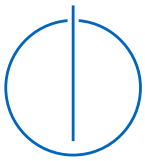
DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Design and Implementation of a Data-Driven, Provider-Independent Test Service for Banking APIs

Josef Maria Kamysek



DEPARTMENT OF INFORMATICS

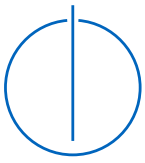
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Design and Implementation of a Data-Driven, Provider-Independent Test Service for Banking APIs

Design und Implementierung eines Datenorientierten, Anbieterunabhängigen Testservice für Banking APIs

Author:	Josef Maria Kamysek
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	M.Sc. Gloria Bondel
Date:	April 15, 2020



I confirm that this bachelor’s thesis is my own work and I have documented all sources and material used.

Munich, April 15, 2020 Josef Maria Kamysek

Acknowledgments

This thesis would not have been possible without the help of many people. In the following lines, I would like to thank those who supported me and contributed to my work. The opportunity to write my thesis at the Chair of Software Engineering for Business Information Systems at the Technical University of Munich was a genuinely great opportunity. Therefore, I would like to thank my supervisor Prof. Dr. Florian Matthes, for giving me the chance to write my thesis in cooperation with the industrial partner msgGillardon.

Furthermore, I would like to thank the msgGillardon AG for providing me access to their network and information for my work. My profound respect and thanks goes to my advisors M.Sc. Gloria Bondel from the chair of Software Engineering for Business Information Systems from the Technical University of Munich and M.Sc. Markus Kraft from my industry partner msgGillardon for their guidance on my topic. Without their help, my thesis would not have been possible. Their advice, constructive feedback and encouragement were a continual source of motivation for me. Whenever I had questions, I could always rely on them and their great feedback. Thank you for your guidance and for giving me the opportunity to work on this exciting and challenging project.

In this context, I would also like to thank all my interview partners who shared with me their valuable knowledge, gave me many insights and the information that I needed for my research. Finally, I would like to express my deepest gratitude to my family and friends for their support, patience and backing throughout my studies.

Abstract

In 2015, the Council of the European Union introduced the Second Payment Service Directive (PSD2), which provided the legal basis for the creation of a uniform and efficient marketplace for payment services within the European Union (EU). It opened the financial market to new market participants and innovative services. The PSD2 requires banks to provide access to their infrastructure. Implementing Application Programming Interfaces (APIs) enables the exchange of financial data of bank customers with Third Party Providers (TPPs). The majority of banks and associations in Germany rely on the concepts and open standards defined by the Berlin Group. This initiative focused on the technical and organizational requirements to establish a standardized communication layer from which both banks and TPPs can benefit.

The verification of API implementations plays an essential role in the financial sector. Flawless functionality of PSD2-compliant APIs is particularly crucial in an industry mainly based on trust and confidence. Tests must be performed to determine whether or not the APIs meet the expectations in terms of functionality, reliability and performance. Early feedback shortens the error correction process and prevents costly, time-consuming problem fixes at a later stage in the development cycle. Therefore it is essential to have a well-functioning, efficient test process that ensures high quality and correct functionality.

This bachelor thesis presents an approach to improve the test process of different bank interfaces. Based on Hevner's Design Science in Information Systems Research, an artifact was developed that makes the test process of PSD2-compliant APIs simpler and more efficient. Before its implementation, the requirements for such a tool were identified through interviews and literature research. Based on these findings, a prototype was developed to demonstrate vendor-independent and data-driven testing for the Access to Account (XS2A) interface of the Berlin Group. The tool uses a workflow from the Account Information Service (AIS) of the Berlin Group to demonstrate the test process. Financial institutions can use this tool to evaluate a predefined PSD2-compliant bank interface by dynamically adding test cases to it. Parallel to its implementation, the extensibility of the program was evaluated.

The results of this work show that the test process for PSD2-compliant APIs faces many challenges. The large number of different standards makes the evaluation of bank interfaces difficult. Despite these challenges, the prototype demonstrates a viable approach to improve and facilitate the test process of PSD2-compliant bank interfaces. Through the automation of this test process both costs and effort can be reduced. Furthermore, recommendations for future functionalities are presented based on an evaluation of the usability and extensibility of the test tool.

Contents

Outline of the Thesis

Part I: Introduction and Theoretical Framework

Chapter 1: Introduction

In addition to the motivation, the research topic, and the applied research approach of this thesis, the research questions and their objectives are presented.

Chapter 2: Theoretical Framework

Theoretical background information is explained to enable an easy understanding of the most important research results.

Chapter 3: Related Work

An overview of different API test approaches and tools for testing PSD2-compliant APIs is given to allow a comparison with the scientific appeal of this research.

Part II: Design and Implementation

Chapter 4: Test Tool for PSD2-compliant APIs

Detailed explanation of the test tool process, the test tool architecture and the functionality of the test case mapping.

Part III: Appraisal and Conclusion

Chapter 5: Test Tool Evaluation from a User Perspective

User-based evaluation of the test tool in regard to usability, user conformity and missing functionalities.

Chapter 6: Extensibility Analysis of the Test Tool

Analysis of the ability to evaluate additional test cases, add new API calls and integrate new PSD2-compliant standards.

Chapter 7: Conclusion

Presentation of key results, problems and learnings of the thesis, followed by an outlook for future development.

Part I.

Introduction and Theoretical Framework

1. Introduction

In November 2014, the British Competition and Markets Authority (CMA) started an investigation of the provision of retail banking services because they conceived the British financial sector as insufficiently customer-friendly and competitive [Smith et al., 2016]. The accompanying report was published in August 2016 and confirmed their assumptions. Larger banks had easier access to the market and less competition than smaller banks [Open Banking Limited, 2016]. According to the retail banking market investigation, larger "banks will only invest in new products or services or reduce their prices and improve service quality, if they expect to win business as a result, or fear losing business if they do not" [Competition and Markets Authority, 2016]. This means that small banks and fintechs need to invest more in order to reach a similar customer base as the established banks. Given this knowledge, the CMA wanted to change the traditional way banks operate in the financial sector [Competition and Markets Authority, 2016]. They aimed to increase competition and to foster the transition to a digitalized banking era. In order to retain customers and adapt to their changed behavior, banks were forced to invest more in innovative products and improved customer relations [Accenture, 2016]. In response to this issue, the CMA formed an implementation entity for open access to the banking infrastructure to enable a more transparent and security-oriented banking experience.

Almost simultaneously, the Council of the European Union introduced the PSD2, which formed the legal basis for the creation of a single and more efficient payment service market within the EU. With similar intentions as those of the Open Banking Implementation Entity, they introduced a clear and comprehensive set of rules. They aimed to open up the banking sector, through the provision of an operational interface for TPPs, thus enabling them to access account information. Of course, the consumer must have given his consent for this access in advance. Furthermore, the directive obliges banks to implement stricter security requirements to guarantee better protection of customer data, and promote transparency of banking conditions [Lynn et al., 2019].

Regulatory pressure and growing interest in open APIs prompted banks to develop a standardized communication interface from which both banks and TPPs could benefit [European Parliament, 2015]. Several standardization initiatives were founded to develop a uniform approach to this topic. These standardization initiatives consist of several dozen financial institutions. Their tasks included the development of implementation and design guidelines to facilitate the implementation of the PSD2. One of the best-known European wide representatives of these standardization initiatives is the Berlin Group [Berlin Group, 2020]. Their goal is an open and standard scheme for

the interbanking domain [Berlin Group, 2020, Kirchmann, 2017]. This prompted a fundamental change in the way financial services are designed and offered to customers. Banks can draw benefits by using their APIs as a marketing tool or by creating services for previously neglected niches. Throughout the API development cycle, developers must ensure the correctness of the implementation. This testing process includes the validation of the API. Testing of these is considered as a rather complicated and error-prone task [Soap UI, 2019, Edgescan, 2018, KPMG, 2019]. A well-functioning and efficient test process is, therefore, essential to ensure high quality and correct functionality [Interview4, 2020]. The following chapters discuss the relevance of such a test process and the additional complexity that emerges through the use of these newly introduced API interfaces from a vendor and user perspective.

1.1. Motivation

The use of APIs gives Third Party Providers (TPPs) the opportunity to make banking more convenient and customer-oriented. The provision of standardized APIs is a significant investment for the future, costing banks a lot of time and money. Ensuring the quality and stability of such APIs must, therefore, be a top priority. However, due to time pressure and high testing efforts, this task is often reduced. The effort required for the testing process of Representational State Transfer (REST) APIs is often underestimated [Interview4, 2020]. In fact, it is seen as a problematic, time-consuming and error-prone task that leads to increased costs and reputational risks [Edgescan, 2018, KPMG, 2019].

To verify the response of an API, complex test scenarios are required. Especially the case of information retrieval for the end-user requires compound workflows consisting of multiple API calls. The current state of the art is that test programs are usually able to perform such workflows. One of the biggest challenges lies in the automation of these processes, especially if several bank interfaces should be tested with one test tool using a single test case catalog [Interview2, 2020, Interview3, 2020]. This is where conventional test programs reach their limits. Especially the differences of the interface structure prevent the execution of the same test cases without adaptation to the new interface. Each and every test case must be adapted to the new structure. For example, specific headers have to be renamed and the structure of the request body has to be changed. These are some reasons why a more flexible and comprehensive test tool is needed.

With the increasing complexity of software solutions, the source of human mistakes increase as well. Simply increasing the number of test engineers, which also means increased costs, is not a plausible solution. The demand for a useful and automated test tool that can perform these tasks must be satisfied. Banks urgently need an efficient testing tool to verify the current and future quality as well as the functionality of their PSD2-compliant APIs [Interview1, 2020]. Banks should work together on a standard test case catalog that is as comprehensive as possible and covers the majority of

problematic situations [Interview1, 2020]. This catalog can then be executed with a program on all diverse interfaces. There are already organizations that deal with this topic, for instance, the NextGenPSD2 Implementation Support Program (NISP) [NISP, 2019].

One of the main objectives of this thesis is to improve and to simplify the testing process while ensuring a more reliable, higher-quality test tool. The accuracy of a test tool is crucial for a good test process. This does not only reduce the test efforts but also the overall development costs [Marvin Zelkowitz, 2009]. Factors such as the profundity of test cases, the velocity of test execution and the data availability must not be neglected during the design and implementation of the test tool [Interview2, 2020]. They should be a top priority and are essential for the overall success of the project. The European Banking Authority (EBA) advocates the use of automated test programs, as these can increase productivity and scalability [European Banking Authority, 2019]. Regression tests facilitate the reuse of test cases and reduce the execution time. This accelerates the entire feedback process for the developer. It also helps to determine whether changes in the code have led to dependencies, defects or malfunctions [Warsi, 2016]. In general, regression testing leads to faster error detection and improved error correction [Mohanty et al., 2017, Jeff Offutt, 2018].

The development of such a test tool allows the testing process to be performed independently of the developers. This allows an objective view and avoids bias by the developers, leading to more error detection. Furthermore it reduces cost and effort of the development process of the API.

In summary, the motivation of this thesis is to develop the requirements as well as the corresponding prototype under consideration of the state of the art research concepts for an efficient, extensible and user-friendly process for testing different API implementations with a large amount of data.

1.2. Problem Statement

Ensuring quality and reliability does not become easier as software projects grow. With increasing complexity, the number of errors during development increases. Errors pose a serious threat to the success of the project [Interview4, 2020]. It is essential to test the software to ensure that it works correctly. Potential errors that are visible to the customer can cause bad publicity for the company [Jeff Offutt, 2018, Qa, 2019].

When projects are behind schedule, testing time is often sacrificed to make up for delays [Interview3, 2020, Interview4, 2020]. The shortening of the testing process is one of the main reasons why rework is necessary and large amounts of money are lost during software development. Software defects damage the company’s reputation and change the user’s attitude towards the product because they can affect its usability and security [Mohanty et al., 2017].

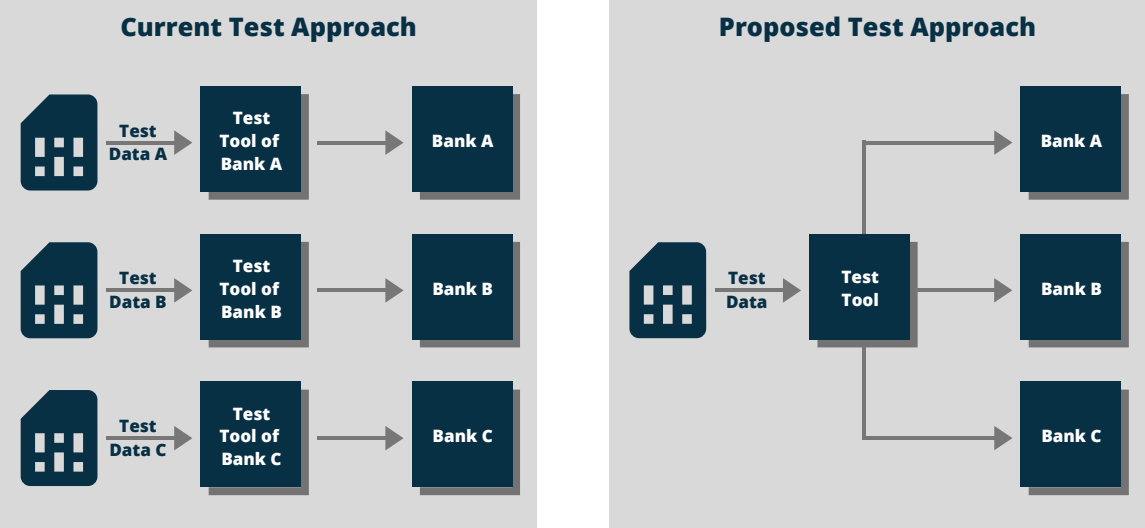


Figure 1.1.: Comparison of Current and Proposed Test Approach

In a large and complex system, undetected errors are more likely to occur. The reasons for these human-made errors are lack of understanding, communication problems and misinterpretations [Marvin Zelkowitz, 2009, Interview4, 2020]. Verifying the logic and compliance with the functional requirements of PSD2-compliant APIs is an enormous and challenging task for banks [Interview1, 2020, Interview2, 2020].

Strict quality requirements and the goal of guaranteed error detection increase the demand for more efficient software testing. The evaluation of APIs with a test tool that can only evaluate them through a Graphical User Interface (GUI) is an expensive and demanding challenge. Often, these applications are used for manual testing because they are quick to configure and easy to use [Interview4, 2020]. However, once the test cases become complex, the overview is lost. That makes it difficult to evaluate many test cases at once.

Additional functionality may also be required to test the interfaces. The integration of third-party software, such as a reporting system, may be necessary to adapt the program to the specific requirements of the user. GUI test tools rarely offer this type of integration. In most cases, it is not available at all.

The problem of the current test process is shown in figure 1.1. Under the present test approach, each organization is responsible for its testing procedures and the correctness of its API implementations. That results in significantly different qualities of test environments and test coverage. Besides, there is a constant need for adaptive test objectives due to adjustments to requirements during different development cycles [Schwertner, 2019].

Many banks use similar PSD2-compliant APIs, which essentially represent one type of bank interface standard. For this reason, it is assumed that it should be possible to

evaluate all APIs of one standard with just one test tool. Consequently, working together on a single test data set and a single test tool would make the testing process more efficient and less time-consuming [Interview3, 2020]. Moreover, the development of a specific test automation tool would allow a better adaptation to the banking environment. Many advantages are associated with the possibility of adapting the test suite to the specific test area. For example, it is possible to perform tests with a large amount of data more efficiently. Beyond that, an intelligent layer can be offered, which allows an adaptation to the respective bank interface.

In this thesis, the efficiency and complexity problems of the individual test procedures are addressed by a uniform test approach. The combination of the individual problems makes it possible to create a more straightforward test procedure for all interfaces.

1.3. Research Questions and Objectives

The goal of this thesis is to develop the requirements and implementation of a test tool for PSD2-compliant APIs. The tool can verify the functionality and correctness of the interfaces of several banks using dynamic test case entries and regression testing. The following research questions were defined to achieve this goal in the best possible way. Both the questions and their objectives are briefly explained below.

1. **What are the requirements for a data-driven and vendor-independent regression testing tool for PSD2-compliant APIs?**

The first research question examines the requirements for a test tool that uses dynamic test case entries to make requests to multiple bank interfaces. The knowledge gained from five expert interviews and a thorough literature research was used to introduce essential concepts of open banking and software testing. Thus, necessary theoretical concepts such as API testing or regression testing were introduced to understand the functionality of the test tool. The respondents enriched both the functional and technical aspects of this thesis with their outstanding expertise. Their know-how and insights served as a basis for the development of the test tool.

2. **How could the design and implementation of a regression testing tool for PSD2-compliant APIs look like?**

The second research question deals with the actual implementation of the prototype. Interviews and findings from literature research were used for its creation. The prototype aims to verify different API implementations. For this purpose, an extensive data set with test cases can be imported to verify bank interfaces. An overview of the architecture of the test tool is given to understand its structure and functionality. Additionally, the mapping of test cases is explained, since this is necessary to address the different interfaces.

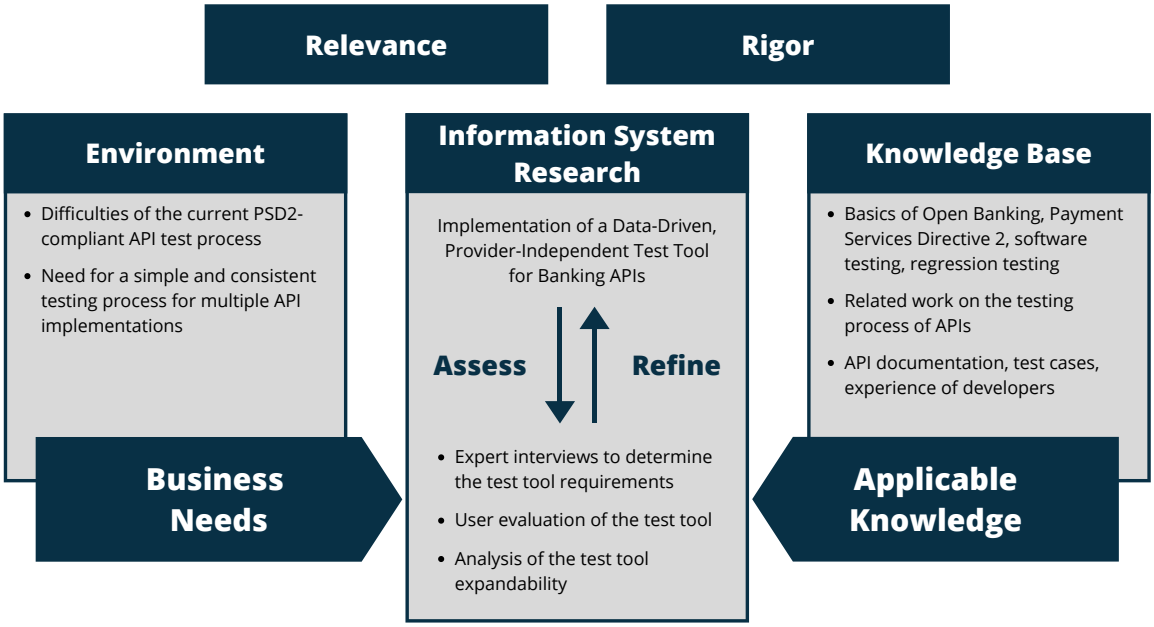


Figure 1.2.: Information Systems Research Approach - Test Tool for Banking APIs

3. **How easily could the test tool be extended to check the functionality and correctness of the entire banking API?**

The third research question evaluates the expandability of the prototype. The goal is to assess how easy and uncomplicated it is to add functionality. For this purpose, we distinguish between three different types of extensibility. First, we examine how easy and uncomplicated it is to add additional test cases. Then we assess how straightforward it is to add new functionalities in the form of new API calls to the program. Finally, we will also consider the possibility of adding a completely different PSD2-compliant API standard. That makes it easier to understand how the program can be expanded and future improvements can be made.

1.4. Research Approach

This thesis aims to contribute to the ongoing test process of PSD2-compliant bank APIs. Using an automated test tool helps to improve the test process. It reduces costs while still ensuring a high-quality product. The framework for Design Science for Information Systems proposed by Hevner et al. 2004 is applied in this thesis [Alan R. Hevner et al.,

2004]. This approach has been utilized in many other research projects and is an excellent method to achieve scientific rigor.

The conceptual framework facilitates the understanding, implementation and evaluation of information systems research [Alan R. Hevner et al., 2004]. This approach is scientifically recognized and provides guidelines for building a relevant artifact based on a thorough knowledge base. It is also driven by a business need from an environment with a defined problem. In our case, we are looking for a more efficient test process for multiple bank interfaces, since the current test process is a very complicated and costly procedure. An overview of the framework used in this thesis is shown in figure 1.2. It mainly consists of three parts, namely the environment, the information system research and the knowledge base. Each of these three areas is briefly discussed below.

The environment reflects the interest in the topic and the problem. The current testing process of bank APIs is a complex and inefficient procedure. In this thesis, a more efficient solution for this problem is presented. The presented approach is of particular interest for banks and the legislator, since a more efficient solution would simplify the task of testing the individual interfaces. According to the Design Science in Information Systems Research (ISR), the environment is used to assess and evaluate "business needs [...] in the context of organizational strategies, structure, culture and existing business processes" [Alan R. Hevner et al., 2004]. This means that fundamental concepts and related research are considered for the design and implementation of our artifact. Therefore, an overview of current research and testing approaches for bank interfaces is given.

The knowledge base provides an introduction to open banking, PSD2 and software testing. In particular, it provides an overview of the most important topics for this thesis, such as API testing and regression testing. In order to create a comprehensive knowledge base, original English and German literature, relevant publications on PSD2 and software testing as well as official papers of the EU are used. Further insights to solve relevant problems are gained through expert interviews. The five respondents contributed know-how and insights to the functional and technical part of this work. A summary of the respondents as well as the duration of the interviews can be found in figure 8.1 of the appendix. Each respondent was asked about the requirements for an efficient and effective test tool for PSD2-compliant interfaces. Moreover, they commented on current problems of the test procedure of PSD2-compliant bank interfaces. The questionnaire used in the conducted interviews can be found in figure 7.3 of the appendix. It is essential to provide an understanding of the basic concepts of the test tool. That helps to gain a deeper understanding of its functionality.

The evaluation of the interviews, as well as the findings from the literature, helped to determine the essential requirements for the design and implementation of the prototype. To be able to make a statement about the functionality, an evaluation of the extensibility and usability of the artifact was carried out. That evaluation is supported

by a refinement process that was established during the development of the test tool. It helped to gain new insights and thus supported the further design and development of the test tool. In general, this research approach aims at a continuous improvement of the artifact through multiple iterations of assessments and evaluations [Alan R. Hevner et al., 2004].

2. Theoretical Framework

This chapter explains the theoretical foundations and the relevance of this scientific research. Since this thesis deals a lot with Application Programming Interfaces (APIs), the fundamentals are briefly explained. This will provide a general understanding of what we are dealing with. In order to understand the necessity of open banking and its relation to the PSD2, the history of both is briefly presented. Together with their development, the expected potentials and their focal points are elaborated.

The description of the legal background is followed by a brief overview of software testing. This introduction focuses on accompanying objectives and concepts such as API testing, regression testing and test automation. A short definition of regression testing and its applicability in the area of API testing is outlined. That helps to answer the question regarding the requirements for a data-driven and vendor-independent regression testing tool for PSD2-compliant APIs. It is essential to understand how the prototype and especially the mapping of test cases, one of the main contributions of this thesis, work. Therefore, the focus is on concepts that are necessary to enable an efficient test procedure.

2.1. Foundations of Application Programming Interfaces

This section deals with the basic concepts of Application Programming Interfaces (APIs). Since these are referred to quite often in this thesis, it is essential to understand what they are and primarily how they work.

According to the Oxford Dictionary, APIs are "a set of programming tools that enables a program to communicate with another program or an operating system" [Oxford Learners Dictionaries, 2018]. In other words, they are used to disclose parts of a software system to a third party [Massé, 2012]. In many cases, these interfaces are also used as a security layer between the application and the server [MuleSoft, 2020]. Most people, consciously or unconsciously, have already used services or applications that leverage APIs, for example, paying online with PayPal, signing in through a social media account or using web applications such as Google Docs. APIs enable a standardized communication between the application and the addressed service [Meng et al., 2018]. To enable such standardized communication, it has to be defined how this interface can be accessed and how the respective access has to look like. That is usually specified in the corresponding documentation. If the request to the respective service corresponds to the actual structure, it is straightforward to realize the access.

APIs are indispensable in today's software projects, since they provide a high level of abstraction for the respective task in the program [Meng et al., 2018]. Through their usage, one attains access to already existing services and this saves the effort of rebuilding this functionality. Nowadays, APIs are often seen as the actual product [MuleSoft, 2020]. That also applies to the PSD2-compliant interfaces that banks have to provide to TPPs.

In general, APIs facilitate smooth integration of already existing services and simplify the development process [Massé, 2012, Meng et al., 2018]. For APIs to be deployed, they must be well structured and clearly organized otherwise they will not be used by a developer.

2.2. Origin and Anticipated Potential of Open Banking

The financial services industry is one of the most important contributors to the successful development of a country. It has a significant impact on a country's competitiveness and contributes to the growth of its economy [Romanova et al., 2018]. Stagnating competition, however, hinders this progress and can endanger the successful development of a country. Long-standing concerns and fears about competitive shortcomings in the financial market were the reason why the CMA[1] launched a market investigation in 2014 to examine these issues [Smith et al., 2016]. They examined the supply of retail banking services for both private and corporate customers. Problems such as complicated fee structures for overdrafts, open-ended account relationships or the costly and challenging task of attracting new customers, lead to the conclusion that older and larger banks, which dominated the market, have it more comfortable to win and retain customers [Smith et al., 2016, Competition and Markets Authority, 2016]. This leads to a less competitive and innovative market as new and smaller banks have to work harder to build a customer base and gain ground with their advanced and pioneering business models.

Following the publication of the findings of this investigation, the Open Banking Initiative (OBI)[2], which aims to help customers by making the financial services simpler and more transparent, was founded [Daniel Gozman, Jonas Hedman, and Kasper Sylvest Olsen, 2018]. This initiative wanted more opportunities for clients to manage their banking experience. Furthermore, they also wanted easier access for small businesses to the financial market. Remedial measures such as APIs make it easier to obtain financial advice or to provide tailor-made products and services over the internet [Competition and Markets Authority, 2016]. The open API standard enables TPPs to access and process financial data of the bank's customers. For this access, the customer must give his consent to the service provider.

The prospects for open banking are immense. Its underlying idea was to promote in-

[1] British Competition and Market Authority: `https://www.gov.uk/cma`
[2] Open Banking Initiative: `https://www.openbanking.org.uk/`

novation and security [Chaib, 2017]. The opening of the banking market gave TPPs the ability to access client data, allowing them to change the way products and services are created [Chaib, 2017]. These modern and highly specialized services can replace overpriced products and fill the gap of undersupplied services from large retail banks [Bramberger, 2019]. To keep pace, larger banks have to adapt as open banking enables a more dynamic and competitive market. Overall, a fairer financial landscape is emerging as new financial technologies challenge business models of established banks. The competitive pressure on the established banks forces them to rethink their business models. Factors such as quality, customer experience and security are more important than ever and are decisive for product choice [Mbama and Ezepue, 2018].

The speed and responsiveness of the respective service provider and the focus on niche customer segments separates success from failure. Rapid adaptation is particularly difficult for banks, as formalized processes and internally regulated procedures are not suitable for quick adjustments. Higher customer expectations and increasing competition force banks to take every opportunity to create new and better products. Open banking is the future-oriented use of data made available via APIs [Bramberger, 2019]. Not only banks but also other service providers can use APIs and benefit from them. It is conceivable that they will use these as a marketing tool to promote their products and to offer better services. They can also start profitable cooperation's by offering additional functionalities that go beyond the minimum requirements of open banking. As margins are getting lower, it is essential to improve profitability. In general, open banking strives for a versatile and more effortless use of external services in the financial sector. APIs enable simple data integration and usage of existing services. That opens up new opportunities for financial institutions and TPPs to make banking more convenient, versatile, easier to use and thus more customer-friendly [Bramberger, 2019]. Furthermore, open banking deals with security issues and the avoidance of customer risks. Technological improvements and advanced security measures ensure a secure banking experience [European Banking Authority, 2017].

2.3. The Second Payment Service Directive

The growing number of online and mobile payments requires improved consumer rights and security [European Commission, 2018]. Therefore, the Council of the European Union introduced the revised Payment Services Directive in order to adapt to the highly dynamic and rapidly developing European payment market [Interview1, 2020]. This regulatory step aims to improve and remedy the shortcomings of the existing directive [European Commission, 2016]. The standardization of payment processes is a major step towards a uniform, digital payment market [Chaib, 2017]. This will make payment transactions easier, more efficient and more secure [European Central Bank, 2018].

Opening the payment market to new entrants increases competition and leads to better prices. The proposal of the new directive was adopted on 8th October 2015 and became legally binding on 12th January 2016 [European Parliament, 2015]. Each EU

member had to transpose the directive into national law by 13th January 2018 [European Parliament, 2015]. With the eventuating of the PSD2, a clear and comprehensive set of rules was established which applies to both new and existing payment service providers [Interview1, 2020]. The PSD2 forms the legal foundation for the creation of a unified, efficient payment services market within the EU [European Union, 2016]. In addition to innovation and competition, the PSD2 lays the foundation for further development of a more integrated and international market for electronic payments within the EU. Consumers and businesses benefit from greater efficiency, a level playing field, more choice, and better transparency of charges [European Central Bank, 2018]. In addition, regulatory technical standards were developed by the EBA [3] in cooperation with the European Central Bank (ECB) [4] to support the security of customer data and payment transactions [European Parliament, 2019]. These standards enforce the usage of rigorous authentication and secure communication protocols based on reliable open standards [European Central Bank, 2018].

As one of the fundamental legislative changes in the financial sector, it is expected that the PSD2 will have long-term, groundbreaking effects on the market [Interview1, 2020]. Although the regulatory requirements were clearly stated in the directive, it focuses mainly on PSD2 API compliance. Nothing is said about the technical implementation and test process [Romanova et al., 2018]. Banks are under pressure to deliver and implement the required functionalities.

2.3.1. Key Points

With a total of 117 articles the PSD2 is quite complex [European Parliament, 2015]. It is, therefore, not possible to cover all individual aspects of the directive in this thesis. In the following, an attempt is made to summarize the essential statements. It is intended to provide a general understanding of the complexity and diversity of the PSD2.

In a time when most payments are made online, reliable and secure customer authentication is more important than ever. The standardization of payment processes achieved by the PSD2 improves customer protection [Lynn et al., 2019]. According to the PSD2, meticulous customer authentication is achieved through a combination of at least two of three possible types of authentication. These three are described as follows: knowledge, possession and inherence [European Union, 2018]. Using the knowledge component for verification requires a password or an authentication code. The payment card, on the other hand, is a typical verification method that proves possession. Moreover, the use of a fingerprint is an example of inherence. The PSD2 refers to Article 97 when rigorous customer authentication is required. It states that whenever "the payer: (a) accesses his payment account online; (b) initiates an electronic payment transaction; (c) carries out an action via a remote channel that carries the risk of payment fraud or other misuse", strong customer authentication must be used [European Union, 2018].

[3] European Banking Authority: `https://eba.europa.eu/`
[4] European Central Bank: `https://www.ecb.europa.eu/`

That increases trust and acceptance of electronic payments since it reduces the risk of fraud and abuse.

Another key element of the regulation is that banks must provide an open interface for TPPs in order to enable them access to the customer's payment accounts. The implementation of such an interface is done by using open APIs. These interfaces must meet specific requirements that are defined in the technical requirements standards of the directive [Innopay DB, 2017]. Article 30 lays down the general obligations regarding access to these interfaces. It specifies, for example, that "payment initiation service providers [...] are able to identify themselves towards the account servicing payment service provider" or that "account information service providers are able to communicate securely to request and receive information" [European Union, 2018]. The APIs are used for payments and to request account information. As intermediaries, Payment Initiation Service Providers (PISPs) are responsible for initiating the transaction without taking possession of the customer's money. Account Information Service Providers (AISPs) provide the requested consolidated information about customer's payment accounts. Both are subject to authorization and registration requirements. If a customer allows a TPP to access his account, the AISPs must provide him with the necessary information [Deloitte, 2017b].

The rules Third Party Payment Service Providers (TPPSPs) must follow to prevent abuse of these services are the following. They may not store information about the customer, nor may personalized security information be saved [Deloitte, 2017b]. Payment Service Providers (PSPs) are obliged to refund any fees, costs or interest incurred by the payer if they are responsible for the error. In addition, they are responsible for carrying out stringent customer authentication and verifying the identity of the payer. In the event of an incorrectly executed payment, they must take full responsibility and provide a refund if the payer's behavior was not fraudulent. This enhances consumer rights in many aspects [Innopay DB, 2017]. In addition, payers can request a refund for direct debits within eight weeks. The directive has laid down clear and customer-friendly liability rules in broad terms.

2.3.2. Challenges and Opportunities

The Second Payment Service Directive (PSD2) is considered as the digital revolution in the financial sector. While regulation is usually not perceived as disruptive, the introduction of the PSD2 is likely to have considerable consequences. It is expected to have a disruptive effect and deregulate the financial market by increasing competition and encouraging innovation [Accenture, 2016].

This section discusses the challenges and opportunities of this directive. In order to be compliant with the regulation, banks have to provide a particular interface for TPPs. The provision of new IT infrastructure to comply with the regulatory requirements is associated with high costs [Deloitte, 2017b]. An additional challenge is the constant evaluation of security measures with regard to customer friendliness. The process-

ing of sensitive information and personal data requires robust security procedures and should not compromise convenience [Wessing, 2017]. Customers will not use a service that is unreliable and lacks security measures, nor will they accept complicated solutions. The challenge for banks is to find the balance between a secure and easy to use solution.

The evolution towards a more digitalized banking experience also threatens the direct relationship between the bank and its customers [Deloitte, 2017a]. Especially the younger generation prefers to do their banking transactions with their mobile phones instead of going personally to a bank office to talk with a bank employee [PWC, 2018]. Using a mobile application is simply more convenient than going to a bank office. Banks often compete with TPPs that specialize in specific segments. In many cases, TPPs are very inventive in terms of usability and responsiveness, both of which contribute significantly to the customer experience. That increases the competitive pressure on banks and challenges their position through innovation [Deloitte, 2017a].

The adoption of these highly complex regulations is very time-consuming and increases the workload for banks [Interview1, 2020]. This time and effort should be used to create new and better services that adapt to legislation and technology. Banks must adapt their strategies and change the way they do business [Interview1, 2020]. Without innovative products, they run the risk of losing market share or, even worse, of becoming a data pipeline [PWC, 2018, Deloitte, 2017a].

With challenges also come new opportunities. The introduction of open APIs not only offers advantages for TPPs, banks can benefit from those as well. A structured and well-designed API helps to simplify and optimize internal processes, as well as facilitates external accessibility and endorsement [Chaib, 2017, Interview4, 2020]. Adaptation to the latest technological standards enables efficient usage. When the API is in production, an extension can easily be implemented. This enables straightforward integration of additional functionalities.

Banks can capitalize on collaborations by offering additional services that go beyond the basic requirements of PSD2 [Interview3, 2020]. Fintech's use such offerings to provide services that appeal to a broader customer audience or to respond to the specific needs of each individual [Mbama and Ezepue, 2018].

In general, an open regulatory environment such as PSD2 offers customers greater freedom of choice and improves their banking experience. Higher standardization and competition will lead to improved business efficiency and reduced service costs. It also enables the development of services related to customer's transaction data and payments, as well as improved approaches for risk assessment.

2.4. Introduction to Software Testing

According to the Economic Times, software testing is "the process or method of finding error/s in a software application or program so that the application functions according to the end user's requirement" [Economic Times, 2000]. Today's industry is highly dynamic and fast-moving. New and innovative solutions are the order of the day. Developers are racing against time to deliver new products that best meet the needs and requirements of customers [Mohanty et al., 2017]. Jeff Offutt and Paul Ammann point out that "paradoxically, free software increases [...] expectations" [Jeff Offutt, 2018]. Free software entails that the quality of paid software and, in particular, its correct execution and functionality must be flawless. Therefore, the primary task of software testing is to measure and improve software quality [Jeff Offutt, 2018].

One of the most important phases in the life cycle of software development is software testing. It is an extremely beneficial and helpful activity to ensure the correct functionality of the software [Qa, 2019, Interview4, 2020]. Although it is considered the most time consuming and expensive part of software development, it is crucial to meet customer expectations and requirements of the customer [Jeff Offutt, 2018]. The early integration of a test process with a good test design prevents errors and drastically reduces the development effort and associated costs [Nguyen, 2008, Interview4, 2020]. Especially when the product is in a later stage of development or already in production, software errors tend to be very expensive and time-consuming to fix [Interview4, 2020]. That leads to a loss of resources and endangers the reputation of the client [Qa, 2019].

Agile development processes force the industry to rethink its test strategies and how to improve them [Jeff Offutt, 2018]. One way to increase the efficiency of the test process is test automation [Interview3, 2020, Interview4, 2020]. The automatic execution of the test process reduces the test effort and time consumption of testing [Mohanty et al., 2017, Nguyen, 2008].

People tend to make mistakes, and errors in software development are no exception. Testing the code is therefore absolutely necessary to solve this problem. It is especially important in the financial industry, an industry mainly based on trust. The secure and correct functionality of PSD2-compliant APIs is essential for their successful integration and adaptation. Software errors such as misunderstood requirements, incorrect software design or faulty code must be avoided and can be reduced with the right software testing approach [Marvin Zelkowitz, 2009]. The test activity helps to validate and verify compliance and assess the reliability of the API [Jeff Offutt, 2018].

A secure and stable interface fosters functionality, increases the user-friendliness and improves customer service [Qa, 2019]. Well-tested software prevents the malfunction of specific operations and helps to avoid syntax and calculation errors [Marvin Zelkowitz, 2009]. One downfall, however, remains. Testing helps to show the presence of errors, but it cannot ensure that there are no remaining errors [Marvin Zelkowitz, 2009, Interview4, 2020].

The following sections elaborate, the main objectives of software testing. These include the basic concepts of software testing, such as different test approaches and guidelines as well as a short introduction to the process of test automation. The respective goals and benefits are examined and the relevance and necessity for automating the test process are shown.

2.4.1. Goals of Software Testing

Software must be tested to ensure that it conforms to the business requirements specification and the system requirements specification [Software Testing, 2016, Try Qa, 2018]. Compliance with such specifications is essential to ensure that the product performs as expected and meets the intended requirements and functionalities [Interview4, 2020].

The validation of the required functionality can be done by comparing the expected and actual results of a test case. An error is detected if the outcome of a test does not match the expected behavior described in the specification. Problems such as faulty calculations, incorrect data processing, insufficient software performance or security leaks must be avoided [Interview4, 2020, Interview2, 2020]. Identifying and correcting such errors prevents their re-occurrence in the future and leads to more efficiency and better functionality of the respective program [Balaban, 2011].

The review of the system and its requirements shows what needs to be done to improve the system and thus increase the usability and service for the customer. Detecting and correcting errors helps in this process [Lingard, 2011, Qa, 2019]. It is important to have good test coverage and to fix errors as soon as possible. Good test cases are neither too simple nor too complex. They help to detect undiscovered errors. Both the identification of an error, which is then corrected, and a successful iteration of the test process provide information about the reliability and quality of the product [Try Qa, 2018].

The test process increases confidence in the release of the product and ensures that the conformity requirements are met. By creating and executing specified test cases, a verification of the system under test is possible. Testing for errors helps to detect incorrect and non-compliant behavior of the program. The main objective of software testing is to find errors in specification, design and implementation [Interview4, 2020, Interview2, 2020].

The testing process of software helps to meet the functional requirements of a system. Financial losses and costs can be minimized by the early detection of errors in the software [Jeff Offutt, 2018, Mohanty et al., 2017]. The goal is to test the software application thoroughly and thereby ensure that it works well. Thereby one tries to achieve the best possible test coverage [Try Qa, 2018]. However, this is not always so easy, as there are often many different scenarios. For example, PSD2-compliant APIs have many possibilities to configure each individual API call [Try Qa, 2018].

The goals of testing is to improve the quality of the product through continuous measurement and verification of design and code [Qa, 2019]. Software testing should follow Dr. W. Edwards Deming's learning cycle of quality and adhere to his statement: "Inspection with the aim of finding the bad ones and throwing them out is too late, ineffective and costly". Therefore the ultimate goal should be to improve processes through inspection. Testing is seen as a means to evaluate and verify the correct functioning of certain processes [Software Testing, 2016].

2.4.2. Basics of Software Testing

Software testing is considered one of the most critical processes in the development cycle. Continually changing and more agile methods of software development require new integrative solutions that improve the traditional way of software testing [Qa, 2019, Mohanty et al., 2017]. Guidelines and various test approaches are presented below.

Software testing starts with the identification of requirements and continues during the development phase [Interview4, 2020, Interview2, 2020]. However, the test procedures and methods vary in each development cycle and are strongly dependent on the software life cycle model used. Since it can never be guaranteed that the entire system is correct, certain assumptions must be made to determine the end of the testing process [Interview4, 2020].

Nobody can actually guarantee error-free software. The end of testing depends on time, budget and duration of the respective test case [Software Testing Help, 2020, Software Testing, 2016]. It is essential to find the right balance between all three constraints [Interview4, 2020]. That varies from project to project and should be determined at the beginning of each undertaking. The test coverage should be high enough to test all specified requirements, and testing should only be stopped when high priority defects have been identified and fixed [IEEE Standard, 1998, Software Testing, 2016, Software Testing Help, 2020].

The Institute of Electrical and Electronics Engineers (IEEE) published several standards such as IEEE Std 829-1998 or IEEE Std 829 2008-1998, which provide a reference framework for software testing. Among other things, these documents provide guidance for creating test design specifications, test case specifications, and test procedure specifications. A basic software test lifecycle phase may have the following characteristics:

1. Requirements study
2. Test Case Design and Development
3. Test Execution
4. Test Closure

5. Test Process Analysis

The requirements study deals with the needs of the customer. Its thorough understanding is essential for the testing process and for determining the customer's expected results [Ostrand and Balcer, 1988]. Furthermore, test objectives, required resources and the need for specific tools can be derived. The analysis of the functional and non-functional requirements facilitates the planning and creation of the test case design and promotes the definition of a tailor-made test process for each project [Interview2, 2020, Ostrand and Balcer, 1988].

According to the IEEE Std 829-1998, test case design and development "refines the test approach and identifies the features to be covered by the design and its associated tests" [IEEE Standard, 1998]. Its purpose is to validate the acceptance of the product and to facilitate the understanding of the test requirements by people outside the test group [Software Testing, 2016]. A traceability matrix can be useful to ensure that all test requirements are considered in the preparatory phase of test planning [Ostrand and Balcer, 1988]. The IEEE requires the specification of the test approach and the accompanying features that are to be tested. Additionally, each test case should specify the input required to perform the test and the expected output [Software Testing, 2016]. That facilitates the execution and validation of the test cases and enables an uncomplicated development of the test cases [Software Testing, 2016, IEEE Standard, 1998]. Test execution offers the possibility to evaluate the performance and functionality of the software. During the execution phase, errors can be easily detected. That is always the case if the test results do not match the expected output.

Test closure and test process analysis are closely related [IEEE Standard, 1998]. They include all test reports generated during the test process. These reports are analyzed to improve and correct the program's misbehavior.

There are different types of tests and various test methods. For the purpose of this scientific work, only the most common approaches are examined. In most cases, the testing approach can be categorized into three main techniques: functional testing, performance testing and security testing [Ostrand and Balcer, 1988]. Within these techniques, the choice is between static or dynamic testing. Static testing, also known as manual testing, checks the requirements specification or the source code of the software. It is usually applied in the initial phase of software development [Nahid Anwar & Susmita Kar, 2019]. Dynamic testing or automated testing "involves testing [...] software for the input values and output values" [Nahid Anwar & Susmita Kar, 2019]. Here the dynamic behavior of the code is analyzed. Both static and dynamic testing can improve the verification and validation of the software.

Functional testing analyzes whether the program meets the specified functionality. It revolves around the workflow and can be performed at several test levels. Conventional approaches include unit testing, integration testing and system testing [Ostrand and Balcer, 1988]. Black box testing and white box testing are also well-known test methods. The former is a technique where the knowledge about the internals is not

visible to the test person. Thus the tester has no knowledge of the code or the internal structure of the program [Nidhra, 2012]. It focuses on determining the requirements of the system and whether the program is doing the intended. Black box testing is particularly well suited when no knowledge of internal design nor code access is required and when inconsistencies in the requirement specifications need to be found [Interview2, 2020, Interview4, 2020, Nahid Anwar & Susmita Kar, 2019]. Due to limited knowledge, testers may find it quite challenging to design test cases, respectively, might not test some functionality at all [Nahid Anwar & Susmita Kar, 2019].

The opposite is white box testing. The tester has a detailed knowledge of internal procedures as well as the code structure [Ehmer and Khan, 2012, Nidhra, 2012]. That facilitates the creation of test cases, making them also less expensive since error-prone segments can be addressed more explicitly, leveraging the knowledge of logic and code [Software Testing, 2016, Nahid Anwar & Susmita Kar, 2019]. This approach allows maximum test coverage and facilitates the detection of errors during the test process [Nahid Anwar & Susmita Kar, 2019].

The test process of PSD2-compliant bank APIs can be mainly considered black box testing [Interview2, 2020]. However, there is a vast amount of documentation that provides information about the internal functionality and the intended behavior of the respective API calls. This provides insight into the functionality of individual methods. The multitude of available manuals and technical specifications simplifies the test process considerably. Error messages can be used to create specific test cases and to check whether certain functionalities work correctly. Nonetheless, experienced testers who are particularly familiar with the functionality of the interface are still required. The testers must understand the various workflows that the API interface offers [Interview2, 2020, Fundamentals, 2010].

Performance tests are often associated with stress testing and load testing of the system [Ostrand and Balcer, 1988]. In this test process, the tester is particularly interested in parameters such as load time, response time, access time or runtime [Ostrand and Balcer, 1988]. These values are crucial and should not be neglected as they are responsible for ensuring good quality, overall reliability and user acceptance of the software. For meaningful and reliable performance testing, testers depend on tools that automate these processes, as it may be necessary to send many different connection requests to the appropriate API interface at once. The test tool that was developed in the context of this thesis for illustrating and improving the test process of PSD2-compliant APIs deals exclusively with functional testing.

Another essential test category that often remains in the background is security testing. At a time when most failures are due to security problems, it is irresponsible to omit this test category. Although it is still impossible to find all defects due to time constraints, every tester should try in the best possible manner to identify problems at the design level that lead to security issues [Nahid Anwar & Susmita Kar, 2019]. Services such as authentication and authorization must be tested.

For the practical part of this scientific work, it is necessary to understand the concept of regression testing and API testing. Regression testing is used to perform tests of PSD2-compliant interfaces efficiently during their continuous development.

2.4.3. Test Automation

Software testing makes up a large part of the entire development cycle and is quite expensive. It is therefore not surprising that the idea of avoiding costly human error through software testing or increasing the speed of development through faster test processes and error correction methods is leading to a rethinking of how test processes can be made more efficient [Kumar and Mishra, 2016]. In this context, automated testing is often proposed as the most promising solution. It pledges for test quality, shortened execution time and lower costs. Many facts must be taken into account when deciding whether or not to implement test automation. The following is a short introduction to the advantages and disadvantages of test automation.

"[In order] to work more efficiently and effectively, test engineers must be aware of various automated testing strategies and tools that assist test activities", says Frank Elberzhager from the Frauenhofer Institute for Experimental Software Engineering [Garousi and Elberzhager, 2017]. Many test engineers consider test automation only as a tool for executing test cases. However, this is not the only domain that can be automated. The process of test automation goes far beyond this point. Activities such as test case design, test evaluation or test management can be automated [Garousi and Elberzhager, 2017]. Modern scientific studies show that sometimes it is possible to generate test cases automatically, but this will not be discussed further in this thesis [Ed-douibi et al., 2018]. Automatic testing brings many advantages to the tester. For example, it can increase test speed and reduce execution time [Garousi and Elberzhager, 2017, Interview4, 2020]. Since no human interaction is required for the individual execution of the test cases, the respective tests can be performed quickly. As a result, the test process can be carried out more frequently [Interview4, 2020]. That brings advantages to modern software development since quick feedback enables the creation of better products through faster error detection [Wiklund et al., 2017]. As a "core component in agile development", it is particularly useful in scenarios where many small code changes occur regularly [Wiklund et al., 2017, Ropota, 2011]. Suppose a data-driven test tool is developed for several bank interfaces. This tool allows for the evaluation of many test cases. An automatic execution would be beneficial since the manual execution of hundreds of test cases could easily cause some to be missed. Automated testing can avoid errors, since repeated manual testing of test cases becomes tedious and thus increases the risk of errors [Ropota, 2011, Interview4, 2020].

Improving efficiency and optimizing the test process is the ultimate goal of test automation [Mahajan et al., 2016]. Since the process of test automation is quite complex and time-consuming, certain conditions should be met to make it worthwhile [Kumar and Mishra, 2016]. The first step is to check whether the types of tests to be performed can

actually be automated. That is not always possible and is considered as a limitation of the automated test processes. For example, a particular interaction may require manual verification, such as the use of a TAN, a one-time password. Automating such a process is often quite difficult or even impossible [Interview2, 2020, Interview3, 2020]. In addition, test cases that are executed very rarely or where manual verification is much easier than an automated solution should not be automated [Ropota, 2011]. In large projects, tests that are repeated regularly or that are prone to human error should be automated [Mahajan et al., 2016]. That improves the test's quality and brings several advantages for the product [Mahajan et al., 2016]. A high degree of automation can reduce test costs and the associated test effort if implemented correctly [Interview4, 2020]. However, even this process, if not implemented carefully, can hinder success and cost a lot of money [Mahajan et al., 2016].

A more cost-effective test procedure, higher efficiency and improved software quality are just a few examples of the advantages of automating software testing. However, there are also several disadvantages. Many people have unrealistic expectations of software test automation [Ropota, 2011, Interview4, 2020]. They believe that it will uncover many new errors and that, once in use, it will not require any further work. Especially the maintenance of test cases in an automated test suite is essential to ensure consistent quality [Ropota, 2011, Interview4, 2020]. These investment costs are also significantly higher than those for manual testing [Kumar and Mishra, 2016]. If a test is performed only once, it is often cheaper and more convenient to test the case manually, since the configuration of the test tool can be time-consuming [Ropota, 2011].

In the case of testing PSD2-compliant APIs, automation of the software testing process brings many advantages. The test time and the costs for the test activity are reduced. The EBA recommends the use of automated procedures for the test process of PSD2-compliant APIs [European Banking Authority, 2019]. In particular, it enables the use of automated test processes since each API call can be executed independently and a simple comparison of the actual results with the required ones is possible. Since the PSD2-compliant APIs are constantly being expanded, the overall effectiveness of test repetitions is high. Here, test automation offers the possibility to serve as good control and assurance of quality [Wiklund et al., 2017].

2.4.4. Regression Testing

For the software testing process to be successful, organizations must use efficient and effective testing strategies [Interview3, 2020]. Regression testing is one such testing techniques. It is the process of repeatedly testing an already validated piece of code/-software after a modification in some other software segment has been made [Huizinga and Kolawa, 2007, Ammann and Offutt, 2008].

There is a distinction between repeated execution and regression testing. The former is the process of repeatedly testing software. A modification is not compelling. Retesting

or repeated testing is, therefore, considered as planned testing for failed test cases. We speak of regression testing when the application has been modified and there is a need to verify that these changes do not affect other parts of the software [Huizinga and Kolawa, 2007, Ammann and Offutt, 2008].

Regression testing is an essential part of a successful and meaningful test procedure during software development [Interview3, 2020, Interview4, 2020]. Since small changes in one part of the software can also lead to a change in the behavior in an unchanged part of the software, it is particularly important to test the entire software after every modification [Huizinga and Kolawa, 2007]. Many different programs are available to automate this process.

In order to make this test process efficient, the tests must be executed automatically [Bitbar, 2019]. In this case, it is particularly important to ensure that the test set is well constructed. The selection of all test cases created for the software can lead to a non-strategically defined regression test set. Such test sets are often very large so that their execution takes longer than necessary and leads to fewer test runs [Huizinga and Kolawa, 2007, Ammann and Offutt, 2008]. In general, it is almost always worthwhile to build a regression test suite if test cases need to be executed frequently. Initially, setting up such test suites usually takes more time than manual tests [Interview3, 2020]. However, during the development process, these tests are usually more profitable, since they can be executed automatically and manual interactions are avoided [Ammann and Offutt, 2008]. That means that as soon as a modification or a new component of the program is developed, a corresponding test case is added to the test suite. Like all other test cases in the test suite, it can then be executed quickly and efficiently. It is also important to continuously adapt existing tests to the new requirements or changed functionality [Huizinga and Kolawa, 2007].

Of course, all test cases can be executed. However, as already mentioned, this is often very time and resource-intensive. Therefore, it is often preferable to execute only a certain part of the test-suite [Interview4, 2020]. This subset must be chosen carefully so that both the area to be tested and the affected methods are thoroughly examined [Ammann and Offutt, 2008, Interview4, 2020]. Another possibility is the prioritization of test cases [Elbaum et al., 2002]. A well-thought-out strategy reduces the complexity of the test suite considerably [Huizinga and Kolawa, 2007]. Minimizing a test suite while maintaining the same test coverage is not easy, but if implemented well, it can save a lot of time and money.

The initial test suite is created when the team starts with the black box testing. New test cases are created for each new module. Consequently, the size of the test suite increases as the program grows. The regression test set should be executed at least once before each new build [Huizinga and Kolawa, 2007]. If an error is found during the execution of the test cases, the developer can fix it immediately.

2.4.5. Data-Driven Testing

In many cases, it is necessary to run the same test case with different input data, since results can vary depending on the input [Stepien et al., 2018, Vijay, 2016]. High test coverage for a test case can be achieved by covering all possible input data sets. That should be done despite the same request. Our PSD2-compliant API is a perfect example. In order to make a single API call, one first has to create a so-called consent. The creation of the consent is done by executing a specific API call and specifying its data object. This call is always the same and nothing changes except the data objects [Interview2, 2020]. Creating a separate test case for each response would be time-consuming and inefficient [Stepien et al., 2018, Vijay, 2016]. Therefore it makes sense to separate the data from the actual test case [Interview2, 2020, Interview4, 2020]. Precisely this abstraction describes the goal of data-driven testing. A data-driven test procedure tries to separate the test data from the actual test cases in the best possible manner [Stepien et al., 2018].

At the start of the test run, the actual data sets are loaded from a file e.g., a Extensible Markup Language (XML) or JavaScript Object Notation (JSON). That avoids hard-coded values in the test cases and generally makes the test procedure more flexible [Stepien et al., 2018, Interview2, 2020]. It also gives the tester the ability to make his test cases clear and understandable. In addition, test duplicates can be avoided, so that the tester now saves time in the otherwise time-consuming and costly creation of test cases [Stepien et al., 2018].

The use of data objects with well-defined data models such as XML or JSON allows a simple and relatively flexible separation between data and the test cases. A particular advantage of data-driven testing is the possibility of having the test cases created by experts in the relevant field [Interview4, 2020]. They do not need in-depth knowledge of programming or the use of the test program [Stepien et al., 2018, Wu et al., 2009]. They are better suited to create the test cases. Their test cases are usually better and more extensive, as they have more knowledge of the functional aspects of the interface.

Such an approach requires a suitable format to represent the test cases in a simple way. This format is used to extract the necessary information for each test case [Stepien et al., 2018, Wu et al., 2009]. The experts can use this structure to fill the individual test cases with information, without knowing the technical implementation of the test case. As a result, the programmers only have to concentrate on the technical implementation of the test cases.

Another advantage is that the tool can process positive and negative tests more straightforward. When executing positive tests, the test result must be correct. A negative test indicates that the tester expects the test case to fail. However, both test cases are still marked as successfully, since they confirmed the desired event.

In general, individual test cases can be managed in a much easier fashion by abstracting the data from the test cases [Interview3, 2020, Interview4, 2020]. Consequently,

the test depends more on variables than on fixed values. Data-driven testing helps to keep the test cases manageable and thus simplifies testing [Stepien et al., 2018].

2.4.6. API Testing

An Application Programming Interface (API) is used to explicitly expose a certain part of a software system to a third party [Massé, 2012]. This third party can then use the predefined structure of the API to communicate with the system. In general, a distinction is made between two different types of web services: Representational State Transfer (REST) and Simple Object Access Protocol (SOAP) [Inflectra, 2020]. In the following, only REST is discussed. APIs allow TPPs to communicate with the server via Hypertext Transfer Protocol (HTTP) and request customer data [Massé, 2012]. In this case, the API acts as a translator between the application and the server. For a smooth communication between both parties, every API must be tested. As explained above, there are several ways to test. The interface must meet certain performance and security requirements. However, the prototype of this thesis will only focus on testing the functionality of the API. That means ensuring that the requested data is returned correctly.

Testing is crucial, especially when it concerns the API development life cycle and verifying the application logic [Soap UI, 2020, Broadcom, 2020]. It is crucial to certify that the returned value matches the expected value and that the predefined format is followed. Since the tester of the API interface usually has no knowledge of the underlying code, it is often difficult to identify all individual test cases [Soap UI, 2020, Broadcom, 2020]. In such a situation, the documentation and specifications of the interface can help to create test cases. The main difficulty in testing APIs is that one needs to understand how the interface works [Interview2, 2020]. Often several API calls are required to get the desired result [Soap UI, 2020]. In the example of our PSD2-compliant API, five different requests are required to retrieve the account lists. In addition to the sequential order of the individual API calls, the combination of various input parameters, such as header, play an important role. There are many different ways to set and select these parameters. The tester needs a reasonable amount of knowledge to decide which combinations are necessary and which do not fit together at all [Interview2, 2020].

Instead of testing software with its Graphical User Interface (GUI), it is preferable to do this by using the API. This avoids the need for GUI testing [Inflectra, 2020]. Maintaining these is generally quite difficult and error-prone due to the frequent changes and resulting adjustments to tests [Inflectra, 2020]. Especially in a time when more and more services are pushed towards cloud solutions, all API functionalities must work properly. The failure of one service can lead to a complete breakdown of the entire program.

2.5. Summary of the Theoretical Framework

The objective of the Theoretical Framework is to provide a general understanding of the requirements for the test tool and the concepts used in its development. This summary provides a general understanding of the problems of the current test process of PSD2-compliant APIs. It also highlights the need for a new test tool specifically designed for the evaluation of banking interfaces. The essence of the current situation is outlined below to make the purpose of this thesis comprehensible.

EU banks had to provide interfaces for TPPs by September 2019. These interfaces allow TPPs to access the bank's customer account information and to carry out transactions. From a technical perspective, there is no clear definition of how these interfaces should actually be implemented [European Parliament, 2015, Interview1, 2020]. This means as long as these interfaces meet the requirements specified by the PSD2 and the Regulatory Technical Standards (RTS), they can be implemented in any desired way.

To achieve the goal of PSD2, communication between TPPs and banks must be standardized [Interview1, 2020]. For this purpose, various organizations were founded that work together on the definition of a standardized interface. One of them is the Berlin Group, a consortium of multiple organizations that are developing a uniform interface standard [Berlin Group, 2020].

If the assumption is made that there are just a few organizations that provide a standard which is then used by many different banks, the misleading conclusion might be that one has to develop test cases only for the standards themselves [Interview1, 2020]. That would make the test process much easier, of course. However, this is unfortunately far away from reality [Interview1, 2020].

Since the legal foundation of PSD2 does not specify the design and implementation of the interface, banks do not have to comply with the standards defined by e.g. the Berlin Group. As a result, many banks choose a standard on which they base their interface, but then make changes to it. This allows them to adapt the interfaces to their own requirements [Interview1, 2020, Interview2, 2020]. While this is logical for the individual banks, it does not contribute to the goal of a uniform and consistent communication between the TPPs and the banks.

Having to live with this status quo, the goal of this work is to develop a concept and a test tool prototype with the ability to evaluate APIs independently of their interface implementation. For efficiency and utility reasons, the used test set must also be equally usable for all banks [Interview1, 2020]. Therefore, the assumption is made that the banks have the possibility to use the same data set, meaning that they test with the same accounts and users. That should not pose a significant problem, as banks can create the necessary test accounts and users in their test environments.

Such a test tool also has to meet different requirements. The concepts introduced in the Theoretical Framework, provide the rational behind the functionalities implemented in the test tool.

The evaluation of the various bank interfaces is a very repetitive task. Ideally, all API endpoints should be tested at least once in each development cycle [Interview2, 2020, Interview3, 2020]. This ensures the correctness of the respective interface. Since manual testing is by far to time-consuming, this process should be automated [European Parliament, 2015]. The concept of test automation was also used in the test tool, making the test process more efficient.

In order to ensure that unchanged, already existing functionalities were not affected by a modification in the software, the test cases covering these functionalities should be re-evaluated [Interview3, 2020, Interview4, 2020]. This concept is called regression testing. It involves re-executing existing test cases that have already been successfully carried out.

Another important point is the possibility to evaluate a large number of test cases. For this purpose, simple integration of test cases is required. It is advantageous if the test cases are separated from the actual test data. This enables the creation of generic test cases which are then filled with the test data [Interview2, 2020]. This is the way to run a data-driven test procedure.

All these concepts give an outlook which requirements must be met by a test tool for PSD2-compliant bank interfaces. Furthermore, the knowledge about these concepts facilitates the understanding of the functionalities presented in this thesis.

3. Related Work

Already several ready-to-use solutions for API testing are available today. Due to the large number of different bank interface standards and the complexity of testing them, it is beneficial to develop a new test tool that is designed specifically for testing PSD2-compliant APIs. Such a test tool can also address potential difficulties in the current test process, such as the complex adaptation of individual test cases to the specific bank interface.

Before introducing the architecture and actual functionality of the prototype, several existing API test tools are presented. These all have the functionality to test bank-specific API implementations. Since there are many different ways to perform similar operations within the XS2A interface, certain assumptions were made for the sake of simplicity.

The following chapter introduces and compares some of the currently available API test approaches and programs that are used to test bank interfaces. This comparison is based on information from various scientific articles as well as field reports from expert interviews. The purpose of this overview is to understand the difference between GUI-based and code-based test tools. In addition, an insight is given into various test approaches and their benefits.

3.1. Test Approaches for PSD2-compliant Interfaces

In recent years, the significance of APIs has grown. That is reflected in the growing interest that companies are showing in suitable API processes, tools and solutions [Benzell and van Alstyne, 2016]. Companies have recognized that they can offer their end-users and partners better software solutions through APIs [Jacobson, 2011]. These interfaces can be used to create new, innovative solutions and, most importantly, they can easily be monetized.

In order to ensure that APIs meet expectations in terms of functionality and performance, they must be tested [Bangare et al., 2012, Interview4, 2020]. This is an essential part of the development process. Testing is performed to verify that both the functional requirements and security aspects are fulfilled. Unfortunately, there is only limited scientific work and few publications on how banks actually deal with this topic. In the following, two existing approaches are presented.

In the paper "API Testing for Payment Service Directive2 and Open Banking", various

workflows that can be performed using the API have been investigated [Coste and Miclea, 2019]. For each of these workflows, test cases were developed to evaluate their implementation. These test cases were executed with Swagger and stored in the test management tool HP ALM [Coste and Miclea, 2019]. The results were displayed and validated with a GUI. This approach's target was to give an overview of the different processes to be tested. In addition, these test cases aimed for a better customer benefit and a cost-efficient adaptation through a correct PSD2 implementation [Coste and Miclea, 2019]. However, this approach only considered the execution of individual API calls. Standardization of the test process in order to create a more straightforward and more efficient test procedure was not examined at all.

The NextGenPSD2 Implementation Support Program (NISP) presents another approach for testing PSD2-compliant interfaces. Its goal is to create a stable and sustainable implementation of the respective interfaces through synergies in implementation and test procedures [Dijkstra, 2009]. For this purpose, a test concept and a detailed test case catalog was developed [Dijkstra, 2009, Interview2, 2020]. The test concept is an interface description that explains what business processes are possible with the interface [Interview2, 2020]. For each process, the possible flow of information was analyzed. Based on this information, the individual test cases could be identified [Interview2, 2020]. All of this was documented in an Excel spreadsheet. If a member of the organization wants to test their interface, they must enter the parameters corresponding to their interface in the Excel document [Interview2, 2020]. The worksheet then generates a human-readable test specification based on these parameters [Interview2, 2020]. To execute the test cases programmatically, this test specification must be enriched with information and adapted to the interface to be tested. Both components may only be used by the members of this organization. The use of the test case concept and the test case catalog helps members to reduce their test effort and to meet the legal requirements [Interview2, 2020]. This approach is actively used by financial institutions to evaluate their interfaces.

3.2. Test Tools for Evaluating PSD2-compliant Interfaces

There are many different tools and frameworks available for evaluating PSD2-compliant interfaces. It is possible to choose from several options depending on the needs of the tester [Interview4, 2020]. If the tester needs extensibility to integrate existing software solutions, it is advantageous to choose a software testing framework instead of an independent test tool. The later is often limited in its extensibility. However, such a decision can also depend on the prerequisite of whether or not the test person can program.

In practice, there are many different tools for testing interfaces. One of the simplest and best known is Postman. This is a REST client that performs API tests using a simple and well-structured GUI [Interview2, 2020]. No programming skills are required. All customization can be made using the GUI. That makes Postman very attractive for

users/testers who are more familiar with the functional side of the interface. The necessary data can be entered via the GUI quite easily. Of course, it is also possible to perform data-driven tests, making the test process of a large number of test cases straightforward [Postman, 2020]. Postman is well suited for a quick run through API tests without much effort. That was confirmed during the conducted interviews. The majority of respondents were familiar with the program and used it in their testing process. However, it was also pointed out that Postman has limitations when it comes to adding custom functionality [Interview2, 2020, Interview4, 2020, Interview3, 2020]. Unfortunately, this is a big problem, especially when an intelligent layer is needed to adapt the test cases to the particular bank interfaces. Postman is not the only program capable of performing such operations. Katalon Studio or SoapUI are alternatives to Postman with similar functionalities.

As mentioned above, testing APIs with a GUI-based test tool works perfectly fine in most cases. However, the integration of own products or third-party solutions with a GUI-based test tool is quite challenging due to the limited possibilities to integrate new [Interview2, 2020, Interview3, 2020, Interview4, 2020]. Using software test frameworks is an alternative in this situation. A test framework can easily be integrated into existing software and allows the inclusion of e.g., special reporting software or functionality to revise data input. Of course, using a framework is a bit more complicated than a GUI-based test tool, but it is more profitable in the long run because adjustments and customization of the functionalities can quickly be made later. For the prototype built in this scientific work, the Java-based test framework Rest Assured is used [REST Assured, 2020a]. This framework provides a domain-specific language to simplify the testing process of a REST service [REST Assured, 2020b]. It includes HTTP methods such as POST or GET, making it easier to send requests or validate responses [REST Assured, 2020b].

Part II.

Design and Implementation

4. Test Tool for PSD2-compliant APIs

This chapter focuses on the design and implementation of the actual test tool. In previous sections, all the essential information and concepts needed to understand how this prototype works have been explained. In the following, the test process of the program and the test tool architecture are described in more detail. Furthermore, the concept that enables the tool to address different bank interfaces and adapt the test data to the respective interface requirements without having to write production-ready code is presented. In order to understand how the mapping process works, a simple example of a possible workflow from the user perspective is given. The correct configuration of the test tool is crucial because if it is misconfigured, the tool will produce false results.

This test tool shows a straightforward and understandable way of addressing different bank interfaces. To keep the project manageable, the scope was limited. These restrictions also facilitate to keep the concentration on special features such as the input and output mapping. For the sake of simplicity, the scope of the PSD2-compliant interface standards covered by the test tool is limited.

The prototype focuses particularly on the standard of the Berlin Group. This decision was made due to the present market position of this organization and its broad acceptance in the market [Berlin Group, 2020]. The majority of German banks rely on the Berlin Group standard. Thus, a large number of different financial institutions should be able to be tested at once. However, this is not the only restriction that was made. Since the interface description of the Berlin Group is very complex and contains many methods, this test tool only examines a subset of the Account Information Service (AIS). More precisely, eight different API calls from the AIS of the Berlin Group standard can be evaluated with this test tool. Facilities such as the Payment Information Service (PIS) are omitted for complexity reasons. To summarize, this test tool only considers the AIS of the Berlin Group standard.

Since banks can choose between several authentication methods for their interfaces, further restrictions were necessary. Banks can choose between the redirect, the decoupled and the embedded approach as well as mixtures of the aforementioned authentication methods [European Parliament, 2019, European Banking Authroity, 2018]. With the redirect approach, customers are usually redirected to the bank's website to perform a part of their authorization there [WSO2, 2019]. The decoupled method, on the other hand, outsources the second factor of the authorization process [Kuang, 2018]. In this case, authorization must be carried out with an additional device, such as a smartphone. One method that is particularly widespread in Germany is the embedded

approach. Here, the user passes his authorization data forward to the TPP. This TPP then handles the authorization process for the user [Kuang, 2018]. The test tool uses exactly this type of authorization. To make this test process efficient and to avoid complications when authorizing the user, the required transfer of the TAN has been defined statically in the bank environment.

Before the test tool process and the architecture of the actual tool are presented, the list of constraints is summarized for clarity. The test tool deals with the test process of PSD2-compliant interfaces, in particular the standard of the Berlin Group. To keep the test process manageable, only a part of the AIS is tested. The actual authorization process is accomplished by an embedded approach with a statically set TAN.

4.1. Summary of the Test Tool Requirements

Prior to examining the test tool, the requirements for a PSD2-compliant test tool are summarized. The aim is to match the concepts presented in the theoretical framework with the corresponding requirements. There are two types of requirements, namely functional and non-functional. The functional requirements define what the test tool has to do and the non-functional requirements describe how the application has to provide these services [Scheler, 2018, Bobrica and Niemela, 2002]. The comprehensive literature review of the concepts presented in the theoretical framework, as well as the interviews, helped to elaborate the test tool requirements. In the following, both the functional as well as the non-functional requirements are presented. How these requirements are implemented in the test tool will be explained later.

The functional requirements mainly refer to the IPO model. [Bobrica and Niemela, 2002, Hawkey, 2013]. This model describes the possible data input, data processing and the resulting data output [Bobrica and Niemela, 2002, Hawkey, 2013].

It must be easy to integrate different test cases into the test tool. It should be possible to add new test cases at any time without having to change the program [Interview2, 2020, Interview4, 2020]. This allows non-static testing of the respective test cases. Our first requirement for the test tool is, therefore, the simple and straightforward integration and modification of test cases.

Another important requirement is the automated execution of test cases. It should be possible to run the test cases without human interaction. This leads to a more efficient execution of the test cases since, for example, it is possible to run the test set in the background while working on other essential tasks [Interview2, 2020, Interview4, 2020]. This automated process also reduces the number of errors which could occur when the test cases are executed manually. Our second requirement is, therefore, an automated test procedure.

In addition to the automated execution of test cases, another important feature is the reevaluation of already executed test cases. This allows previously evaluated, unmodi-

fied parts of the interface to be retested. This ensures that the corresponding areas of the software are not affected by changes and that everything still works [Interview3, 2020]. The test tool should, therefore, be able to perform regression tests, e.g. the re-verification of successfully tested software modules.

As explained previously, it is useful to separate test cases from test data. This allows the test data for the test case to be changed before the program is executed or at runtime. The test tool imports the test data from a separate file and assigns it to the relevant test case. Separating test cases and test data facilitates the creation of a large amount of test data for one test case. This is particularly important because a large number of different test situations can be covered, which are then processed by only one test case [Interview3, 2020]. The fourth requirement for a PSD2-compliant test tool is, therefore, a data-driven test approach. This separation also simplifies the adaptation of test cases when changes are made to the interface.

After the bank interface has processed the request, the results must be evaluated to see if everything functioned correctly and no errors occurred. Before the test run, the tester had to define the expected results in the input data. The test tool now compares these values with the attained results and can determine whether or not the request was processed correctly. Thus, the test tool must also be able to verify the test results simply and efficiently.

In addition to the functional requirements for a test tool of PSD2-compliant interfaces, there are also non-functional requirements. The non-functional requirements reflect the quality requirements and the usability of the test tool. For a program to be usable, its performance and reliability are essential. Modifiability and portability must also not be neglected.

The user guidance of a test tool is essential [Interview2, 2020, Interview4, 2020]. Manual entry is quite error prone. Since many features and aspects affect user guidance, just one example is considered. If the user wants to execute test cases, he must be guided through this process in an intuitive way [Interview2, 2020]. It must be easy to choose which test cases to execute. This is especially important when a particular test case has failed, since it must be possible to execute this case again without having to go through the entire test suite [Interview4, 2020]. If this is not possible, the usability of the test tool becomes quite cumbersome. The primary non-functional requirement for the test tool is therefore that it must provide a simple and intuitive user experience.

Another important aspect is the high-performance and reliable execution of the respective test cases. The test tool should be more time efficient than manual testing. It should also ensure that the test cases are executed in the same manner everytime and that there are no unexpected changes in the test environment. The second non-functional requirement is, therefore, to ensure that the test tool works correctly and reliably.

Especially when testing different PSD2-compliant interfaces, it is important to be able

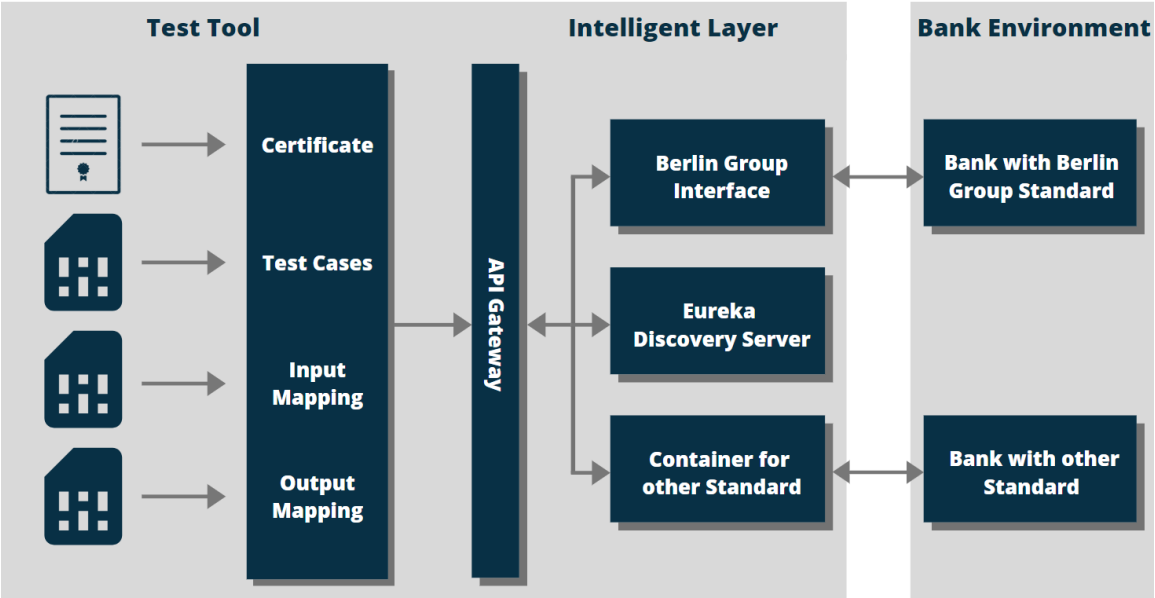


Figure 4.1.: Overview of Test Tool Process

to integrate new functions or even new interfaces [Interview1, 2020]. It must be ensured that there is an easy way to integrate such functionalities. The third non-functional requirement is that the test tool must be conveniently extendable.

Since the development of such a test tool is quite complex and the time available for development was limited, the implementation of the test tool mainly focused on the functional requirements. Aspects such as user guidance, for example, were left out completely.

4.2. Overview of Test Tool Process

This section describes the process of testing a PSD2-compliant interface with the test tool presented in this thesis in more detail. The program consists of several components. Each component is shown and explained in a high-level overview as shown in the figure 4.1. The first component is the actual test tool, which is responsible for processing the input data and evaluating the requests. For the test tool to work successfully, input data such as the test certificate, the test cases and the corresponding mapping structures must be provided.

The second component includes the API gateway and all of the interface standards. This is essentially an intelligent layer that decides which API standard is used and how the content of the request, namely header and body, must be changed to be compatible with the respective bank interface. Once the transformation is complete, the altered request is sent to the actual bank interface.

```
//file: input.json

{
   "nameOfAPIEndpoint" : [
      {
         "testId" : " ",
         "testName" : " ",
         "testDescription" : " ",
         "testConnectionId" : " ",
         "header" : {

            //add header information here e.g. tpp-qwac-certificate
         },
         "body" : {

            //add body information here
         }
         "result" : {

            //add expected results here
         }
      }
   ]
}
```

Figure 4.2.: Basic Test Case Structure

The third component is the banking environment. It contains the addressed bank interface that receives and processes the request. This is not part of the program developed in this work. The bank to be tested must provide the tester with this environment.

The details of these components and the information that the user must provide during the test process are explained below.

In general, each test case has a similar structure, as shown in figure 4.2. It is essential that the key names shown here always remain the same. These are used by the program to extract the relevant values, such as header or body information.

Every test case is represented by a unique value. It is stored in the key `testID`. The `testName`, `testDescription` and `testConnectionID` are not unique and may occur frequently. To indicate that several test cases belong together the `testConnectionID` is used. Here the user must specify a unique string and insert it in all relevant related test cases. The `header` and `body` contain all the necessary information for the request. The `result` contains the information that the tester has provided in order to evaluate

```
//file: input.json

{
   "nameOfAPIEndpoint" : [
      {
         "header" : {
            "tpp-qwac-certificate" : "<<< ADD CERTIFICATE HERE >>>"
         }

         //other values e.g. body, result
      }
   ]
}
```

Figure 4.3.: Example Certificate Header

the request. It is used to check whether the request has returned the expected values. For example, this could be the status code of the HTTP response.

Before an organization or tester can access the bank interface to be evaluated, they must obtain a specific certificate. This certificate can be acquired after a very complex registration process with the national authorities [Barzachki, 2018, Interview1, 2020]. In Germany this registration must be done through the Bundesanstalt für Finanzdienstleistungsaufsicht (BaFin) [Bundesdruckerei, 2019]. After successful completion, the applicant obtains the Electronic Identification, Authentication and Trust Services (eIDAS) certificate. This certificate must be included in the API request sent to the bank interface. Banks use the certificate to verify the identity of the requester. The test tool expects the tester to insert the certificate as a header in the respective test case. A look at the test case structure will make it clear where the certificate must be inserted see figure 4.3. The `nameOfAPIEndpoint` tells the program which API endpoint will be called. This `nameOfAPIEndpoint` is a list containing all test cases that should be sent to the defined API endpoint. In this example, a request with the header element `tpp-qwac-certificate` is sent to the API endpoint `nameOfAPIEndpoint` of the specified bank interface.

The header name, which in this case is `tpp-qwac-certificate`, is not fixed. This means that any header name can be used as long as the mapping file contains the name to which this header has to be translated. For example, the mapping contains the entry `"tpp-qwac-certificate" : "tpp-certificate"`. This tells the test tool that the `tpp-qwac-certificate` must be translated into `tpp-certificate` before the actual request can be sent to the bank interface.

Similar to the header, the `body` and `result` are filled with the necessary information for the respective request. This is, of course, dependent on the API endpoint. How this

is done can be seen in a later example.

The test certificate application and the creation of the test case structure is usually done only once. That is because both of these components can be used for testing several bank interfaces. The only prerequisite is that banks can import the test accounts and test users specified in the test cases into their system.

To ensure that the communication between the test tool and the respective bank interface works, two mapping files must be provided. These contain all information regarding how the interface can be accessed and how the test cases must be modified. Relevant information such as the `host` and `port` are stored in the `serverInformation` of the input mapping. They describe how the request and response have to be mapped to the structure of the bank interface and vice versa. The exact procedure of this mapping is explained in a separate chapter. Once the tester has provided all necessary data – the test certificate, the test cases and the two mapping files – the test tool can be started. It will then import all provided information and execute the test case.

In the first step of the test case execution, all elements are sent to the container that holds the "standard" used by the bank. There, the requests are processed and forwarded to the respective banking interface. In the case of this test tool, all information is sent to the Berlin Group Interface container shown in figure 4.1. The mapping file defines to which standard the request is send. By extracting this information, the gateway knows to which service the information should be sent. Since the gateway does not know the exact address of each service, it first sends a request to the Eureka Discovery Server which manages the addresses of all containers. All individual services must register with this service before they are started. Therefore, if the gateway requests the address of the Berlin Group Interface container, the Eureka Discovery Server can provide this information immediately. Thus, enabling the gateway to send the information to the specified address.

The respective bank interface container receives the request and translates it to the target bank interface. Once the transformation is complete, the actual bank request can be sent. The bank environment processes the request and returns a response. Just like the request, the response must also be transformed. In this case, it has to be mapped to the standard defined by the test tool. This is necessary to ensure that the test tool knows exactly where the values are stored and how to interpret them. After the translation, the response is sent back to the test tool, where it is evaluated.

4.3. Test Case Mapping

In order to be able to use the test cases from the test tool for different bank interfaces, they have to be adjusted to the specified interface definitions. That means that the test tool must know how to adapt the structure of its test cases so that they are understandable for the bank interfaces. Of course, there are several ways to implement such a transformation from a technical point of view. One possibility is to write a separate

```
//file: input_mapping.json

{
   "nameOfAPIEndpoint" : {
      "header" : {
         "x-request-id" : "y-request-id"

         //add additional header information here
         //the strucutre always looks like this:
         //"header name test tool" : "header name bank interface"
      }

      //other values e.g. body
   }
}
```

Figure 4.4.: Sample Header Mapping

module for each individual bank, which then performs the translation. However, for reasons of efficiency and complexity, this is not possible because a specific translation module must be written in advance for each bank. Moreover, this approach also has certain limitations. For example, if a new bank is established and wants to use the test tool, no module for translation will exist. It may also happen that an implemented module is not used at all, a waste of resources. Although these modules can be changed easily to adapt to updates, this is far more complex than necessary. The test tool presented in this thesis uses a different approach. In order to explain how this is done, it is necessary to go back to the fundamental problem of why a translation is necessary.

Banks are legally not bound to follow a certain standard when they implement their interfaces. The standardization of payment transactions and interfaces is one of the fundamental goals of PSD2. Various organizations have been founded to address this problem in a uniform manner e.g., via interface descriptions that banks can use. Banks often change individual components making a uniform communication between these interfaces difficult. That is of course a challenge when several interfaces should be evaluated with the same test tool. In order to solve this problem, an intelligent layer is needed that allows a translation between the individual requests and the bank interfaces. The concept of this intelligent layer is explained below.

Each request consists of two parts, a header and a body, although the body is not always necessary. The translation is done in two steps. In the first stage, the header is adapted to the respective specifications of the bank interface. That is essentially a one-to-one translation. Figure 4.4 shows how this looks like. The test case contains the header `x-request-id`. The bank interface does not understand this header. In its specification, this header is called `y-request-id`. Therefore, it must be translated. Figure 4.4 shows

```
//file: input.json

{
   "nameOfAPIEndpoint" : [
      {
         "body" : {
            "objectA" : {
               "arrayB" : [
                  {
                     "keyC" : "valueC"
                  },
                  {
                     "keyC" : "valueC"
                  }
               ],
               "objectB" : {

               },
               "keyB" : "valueB"
            },
            "arrayA" : [

            ],
            "keyA" : "valueA"
         }

         //other values e.g. header, result
      }
   ]
}
```

Figure 4.5.: Sample Test Case Body

how such a translation must be addressed to be understood by the test tool. Essentially what happens is that the test tool swaps the header names, enabling the bank interface to process the header information.

The translation of the header is quite easy since only the header names have to be swapped. Translating the body is a bit more complicated. For the sake of simplicity, the only standard examined in this thesis is the one defined by the Berlin Group. Therefore, only bodies with JavaScript Object Notation (JSON) are discussed in the following, although there are several other ways to represent the information in a HTTP body.

The translation of the body is associated with some challenges. For example, not only

```
//file: input_mapping.json

{
   "nameOfAPIEndpoint" : {
      "body" : {
         ".arrayA:" : ".arrayA.",
         ".keyA" : ".objectA.objectB.keyC",
         ".objectA.arrayB:0.keyC" : ".objectA.arrayB.keyC0",
         ".objectA.arrayB:1.keyC" : ".objectA.arrayB.keyC1",
         ".objectA.keyB" : ".objectA.keyB"
         ".objectA.objectB." : ".objectA.objectB.",
      }

      //other values e.g. header
   }
}
```

Figure 4.6.: Sample Body Mapping

key names can change. Theoretically, the whole structure of the body can change. For this reason, it is important to understand the different ways JSON can look like. There are two different data structures in JSON [JavaEE, 2014]. Firstly objects that are characterized by curly brackets and secondly arrays that are defined by square brackets. There are also different value types, such as objects, arrays, and strings [JavaEE, 2014]. These types are relevant for the translation. In each mapping file, the left side describes the information to be translated. In the case of the test tool prototype, this is the test case representation. The right side shows how this information must be translated to be understood by the bank interface.

A simple way to visualize this translation is to think of it as a tree, starting at the root and describing all possible occurrences up to a leaf. A more detailed explanation is given on the basis of the JSON structure shown in figure 4.5. It shows the representation of a request body of a possible test case in the test tool. This body contains various objects, arrays and values. The addressed bank interface cannot work with the structure shown in figure 4.5. It requires a different format.

To perform a translation, the tester must specify the required format for the bank interface. An example of such a format is shown in figure 4.6. Looking at `.objectA.arrayB :0.keyC` will clarify how the program performs the translation. The `. (DOT)` shows that `objectA` is located in a JSON object. The next element the `. (DOT)` indicates that `arrayB` is also stored in an object. More interesting are arrays, which are represented by a `: (COLON)`. A subsequent number indicates that the next element is stored at the position of this number in the array. The object containing `keyC` is thus at the first position in the array of `arrayB`. This translation works the same way in the reverse

```
//translated request for bank interface

{
   "nameOfAPIEndpoint" : [
      {
         "body" : {
            "objectA" : {
               "arrayB" : {
                  "keyC0" : "valueC"
                  "keyC1" : "valueC"
               },
               "objectB" : {
                                 "keyC" : "valueA"
               },
               "keyB" : "valueB"
            },
            "arrayA" : {

            },
         }

         //other values e.g. header
      }
   ]
}
```

Figure 4.7.: Request For Bank Interface

direction.

Technically, this is accomplished by a tree, which represents the JSON body. This tree is mapped to the representation shown in figure 4.6. The mapping file is the key for the program to determine what the information for the other side must look like. In our example this is `.objectA.arrayB.keyC0`. With this representation, a new tree can be constructed. The complete translation can be seen in figure 4.7. Now the translated request can be sent to the bank interface.

In general, the concept of translation makes it possible to map all test cases from the test tool to any bank interface without changing the program itself. That allows a very flexible test process. If the bank interface sends a response, the translation process is performed the same way and therefore requires no further explanation.

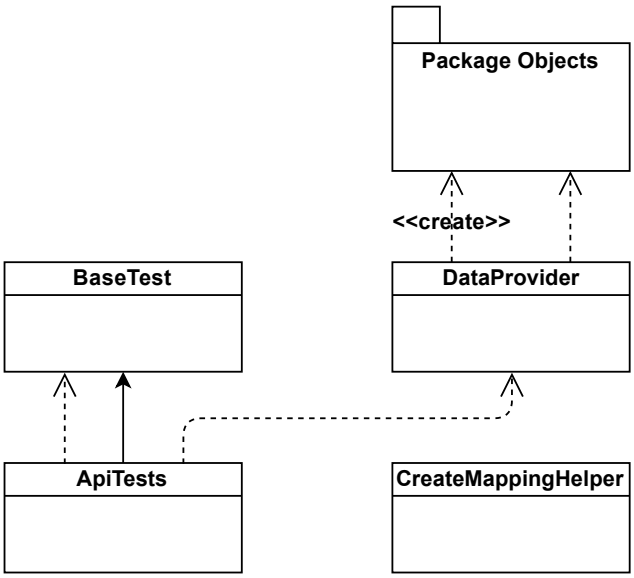


Figure 4.8.: Architecture Extract of the testservice Module

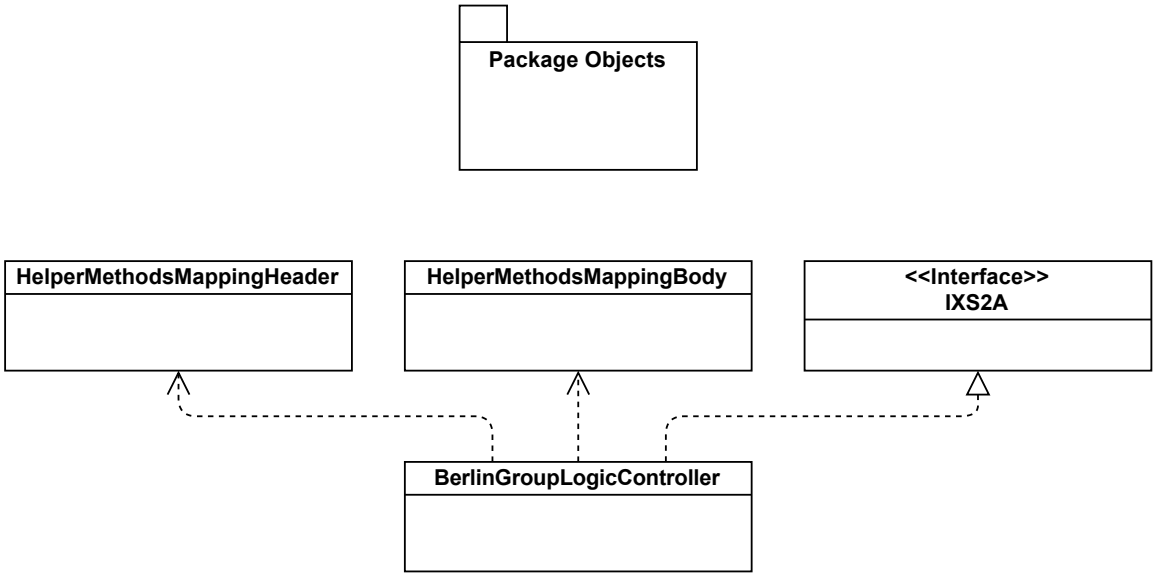


Figure 4.9.: Architecture Extract of the berlingroupinterface Module

4.4. Test Tool Architecture

This section describes the architecture and technical implementation of the test tool. It essentially consists of four different modules, the `testservice`, the `apigateway`, the `eurekadiscoveryserver` and the `berlingroupinterface`. Since the `testservice` module and the `berlingroupinterface` module contain all the crucial programming work, figure 4.8 and figure 4.9 show a simplified picture of their architecture. A de-

tailed version of the entire test tool architecture is shown in figure 8.3 and 8.4 in the appendix of this thesis. The following briefly describes each module and the relationship between the individual components. This provides a basic understanding of the system and therefore a better comprehension of the relationship between the deduced requirements and the actual implementation of the test tool.

Each of these modules is a microservice. This allows each module to focus on just one task. Microservices enable larger projects to be divided into smaller units [Redhat, 2020, Mak, 2017]. All of these modules can be used independently of each other. This gives great flexibility during development, facilitates the modularization of the test tool, and assures easy maintenance of the respective modules [Redhat, 2020, OpenSource, 2020, Mak, 2017]. It also has significant advantages in terms of expandability, e.g. not having to stop everything when adding something new [OpenSource, 2020]. The test tool presented here, was implemented with Spring, a Java-based framework that allows microservices to be created easily [Spring, 2019]. The Spring framework already offers its users a variety of different functionalities that facilitates the implementation of a project [Spring, 2019]. For example, a simple API gateway, as used in the test tool, can be implemented relatively quickly. The Spring framework offers an "out of the box" solution for this task.

The `testservice` module uses TestNG framework and RestAssured library to facilitate the testing process of PSD2-compliant interfaces. TestNG allows a straightforward way to perform data-driven testing and RestAssured provides an excellent method to create simple test cases for RESTful APIs [TestNG, 2019, REST Assured, 2020a]. The `testservice` module loads both the test cases and the two mapping files and processes the execution of the test cases. The `DataProvider` extracts all necessary, test data and sends them to the class `ApiTests` which then assigns them to the relevant test cases. Afterwards the respective request is sent to the API gateway.

The `apigateway` module is based on the Spring Cloud Gateway. The task of this module is to forward the individual requests to the correct PSD2-compliant standard. The addresses of the individual services are not statically defined. This means prior to the gateway forwarding something, the address must be requested from the Eureka Discovery Server.

The task of the `eurekadiscoveryserver` is to collect information and to store the addresses of the individual services. A new service has to register with the Eureka Discovery Server before it can start. Should the API gateway not know the destination address of a service, it can ask the Eureka Discovery Server for the destination address of the request.

The `berlingroupinterface` module represents the Berlin Group Standard. Each API endpoint of the standard is implemented in the `BerlinGroupLogicController`. If a specific API call is made, the test tool sends the request to this service. There the respective request is translated and then forwarded to the actual bank interface.

4.5. Summary of the Design and Implementation

The aim of the Design and Implementation was to give an overview of the test tool developed in this thesis. It illustrated how the process of evaluating an interface with this test tool, gave both an insight and an overview of the input data needed to execute the test tool and presented an mechanism which allows the translation of test cases to the individual bank interfaces. With this approach a large number of different bank interfaces can be evaluated with one test set. An insight into the architecture and the technical implementation of the test tool prototype was also given. In the following, difficulties and problems during the implementation are briefly discussed as well as if and how they were solved.

In general there were several attempts to develop the prototype of this test tool. As soon as the requirements for the test tool were identified, the implementation started. For a test tool to be able to perform various operations, certain functionalities are required. However, it was not clear how automated testing or data-driven testing could be technically implemented. For this reason, the development phase began with the technical realization of these requirements. After a better understanding of the individual functionalities was attained, a new project was started which resulted in the current test tool.

The second challenge was the feasibility of evaluating different interfaces with only one test set. In the first step, the test tool was built with the assumption to create a separate module for each bank interface. Each standard should have had a separate container. This container would have contained a single translation module for every bank. However, this way forward would have only been possible with a great deal of effort. Another solution had to be found in order to implement this in a more efficient way. In the end, the translation of the test cases for the respective interfaces was realized by the mapping process presented above.

A further obstacle was the implementation of the authorization process. Since there are various ways to implement this, it was necessary to understand the functionality of the particular methods first. Automating this process is not trivial. Currently this is solved through the usage of a statically set TAN. It would be very interesting, to implement this without a fixed value, however, this would be topic of another scientific contribution.

In summary, it can be said that the implementation of such a test tool is relatively complex. Many functionalities that initially appear quite simple are not as easy to implement as expected. The prototype presented here shows a possible approach how a large number of different PSD2-compliant interfaces could be evaluated with a single test set.

Part III.

Appraisal and Conclusion

5. Test Tool Evaluation from a User Perspective

For a tool to be successfully adopted by a user, several aspects need to be considered. Apart from being feature-rich and extremely reliable, the user interface and usability play a crucial role. There are several ways to evaluate the quality of software. The evaluation of the test tool presented in this thesis is based on the ISO 9126 and 25010:2011 standard [Liliana Bobrica, 2002, ISO, 2011, SQA, 2001]. This approach considers six major aspects in order to make a statement about software quality. These include functionality, reliability, usability, efficiency, maintainability and portability [Liliana Bobrica, 2002, SQA, 2001]. In the following, each of these factors is briefly discussed and the test tool congruence to each factor will be appraised. This gives an overview of the usability and practicability of the test tool from a user's perspective and will contribute to a better understanding of the test tool's software quality.

The functionality aspect requires features that are necessary to ensure that the test tool can meet the desired needs of the user [Liliana Bobrica, 2002]. Currently, the prototype is capable of testing eight different API calls from the AIS of a bank interface supporting the Berlin Group standard. The test tool uses an automated and data-driven test procedure to evaluate these requests. This allows a large number of different test cases to be evaluated. Regression tests, e.g. the re-execution of already correctly evaluated test cases, are also possible. Such functions improve the efficiency of the test process considerably and simplify the process to ensure that the individual API calls function correctly.

Another aspect listed in the ISO standard deals with the reliability of the software [Liliana Bobrica, 2002]. In this case, the test tool always ran smoothly during the test process of the individual interfaces. The tests of both complete workflows, as well as individual API calls, behaved correctly and without any problems during the execution. However, only a small amount of test cases was executed never more than 20. How the program behaves with a larger number of test cases was not examined.

The usability of a software program is another aspect that is considered in the ISO standard [SQA, 2001]. Sufficient functionality as well as factors such as user guidance and the user interface play an important role regarding usability. For this test tool, only quite elementary functions of a test tool - the import and processing of test cases - were implemented. The necessary simple and self-explanatory user interface is not available in the test tool presented in this thesis. Unfortunately, the tool has no GUI at

all. Everything is controlled from the Integrated Development Environment (IDE) that the tester uses. This makes the use of the test tool difficult. It is also expected, that a user can use the test program without detailed knowledge of its implementation. The tool must, therefore, support the user constructively in order to avoid mistakes during use. The prototype lacks a mechanism that checks whether all necessary information has been provided before starting the translation of a request. For example, if the user forgot to define the bank interface standard in the request, or if the mapping is incorrect, the program will fail. A mechanism that does not allow the program to start until all the necessary information has been provided would avoid such errors and contribute considerably to the user-friendliness of the program. Such a feature needs to be added to optimize the use of the test tool. The way in which test cases are imported into the test tool also plays an important role in terms of usability. Currently, the test cases and mappings must be entered manually into the JSON file. Manual input is always a very error-prone process, not only can mistakes be made, but it is very likely that necessary information for the program will be forgotten. If an error occurs, it is more likely to be caused by an incorrect input than by a malfunction of the tested bank interface.

In terms of efficiency and maintainability, the test tool performs well [Liliana Bobrica, 2002]. The use of the latest technologies guarantees a resource-saving execution of the test tool. This statement depends of course on the number of evaluated test cases. More resources are required if several thousand test cases are evaluated, as opposed to just 20 test cases. In general, however, it can be said that the test tool is relatively resource-efficient. The use of the latest technologies also facilitates the maintainability of the test tools. Especially the use of microservices allows good scalability and expandability of the test tool. Therefore it can be said that due to the use of latest technologies and a good software architecture, the test tool satisfied the criteria of efficiency, extendability and maintainability.

The last point of the ISO standard deals with the portability of software [Liliana Bobrica, 2002, SQA, 2001]. In theory, the test tool can be transferred from one environment to another in a very simple way. Since the test tool is written in Java, it can be executed on any machine with a Java Virtual Machine. The Java Virtual Machine ensures the interoperability of the individual Java programs on different machines.

The test tool presented in this paper scores quite well when evaluated against the ISO standard. From a functional point of view, many features are covered. Additionally, the test tool is extensible and maintainable. The non-functional requirements, which are covered in the usability aspect of the ISO standard, are not satisfied. A GUI as well as a simpler way to add new test cases would be desirable. Such features would contribute substantially to the handling of the test tool. However, the non-functional requirements were not the focus of the test tool prototype presented in this thesis. Decisive for this thesis is the newly gained scientific contribution. The thesis's goal was to provide a method to evaluate different PSD2-compliant API interfaces with a single test data set. Through intelligent mapping, the test tool presented in this thesis can easily translate

requests and responses enabling the communication with a variety of bank interfaces.

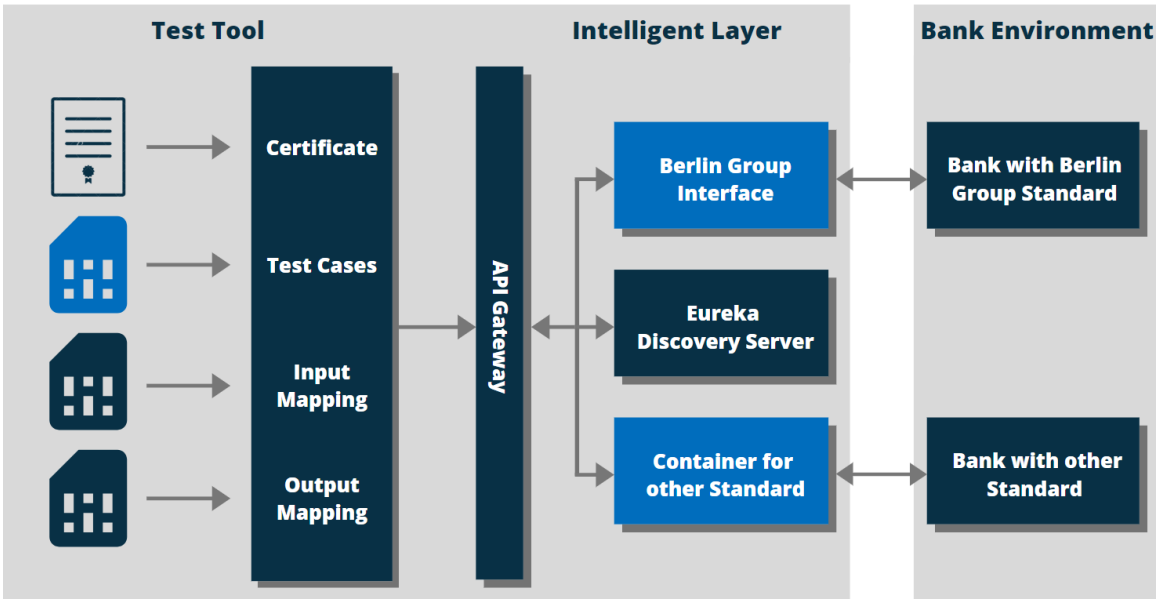


Figure 6.1.: Overview of Test Tool Process - Extendable Parts

6. Extensibility Analysis of the Test Tool

One of the main topics of this thesis is the evaluation of the extensibility of the test tool. It is, therefore, essential to distinguish between different types of program extensibility. Figure 6.1 highlights the different ways of enhancing the program. These three possibilities are illustrated in the following.

The first is to create new test cases, the second is to add new API calls to extend the functionality of the tool and the third is the possibility of adding additional interface standards. Of course, there are other ways to extend the application as well, such as allowing new authorization methods. Since the goal of this evaluation is to assess the general extensibility of the tool, small and really specific functionalities like adding new authorization methods are not examined.

The test tool can evaluate eight different API calls of the AIS of the Berlin Group standard. The particular test cases are stored in a JSON file. Of course, this file can be supplemented with further test cases at any time. The tester only has to add them to the list of the corresponding API calls. Depending on whether the examiner wants to evaluate a complete workflow, meaning the execution of several dependent API requests, or only a single request, the corresponding parameters have to be set. If several test

cases belong together, the `testConnectionID` specifies the link between them. It is not possible to add test cases that attempt to access unimplemented API calls. Such API calls are not recognized by the test tool because it does not know their structure and test names. Therefore the tool can not process the request. In general, it is possible to extend the program by adding new test cases for all implemented API interface calls. This also happened repeatedly during the evaluation of each individual API call.

The second alternative, the implementation of new API calls, is explained. Every API call is part of a standard that has its own container in the program. For the sake of simplicity, only the standard of the Berlin Group is considered. A standard provides the interface description for a possible realization of a PSD2-compliant API. The information from this interface description was used to create the Berlin Group container utilized in the test tool. It contains a Java interface that defines abstract methods for all available API calls. To enable a unique identification, these methods are named corresponding to the `operationIDs` specified in the interface description of the Berlin Group standard. Each individual method is implemented in a controller. Here the headers and bodies are translated to ensure a smooth communication between the test tool and the bank interface. So if more API calls are added, the following steps must be performed. The new API calls must be defined as abstract methods in the Java interface. In the `berlingroupinterface` module these are added in the `IXS2A`. Then they can be implemented in the designated controller. In our case this is the `BerlinGroupLogicController` class. Once these are implemented, the program can call them. However, before the actual test case can be created in the test tool, the Plain Old Java Object (POJO) containing the test data must be updated. POJOs are used to display data in a readable and more explicit format. In the `berlingroupinterface` module they are stored in the `Objects` package. Once these steps are completed, it is possible to create a new data-driven test case for the newly added API call. The last step is to add the test case information to the JSON file with the test cases. Now the test tool can evaluate the newly added API call. Although this process of adding new API calls is not as easy as adding new test cases, it is very flexible and still manageable. Nevertheless, the integration of new API calls requires a good understanding of the architecture and functionality of the tool.

Adding a new standard to the test tool is the third aspect to be evaluated. That is definitely more complex than adding new test cases or API calls. As seen in the overview of the test tool shown in figure 4.1, a new container for another standard can be built in the same manner as the existing Berlin Group container. First of all, the individual methods have to be specified in the Java interface. Afterwards, they can be implemented in the respective container. Everything is basically done in the same way as when adding a new API call, which is described above. Once this is done, the API gateway must be extended with the new standard to allow the transfer of information to the corresponding service. Adding a new standard requires some effort, but if an organization wants to test multiple standards with the same test set, this effort is manageable. For time reasons and to keep the implementation of the test tool as simple as possible,

only one standard was integrated into the test tool.

With these three options for extending the test tool, an extensive, flexible, efficient test process for PSD2-compliant interfaces can be created.

7. Conclusion

"Open Banking is a ten-year project", this is how Andreas Pratz from STRATEGY& described the current status of Open Banking in an interview with the Börsen Zeitung. [Börsen Zeitung, 2020] There are still many challenges to be overcome in the next years. Nevertheless, OpenBanking and PSD2 have already laid a very important foundation for the standardization of payment markets in Europe. It is unclear whether there will be changes to the current directive or even a new one in the future. Maybe a uniform standard will be introduced. What is certain, however, is that many of these problems could have been avoided if the Directive had prescribed a uniform standard instead of each bank bringing its own tailor-made solution to the market [Interview1, 2020]. Pratz also explained in his interview that for many banks the so-called multibanking, where several accounts are integrated into one account, is not manageable from a purely technical point of view [Börsen Zeitung, 2020]. He does not say what exactly the reason for this is, but the variety of standards offered is certainly a decisive factor. One of the key learnings, while working on this thesis was the realization that a uniform standard would have had considerable advantages. A test procedure for PSD2-compliant interfaces with a standardized interface would have made testing much easier. Since this is not the case, the question was raised whether there is a simple way to evaluate several bank interfaces with only one test data set. At the end of this thesis, the most important results, as well as the problems and limitations that arose during the work on this topic, are summarized. Finally, a short outlook on what could be done in the future is given.

7.1. Key Findings

This thesis deals with the requirements, the implementation and the extensibility of a test tool for PSD2-compliant bank interfaces.

The test process for PSD2-compliant APIs is not trivial. Additionally, it is required that certain prerequisites need to be observed, when evaluating such interfaces using a test tool. The test tool must also provide an easy way to evaluate a large number of different PSD2-compliant interfaces. The test process must be automated and data-driven to allow for efficient testing of a large number of test cases. Of course, the test tool should be able to re-execute already successfully evaluated test cases, i.e. perform regression testing. A test tool that is specifically designed for testing different bank interfaces should also have the possibility to add functionality on the fly. For

example, a specialized reporting system or an exclusive type of authentication method could be integrated into the test tool to allow a simpler test approach. Besides the functional aspects, considerations regarding the user guidance and the user interface must not be omitted. Due to the complexity and time constraints of this thesis, the last two points were not take into account when implementing the test tool.

Besides the research inquiry of functional and non-functional requirements for a PSD2-compliant test tool, a prototype was developed based on these research results. This prototype is capable of evaluating eight different API endpoints from a bank interface that is compliant with the Berlin Group standard. The test cases are imported from a file provided by the tester. These test cases are then transformed via a user defined mapping into a structure which the addressed interface is able to process. This is exceptionally difficult if several interfaces are to be tested with a single test tool. The challenge here lies in the variations of the individual interfaces. Although many APIs have a very similar structure, individual providers usually add minor changes such as changed header names or different body structures. The mapping mechanism used allows the test tool to address a large number of different interfaces and evaluate them with one test set. This simplifies the test process for PSD2-compliant interfaces considerably.

Besides the technical aspects of the test process, the functional understanding of the individual components plays an important role. For example, the tester must know how the individual headers have to be set up and what options are available for the different bodies of the request. If these are not set correctly, the request may not work at all. It is also very important to have a good understanding of the individual processes that can be implemented with a PSD2-compliant interface. This knowledge is necessary in order to be able to use the test tool. Nevertheless, financial institutions can save a lot of time and effort during the test process if they use the concepts and leverage the test tool prototype developed in this thesis. Especially since the tool offers the functionality to process a test case catalog defined together with other financial institutions for the evaluation of the bank specific APIs.

The third part of this thesis focused on the extensibility of the test tool. A differentiation was made between three different extensibility types. The first considered the addition of new test cases. Since the test tool imports test cases from a file, it is possible to add new test cases at any point in time. The second factor dealt with the extensibility in regard to adding new API calls. This requires a bit more effort than that of adding new test cases, but is also possible. The third and probably the most critical point in extending this test tool is the integration of a new standard. Since the test tool consists of individual microservices and each standard is a separate container, meaning a separate microservice, new standards can be merged smoothly into the existing test program. During the development of the prototype, certain design decisions were made to facilitate the implementation of the adaptation and modification of requirements and the updating of interface standards. Therefore, special attention was paid to the extensibility of the test tool. In general, the architecture of the test tool

allows a flexible integration of different interfaces and new API calls.

In summary, the test process was simplified by providing a data-driven, vendor-independent test tool for different bank interfaces. Currently, the developed prototype is limited in its functionality and can only evaluate eight different API calls from the AIS of the Berlin Group standard. However, the test tool contains an intelligent layer that allows the transformation of test cases to different interfaces via a predefined mapping. This mapping mechanism allows the test tool to address a large number of interfaces. The concept of this mapping process is the major insight and scientific contribution of this thesis. It provides a transparent solution to simplify the current test process for several different PSD2-compliant API interfaces.

7.2. Problems and Limitations

As in every project, this thesis and the development of the test tool prototype were also confronted with several obstacles as well as some unexpected results and situations.

The initial idea was to test the tool with the bank's test environment, the so-called sandbox. According to PSD2, the banks must make these sandboxes available to TPPs for testing purposes. In principle, this is a very convenient concept, as these sandboxes already contain test data in form of user accounts and transactions. They also allow a TAN to be set statically, which simplifies the test process considerably. However, when confronted with this topic for the first time, it became obvious that a "simple developer" would not be able to access these sandboxes, since they require an Electronic Identification, Authentication and Trust Services (eIDAS) certificate. With such a certificate, a third party can prove its identity to another party. The TPP can only obtain such a certificate if they are registered with a corresponding national authority. In Germany, this function is carried out by the BaFin. During the interview process of this thesis, it was also confirmed that this registration could take several months and that an individual has no possibility to obtain such a certificate [Interview1, 2020]. Even if this dilemma could have been solved, the next challenge that the sandboxes do not allow the replication of a business case would have led to a dead end. This means, that although it is possible to access the bank interfaces, it is not possible to test them with the test tool, because the interface always returns the same static information. However, there is a sandbox that can be configured and run locally on a machine [Adorsys, 2018]. Additionally, this sandbox provides a test certificate for accessing the PSD2-compliant interface and allows a business case to be executed. With this sandbox, it was possible to test the prototype.

The second major obstacle was the authentication process of the individual users. As already mentioned, there are several ways to do this. In most cases, the user gets an One Time Password (OTP) after entering his credentials. The TAN is then used to finalize the authentication. From a theoretical point of view, this seems relatively easy to automate. In reality, however, this is a complex process. Several solutions to

this problem were discussed during the interviews. If the banking environment has the ability to set a static TAN, the test process of the authentication can be automated easily. Otherwise, a bank can offer a service that always sends the correct TAN when requested. The sandbox used for this test tool prototype can set a static TAN for every user.

In addition to the challenges listed above, some restrictions had to be made. Currently, the test tool can only evaluate eight different API calls from the AIS of the Berlin Group standard. Although this limits the conclusion about the overall functionality of the test tool, a thorough insight into its concept, architecture and functionality can be conceived. The Berlin Group standard was chosen because of its high market position. That gives the program the ability to cover a particularly broad spectrum of different bank interfaces. Since the interface description of the Berlin Group standard is very extensive, the test tool only evaluates a few API calls from the AIS. That is acceptable since it allows this work to concentrate on scientific contributions such as the mapping of test cases.

7.3. Future Work

Due to the time constraints of this thesis, several aspects had to be simplified and limited.

The test tool covers only a fraction of the functionality defined in the specification of the Berlin Group standard. This restricts the evaluation of the functionality of the entire test tool somewhat. It would, therefore, be interesting to take a closer look at more difficult and complex API requests. The inclusion of such API requests would certainly broaden the picture of the capabilities of the test tool.

Another relevant consideration would be to evaluate the tool using a different banking interface. So far, this test tool has only been evaluated using one sandbox, the reason being that no official eIDAS certificate was available. Such an option would determine how well an adaptation of the test tool to another interface really works. This would also verify how promising this test approach really is.

Finally, it should be noted that this test tool has no GUI. Not only would a GUI simplify the test process, it would also provide an elegant and clear way to integrate new test cases and mappings. If the tester could enter these directly to the application only an understanding of the functionality of the test cases and interfaces would be necessary. Currently knowledge of the workings of the underlying tool is necessary in order to make such changes.

Of course, these are just a few possibilities that could be considered when further researching this project.

Appendix

Interview Partner and Questions

ID	Role	Duration [hh:mm]
Interview1	Vice President Marketing / Sales	0:45
Interview2	Head of Test and Quality Assurance Department	0:44
Interview3	Process Analyst and Project Management, Test Management	2:44
Interview4	Head of Software Engineering & Application Management	0:54
Interview5	Managing Director Financial Startup	0:17

Figure 8.1.: Interview Participants

1. **Wie sieht der aktuelle Testprozess für PSD2-konforme APIs aus?**
   a) Was sind die Anforderungen und Merkmale für das Testen von PSD2-konformen APIs?
   b) Verlassen Sie sich auf API-Testwerkzeuge wie Postman, SOAPUI, etc.? Warum haben Sie sich für dieses spezielle Tool entschieden? Was spricht dagegen?
   c) Spielen Regressionstests eine wichtige Rolle in diesem Prozess? Wie würden Sie die Relevanz und Anwendbarkeit von Regressionstests einordnen?
2. **Wie könnte ein automatisiertes Testwerkzeug beim Testen von PSD2-konformen APIs helfen?**
   a) Welche Herausforderungen und Möglichkeiten sehen Sie in einem solchen Testwerkzeug?
   b) Inwieweit halten Sie es für sinnvoll, den Testprozess von PSD2-konformen APIs zu automatisieren?
   c) Wie stellen Sie sicher, dass Sie alle für die Testautomatisierung geeigneten Testfälle finden?
3. **Welche Funktionalitäten werden für ein solches Testwerkzeug benötigt?**
   a) Was sind typische Probleme bei der Einrichtung und Durchführung von Testautomatisierung? Wie könnte man diese Probleme einfach lösen?

   b) Glauben Sie, dass ein datengetriebenes und verhaltens-getriebenes Testwerkzeug den Testprozess verbessert? Was genau kann dies Ihrer Meinung nach verbessern?
   c) Welche Testframeworks würden Sie für ein solches Projekt verwenden? Was halten Sie von Rest Assured?
4. **Wie können Regressionstests bei diesem Problem helfen?**
   a) Welche Strategie verfolgen Sie bei der Erstellung von Regressionstests für APIs?
   b) Was denken Sie über die Verwendung von Heuristiken für die Generierung von Testfällen?
   c) Würden Sie Ihre Regressionstestfälle manuell oder automatisch erstellen und wie würden Sie sicherstellen, dass Sie eine hohe Testfallabdeckung erhalten?
5. **Wie hoch ist der Nutzen einer automatisierten Testfallgenerierung wirklich?**
   a) Kann eine rein randomisierte Generierung von Testfällen überhaupt gute und qualitativ hochwertige Testfalle erstellen? Was verbessert eine randomisierte Generierung von Testfällen Ihrer Meinung nach genau am Testprozess?
   b) Inwieweit halten Sie es für sinnvoll Testfälle von einem Modell automatisiert generieren zu lassen?
   c) Welche Methoden zur Testfallgenerierung für eine OpenAPI Spezifikation würden Sie empfehlen?
6. **Wie würden Sie die Erweiterbarkeit eines solchen Tools sicherstellen?**

Enumeration 8.2.: Interview Questions

Test Tool Architecture

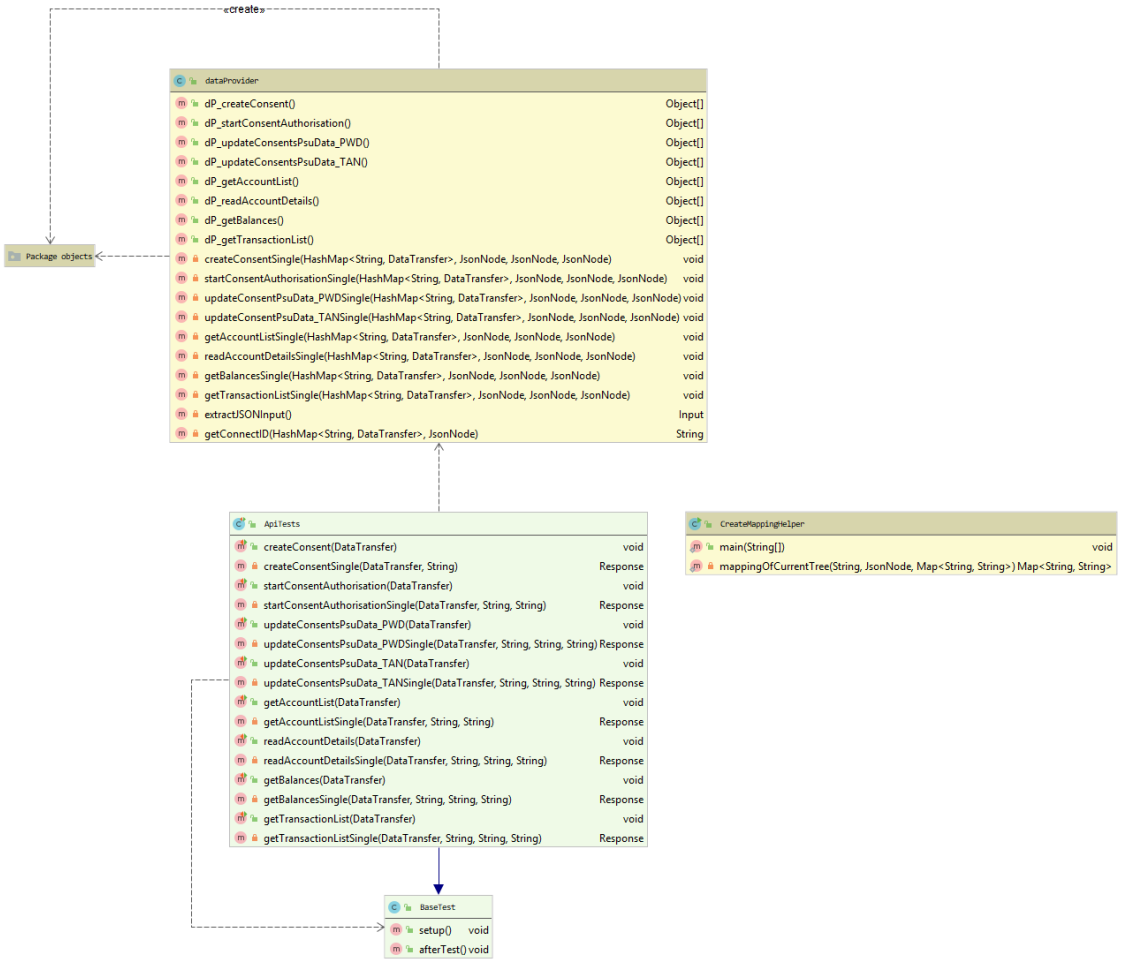


Figure 8.3.: Architecture of the testservice Module

Figure 8.4.: Architecture of berlingroupinterface, apigateway, eurekadiscoveryserver Module

List of Figures

Acronyms

AIS Account Information Service.

AISPs Account Information Service Providers.

API Application Programming Interface.

APIs Application Programming Interfaces.

BaFin Bundesanstalt für Finanzdienstleistungsaufsicht.

CMA British Competition and Markets Authority.

EBA European Banking Authority.

ECB European Central Bank.

eIDAS Electronic Identification, Authentication and Trust Services.

EU European Union.

GUI Graphical User Interface.

HTTP Hypertext Transfer Protocol.

IDE Integrated Development Environment.

IEEE The Institute of Electrical and Electronics Engineers.

ISR Information Systems Research.

JSON JavaScript Object Notation.

NISP NextGenPSD2 Implementation Support Program.

OBI Open Banking Initiative.

OTP One Time Password.

PIS Payment Information Service.

PISPs Payment Initiation Service Providers.

POJO Plain Old Java Object.

POJOs Plain Old Java Objects.

PSD2 Second Payment Service Directive.

PSPs Payment Service Providers.

REST Representational State Transfer.

RTS Regulatory Technical Standards.

SOAP Simple Object Access Protocol.

TPP Third Party Provider.

TPPs Third Party Providers.

TPPSPs Third Party Payment Service Providers.

XML Extensible Markup Language.

XS2A Access to Account.

Bibliography

[Accenture, 2016] Accenture (2016). Seizing the opportunities unlocked by eu's revised payment services directive.

[Adorsys, 2018] Adorsys (2018). Xs2a-sandbox.

[Alan R. Hevner et al., 2004] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram (2004). Design science in information systems research.

[Ammann and Offutt, 2008] Ammann, P. and Offutt, J. (2008). *Introduction to software testing*. Cambridge Univ. Press, Cambridge, 1. publ edition.

[Balaban, 2011] Balaban, M. (2011). Software testing.

[Bangare et al., 2012] Bangare, S., Borse, S., Bangare, P., and Nandedkar, S. (2012). Automated api testing approach. *International Journal of Engineering Science and Technology*, 4.

[Barzachki, 2018] Barzachki, A. (2018). Eba draft opinion on the use of eidas certificates under the rts on scacsc.

[Benzell and van Alstyne, 2016] Benzell, S. G. and van Alstyne, M. W. (2016). The role of apis in firm performance. *SSRN Electronic Journal*.

[Berlin Group, 2020] Berlin Group (2020). Berlin group.

[Bitbar, 2019] Bitbar (2019). How can regression testing benefit banks?

[Bobrica and Niemela, 2002] Bobrica, L. and Niemela, E. (2002). A survey on software architecture analysis methods.

[Börsen Zeitung, 2020] Börsen Zeitung (2020). Open banking ist ein zehnjahresprojekt.

[Bramberger, 2019] Bramberger, M. (2019). *Open Banking: Neupositionierung europäischer Finanzinstitute*. essentials. Springer Fachmedien Wiesbaden and Springer Gabler, Wiesbaden.

[Broadcom, 2020] Broadcom (2020). Api testing guide: An automated approach to api testing transformation.

[Bundesdruckerei, 2019] Bundesdruckerei (2019). Zertifikate für psd2.

[Chaib, 2017] Chaib, I. (2017). Regulating open banking - how regulators around the world are shaping the future of financial services.

[Competition and Markets Authority, 2016] Competition and Markets Authority (2016). Retail banking market investigation: Overview of final report.

[Coste and Miclea, 2019] Coste, R. and Miclea, L. (2019). Api testing for payment service directive2 and open banking. *International Journal of Modeling and Optimization*, 9(1):7–11.

[Daniel Gozman, Jonas Hedman, and Kasper Sylvest Olsen, 2018] Daniel Gozman, Jonas Hedman, and Kasper Sylvest Olsen (2018). Open banking: Emergent roles, risks and opportunities.

[Deloitte, 2017a] Deloitte (2017a). Open banking and psd2.

[Deloitte, 2017b] Deloitte (2017b). Payment service directive 2.

[Dijkstra, 2009] Dijkstra, P. (2009). Anforderungsanalyse.

[Economic Times, 2000] Economic Times (2000). What is software testing? definition of software testing, software testing meaning - the economic times.

[Ed-douibi et al., 2018] Ed-douibi, H., Canovas Izquierdo, J., and Cabot, J. (2018). Automatic generation of test cases for rest apis: A specification-based approach.

[Edgescan, 2018] Edgescan (2018). Payment services directive (psd2) - opening the doors to a secure business.

[Ehmer and Khan, 2012] Ehmer, M. and Khan, F. (2012). A comparative study of white box, black box and grey box testing techniques.

[Elbaum et al., 2002] Elbaum, S., Malishevsky, A. G., and Rothermel, G. (2002). Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182.

[European Banking Authority, 2017] European Banking Authority (2017). Final report on strong customer authentication and common and secure communication under article 98 of directive 2015/2366 (psd2).

[European Banking Authority, 2019] European Banking Authority (2019). Eba publishes clarifications to the first set of issues raised by its working group on apis under psd2.

[European Banking Authroity, 2018] European Banking Authroity (2018). Opinion on the implementation of the rts on sca and csc (eba-2018-op-04).

[European Central Bank, 2018] European Central Bank (2018). The revised payment services directive (psd2).

[European Commission, 2016] European Commission (2016). Payment services.

[European Commission, 2018] European Commission (2018). Payment services directive: frequently asked questions.

[European Parliament, 2015] European Parliament (2015). Directive (eu) 2015/ of the european parliament and of the council of 25 november 2015 on payment services in the internal market, amending directives 2002/65/ec, 2009/110/ec and 2013/36/eu and regulation (eu) no 1093/2010, and repealing directive 2007/64/ec.

[European Parliament, 2019] European Parliament (2019). Commission implementing regulation (eu) 2019/ 410 - of 29 november 2018 - laying down implementing technical standards with regard to the details and structure of the information to be notified, in the field of payment services, by competent authorities to the european banking authority pursuant to directive (eu) 2015/ 2366 of the european parliament and of the council.

[European Union, 2016] European Union (2016). Revised rules for payment services in the eu - eur-lex.

[European Union, 2018] European Union (2018). supplementing directive (eu) 2015/2366 of the european parliament and of the council with regard to regulatory technical standards for strong customer authentication and common and secure open standards of communication.

[Fundamentals, 2010] Fundamentals, S. T. (2010). Software testing fundamentals.

[Garousi and Elberzhager, 2017] Garousi, V. and Elberzhager, F. (2017). Test automation: Not just for test execution.

[Hawkey, 2013] Hawkey, K. (2013). Overview of software processes.

[Huizinga and Kolawa, 2007] Huizinga, D. and Kolawa, A. (2007). *Automated defect prevention: Best practices in software management*. IEEE Computer Society Wiley-Interscience a John Wiley & Sons Inc. Publication, Hoboken, New Jersey.

[IEEE Standard, 1998] IEEE Standard (1998). Ieee standard for software test documentation.

[Inflectra, 2020] Inflectra (2020). What is api testing?

[Innopay DB, 2017] Innopay DB (2017). Psd2 open banking ecosystems.

[Interview1, 2020] Interview1 (2020). Interview 1 - vice president marketing / sales.

[Interview2, 2020] Interview2 (2020). Interview 2 - head of test and quality assurance department.

[Interview3, 2020] Interview3 (2020). Interview 3 - process analyst and project management, test management.

[Interview4, 2020] Interview4 (2020). Interview 4 - head of software engineering and application management.

[ISO, 2011] ISO (2011). Iso/iec 25010:2011.

[Jacobson, 2011] Jacobson, D. (2011). *APIs*. O'Reilly Media Inc, Sebastopol.

[JavaEE, 2014] JavaEE (2014). Introduction to json.

[Jeff Offutt, 2018] Jeff Offutt, P. A. (2018). Why do we test software?

[Kirchmann, 2017] Kirchmann, A. (2017). Overview of psd2 - related market initiatives.

[KPMG, 2019] KPMG (2019). Psd2 testing considerations.

[Kuang, 2018] Kuang, H. (2018). Api eg authentication.

[Kumar and Mishra, 2016] Kumar, D. and Mishra, K. K. (2016). The impacts of test automation on software's cost, quality and time to market. *Procedia Computer Science*, 79:8–15.

[Liliana Bobrica, 2002] Liliana Bobrica, E. N. (2002). Survey software architecture.

[Lingard, 2011] Lingard, B. (2011). Software testing.

[Lynn et al., 2019] Lynn, T., Mooney, J. G., Rosati, P., and Cummins, M. (2019). *Disrupting finance: FinTech and strategy in the 21st century*. Palgrave Pivot. Palgrave Macmillan, Cham.

[Mahajan et al., 2016] Mahajan, P., Shedge, H., and Patkar, U. (2016). Automation testing in software organization. *International Journal of Computer Applications Technology and Research*, 5(4):198–201.

[Mak, 2017] Mak, S. (2017). Modules vs. microservices.

[Marvin Zelkowitz, 2009] Marvin Zelkowitz, Manav Desai, T. (2009). Cmsc 435: Software engineering - software testing.

[Massé, 2012] Massé, M. (2012). *REST API design rulebook: Designing consistent RESTful Web Service Interfaces*. O'Reilly, Beijing.

[Mbama and Ezepue, 2018] Mbama, C. and Ezepue, P. (2018). Digital banking, customer experience and bank financial performance: Uk customers perceptions. *International Journal of Bank Marketing*, 36:00–00.

[Meng et al., 2018] Meng, M., Steinhardt, S., and Schubert, A. (2018). Application programming interface documentation: What do software developers want? *Journal of Technical Writing and Communication*, 48(3):295–330.

[Mohanty et al., 2017] Mohanty, H., Mohanty, J. R., and Balakrishnan, A., editors (2017). *Trends in Software Testing*. Springer Singapore, Singapore and s.l.

[MuleSoft, 2020] MuleSoft (2020). What is an api?

[Nahid Anwar & Susmita Kar, 2019] Nahid Anwar & Susmita Kar (2019). Review paper on various software testing techniques & strategies.

[Nguyen, 2008] Nguyen, C. D. (2008). Testing techniques for software agents.

[Nidhra, 2012] Nidhra, S. (2012). Black box and white box testing techniques - a literature review. *International Journal of Embedded Systems and Applications*, 2:29–50.

[NISP, 2019] NISP (2019). Nisp.

[Open Banking Limited, 2016] Open Banking Limited (2016). Open banking.

[OpenSource, 2020] OpenSource (2020). What are microservices?

[Ostrand and Balcer, 1988] Ostrand, T. J. and Balcer, M. J. (1988). The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, 31(6):676–686.

[Oxford Learners Dictionaries, 2018] Oxford Learners Dictionaries (2018). Api noun - oxford advanced learner's dictionary.

[Postman, 2020] Postman (2020). Postman - working with data files.

[PWC, 2018] PWC (2018). Customer experience and payment behaviours in the psd2.

[Qa, 2019] Qa, P. (2019). Testing objectives.

[Redhat, 2020] Redhat (2020). What are microservices?

[REST Assured, 2020a] REST Assured (2020a). Rest assured.

[REST Assured, 2020b] REST Assured (2020b). Rest assured github.

[Romanova et al., 2018] Romanova, I., Grima, S., Spiteri, J., and Kudinska, M. (2018). The payment services directive 2 and competitiveness: The perspective of european fintech companies.

[Ropota, 2011] Ropota, A. (2011). Automated testing.

[Scheler, 2018] Scheler, F. (2018). Anforderungsanalyse.

[Schwertner, 2019] Schwertner, C. (2019). In 6 monaten sind psd2 apis realität: Developer sind jetzt unsere hoffnung!

[Smith et al., 2016] Smith, A., Hoehn, T., Marsden, P., May, J., and Smith, E. (2016). Retail banking market investigation - final report.

[Soap UI, 2020] Soap UI (18.3.2020). Api testing guide: An automated approach to api testing transformation: Broadcom inc. | connecting everything.

[Soap UI, 2019] Soap UI (2019). Why your api testing goals are missing the mark.

[Software Testing, 2016] Software Testing (2016). Learn software testing.

[Software Testing Help, 2020] Software Testing Help (2020). When to stop testing (exit criteria in software testing).

[Spring, 2019] Spring (2019). Spring.

[SQA, 2001] SQA (2001). Iso9126 - software quality characteristics.

[Stepien et al., 2018] Stepien, B., Peyton, L., and Alhaj, M. (2018). Data-driven testing using ttcn-3.

[TestNG, 2019] TestNG (2019). Testng.

[Try Qa, 2018] Try Qa (2018). What are software testing objectives and purpose?

[Vijay, 2016] Vijay (30.5.2016). How data driven testing works (examples of qtp and selenium). *Vijay*.

[Warsi, 2016] Warsi, T. (2016). Why is it important to use regression testing?

[Wessing, 2017] Wessing, T. (2017). Security vs. convenience: strong customer authentication under psd2.

[Wiklund et al., 2017] Wiklund, K., Eldh, S., Sundmark, D., and Lundqvist, K. (2017). Impediments for software test automation: A systematic literature review. *Software Testing, Verification and Reliability*, 27(8):e1639.

[WSO2, 2019] WSO2 (2019). Strong customer authentication and dynamic linking for psd2: Strong customer authentication and dynamic linking for psd2.

[Wu et al., 2009] Wu, T., Wan, Y., Xi, Y., and Chen, C. (2009). Study on the automatic test framework based on three-tier data driven mechanism. In *2009 Eighth IEEE/ACIS International Conference on Computer and Information Science*, pages 996–1001.