The **API provider** can exploit API usage data to provide proactive insights into potential problems for the API consumer. For instance, an API consumer dashboard that shows the different experienced errors with pointers to detailed error explanations and mitigation documentation. Using alerts, API consumers can be notified instantly of new potential problems.

9.2 Fault type prevention

The impact of API faults can most effectively be reduced by preventing these faults from occurring. We discuss two types of prevention: *validation* of the input and *fixing* the underlying problem. These are discussed in Section 9.2.1 and 9.2.2 respectively. Table 9.1 provides an overview of the types of prevention per category.

9.2.1 Prevent by validation

Two types of faults can be prevented by validation. *Invalid* or *missing user input* faults are caused by the end user in an API integration. The API consumer can limit the number of faults by performing both client-side and server-side validation. Using client-side validation the end user receives immediate feedback while filling out or submitting a form, while potentially increasing the user experience. In case of server-side validation, the end user submits information to the server, that performs the validation and informs the end user about any mistakes made. Both solutions avoid a call being made to the API. This removes an opportunity for network connectivity issues and saves the additional request and response time. Client-side validation has the preference as this reduces client-server interaction. However, for security reasons, server-side validation is still necessary.

To help the API consumer prevent *invalid* and *missing user input* faults the system under study offers a JavaScript library that does client-side card data validation, which checks whether the card number is valid according to the Luhn algorithm [19] and offers basic validation of other fields. The form cannot be submitted when one or more fields are not valid. Using client-side validation API consumers are able to reduce end user drop-off and increase the number of payments made.

> ***Recommendation 3****: Validate user input*
>
> The **API consumer** should validate the end user's input to avoid redundant calls to the API. Client-side validation can improve user experience and server-side validation is required for security purposes.

Survey participants mentioned that the restrictions of calls and parameters should be more clearly documented. The API provider can contribute to the quality of the validation logic of the API consumer by providing elaborate field specification information in the API documentation, such as:

Requirement States whether the field is required or not.

Field type Denotes the data type of the field, e.g., integer, decimal or string.

Input constraint Explains what format a value should have, e.g., a valid email address or the minimum and maximum range for digit values.

> ***Recommendation 4****: Provide parameter information to enable validation*
>
> The **API provider** should provide the following detailed API request parameter specifications to enable proper validation by the API consumer: whether the field is required, the field type and the input constraints.

9.2.2 Prevent by fix

In this section we discuss the following categories of faults: *invalid request data*, *missing request data*, *insufficient permissions*, *double processing*, *configuration* and *missing server data*. The errors caused by these problems are all caused by the API consumer, who is responsible for sending in the request in the first place. For this reason the only way to prevent faults of these types from occurring again is to identify the underlying problem and fix it. Although this sounds like an intuitive way to reduce the number of faults, it is not a simple process by default.

Invalid or *missing request data* faults can be caused by misinterpretation of the specifications, or due to a bug which causes the request data to be malformed or missing. The API consumer can reduce the number of errors by including integration tests in the development process. The integration tests should call the API and cover the intended behavior of the consumer's application. The API responses should match the expected responses defined in the tests. This approach's effect is twofold. First, during development the developer can verify the correctness of the API integration by rerunning the tests after making changes. Second, if run regularly, the tests can verify that changes made to the API do not impact the API integration.

Configuration and *insufficient permissions* faults are caused by incorrect API settings and insufficient rights respectively. These mistakes can occur due to incorrect assumptions about what is possible and what is not. Also, changes to the configuration or permissions may break existing functionality. For this reason tests to verify the intended functionality will help during development to avoid faults during integration. Once the tests are in place they should be run regularly to verify that changes to the application, API, configuration and permissions do not break any functionality. If new issues are detected tests should be added to cover the functionality.

Recommendation 5*: Use continuous testing and periodic testing*

The **API consumer** should have API integration tests in place to verify the correctness of the integration as part of the software delivery pipeline; continuous testing. In addition, these tests should be run periodically, apart from the delivery pipeline, to verify that changes to the API, configuration and permissions do not break functionality.

To aid the testing process of the API consumer, the API provider can supply a set of test cases that each comprise of request parameters and the expected response. The test cases should be available per feature, such that the API consumer can quickly select the applicable tests for their integration. The API provider may be able to supply more effective tests cases compared to those the API consumer can come up with as the knowledge of the API design and implementation lies with the API developers.

Recommendation 6: *Offer a complete set of test cases*

The **API provider** should provide a complete set of test cases for the API consumer that cover request parameter specifications and expected responses.

Double processing faults are the results of sending the same request multiple times, while this is not allowed. We found these faults to be introduced by accident in the form of a bug, and not because of incorrect assumptions. Testing in this case is not a feasible solution, because it can be hard to identify the cases in which this problem may occur. Therefore faults in this category should be investigated when detected and fixed if deemed important.

The same holds for *missing server data* faults. We found errors related to these faults, caused by incorrect management of resource identifiers, to be most often introduced by bugs. Test cases to cover these types of faults can be difficult to identify and therefore detecting them is essential. API consumers can limit the number of these errors by not managing these resources locally, but instead requesting them on-demand using an identifier which is fixed. In case the consumer chooses to manage resources locally for performance reasons they should be fully aware of what actions may alter a remote resource, such that they can update it.

Recommendation 7: *Fix API consumer faults*

Errors related to API consumer categories can only be prevented by the API consumer themselves. The **API consumer** should therefore implement a fix for all errors in these categories to eliminate the potential problem impact.

Similar to what is explained in Section 9.2.1, the API provider can help the consumer prevent API errors by providing an accurate and elaborate API reference manual including the field requirement information, field types and input constraints. As suggested by survey participants, high level architectural advise will help to understand how the API is intended to be implemented. This can help avoid faults related to *missing server data* for example. In addition the documentation in terms of manuals, tutorials and FAQs should be update to date, correct, explain all functionality and leave no room for interpretation. Finally, enough code samples should be supplied to illustrate the appropriate use of the API.

Recommendation 8: *Provide implementation architecture recommendations*

The **API provider** should provide an architectural recommendation for API integration by the API consumer. For instance, by providing a reference implementation.

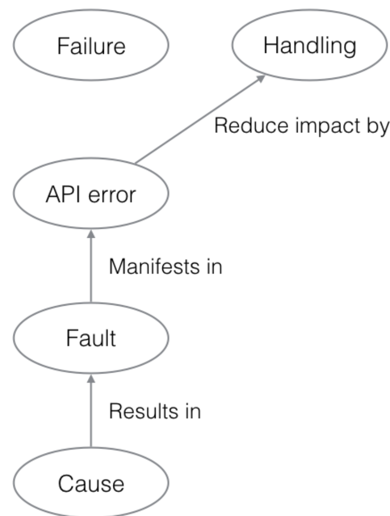


Figure 9.2: The impact of error handling on the cause-failure chain.

***Recommendation 9**: Provide effective documentation*

The **API provider** should provide complete, correct, up-to-date and unambiguous documentation. Code examples are a must to illustrate the appropriate use of the API.

9.3 Fault type handling

Faults are bound to occur and detection is essential. Although detection can help identify problems and prioritize what to improve, it is still necessary to handle known and unknown errors that occur in production properly to reduce the impact that they would have without handling. Figure 9.2 shows the result of handling API errors based on the cause-failure chain explained in Section 3.2. Although the failure scenario is mitigated, the underlying cause and the accompanying faults and API errors remain.

We distinguish between three different ways of handling requests: *feedback and retry*, *recover* and *retry or recover*. These are discussed in Sections 9.3.1, 9.3.2 and 9.3.3 respectively. Finally, in Section 9.3.4 we suggest API error response practices for the API provider to help the API consumer handle faults. Table 9.1 provides an overview of the fault handling methods per category.

9.3.1 Feedback and retry

The first way of handling API errors is related to all end user fault categories: *invalid user input*, *missing user input* and *expired request data*. A retry of the same request will result in

the same error as the request failed because of a problem with the data. Therefore changes are to be made, before the request can be resubmitted. To this end the API consumer should give the end user *feedback* in terms of what has gone wrong and what actions should be taken to resolve the issue. Subsequently the request should be *retried* with different input.

The API consumer may deliberately use the API provider for validation. In this case the consumer accepts the extra request time and should have the described handling in place to inform the end user. In this case the API consumer is aware of the fault, but avoids failure by handling appropriately.

Recommendation 10: *Handle end user faults by providing feedback and retrying*

The **API consumer** should provide actionable feedback to the end user when the end user causes an API error. After making changes to the request, the request should be retried.

9.3.2 Recover

Recovering is the error handling solution for faults that cannot be retried by the end user as the origin lies at the API consumer. Furthermore, a second attempt for faults of type *invalid request data*, *missing request data*, *insufficient permissions*, *double processing*, *configuration* and *missing server data* will result in the same error unless the implementation is changed, or the configuration or permissions are updated. There recovering is the best option to reduce the impact of the fault. In addition, the API consumer should look into the problem and fix the issue if deemed important.

Recommendation 11: *Recover from API consumer faults*

The **API consumer** should recover from API consumer related faults to reduce the impact, as these requests cannot be retried. Identifying the problem and implementing a fix is recommended.

9.3.3 Retry or recover

The third way of handling API errors considers *internal* and *third party* faults. In case the fault is caused by, for instance, a temporary service outage or a connection timeout, the original request may be handled correctly in a second attempt. The requests resulting in this sub-type of fault could be retried a predefined number of times on a certain interval, before resorting to *recovering*. It is however possible that there is an implementation mistake on the API provider or third party side. In that case a retry will yield the same error every time. Faults that cannot be retried should be handled by recovering only.

If possible, a distinction should be made by the API provider between retriable and non-retriable errors, such that the API consumer can effectively handle the underlying types of faults.

> ***Recommendation 12****: Distinguish between retriable and non-retriable errors*
>
> The **API provider** should distinguish between retriable and non-retriable errors to facilitate the error handling process of the API consumer.

> ***Recommendation 13****: Retry or recover from API and third party faults*
>
> The **API consumer** should recover from by API and third party related errors by retrying a predefined number of times if the request is retriable. In case the request is not retriable, or the defined number of retries has been reached, the API consumer should resort to a recover mechanism to reduce the impact.

9.3.4 API error response proposal

The API provider can help the consumer make the choice of what handling mechanism is appropriate. An error code and message allow for specific error handling logic to be implemented. Interview and survey participants however noted that mapping all errors is unfeasible because of the required development time and the often incomplete or insufficient error details that are given. Generic error handling can therefore be used to capture groups of error messages which are not handled specifically. This approach also allows previously unknown errors to be captured, which would be missed by specific handling logic. Generic error handling logic is also found to be used most often.

> ***Recommendation 14****: Use generic error handling for unknown errors*
>
> The **API consumer** should implement generic error handling for unknown errors, such that the impact of these errors is limited. If possible, groups of similar errors should be targeted with group-specific error handling mechanisms to reduce the impact even more.

Survey participants mentioned that it is often unclear what the right way to handle errors is. To this end we propose to make API error responses more informative and actionable to the API consumer. Besides more accurate documentation, the API developer can leverage the error responses to help the API consumer handle API errors. In addition to a specific error code and message we suggest to return an *error type* based on message similarities as mentioned by one of the survey participants. The error type, based on the 11 categories, could have the following values:

validation Denoting end user input related faults. (*invalid user input* and *missing user input*)

expired Indicating an expired request caused by the end user. (*expired request data*)

implementation Implying an implementation related fault caused by the API consumer. (*invalid request data*, *missing request data*, *double processing* and *missing server data*)

configuration Translating to faults related to misconfiguration. (*configuration*)

permission Expressing a fault caused by insufficient rights to make the request. (*insufficient permissions*)

internal Specifying a fault caused by the API or third party. (*internal* and *third party*)

These error types describe the specific error on a higher level, which can help the consumer to better understand the impact and appropriate actions to take. The API provider has to determine these error types manually for each error message. More detailed error types, e.g., using the 11 categories, can add the unnecessary overhead of having to understand and deal with 5 additional error types.

Instead of, or in addition to, the error type the API provider may decide to include the suggested *handling action* to be taken.

feedbackAndRetry Specifying that the end user should be informed and suggested to retry the action.

recover Implying that the request cannot be executed and that recovery is the only option.

retry Indicating that a retry with the same request data may result into a success.

The *retry* value could be accompanied by a suggestion in terms of the number of retries that is recommended before resorting to recovering. Similarly a recommended time interval between the requests can be included. Furthermore, the API provider could track retries and decrement the suggested number with each request. When retrying is no longer expected to result in success the API error should include *recover* instead of *retry* as handling action. In that case generic logic implemented by the API consumer based on the handling action should proceed to recovery.

Besides the challenges of handling errors, it is experienced difficult to translate error messages to something actionable by the end user. To help the API consumer the API provider can include *user messages* in API error responses. These messages can describe what has gone wrong and what can be done to fix this.

The request below is an example of what the error response related to FC9 described in Appendix A would look like using the error response proposal.

```
{
    "errorCode" : "128",
    "errorMessage" : "Card Holder Missing",
    "errorType" : "validation",
    "handlingAction" : "feedbackAndRetry",
    "userMessage" : "The name of the card holder is required.
                     Please try again."
}
```

***Recommendation 15**: Provide enriched API error responses*

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.

The Adyen API is capable of returning over 150 different error messages, which makes it difficult for merchants to implement error handling logic. To ease the error handling process for merchants Adyen uses error types, similar to the suggestion made above. The following four types are used: *validation*, *security*, *configuration* and *internal*. We however found these types to not accurate describe the underlying problem. For instance, the error “Invalid merchant account” indicates that the account is missing or is not valid in context of the request. This error is labeled with error type *validation*, described as “the request does not pass validation.” This description explains why the request cannot be processed by the API, instead of suggesting the cause of the error. We believe Adyen can improve their error messages and help their merchants handle errors by looking into the suggestions made in this section.

9.4 Problem priority

Based on API data of the system under study we deduced the impact of faults in terms of number of occurrences and number of impacted API consumers. Furthermore the survey participants were asked how often they experienced faults in the different categories and what the impact of those faults was. This information can be useful for the API provider to prioritize efforts to improve API integration and for API consumers to determine what type of faults to pay more attention to.

Both according to the data of the system under study and the survey participants faults related to *invalid user input*, *third party* and *configuration* occur the most. Only *third party* and *configuration* faults are regarded to have a high impact by the survey respondents. *Invalid user input*, although experienced often, however is not regarded as very impactful. The API provider can prioritize by looking into error returned caused by third parties and provide more clarity on the different configuration options.

According to the survey participants *missing server data* and *internal* faults occur relatively often compared to the other categories. *Internal* problems are even considered the most impactful. The findings in the system under study however show that *internal* faults do not occur for many consumers at all. For *missing server data* the same holds. Similarly, the API error data suggests *invalid request data* and *missing request data* faults to impact a relatively large amount of API consumers. The survey respondents however regarded these categories to not occur very often. For these reasons the impact of certain types of faults seems to depend very much on the API and its functionality. The findings based on the survey can be used to give a general idea of what is considered important and impactful, however there may be large deviations based on the implementation of the API and its use cases.

Both the survey respondents and the API error data suggest that *expired request data*, *insufficient permissions* and *double processing* related faults are not experienced very often by API consumers. The *expired request data* category is perhaps not generic enough. During the analysis of the system under the study we only found one problem in this category with a very small number of errors. Although these faults can result into problems for the API consumer, the API provider would benefit more from improving in other categories first.

> ***Recommendation 16****: Prioritize on problem impact*
>
> **API consumers** should focus their efforts on fault categories that have a large potential problem impact. To this end, the impact of *third party*, *internal*, *configuration* and *missing server data* faults should be reduced first with the appropriate prevention and handling actions.

Chapter 10

Discussion

In this chapter we outline research conducted related to this work in Section 10.1. Next, in Section 10.2, we list the internal and external validity concerns of our research. The envisioned future direction of research in the area of faults in Web APIs is elaborated on in Section 10.3. Finally, in Section 10.4, we explain the lessons learned during this research that we believe are relevant for other researchers.

10.1 Related work

We provide an overview of research conducted in the field of Web APIs as well as current challenges and research opportunities described in the literature. In addition we describe work done in the field of traditional offline APIs that may be applicable to Web APIs.

Wittern et al. [41] identified four challenges for developers calling Web APIs and argue in favor of corresponding research opportunities to support API consumers. We describe three of them: 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.

Wittern et al. [41] their vision is that the first line of research focuses on static analysis and IDE support for API consumers. To this end Wittern etl al. [40] 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 takes Open API [4] specifications as input. The Open API Initiative is created by industry experts who recognize the value of standardizing how REST APIs are described. Using the Open API specification of an API as input the static checker aims to check whether the API requests in the code conform to the specifications. The authors report a 87.9% precision for payload data and a 99.9% precision on query parameter consistency checking.

Wittern et al. [41] also regards mining web API usage for more advanced API consumer support. For example, they recommend to identify effective API usage patterns, and recommend to investigate effective API compositions. Our work coincides with this vision as we provide insights into erroneous usage of web APIs and make recommendations to help the API consumer. API composition is said to be limited by the lack of machine-readable API descriptions. Furthermore, missing API specifications limit the creation of tools for auto-completion, automatic testing and static request checking.

Wittern et al. [41] mention that the specification issues can be solved by the API provider, although this practically is not a given. Swagger [34] is a tool that allows for API specifications to be created using source code annotations. Such a solution depends on the API provider as API consumers do not have access to the source code of the API. Suter and Wittern [37] 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. [35] 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 [8] 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.

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. [30] 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. The survey provides an overview of the different properties inferred, the mining techniques used and the corresponding empirical results for five categories. The *behavioral specifications* category has commonalities with the validity of Web API requests and work in this category aims to complement incomplete or incorrect API documentation using automated techniques. The *migration mappings* category covers API evolution and its challenges, and work in this field may be applicable to Web APIs as well. Finally, the work describes a *general information* category that covers work that attempts to learn from API usage to aid the presentation of API documentation, which can translate to Web API documentation.

Robillard [29] 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. Inadequate or absent learning resources were considered an obstacle by

most respondents. To mitigate the obstacles the participants mentioned accurate examples, completeness, support for complex usage scenarios and relevant design elements. All of these were mentioned by our survey respondents as well. This suggests that the obstacles in learning how to use traditional APIs and Web APIs are similar, and may be resolved in a similar manner.

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. The angle of API evolution is interesting as it can potentially be the cause of many of the API related issues, such as the ones we identify in this work. 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. This understanding can be useful for maintenance purposes where the impact of changes can be evaluated and predicted.

Venkatesh et al. [38] mention that to help the integration process one should understand the challenges that are encountered by client developers. Our work uses an API usage-driven approach to identify categories of faults, and a survey to identify current practices and challenges. Venkatesh et al. base their analysis on developer forums and Stack Overflow[1] 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.

10.2 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 fault explanations 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. Similarly, the categorization process was manual and could therefore have been subject to bias. To limit the effect of bias, we verified the categorization step with an expert.

[1] https://stackoverflow.com/

3. To discover possible multiple explanations of error tuples, we analyzed the error messages for several API consumers. However, it is possible that a fault remained undiscovered because it occurred less frequently. 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 categories. 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 platform to determine the fault categories 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 categories 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. The case described in Section 7.3 is an example of this, as this problem does not occur inside the 28 day window selected for the data analysis. It would be useful to replicate the analysis based on a different time window to investigate the impact on the results.

3. The analyzed data only covers API error responses, and not the successful responses. For this reason we were unable to analyze the proportion of requests that resulted in an error, making it more difficult to generalize the results.

4. It turned out to be difficult to reach the target audience of the survey. This resulted in 40 responses in total of which 11 were partial. In addition, some of the questions were only answered by 16 to 22 participants. This sample is insufficient to generalize results about integration, detection, handling and prevent practices. Using these results, in combination with the API error data analysis and the illustrative interviews, we attempted to strengthen the results.

10.3 Future work

In this section we propose future work in the field of faults in API integration. We elaborate on future research that could strengthen the results of this work, improve the recommendations, and suggest other angles of research that we see.

For the generalization of the results, in terms of the identified categories and number of impacted API consumers, we believe it is worthwhile to replicate this part of our work

using data from various other Web APIs. This will give insights into the applicability of fault categories in other environments. Furthermore, the results may indicate that, for other APIs, it makes sense to split or combine certain categories.

The survey results give an initial understanding of the current practices and challenges in terms of integration, error prevention, handling and detection. It would be worth investigating each of these aspects in detail by means of qualitative interviews with API consumers. Furthermore, more insights can be obtained by investigating the source code of projects that have implemented an API, to learn more about these aspects. This improved understanding can be used to improve the proposed recommendations and add new suggestions.

In this work we propose 16 recommendations for both API consumers and API providers. To be able to give a definitive answer to the question, how the impact of API errors can be reduced, the effect of the recommendations needs to be observed.

To this end we propose to implement the dashboard proposed in Section 9.1.2. The next step is to understand the dashboard usage, study the users and quantify the effects of the tool in terms of API error reduction.

It will be interesting to see the effect of the enriched error messages as proposed in Section 9.3.4. The study can be data focused to see the effect on the number of errors and impacted consumers, or API consumer focused by studying the perceived impact of errors by consumers and the practices of dealing with errors used the extra information in the error messages.

More research is needed in the field of API integration testing. Some APIs, such as Adyen's or PayPal's[2], offer a test environment that acts as development sandbox for API consumers. The API consumer can use this environment to trigger API errors and test whether error handling is done correctly. However, certain errors, such as those related to *third party* or *missing server data* faults, are often only experienced in production. The API consumer however may want to test for these cases as well. We therefore feel it is worth investigating the possibilities for API providers to facilitate the testing needs of the API consumer. For instance, the API provider may be able to generate a set of test cases, based on the API specification, for the API consumer to use in their continuous integration.

There are a large number, and wide variety, of different APIs available. We expect there to be differences in the occurrence of the API faults and their impact on different APIs. This means that the best way to reduce the impact of API faults will likely also differ per API. For this reason we believe it will be beneficial for the field of Web API research to characterize the landscape of Web APIs. Subsequent work in the field can then define a more fine-grained scope and with greater detail expand on the applicability of the research.

During the manual analysis process we discovered that certain API errors had been occurring for months. We think it will be interesting to investigate how long errors in each fault category go undetected for. Another angle would be to research the time it takes for faults to be resolved after they have started to occur in the form of API errors.

[2] https://developer.paypal.com/docs/classic/lifecycle/ug_sandbox/

Continuing the previous idea, it will be interesting to look into the effects of API evolution on the occurrence and impact of API faults. This work and the current work on API evolution can be taken as starting point [11, 12].

10.4 Lessons learned

In this section we outline some of the lessons learned during the course of this research project that we feel are useful for any research project. We elaborate on understanding the data and verifying the correctness of data analysis in Sections 10.4.1 and 10.4.2.

10.4.1 Understanding the data

We found that fully understanding the data under investigation is of utmost importance. The data set used in the scope of this works consists of API error responses which contain an error message and error code. A naive approach would have been to start categorizing the errors based on the messages, or to apply machine learning techniques based on the different features in the data set. However, after looking into the requests that resulted in errors in more detail, we discovered that the error messages in many cases were not describing the actual fault that had occurred. Also, some of the messages were the result of different underlying faults. In addition, we found that our data set included API responses from internal processes, which are outside the scope of this work and therefore were excluded. If we had not done this we would have based our results on seemingly correct data, which would not accurately describe the actual situation. It took a significant amount of our research time to obtain the necessary domain knowledge and identify the underlying faults, but this allowed us to base our results on the actual faults resulting in errors instead of the misleading error messages. The lesson learned here is that understanding the data is critical to ensure accurate results. The approach to achieve this is to combine quantitative and qualitative research methods in order to arrive at rich, well understood data.

10.4.2 R testing

The second important lesson we learned is related to data analysis, mutations and aggregations for which we used R[3]. In Section 5.1 we described the steps taken to obtain a set of unique error messages. This involved splitting the data set based on messages that do and do not have an unique error code. For both sets a different approach was necessary to obtain the unique error messages after which the results were combined. In addition we had to preprocess the data set to remove messages that were out of scope. We experienced that wrong assumptions or bugs were easily missed and that the resulting data set would be incorrect. Especially when the logic is complex and changes have to be made at a later stage bugs were introduced. To verify the correctness of our R data processing and analysis we made use of the *stopifnot* function which translates to an assert in Java. We added asserts throughout the code to test whether the result of a series of statements was indeed

[3]https://cran.r-project.org/

expected. The lesson learned is that testing is important even when working in data analysis and statistic tools, such as R.

Chapter 11

Conclusion

API errors can indicate significant problems for API consumers. In the system under study over 60 thousand 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. To fill this gap of knowledge we researched the domain of Web API errors to investigate how the impact of API errors can be reduced.

To reduce the impact of problems caused by mistakes in API integration two things can be done: 1) Reduce the number of problem occurrences and 2) reduce the impact of individual problem occurrences. We reasoned that to be able to do this an understanding is needed of the errors that occur in an API integration. Furthermore, to make effective recommendations, an understanding is needed of the current practices and challenges of reducing the impact of API error related problems. To this end we answered five research questions using a combination of API error data and insights from API consumers.

RQ 1*: What type of faults, resulting in API errors, are impacting API consumers?*

Based on the API faults data, faults in API integration can be grouped into 11 categories: *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 category can be contributed to one of the four API integration stakeholders: *end user*, *API consumer*, *API provider*, and *third parties*.

The survey results support the 11 categories. Although the participants propose additional categories we regard them as part of previously identified categories.

RQ 2*: What is the prevalence of these fault types, and how many API consumers are impacted by them?*

Based on the API faults data, from a stakeholder perspective most faults, 39 out of 69, can be contributed to the API consumer. This compares to 86.3% to 87.7% of 2.43 million API errors and between 67.5% and 84.6% of the 1464 impacted API consumers.

From a category perspective most faults, 17 out of 69, can be contributed to the *invalid request data* category. However most errors, 36.0%, are related to *double processing* faults. Most API consumers seem to be impacted by faults in the *invalid request data* and *third party* categories.

Based on the survey results, *missing server data* and *configuration* faults were experienced the most by API consumers. Faults caused by the API provider and third parties, categories *internal* and *third party* respectively, were also found to impact relatively many API consumers. *Double processing* and *expired request data* faults were not experienced by over half of the participants.

RQ 3*: What type of faults do API consumers consider the most impactful?*

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 are regarded as having the least impact.

RQ 4*: What are the current practices to avoid and reduce the impact of production problems caused by faults in API integration?*

API consumers most often use official API documentation to implement an API correctly, followed by code examples. The impact of faults and potential problems is reduced via specific error handling, or, more frequently, via generic error handling. Most often, API related production problems are detected by the end user of the application, followed by log analysis and monitoring dashboards.

RQ 5*: What are the current challenges to avoid and reduce the impact of production problems caused by faults in API integration?*

The challenges of preventing problems from occurring are the lack of implementation details, insufficient guidance on the way to approach certain aspects of the integration, and lack of insights in problems and changes. Handling errors is experienced challenging due to an insufficient understanding of the impact of problems, missing guidance on how to recover and a lack of details on the origin of errors. Detection is challenging because of the unfeasibility of testing all scenarios, a lack of detailed exceptions, and the amount of work required to analyze logs regularly. In addition, missing, incomplete, incorrect, and unspecific documentation makes avoiding errors more difficult as well as APIs that cannot handle specific input, or do not do versioning properly.

To categorize faults, we obtained a set of fault descriptions from a set of 2.6 million API error logs. The approach used for this is illustrative and reproducible, and can possibly be used to replicate this work based on the API error logs from other APIs.

Based on the answers to the research questions we formulated 16 recommendations for API consumers and providers to reduce the impact of API faults:

1. **API consumer**: Be aware of the detection action per fault category
2. **API provider**: Provide API error insights
3. **API consumer**: Validate user input
4. **API provider**: Provide parameter information to enable validation
5. **API consumer**: Use continuous testing and periodic testing
6. **API provider**: Offer a complete set of test cases
7. **API consumer**: Fix API consumer faults
8. **API provider**: Provide implementation architecture recommendations
9. **API provider**: Provide effective documentation
10. **API consumer**: Handle end user faults by providing feedback and retrying
11. **API consumer**: Recover from API consumer faults
12. **API provider**: Distinguish between retriable and non-retriable errors
13. **API consumer**: Retry or recover from API and third party faults
14. **API consumer**: Use generic error handling for unknown errors
15. **API provider**: Provide enriched API error responses
16. **API consumer**: Prioritize on problem impact

We envision several angles of future work. To strengthen the results of this work the study should be replicated within the context of other Web APIs. More elaborate work on the current practices and challenges of reducing the impact of API related problems can be used to improve the proposed recommendations and add new suggestions. We propose to evaluate the recommendations by implementing the suggestions for the API provider and studying the effects in terms of the API error rate and problem impact. Furthermore, a characteristic study of Web APIs can help define the scope and applicability of future research. Other future work may include investigating the longitudinal effect of the occurrence of API errors and API evolution.

This study is the first to explore faults in Web API integration in an attempt to understand the type of faults that occur and their impact. By investigating the current practices and challenges of reducing the impact of these faults we proposed a set of 16 recommendations. 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.

Bibliography

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

[2] Handle 4xx and 5xx responses. https://docs.adyen.com/developers/ecommerce-integration/response-handling#errorcodesreturned. [Online; accessed 07-Aug-2017].

[3] Marketing Cloud API: Error codes. https://developer.salesforce.com/docs/atlas.en-us.noversion.mc-apis.meta/mc-apis/error_codes.htm. [Online; accessed 07-Aug-2017].

[4] Open API Initiative. https://www.openapis.org/. [Online; accessed 06-Sep-2017].

[5] Search the Largest API Directory on the Web. https://www.programmableweb.com/category/all/apis. [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. https://pages.apigee.com/eBook-Web-API-Design-The-Missing-Link-reg.html. [Online; accessed 28-Jun-2017].

[8] David Bermbach and Erik Wittern. Benchmarking web api quality. In *International Conference on Web Engineering*. Springer, 2016.

[9] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (SOAP) 1.1, 2000.

[10] David De Vaus. *Research design in social research*. Sage, 2001.

[11] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. Understanding the interactions between users and versions in multi-tenant systems. In *Proceedings of the 2013 International Workshop on Principles of Software Evolution*. ACM, 2013.

[12] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. 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] Peter Evers. Finding Errors in Massive Payment Log Data. 2017.

[14] Roy Fielding. REST APIs must be hypertext-driven. `http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven`. [Online; accessed 28-Jun-2017].

[15] Roy Fielding and Richard Taylor. *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation, 2000.

[16] Arlene Fink. *The survey handbook*, volume 1. Sage, 2003.

[17] Ceki Gülcü. *The complete log4j manual*. QOS. ch, 2003.

[18] Siw Elisabeth Hove and Bente Anda. Experiences from conducting semi-structured interviews in empirical software engineering research. In *IEEE 11th International Sympsium on Software metrics*, 2005.

[19] ISO/IEC 7812-1. Identification cards – Identification of issuers – Part 1: Numbering system, 2017.

[20] Barbara Kitchenham and Shari Lawrence Pfleeger. Principles of survey research: part 1-6. *ACM SIGSOFT Software Engineering Notes*, 26-28, 2002.

[21] Kin Lane. History of APIs. `http://apievangelist.com/2012/12/20/history-of-apis/`. [Online; accessed 28-Jun-2017].

[22] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, 1966.

[23] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

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

[25] Oracle. Java Platform, Standard Edition 8 API Specification. `http://docs.oracle.com/javase/8/docs/api/`. [Online; accessed 28-Jun-2017].

[26] Aniket Patil. Get developer hugs with rich error handling in your API. https://blog.box.com/blog/get-developer-hugs-with-rich-error-handling-in-your-api/. [Online; accessed 28-Jun-2017].

[27] Jane Radatz, Anne Geraci, and Freny Katki. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, 1990.

[28] Julian Reschke and Roy Fielding. RFC 7231-Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. June 2014. http://tools.ietf.org/html/rfc7231. [Online; accessed 03-Aug-2017].

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

[30] Martin Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering*, 39(5), 2013.

[31] Lior Rokach and Oded Maimon. Clustering methods. In *Data mining and knowledge discovery handbook*. Springer, 2005.

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

[33] Carolyn Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering*, 25(4), 1999.

[34] SmartBear Software. Swagger - The world's most populair API tooling. https://swagger.io/. [Online; accessed 06-Sep-2017].

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

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

[37] Philippe Suter and Erik Wittern. Inferring web API descriptions from usage data. In *Third IEEE Workshop on Hot Topics in Web Systems and Technologies*, 2015.

[38] Pradeep Venkatesh, Shaohua Wang, Feng Zhang, Ying Zou, and Ahmed Hassan. 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.

[39] Rick Wieman. An Experience Report on Applying Passive Learning in a Large-Scale Payment Company. In *Industry track-IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017.

[40] Erik Wittern, Annie Ying, Yunhui Zheng, Julian Dolby, and Jim Laredo. Statically checking web API requests in JavaScript. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017.

[41] Erik Wittern, Annie Ying, Yunhui Zheng, Jim Laredo, Julian Dolby, Christopher Young, and Aleksander A Slominski. Opportunities in software engineering research for web API consumption. In *Proceedings of the 1st International Workshop on API Usage and Evolution*. IEEE Press, 2017.

Appendix A

Adyen Fault Cases

FC1) "5_001 - ApplePay token amount-mismatch"

A mismatch occurred between the amount in the payment request and the amount in the ApplePay token due to a rounding problem. A one cent difference is detected where the payment request amount is, e.g., 72.20, while the amount in the ApplePay token is 72.21.

FC2) "110 - BankDetails missing" (1)

Due to missing response data from a third party payment method, recurring payments for a specific bank and payment method are failing. In this case Adyen resorts to a backup mechanism, which is unsuccessful and causes an internal error.

FC3) "110 - BankDetails missing" (2)

Bank details are required when storing details for shopper payouts in case there is no credit card information supplied. The bank details are missing in the request, hence this error is returned.

FC4) "131 - Billing address problem (City)"

The city name is a required field of the billing address. This value is not supplied.

FC5) "134 - Billing address problem (Country 0 invalid)" (1)

The country code should be a valid value ISO 2-character country code. This error is returned because the value is missing.

FC6) "134 - Billing address problem (Country 0 invalid)" (2)

The country code should be a valid value ISO 2-character country code. This error is returned because it does not conform to the standard.

FC7) "135 - Billing address problem (StateOrProvince)"

A valid 2-character abbreviation for the state or province is required for the United States and Canada. This value was not supplied causing this error.

FC8) "132 - Billing address problem (Street)"

The street name is a required field of the billing address. This value is not supplied.

FC9) "128 - Card Holder Missing"

The name of the card holder is required field. In case this value is missing this error is returned.

FC10) "0 - Configuration Problem mpiImplementation"

3D secure authorize calls require a payment authorization response from the issuer and a payment session identifier. Requests that do not contain these values fail with this error.

FC11) "800 - Contract not found" (1)

This error occurs when one attempts to disable a shopper contract that has been disabled in the past. The reference associated with the contract thus not found.

FC12) "800 - Contract not found" (2)

This error is caused by an internal database replication delay. When a recurring contract is created it takes a few minutes to be replicated to the slaves. If the contract is requests before replication the request fails with this error.

FC13) "800 - Contract not found" (3)

Attempting to use a recurring contract in combination with a payment method that does not support recurring payments results in this error. At the moment of writing Bancontact does not support this functionality there causing errors for merchants attempting recurring Bancontact payments.

FC14) "0_0 - Could not read XML stream.."

The values are not URL-encoded causing a failure in the XML parsing. The ampersand sign should be URL-encoded as '&,

FC15) "0 - Couldn't parse expiry year"

The expiry year, which is part of the credit card information, cannot be parsed to an integer. This happens when it contains letter or other non-digit characters. E.g., '20/2 cannot be parsed, in contrast to the likely intended '2012.

FC16) “103 - CVC is not the right length”

The credit card CVC code is not the expected length. Generally the code consists of three digits, with American Express cards being the exception with four digits.

FC17) “0 - CVC is required for OneClick card payments”

The credit card CVC code is required for OneClick card payments for which the credit card details are already stored. This value is however missing in the request.

FC18) “0 - Double processing”

This error occurs when a 3D authorize request is submitted more than once. The request has already been processed, hence an error is returned.

FC19) “172 - Encrypted data used outside of valid time period” (1)

The timestamp included is outside the valid time range. The timestamp is generated for the end user, but the end user takes too long to initiate the request.

FC20) “172 - Encrypted data used outside of valid time period” (2)

The timestamp included is outside the valid time range. Either the timestamp is off by a mistake on the server side, or the request is sent in for processing too late.

FC21) “0 - Expiry month not set”

The expiry month of the credit card is required in an authorization request. The value is missing in the request. This error most often occurs in case all credit card details are missing. This is because the expiry month is validated first.

FC22) “0 - Expiry month should be between 1 and 12 inclusive”

The expiry month should be a value between 1 and 12. The value in this case is lower than 1 or higher than 12, causing this error.

FC23) “0 - Expiry year should be a 4 digit number greater than 2000”

This error occurs when the expiry year of the credit card is not higher than 2000. When the shopper does not fill in this year, some merchants default to 0, causing this error to occur.

FC24) “0 - iDEAL communication error”

The select iDEAL bank is not available. This can be because of planned maintenance or a temporary connection problem.

FC25) "0_0 - internal Configuration Problem mpiImplementation"

3D secure authorize calls require a payment authorization response from the issuer and a payment session identifier. Requests that do not contain these values fail with this error.

FC26) "903 - Internal error" (1)

This error is caused by a configuration mistake. The configuration key, required for installments, is not found.

FC27) "903 - Internal error" (2)

For airlines additional airline data can be enabled to capture flight information such as the leg data. This error is thrown when airline leg data is missing.

FC28) "903 - Internal error" (3)

Invoice payment requests require order lines and an invoice number. This error is returned when either is missing.

FC29) "137 - Invalid amount specified" (1)

Captures are used to transfer money from one credit card account to another. The amount specified for a capture request has to be higher than zero. A problem with this error occurs when the amount is zero.

FC30) "137 - Invalid amount specified" (2)

Adyen offers the functionality to initiate a recurring contract by means of a zero amount authorization call. However, the minimum amount of iDEAL transactions is one cent, causing this error when a zero amount authorization call is made.

FC31) "137 - Invalid amount specified" (3)

An amount is invalid when it is below zero or higher than a specific limit. Transaction requests containing an out of range amount result in this error.

FC32) "137 - Invalid amount specified" (4)

A refund request requires an amount higher than zero. Zero amount refund requests fail with this error.

FC33) "101 - Invalid card number"

The supplied credit card number does not validate. It is either not the right length or does not validate against the Luhn formula.

FC34) “153 - Invalid CVC”

The CVC code does not validate. In this case it contains non-digit characters, which are not allowed.

FC35) “116 - Invalid date of birth”

For Open Invoice transactions the shopper has to be at least 18 years of age. This error is returned if the shopper is younger.

FC36) “161 - Invalid iban” (1)

The IBAN supplied for the transaction does not have a valid length. The maximum length is 34, but depends on the country.

FC37) “161 - Invalid iban” (2)

The country code supplied in the payment request has to match the country code of the IBAN. In case of a mismatch this error is returned.

FC38) “161 - Invalid iban” (3)

The IBAN is required for SEPA Direct Debit transaction. This error is returned when the IBAN is missing.

FC39) “901 - Invalid Merchant Account” (1)

In order for a merchant to capture an amount from a shoppers account Adyen requires the merchant account to be included in the request. In case this field is missing the request is rejected with this error.

FC40) “901 - Invalid Merchant Account” (2)

In order to process on Adyens platform the merchant account needs to be active. In this case an inactive account is used to process payments, resulting in an error.

FC41) “901 - Invalid Merchant Account” (3)

To process payments the supplied merchant account has to exist. Requests result in an error when an invalid account is used.

FC42) “105 - Invalid paRes from issuer” (1)

3D secure authorize calls require a payment authorization response from the issuer and a payment session identifier. This error is returned when Adyen tries to process the payment with the issuer, which is unable to process their own payment authorization response.

FC43) "105 - Invalid paRes from issuer" (2)

3D secure authorize calls require a payment authorization response from the issuer and a payment session identifier. This error is returned when the merchant fails to forward the valid payment authorization response and payment session identifier.

FC44) "906 - Invalid Request: Original pspReference is invalid for this environment!"

Adyen provides a test environment for its customers to test their integration with test payments. Payment references in the production environment are invalid in the test environment, and vice versa. This error is returned when a merchant attempts to use a payment reference of a different environment.

FC45) "109 - Invalid variant"

Credit cards come in different variants. Some variants are a subtype of another variant. For instance, the Mastercard commercial premium credit card is a subvariant of the regular Mastercard card variant. An error occurs when the payment contains a credit card or one variant which is not a sub(variant) of the variant supplied in the variant field.

FC46) "100 - No amount specified"

Payment requests such as authorize, capture and refund require an amount. This error is returned when the amount is not supplied.

FC47) "0 - No InitialPspReference provided"

The status of iDEAL payments can be checked by sending in the payment reference in combination with issuer information. A signature is required so that the request can be verified. When the signature is not valid this error is returned.

FC48) "113 - No InvoiceLines provided"

Open Invoice requests require invoice lines with details about the purchased items. When these lines are omitted this error is returned.

FC49) "10 - Not allowed" (1)

API endpoints have to be configured, before they can be used. When a Point of Sale configured account tries to use the ecommerce endpoints an error is returned indicating that the request is not allowed.

FC50) "10 - Not allowed" (2)

A merchant has to be PCI Level 2 certified to handle credit card information. Only then they can use certain request fields for the credit card. Else they have to encrypt the card data

on the client side and send the data in an encrypted blob. Merchants that are not certified and attempt to use the former method receive this error.

FC51) “167 - Original pspReference required for this operation” (1)

For modification action such as capture, cancel and refund a valid reference to the authorization is necessary. In this case the merchant sends in invalid references.

FC52) “167 - Original pspReference required for this operation” (2)

For modification action such as capture, cancel and refund a reference to the authorization is necessary. The merchant in this case neglects to send in this reference.

FC53) “905 - Payment details are not supported”

An acquirer configuration is needed to do a transaction in a specific currency using a specific payment method. If this configuration is missing or incorrectly set up, then payments for these combinations will fail with this error.

FC54) “907 - Payment details are not supported for this country/ MCC combination” (1)

A Merchant Category Code (MCC) is a classification used to describe the services or type of products offered by that merchant. Depending on this classification accepting payments in certain countries is not allowed, and payments are rejected with this error message.

FC55) “907 - Payment details are not supported for this country/ MCC combination” (2)

Adyen doesnt support payments in certain countries. Payments attempted in these countries are rejected with this error.

FC56) “803 - PaymentDetail not found” (1)

The merchant in this case attempts to disable a contract that does not exist. This situation occurs when the references to contracts are not properly managed or the contract has been disabled in the post.

FC57) “803 - PaymentDetail not found” (2)

The merchants attempts to authorize a payment using a recurring contract that has been disabled, updated or removed in the past. The contract can no longer be found, hence this error is returned.

FC58) “0 - Please supply paymentDetails”

To authorize a transaction payment details are required. This error is returned when no payment details are in the request.

FC59) “0 - Recurring requires shopperReference”

Recurring payments require the shopper reference field to be included, although optional for normal payment requests. This error is returned when merchants neglect to send in the shopper reference.

FC60) “130 - Reference Missing”

A merchant reference, created by the merchant, is used to uniquely identify payments. This value is required for all payment authorization and this error is returned if missing.

FC61) “174 - Unable to decrypt data” (1)

For non-PCI certified merchants Adyen offers a client-side encryption library that encrypts credit card information before submitting it via the API. In case requests are submitted with an empty encrypted data field the system is unable to decrypt the data, hence returning this error.

FC62) “174 - Unable to decrypt data” (2)

Credit card information encrypted in the shoppers browser is to be decrypted by Adyen. In a limited number of cases this encryption fails with an internal exception, causing this error to be returned.

FC63) “102 - Unable to determine variant”

To determine how to process a transaction Adyen determines the payment method variant from the supplied card number. When the card number is invalid this process fails, causing this error to occur.

FC64) “175 - Unable to parse JSON data”

This error occurs when the encrypted data containing credit card details cannot be parsed to JSON after decryption. This can happen when a shopper maliciously alters the data before it is encrypted in the browser.

FC65) “138 - Unsupported currency specified” (1)

A payment request requires the currency to be specified. When the value is omitted the request cannot be processed and this error is returned.

FC66) “138 - Unsupported currency specified” (2)

SEPA Direct Debit recurring payments can only be processed in euros. Although initiating the contract can be done in different currencies, not being aware of the currency restriction for subsequent payments can cause this problem.

FC67) “0_0 - validation expired, pspReference=*************”**

For Direct e-banking payments a session validity of 30 minutes is set. Requests that are processed outside the valid timeframe are rejected with this error.

FC68) “0_0 - ” (1)

This request fails because the merchant sends in the amount value of a payment request as a floating point number, instead of an integer. The API fails to respond properly and returns an empty message.

FC69) “0_0 - ” (2)

The merchant in this case tries to process a 3D secure payment request multiple times, which results in an error. The API does not handle the error well resulting in an empty response.

Appendix B

Interview Guide

Introduction

- Introduction of the interviewer
- Introduction to the research project
- Asking for permission to record the interview
  - Mention anonymity of the company and interviewee
  - Explain the recording is only processed by the interviewer

Background questions

1. Could you tell me about what it is you do at <**company**>?
2. How many people at <**company**> are working on your integration with Adyen?
   a) How many of these are developers?

General questions

- Introduction to the general questions; what they are about and why they are useful

1. What kind of testing do you have in place to verify the correctness of your API integration?
   - Optional examples: unit tests and integration tests

   a) Why do you have this in place? *(conditional)*
   b) Why is this not the case? *(conditional)*
2. How do you handle API error responses?

- Background: Adyen has over 150, for example <**error experienced by interviewee**>

a) What is the default fallback for other response codes?
b) Why are these errors not handled with specific logic? *(conditional)*

3. What methods do you use to monitor your application?

- Optional examples: logs and alerts

a) Why do you have this in place? *(conditional)*
b) Why is there no monitoring in place? *(conditional)*

4. How do you experience the reporting of errors by Adyen?

a) How can the reporting be more clear?

5. How did the integration process with the Adyen API go?

a) Could this have been better, and how?

6. During integration, what did you use to learn how to use the API?

- Optional examples: documentation, code examples, technical support and implementation managers

Problem specific questions

- Introduction to the problem specific questions; what they are about and why they are useful

1. How would you describe the problem?

2. What was causing the problem?

3. What could have been done to prevent this problem from happening?

a) Why was this not in place/done?

4. What was causing the problem to go undetected? *(conditional)*

5. What were the consequences of this problem for you as a merchant?

a) Why is this a problem? *(conditional)*
b) Why is this not a problem? *(conditional)*

6. What can Adyen, as API provider, do to help you as a merchant prevent situations like this?

Appendix C

API Integration Survey

Problems in Web API Integration

API integration survey

Dear participant,

In a perfect world the integration between a web API and a deployed application works without any issues. In reality, however, API related problems happen and may have a large impact on the user and business. Although there are numerous API best practices guides out there, these seem insufficient to prevent API problems.

We are analyzing *what* type of problems occur in web API integrations and *why* they happen. With web API integration we refer to applications using remote services over the web by means of communicating with an API.
By collecting enough answers, the published results will include advice for API designers on how they can help the API consumer prevent problems, and advice for API consumers on how to improve their integration. In the end of the survey you can register your email if you wish to receive the results of this study.

This survey takes about 12-15 minutes to complete and the answers will solely be used in anonymized form.

Thank you!

Web API integration experience

VALIDATION Must be numeric Whole numbers only Positive numbers only

1. How many years of experience do you have in software development? *

VALIDATION Must be numeric Whole numbers only Positive numbers only

2. How many years of experience do you have in integrating with web APIs? *

3. Have you worked on integration with a web API for an application in the past 3 years? *

- Yes
- No

4. Is/was the application using the API integration used in production? *

- Yes
- No

Web API characteristics

In this survey we would like you to answer the questions based on an **integration with a web API** that **you** worked on for an application that is/was **used in production**. Please select the integration project from the **past 3 years** which you consider to be the **most complex**. This can be based on the offered features and functionality, number of required/optional parameters and/or number of possible error scenarios.

Please answer the questions with **your experience** in mind, instead of what would be the ideal situation or conforms to best practices that you are aware of.

5. How would you categorize your application that is integrated with the API?

- [ ] Professional
- [ ] Hobby
- [ ] Research
- [ ] Open source
- [ ] Other - Write In

6. What does the API you integrated with do? *

VALIDATION Min = 1 Must be numeric Whole numbers only Positive numbers only

7. In your project, how many developers are/were working on integrating with the API? (Including yourself) *

8. How complex do you consider the API to be on the following properties? *

	Do not recall	Not at all	Slightly	Moderately	Very	Extremely
Number of different features and complexity of the functionality	○	○	○	○	○	○
Required/optional parameters and parameter formatting constraints	○	○	○	○	○	○
The number of possible different errors returned by the API	○	○	○	○	○	○
Overall	○	○	○	○	○	○

API integration process

Please answer the following question regarding the integration with the API you selected in mind.

9. During the integration process of the API with the application, I made use of the following information sources to learn how to use the API:

	Never	Rarely	Sometimes	Often	Very often
Official API documentation	○	○	○	○	○
Code examples	○	○	○	○	○
Question and Answer sites, such as StackOverflow	○	○	○	○	○
Support team (API provider)	○	○	○	○	○
Enter another option	○	○	○	○	○

10. How can the API provider help improve the integration process?

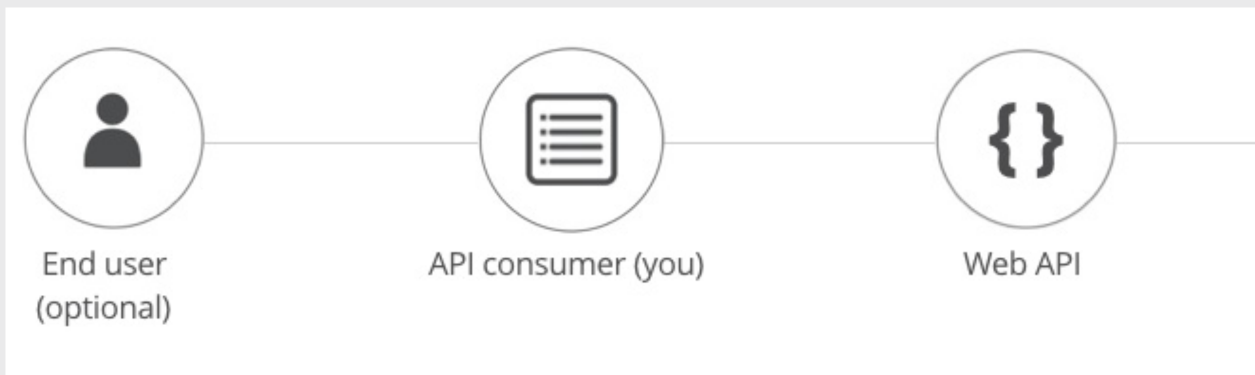
Production problems in the API integration

VALIDATION **Min. answers = 11** *(if answered)*

We now move away from the integration process and instead consider the i**n production use of the API**.

The following questions relate to problems identified **in production** that are related to the API integration.

Please take the following stakeholders and their interaction in mind.
You may have an *optional* **end user** interacting with your application.
The **application** in turn interacts with the **API** via the integration.
The API may *optionally* communicate with **third parties** to utilize other services.



11. My application experienced problems **in production** with the API in the following categories: *

	I don't know/Not applicable	Never	Rarely	Sometimes	Often
Invalid user input (e.g., malformed input by the end user of the application)	○	○	○	○	○
Missing user input (missing input by the end user of the application)	○	○	○	○	○

application)					
Expired request data (the input data was no longer valid at the moment of processing)	○	○	○	○	○
Invalid request data (e.g., malformed input data caused by the API consumer)	○	○	○	○	○
Missing request data (missing input data caused by the API consumer)	○	○	○	○	○
Insufficient permissions (not enough rights to perform the intended action)	○	○	○	○	○
Double processing (the request was already processed by the API)	○	○	○	○	○
Configuration (a problem caused by missing/incorrect API settings)	○	○	○	○	○
Missing server data (the API does not have the requested resource (e.g., document or object))	○	○	○	○	○
Internal error (a problem caused by the API)	○	○	○	○	○
Third party error (a problem caused by a party integrated with the API through which services are offered)	○	○	○	○	○
Enter another option	○	○	○	○	○
Enter another option	○	○	○	○	○
Enter another option	○	○	○	○	○

Production problems in the API integration

The following questions relate to problems identified in production.

12. The impact of the following problems **in production** in my API integration is/was:

	I don't know	None	Low	Moderate	High
Invalid user input (e.g., malformed input by the end user of the application)	○	○	○	○	○
Missing user input (missing input by the end user of the application)	○	○	○	○	○
Expired request data (the input data was no longer valid at the moment of processing)	○	○	○	○	○
Invalid request data (e.g., malformed input data caused by the API consumer)	○	○	○	○	○
Missing request data (missing input data caused by the API consumer)	○	○	○	○	○
Insufficient permissions (not enough rights to perform the intended action)	○	○	○	○	○
Double processing (the request was already processed by the API)	○	○	○	○	○
Configuration (a problem caused by missing/incorrect API settings)	○	○	○	○	○
Missing server data (the API does not have the requested resource (e.g., document or object))	○	○	○	○	○
Internal error (a problem caused by the API)	○	○	○	○	○
Third party error (a problem caused by a party integrated with the API through which services are offered)	○	○	○	○	○
Other (if applicable)	○	○	○	○	○

13. In your experience, what causes these types of API errors happen in production?

API error response handling and detection

The following question is related to the handling of error responses returned by the API.
Often these errors are uniquely identified by an error code and message.
For example the Twitter API can return message "User not found" with error code "50".

We refer to specific, generic and no logic to handle errors:

- **Specific logic**: custom error handling for unique error codes or messages, such as the Twitter error described.
- **Generic logic**: error handling is the same for a group of error messages, such as all error messages returned with HTTP status code 500.
- **No logic**: no error handling in which case errors are not acted upon.

14. In your application API errors are/were handled using:

	I don't know	Never	Rarely	Sometimes	Often	Always
Specific logic to handle unique API errors	○	○	○	○	○	○
Generic logic to handle multiple API errors	○	○	○	○	○	○
No logic to silently swallow API errors	○	○	○	○	○	○

15. In your experience, what are the challenges of handling API error responses?

16. In the project, problems in production with the API integration are/were detected via:

- ☐ Alerts (sms/email/etc.)
- ☐ Monitoring dashboards
- ☐ API integration tests
- ☐ Log analysis
- ☐ End user inquiries
- ☐ Other - Write In
- ☐ None of the above

API problem detection

17. Why is there no detection mechanism in place?

18. What could you do to improve the detection of problems with the API?

19. Why is this not in place?

20. How can the API provider help you prevent problems you experience with the API?

Wrapping up

21. Feel free to add any further comment on this survey, topic, etc.

VALIDATION %s format expected

22. If you would like to receive the results of this study, please leave your email here.

Results are only used **anonymously**. Your email will **not be shared** with anyone and you will **not receive** any other emails than the results of this study.

Thank You!

Thank you for taking our survey. Your response is very important to us.