TÉCNICO LISBOA

Secure Model of Web APIs development in the Cloud

João Miguel Correia de Sousa Silva Santos

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisor(s): Prof. José Manuel da Costa Alves Marques
Doutora Ana Margarida Seara Madeira

Examination Committee

Chairperson: Prof. Alberto Manuel Rodrigues da Silva
Supervisor: Prof. José Manuel da Costa Alves Marques
Member of the Committee: Prof. Miguel Filipe Leitão Pardal

November 2018

For my Parents, who were always there for me
with their unwavering support

Acknowledgments

I would like to thank Professor José Alves Marques for giving me the chance to work on this Project and being my advisor. Rui Rasteiro for all the technical help and other insights provided. Ana Madeira and Tomás Teixeira for their help and inputs provided.

I would also like to thank Professor Miguel Pardal for his feedback in regards to this Thesis.

A special and loving thanks to my parents, Maria and Augusto, and girlfriend, Barbara, for all their strength and support in the last couple of years. A special thanks to my friends Dina, Clara Rocha, Pedro Mendes, Francisco Santos, Manuel Silva, Diogo Pereira, Cândida Parente, Paulo Anjos, Aquilino Silva, Francisco Martins, Daniel Caramujo, Rui Miguel, Carina Martins, Diogo Ferreira, João Pedro, Sheng Wang, Débora Abreu, Eduardo Rodrigues for their friendship and companionship during this journey.

Resumo

As informações bancárias de clientes fazem parte dos tipos de dados mais sensíveis que existem, porque é aí que está indicado quanto dinheiro uma pessoa tem. Devido às necessidades decorrentes nos dias de hoje, surgiram alterações à Legislação Europeia, nomeadamente a Directiva de Serviços de Pagamento revista (PSD2) e Regulamento Geral sobre a Proteção de Dados (GDPR). O PSD2 veio com o intuito de forçar os bancos a exporem as suas Application Programming Interfaces (APIs), enquanto que o GDPR consiste em que toda a informação sensível relativamente ao utilizador só possa ser partilhada com o seu consentimento. Ao PSD2 está ainda associada um Norma Técnica de Regulamentação (RTS) para que haja um mínimo de segurança no que toca às interacções entre entidades, autenticação do utilizador e autorização dada às aplicações.

Como não há Normas definidas pelo PSD2, então pesquisar por soluções existentes é uma mais-valia. OpenBankProject e Apigee são duas entidades que já têm soluções implementadas e que fornecem o necessário para cumprirem com PSD2. Tendo obtido conhecimento, com base nestas soluções, é então possível implementar um conjunto de APIs que siga o mesmo caminho, desenvolvendo assim uma solução que cumpra com PSD2 relativamente ao produto de Internet Banking, BankOnBox. Para que seja possível providenciar os mecanismos de Autenticação e Autorização, implementa-se um servidor com base no protocolo OpenID Connect. Este protocolo demonstra como fornecer esses mecanismos para Autenticar o utilizador com o Banco, dar Autorização à aplicação para aceder a informações das contas do utilizador.

Palavras-chave: PSD2, RTS, API, BankOnBox, OpenID Connect

Abstract

Banks store important client information, making this one of the most sensitive types of information that currently exist. With the needs of today, certain legislature changes came into force. PSD2 forces Banks to be more open. letting interested third parties access client data. GDPR also plays a role in this, in which sensitive data is not shared unless consented to by the owner. PSD2's associated RTS mandates that some degree of security, authentication and authorisation exists in these changes.

To provide this openness, Banks and interested companies have been looking into possible secure solutions. No Standards to existing solutions have been available, which allowed core parameters to be implemented. Therefore, researching for other defined solutions was needed. OpenBankProject and Apigee are two entities that have already provided such PSD2 compliant solutions, making it easier and faster to implement a dedicated set of interfaces.

With this knowledge, developing a solution for providing PSD2 compliance in regards to BankOnBox, was possible. By implementing a set of APIs that would connect the BankOnBox's internal system and third-party applications, PSD2 compliance was half-way met. The other half required providing Security, Authentication and Authorisation mechanisms. For this part, making use of the OpenID Connect protocol was a necessity. This protocol provides Access Tokens to the applications and reliably authenticates the End User with the Bank, in turn confirming with the Application that this was done correctly.

Keywords: PSD2, RTS, API, BankOnBox, OpenID Connect

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The world is ever-changing and in technological advancement. An example of this is mobile devices, which can be used to access the internet. Mobile phones also have a range of applications, such as agendas that allow scheduling events, and Bank applications that allow access to Bank Account details. Not only has this affected every aspect of our daily lives, but that of businesses too. The Banking Industry, in particular, is no exception, which can be noticed by the alterations to the European Legislation, as well as in the U.K. In the last couple of years, Internet Banking has been the main target.

There was a time when Banks would keep all the data and information regarding their customers, which could only be accessed via the bank's on-line portal. However, this is about to change with new legislative rules, namely Second Payment Services Directive (PSD2) in the E.U. and Open Banking in the UK. PSD2 and Open Banking are both aimed towards the same core goal, i.e. to force banks to be more open. Banks being more open means that bank customers information can be accessed by interested and regulated third parties in an on-line and distributed environment. This means that Banks are forced to expose at least a handful of specific interfaces, also called Application Programming Interface (API), to these interested parties. Nevertheless, such access to the bank customers information **must** be consented by said customers and **should not** be accessible otherwise. This last sentence brings into attention another piece of European Legislature, General Data Protection Regulation (GDPR), concerning the protection of customer information and sensitive data, allowing the users to have the "right to be forgotten" and also user consent acceptance to share their information with third parties. This works quite well in conjunction with PSD2 because even though it is about forcing banks to open themselves and ease the access to bank account information, it is also deemed that

no sensitive information (e.g., real bank account number, address, phone number, etc.) can be shared without consent. Which means that GDPR and PSD2 align together nicely, bringing a better sense of security to the customers regarding the Bank's systems. Between these Legislation changes, PSD2 will be the main focus of this Thesis.

Providing customers with a better sense of security means that there is a need to analyze and check how the legislation might affect certain bank systems. It is also important to determine what needs to be changed, in the bank systems, for them to be compliant in regards to PSD2. So far, Banks have mostly used their own Internet Banking platforms or had companies like Link Consulting to develop and provide such products. But in light with recent alterations to the European legislation, it made all of the involved parties to be interested and motivated on changing and improving their product(s) to be fully compliant with PSD2.

The underlying motivation in all of this is to expose the Internet Banking product's APIs in a secure fashion and perform the needed adjustments to these Internet Banking products. These adjustments can range from implementing new components in the system to modifying how the existing components interact with one another and with the outside of the system's boundaries. In light of PSD2, these adjustments are about exposing the Internet Banking product's interfaces to provide, for the regulated third parties, authorized access to the customer's bank account information.

The rest of this document is structured as follows: In Chapter 2 analyzes of the legislations, APIs, BankOnBox is provided; Chapter 3 describes the solution's architecture, its components and important decisions taken; Chapter 4 shows a more specific implementation of each element of the architecture and how they connect, as well as particular implementation decisions taken; Chapter 5 presents an evaluation of the system and conclusions drawn from it, e.g. its efficiency, reliability; Lastly, Chapter 6 presents the conclusions in regards to this Thesis, problems found and future work to be done.

1.2 Objectives

The objective of this Thesis can be divided into smaller sub-objectives that fulfill the intended goal, which is to make use of the needed technologies to establish an Internet Banking product that is fully PSD2 compliant. In regards to this Thesis, the Internet Banking product to be focused on is the Bank On Box, a proprietary product of Link Consulting.

The smaller objectives to be fulfilled are as follow:

- The first objective is to analyze PSD2 to get to know what this legislation is about, what changes it brings to the table, if there are any specific Standards or Frameworks associated

with it that were defined by the governing bodies for the Banks and other companies in the industry to follow. This includes Security-related specifications in regards to sharing customer information over an internet connection.

- The second objective is to analyze the Bank On Box product to get to know how it works, the models of the APIs, exchanged information (request-response messages), and so on.
- After the two previous analyzes, a gap analysis between Bank On Box and PSD2 is to be done in order to ascertain what APIs need to be developed to connect with the Bank On Box interfaces to have a PSD2 compliant Internet Banking product.
- Research and find any necessary technology to help provide and implement a feasible solution.
- Propose and implement necessary changes to the system that is currently in place in order to provide PSD2 compliance as well as taking advantage of the researched technologies.

To fulfill these objectives, part of the research falls upon API Management and Security, as well as searching for already existing solutions and gathering well-formed opinions and ideas to better present a solution to the problem at hand.

Chapter 2

Background

The experience provided by the Financial Technology (FinTech) community around this subject, introduced in Chapter 1, had a significant impact on the conclusions drawn. By gathering these trusted opinions, research and projects already conducted by others, made it possible to get a considerable understanding about Open Banking, PSD2, the FinTech industry, API Management and Security, as well as ongoing projects regarding the subject in question.

To begin with, there was the need to understand what Open Banking and PSD2 are, where they come from and what they entail. Both of these legislation changes aim to bring about disruption and innovation into both the U.K. and the European Banking industries for Banks to expose their APIs and share customer data. The sharing of customer bank information is nothing new, though, considering that changes similar to this were already happening around the rest of the world, for example a handful of Banks in the U.S.A. have allowed for certain Personal Financial Management (PFM) and other data aggregation applications (e.g., Mint [1]) to access customer data. This part of the research will be talked about in more detail in Section 2.1.

To be able to understand more about the problem at hand, there was also the need to understand better the type of technology needed to deal with in regards to the Banks, Third Party Provider (TPP) and any existing APIs that were developed in order to better co-exist in a more API- and mobile-centric world. APIs are an important part of this work, considering that the openness of Banks, application development, TPPs, and even other businesses, are revolving more around APIs and the security involved in the communication. Section 2.2 covers details on APIs and how to provide security for it.

Since APIs can deal with numerous use cases and situations (e.g. from geolocation to accessing bank account information), the need for certain security requirements arose. Some of these situations are more sensitive than others in regards to the information that is passed during communication with the servers (e.g. bank account information), which leads to having better

defined security requirements to access said information. This involves Authenticating the End User, to be sure that there is no one attempting to maliciously obtain his information, as well as Authorisation, in which the end user permits someone or something (e.g. mobile application) else to access his information on his behalf. This is important when it comes to APIs and will be explained in Section 2.3.

Subsequently there was the need to understand how to connect these technologies into the ubiquity of today in regards to an API Management Platform and APIs of a proprietary Internet Banking product, called Bank On Box. From there, arose the need for an analysis of what the BankOnBox (BOB)[1] product is and how it works, what are its APIs, models and information that is passed from, and to, the caller. It is also important to mention that BOB is installed on bank premises, making each bank that uses it to be inside the scope of this work.

Finding a suitable API Management Platform, to provide openness, was needed. This led to checking for a Cloud Service Provider (CSP) that could offer such a service, along with other potential services that could be integrated with the Platform to help provide a different degree of innovation. It also came into consideration the fact that CSPs have been increasing in demand due to their services portfolio providing from Virtual Machines to Data Storage, Machine Learning to Chatbots. It has turned into a myriad of choices and possibilities from which to choose from, which is quite important considering that these CSPs also provide an API Management Platform. This means that it is easier to connect this platform to other services from their own portfolio, since there is the backing support of the company and the community that uses these services. In the end, it led to choosing a CSP to help provide, at least, the required openness via the API Management Platform.

2.1 Open Banking and PSD2

As was mentioned in the introduction of this Chapter, there is a need to understand Open Banking and PSD2, since these are recently passed legislations that have come into force, although still being underway. The due date for companies within the E.U. to comply is September 2019, 18 months after the Regulatory Technical Standard (RTS) in regards to PSD2 was published [2]. The main focus of this Thesis therefore remains PSD2, because it concerns Banks in, or that provide services within, the E.U.

Open Banking is, at times, not used to reference the change in U.K. legislation regarding the Banking industry, but instead, the concept of "Open Banking", i.e. the notion of Banks being more open by providing open APIs that enable interested third-party developers in the

[1] `http://www.linkconsulting.com/bankonbox/en/`

FinTech industry to provide services and products of their own. This is something that Banks have been reluctant to do for years, since customer banking information is deemed to be part of the most sensitive there is. But with PSD2 Banks seem to finally realize that to reach a broader audience more efficiently, they must look at exposing their APIs and letting third parties make use of it to provide new services and applications to the customers. A good example of this is the Mint application used in U.S.A. and Canada, since circa 2006, which nowadays lets its users access information about their bank accounts. This can range from 1 to many: from various banks making it function as an aggregator of banking information, as well as providing various functionalities, such as permitting payments, budgeting and transferring money from one bank account to another. However, this is not done by using APIs, but rather by using a method called "screen scraping", also known as "Direct Access" (this method will be talked about in Section 2.2). This has not been done only by TPPs in America but in Europe as well.

As a means to effectively accelerate these changes in Europe, the European Commission decided to review its Payment Services Directive (PSD) and modernize it by bringing about a new package of legislative measures called PSD2. Right around the same time, the Competition and Markets Authority (CMA) [3]], in the U.K., conducted a review about the Banks concerning how open and innovative they were regarding their services and products. After getting the results, the CMA concluded that pushing for an Open Banking Standard (OBS) was needed.

Regarding European legislations on the matter, there is more to it than just PSD2, there is also another set of important regulations called GDPR [4] which demands the need to provide customers with information about how their data is saved, where it is processed and by who, and to securely protect it. Not to mention if the customer allows this data to be shared with another entity (a third party) then the information provider must be able to provide the customer with information regarding on who has access to customer data.

In the end this is what Open Banking is all about, Banks opening up to the world and letting interested third-party developers make use of their APIs to improve the offers of their services and competitiveness of the industry. This being done in a way to achieve an easier financial transparency and being more cost efficient at the same time while still being able to provide confidentiality of the customer's data. Of course that in the end these Banks will also have to choose if they let third parties take the advantage or if they decide to pursue to be on the vanguard of it as well, i.e., not only comply with the upcoming changes but also transform themselves to be competitors in the playing field.

2.1.1 OBS

Before delving into PSD2, getting to know what OBS was about was also important, since in the U.K. it was not only meant for Banks to be OBS compliant but also because they would need to provide PSD2 compliance as well.

As was mentioned before, there has been some work being done in the U.K. called OBS. It came to exist after the CMA reviewed the state of Small and Medium Enterprises (SME) and Personal Current Account (PCA) services provided by the banks. After noticing that the big banks in the U.K. were not competitive enough it led to a decision from the CMA to call for an Open Banking revolution, leading to the creation of the Open Banking Working Group (OBWG) at the request of Her Majesty's Treasury. This group brought together banking and financial industry experts to develop the first OBS framework which would guide how data should be created and used [5][6][7].

OBS is part of the reforms that the CMA envisioned to implement to enable customers, in case they consented, to share their bank data securely with third parties through these standardized APIs. This reform also dictated that the nine largest banks in the U.K. (afterwards called CMA9) would build and fund an Open Banking Implementation Entity (OBIE) to take care of all the needed steps regarding the APIs development and publishing broadly and inclusively through consultation with various parties (challenger banks, FinTech companies, consumer groups).

Despite OBS and PSD2 being different, there are requirements from the CMA to align OBS with PSD2 in such a way that makes it easier for U.K. banks to also be compliant with PSD2 regulations whenever there is an overlap in scope. To help and ease the convergence from OBS towards PSD2 a Stakeholder Group was created, this group aims to help prevent fragmentation and to work in a more widespread manner with the industry on the practical implementations of PSD2. While PSD2 was already being taken care of and towards better discussions, OBS was still in its early stages of creation [8][9].

2.1.2 PSD2

PSD2 is the revised Payment Services Directive which came to fruition due to the needs of changing, more rapidly, how Banks act and react towards its customers regarding their services and products. Before PSD2 there was the PSD, which came in late 2007 and became law in 2009, with the idea to make easier, more efficient and secure payments in the E.U. [10]. The intentions behind PSD2 come from the basis of PSD and to force a more homogeneous and competitive market in the Banking industry by bringing about this new set of legislations to

accelerate the openness of Banks towards European citizens, markets and information they held.

This came to light as a way to accelerate Banks to open themselves such that it lets other interested parties develop applications that make use of the Banks APIs with all due confidentiality in the exchange and supply of all personal (customer's), payments, transactions, etc. information, at the same time guaranteeing the integrity of the same [information].

With PSD2 there is a relatively big change in how payments and transactions are made. To begin with there are three roles in the participating entities: Payment Initiation Service Providers (PISP), to initiate payments or transactions from the bank to the designated destination, i.e. initiate the transfer of funds on behalf of the customer with his consent; Account Information Service Providers (AISP) to provide information regarding a customer's bank account with due consent from said customer; Account Servicing Payment Service Provider (ASPSP), which are commonly referred to as Banks at the moment. Any Institution, Bank or FinTech company can turn out to have one or both of these roles as long as they are regulated and there is compliance with PSD2.

Afterwards [executives of] Banks need to think on how and what they want to pursue concerning PSD2. There are four strategical options [11]] to choose from, as can be seen in Figure 2.1, in which banks can evolve from one to the other at any time, only depending on the bank's choices regarding its positioning and service portfolio:

1. Comply: The banks that choose only to comply will have to open up the most as they can and let third parties provide the Payment Initiation Service (PIS) and Account Information Service (AIS). This implies that these banks will need to directly cooperate with the third parties to ensure PSD2 compliance and security rules are followed, not to mention that they will eventually be surpassed by others [Banks].

2. Compete: This is similar to the above, but banks decide to also go on the offensive by offering innovation in their PISs and AISs, i.e. these banks choose to have both roles. This implicates that the banks will compete, head-on, for customer relevance against other banks and other third parties (whether FinTech or not). It is also a bit easier for the banks to choose this option since they already have most of what they need in place, just needing to innovate a bit in such a way that they have both roles of PIS and AIS and stay competitive in the market.

3. Expand: If banks choose to expand, they choose not only to ensure compliance with PSD2 but also increase their portfolio of services to provide to their customers via their APIs, services that go beyond PSD2 compliance to bring more competitiveness to the market.

But there is a catch, the banks that choose this option are relinquishing the roles of being PIS or AIS, i.e. the only thing the banks do is provide an increased portfolio of services for the interested third parties (whether FinTech or not) to make use of.

4. Transform: Transformation is simply a fusion of the other three options. Its purpose is to comply with PSD2, extend the services portfolio and at the same time provide their customers with their own and dedicated application and Open APIs, competing as PISs and AISs against others and still collaborate with them to improve their own services and be able to extend as far as they can in the industry. For that they will, of course, have to use Open APIs and collaborate with other banks, third parties (whether FinTech or not), developer communities, etc., as a way to increase adoption and market growth of their services.

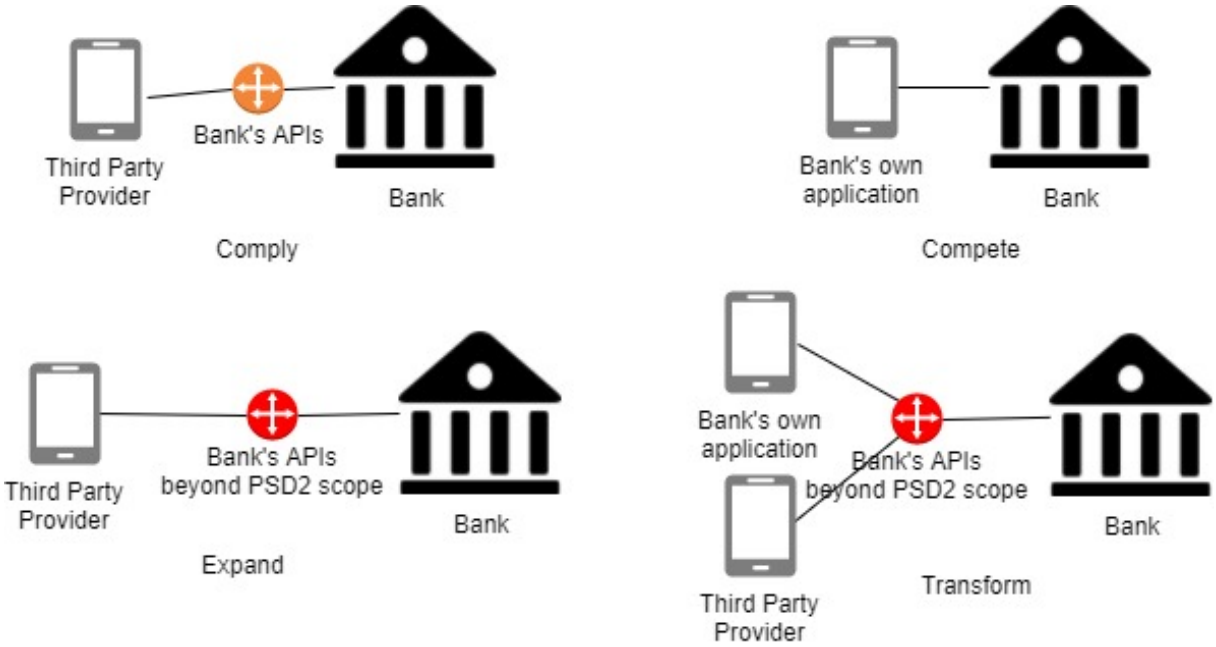

Figure 2.1: The 4 options banks can take

Although there are these four options, for any of those to happen banks will need to comply with PSD2 regulations and have strong security mechanisms in place to provide confidentiality, authentication, integrity of the information and control to the customer regarding his information and consented sharing of it.

The security requirements that need to be met, were being defined by European Banking Authority (EBA) in a collection of documents called RTS, the one most pertaining to the context of this project is related to Strong Customer Authentication (SCA) and Common and Secure Communication (CSC) [12]. This particular RTS specifies technical security conditions, except that it does not do so in any detail, but it is done less extensively, i.e. the specifications presented

in the document do not specify exactly what to use or how to use it. They [specifications] are a bit generalized, in which it just says, for example, "shall ensure that interfaces follow standards of communication which are issued by international or European standardization organizations". Although, there is the case of one specification that provides more concrete conditions. This specification requires the user to provide authentication before the Bank with at least two out of three of the following categories: Knowledge, something the user knows (e.g. PIN code); Possession, something the user possesses (e.g. OTP SMS Token); Inherence, something that the user is (e.g. Fingerprint).

While still on the note of the RTS and its specification, it is also of significance that after the acceptance of the PSD2 proposal, it has been a concern for interested TPPs on whether they could continue to use the "screen scraping" method or if things would change. This concern grew even more after an initial draft of the RTS in regards to CSC and SCA was published, since it meant that the method of "screen scraping" would be, henceforth, unusable after RTS came into force. A Manifesto, undersigned by 77 companies and associations, was released after this draft, due to the mentioned concerns [13], which turned out to be, in some way, successful, considering that there was a change in the RTS that provided a "fallback solution", stating that "Payment Service Provider (PSP)s shall be allowed to make use of the interfaces made available to the Users for authentication and communication with their Bank until the dedicated interface is restored to the level of availability and performance provided for in Article 32." [12], and also that "Competent authorities (...) shall exempt the ASPSPs that have opted for a dedicated interface from the obligation to set up the contingency mechanism described under Paragraph 4 (...)". Although it was not quite the intended resolution they wanted, it was still accepted.

Even though the specification of the RTS document is not thoroughly specific, upon further analysis, certain requirements, when it comes to security and functionality, should be met regardless:

- Authentication - Able to authenticate both the Application that is making the requests and the End User.
- Confidentiality - Non-disclosure of any information with unintended parties, i.e. only the End User and authorized third parties may view the information.
- Integrity - Every bit of information that is either transmitted or kept saved does not change, keeping it accurate and consistent.
- Authorisation - End user permitting regulated third parties to access their bank account information (consent), and in case no permission was given then the application cannot

see and/or do anything in regards to that bank account.

- Login - End User being able to authenticate itself towards the Bank by logging in with its Internet Banking credentials.
- Consent - End User choosing which of his Bank Accounts the Application can have access to and what it can do.
- Access to Bank Account - Providing Bank Account access for the Applications, thus being able to get bank account information details or initiating direct payments between the user's bank account and the intended party.

2.1.3 GDPR

After mentioning the requirements that should be met, it can be seen that these go along with the core requirements of the GDPR regulations that are already in force, therefore complementing with PSD2 requirements.

The GDPR defines rules related to processing and movement of personal data, and according to it personal data cannot be kept in such a way that can identify the subject, to which it pertains to, for no longer than which the data is processed for [4]. Although it can be stored for longer periods of time if it is to be used for public interest, scientific research or statistical purposes, but in the end this data still needs to be processed in a properly secured manner.

The user to whom this data pertains to has the right to access, remove, correct and identify any entity which has, or had, access to their data. This is also part of the "right to be forgotten" rule imposed by the GDPR regulations. This comes to show that, even if a user consents a third party to have access to certain information, such consent can always be changed in a later occasion or even removed entirely, thus un-authorising access to his data.

From this presented information it was possible to note that it is what aligns with the requirements of PSD2 as well. The need to authorise third parties to have access to information of the user (Consent and Authorisation) and keeping this data securely guarded from any malicious intents (Confidentiality and Integrity).

2.2 API – Application Programming Interface

PSD2 regulations ask for the development of dedicated interfaces that TPPs can use to access user's bank account information. This means that what is needed is to develop a set of APIs that meet with the intended requirements set forth by the legislation. API usage has gained more traction over the years [14] and in certain cases some APIs are more dominant than others

(e.g. Google Maps [15]). This means that the interfaces to be developed for PSD2 compliance will most likely turn out to be dominant in the FinTech industry.

There are different types of APIs though, but in the end they all serve the same core function, to be no more than an interface for server-to-server, or application-to-server, communication through a well-defined contract between the two entities, the provider and the consumer. Even so, the contract is still valid for multiple consumers, since the provider is the point of origin of said APIs. This contract serves as a means to find a way for these entities to interact and understand one another by using certain protocols, models and definitions (input/output message format, headers, etc.) on the exposure of services from the provider to be more easily consumed by the consumers, and can also include Terms of Service (ToS) and Service Level Agreement (SLA). These contracts should always stay constant even if there are changes to the underlying implementation of the APIs, making them more reliable over time.

Since the interfaces that are asked to be developed per PSD2 legislation are to be used in an on-line environment, it can be pinpointed to a specific type: Web APIs. This is a subset of APIs that is defined to work on top of HTTP request-reply message exchange. Web APIs, nowadays, allow for easier and better exposure of services to be consumed remotely via Hypertext Transfer Protocol (HTTP) or Hypertext Transfer Protocol Secure (HTTPS). Since the need for APIs has been growing [16], it has turned into more of a necessity in order for developers to be able to build applications. Developers that are interested in using a certain set of APIs tend to first look for a well-defined contract, making them more or less likely to use the APIs depending on the contract's definition.

With the passing of time, not only have Web APIs seen more use, but developers have also been looking towards using an easier and more light-weight solution within the Web APIs communication. Eventually this led to the appearance and growth of Representational State Transfer (REST), with JavaScript Object Notation (JSON) being used in conjunction with REST to be lighter, and since it is more simple to use, it has better performance and reduced network traffic [17]. Still within this subject, different ways have been found on how to secure Web APIs, whether during information exchange or while protecting access to the APIs and information.

With this introduction on APIs it will henceforth be explained, in regards to the context of this work, the most important parts that complement one another.

2.2.1 API Management

Every API needs to be managed to more easily and efficiently design, maintain and publish them after being developed. This leads to the need of an API Management Platform, which provides a framework that allows the two mentioned traits as well as traffic management, documentation, security, etc. This allows everything that is API related, to be seen in a secure and scalable environment. The usage of a management platform has only increased over time with the growth of API usage,i.e. the more businesses depend on APIs, the more there is a need for a dedicated management platform.

There are three core parts to an API Management Platform:

1. Gateway - deals with creation and management of APIs, how to manage the traffic, security, policies, etc. by providing services that form the core for API communications between the applications, APIs and back-end services.

2. Developer Portal - takes care of the provisioning of well-defined documentation, application registration to use the APIs. This serves as a portal for developers to get to know the catalog and also capable of providing a testing playground.

3. Analytics - provides metrics related to performance, operations, throughput bandwidth, etc. by monitoring all the traffic concerning the APIs

The Gateway is one of the most important parts of an API Management Platform, because it helps on protecting the back-end server by being an extra layer for the incoming requests. By using a Management Platform, these requests can be verified for specific information (e.g. Headers), or to limit how many requests can be made per second. After having performed the needed verifications, either a response is immediately generated at the Management Platform or the request is sent to the back-end for processing. When the back-end wants to send an answer back, these responses go through the Management Platform as well and can also be verified before being passed through to the original requester.

With this, a Management Platform provides developers with mechanisms to ease and fasten the publishing of APIs. By having this extra layer towards where requests are made, instead of directly to the back-end, helps to provide scalability and reliability to the system. This is due to being able to evaluate the incoming requests before being passed on to the back-end (in case no errors were encountered) and also limit traffic as needed. Other features provided by this Management Platform will be discussed in the remainder of this Section.

2.2.2 API Documentation

The requests and response messages mentioned, need to be documented so that developers know how to build the request messages and what answers can be expected to be returned. This leads to another part of APIs and their management, the documentation. Documentation of an API helps developers in many different ways, from understanding how the calls work, what parameters are needed and even tutorials to help guide developers in the right direction.

Since PSD2 asks for a dedicated interface, this means that it must be well documented and explained too, whether there are existing standards or not, the documentation behind the provided solution has to be well formed regardless. This documentation has to be easy and simple to read, user-friendly, provide examples and show how it complies with PSD2 requirements.

This is why good documentation must be able to provide all the necessary information about the usage of an API by including various aspects of it, such as:

- Endpoints: the URL that applications need to use to make calls.
- Method: defines the HTTP verbs to use when accessing the APIs.
- URL Parameters: defining parameter names and their formats.
- Message Payload: specifying the structure and format of the request and response messages.
- Header Parameters: specifying standard and custom headers included in the request and response messages.
- Response Code: is the response code that the API sends back to the client (e.g. status code 200=OK, 400=Bad Request, 404=Not Found).
- Error Codes and Responses: response status codes which are considered errors.
- Sample calls: include samples with pre-defined parameters and mocked responses.
- Tutorials: includes examples to show various ways how the APIs can be called and expected responses.

To be able to provide this documentation, one can use the Management Platform tools, but there are also some out-of-the-box tools and services provided by dedicated developers. These tools help on synchronizing the documentation with the developed APIs. This is helpful, because any change that is done to an interface is henceforth shown in the provided documentation as

well, e.g. if a parameter in a request message had its name changed or removed, it would be shown in the documentation so that all developers can see it.

One of the most popular and widely used documentation tools is Swagger [18], it provides a framework to document APIs in a language-agnostic way, making it easier to define a REST interface and thus allowing the client to fully understand the services to be used without prior access. The main goal of this tool is to keep synchronization between the involved parts (client, documentation and implementation). It uses what is called OpenAPI specification, usually written in JSON format and with a pre-defined structure, for its documentation, which is aimed mostly towards RESTful APIs. It also has its own User Interface (UI) to display the interfaces in a more user-friendly manner.

Another tool that provides documentation synchronization is RAML, RESTful API Markup Language, used to ease the management of an APIs entire lifecycle, beginning at the design phase and ending with publishing, by being used to design developer- and user-friendly APIs. It also provides a UI to display the interfaces.

2.2.3 RESTful API

Considering that nowadays APIs are developed and used widely by mobile applications, there is the need for communications to be lighter and faster. With this increasing need came REST, and as *Roy Thomas Fielding* put it in his dissertation about REST, "REST emphasizes scalability of component interactions (...) to reduce interaction latency, enforce security and encapsulate legacy systems" [19]. This leads to the perception, and reality of today, of an easier to use architectural style that works on top of HTTP/HTTPS using the already existing provisioned guidelines and constraints that must be followed to the letter, thus ensuring a scalable, fault-tolerant and easily extendible system.

Since the communications are made through a uniform interface, it means that each request to the server must, clearly and concisely:

- Identify the resource it is requesting, for the server to more easily understand what resource it must be looking for. This is done easily by using a Uniform Resource Identifier (URI), e.g. http://example.com/v2/banks identifies a resource by the name "banks"

- Manipulate the resource the client wants to modify by presenting a representation of the resource instead of the resource in question, this way it is easier to communicate with the resources

- Self-descriptive messages, such that both the client and the server know if there is any

additional information to process and in case there is, how to do it.

To effectively use REST, one must also abide by the conventions, which lead to an easier way to understand what it is the requester wants to do. This involves naming resources with nouns instead of verbs, and not use CRUD (Create, Read, Update, Delete) function names (e.g. "getBanks"), so that the resource being referred to is an item and not an action. These resources can be a single object or a collection of objects, and it is custom to follow a hierarchy in such a way that the URI path can be modelled more easily and made less confusing, thus playing an important role in making and API easier to read (e.g. "/getBanks/Bank-ID/getAccount").

Apart from this, and since REST is used on top of HTTP/HTTPS, it must also abide by the verbs POST, GET, PUT and DELETE. These verbs identify the type of action that is to be performed in the server when a request is made, in regards to the resource contained in the URI. After the request is processed, a response is given with the corresponding message payload dependent on the type of action that was performed, which lets the requester know in which state the resource is currently at as well.

2.2.4 API Security

To make the APIs even more reliable the access to them needs to be secure and for that they need to meet a certain set of requirements in which these can vary on how secure they are depending on the value of the data to be shared, cost vs. risk, etc.

The main considerations that must be carried out concerning the security of APIs are:

- Confidentiality and Integrity of exchanged messages by means of a secure communication channel.
- Authenticating the end users, when required.
- Authorising applications to access user's information, but only with user consent.
- Making sure that the application making requests is trustworthy, i.e. that it has access to make the requests by means of a provided API key.
- Confidentiality and Integrity of the underlying data and assets.

Starting with the first item, the way to provide confidentiality and integrity during the exchange of messages, in the communications channel, is by using HTTPS, since it has built-in mechanisms to provide these requirements as well as server-side authentication. At times, client-side authentication might also be requested, which leads to what is called "Mutual Authentication". Using HTTPS is a necessity nowadays since it protects against eavesdropping and

tampering with the communication, providing additional reassurance that the communication is happening with the intended server and not an impostor and hence without the interference of malicious third parties. This is done by using Secure Sockets Layer (SSL)/Transport Layer Security (TLS), with the former being the predecessor of the latter. With SSL being deprecated, TLS is considered more secure, and thus capable of providing the needed security for the communication. Nevertheless, SSL/TLS is used in the beginning of the communication, during what is called "the handshake"; to agree on a shared secret and type of encryption that is going to be used during the exchange of messages. It is during this handshake that a client application verifies the identity of the server by validating its certificate, and for the server to verify the client application in case it shares its own (optional decision implemented in the server-side).

Authentication and Authorisation are extremely important, not only when it comes to client<->server communication but API security in general. API keys, for example, since they are unique, are used to identify which client application made calls to the APIs. This also means that the application can be denied access if a key was not provided. Access may also be denied when a key was provided but its access was restricted to a handful of APIs instead of the whole set. Such restriction is usually done by using "Policies" that help on restricting access in many different ways (Section 2.2.5).

Another type of Authentication used is via **username** and **password**, which has been a widely used method for many years. In this case, these two fields can be passed in a HTTP header parameter called "Authorization", with the designated type of "Basic" (e.g. Authorization: Basic QWxhZGRpbjpPcGVuU2VzYW1l). Nevertheless, there is no security involved, since this type of authentication is not encrypted or hashed, but rather just encoded in Base64, which is why to provide any sort of protection it should be used in conjunction with HTTPS. Another type of authorisation, known as "Bearer", has grown over the last years. "Bearer" pertains to the usage of Tokens and is associated with an authorisation framework called OAuth2.0 [20].

OAuth2.0 [21] is an Authorisation framework aimed at providing tokens to applications that were consented to do certain actions by the end user in regards to a certain service, for example connecting a mobile application with Facebook and permitting it to access the list of friends. When received by the server these tokens are checked to see if the calling application has permissions to do what it is asking for with consent of the user. This, however, does not authenticate the end user, which is why an Identity Layer that uses OAuth2.0 exists, namely OpenID Connect [22]. This OIDC is a protocol used to enable clients to authenticate that the user is who he says he is when requesting initial access and permissions to be consented for in regards to a service's account. More about how these work will be covered in Section 2.3

Something that needs to be mentioned in regards to API security is that unlike what has been happening recently in Europe, Banks in the U.S.A. have allowed PFMs to access customer's information. The problem here does not lie in information sharing, but in the method used [23]. Although the PSD2 legislation envisions and forces the usage of APIs and the underlying security to be provided to reach the desired outcome, in the U.S.A. the method used is called "screen scraping", or "Direct Access", which has been deemed to be sketchy and un-secure [24] since it forces the customers to share their login credential with the third-party applications. This can lead to an increase in successful phishing attacks, identity theft, etc. This method also forces the applications to use the user's login credential to log into each Bank every time the application needs, or wants, to update the data, over time and with a large number of users this brings more bandwidth expenditure and can overload the Bank's servers [23].

2.2.5 API Policies

Apart from security, and this also helps when it comes to safety, there are API Policies that can be used to help on further protecting and restricting the access to the APIs by providing additional features in the API Management Platform. This is a powerful capability that allows changing the behaviour of the interfaces through a simple configuration document describing, in sequence, the statements to be executed in each part of the communication. I.e., it executes the defined statements for the request message, when it passes the message to the back-end, when it sends back a response to the caller and in case there was an error while executing these statements. Policies come as a way to complement the security features that are developed to protect the APIs.

There are different types of Policies and these range from simply limiting the call rate of an API to message transformation, i.e. receiving a request message in JSON and converting it to its eXtensible Markup Language (XML) equivalent. The different sets of existing Policies, and what they are concerned about are as follows:

- "Access Restriction Policies" – This set is used to: limit the call rate of an API depending on the application; Filter calling specific IP addresses; Validation of JSON Web Token (JWT); Verification of HTTP header existence and/or value.

- "Advanced Policies" – This set is a bit more complex and can: Send requests to specified Uniform Resource Locator (URL)s without waiting for a response; Forward the requests directly to the back-end; Aborting the pipeline of statements and immediately return a response; Changing the HTTP method of the request; Logging a message in a specified format to a Logger entity.

- "Authentication Policies" – Authentication with a back-end service by using Basic or Certificate authentication.
- "Caching Policies" – Cache storage of response messages to certain requests if needed, therefore using the cached values afterwards in case it is needed.
- "Cross Domain Policies" – Allows for cross-domain calls, to provide flexibility for browser-based client to fetch resources deriving from a different origin than from where the original response comes from.
- "Transformation Policies" – This set performs specific transformations to the request/ response message, either by converting the body from JSON to XML, masking URLs in the body or even changing/adding a specific HTTP header.

These sets of Policies help on improving security, reliability and functionality of the overall system, which is a convenient addition to the Management Platform when it comes to managing the APIs after they have been deployed.

2.3 Authorisation and Authentication

As was mentioned in Section 2.2.4, Authorisation and Authentication are extremely important when it comes to safeguarding an API and the user's data. Which is why, over time there has been growth in the use of technologies to better provide users with such features.

Authentication is about obtaining identification credentials, such as **username** and **password** from a user and validating those credentials against the authority. In case these are valid, then the entity that submitted the credentials is considered authenticated, allowing an authorisation process to determine whether that identity has access to given resources or not. In the case of PSD2, once a user is authenticated, the authorisation process determines which permissions the target application will have.

Authorisation is, as just mentioned, about obtaining a consented access to the user's information via the APIs. To be able to do so, the application needs to pass an object which contains information of the consent that the user gave (e.g. an Access Token).

Part of these two properties are OAuth2.0 and OIDC to provide the needed assurances to the users. As was mentioned in Section 2.2.4, OAuth2.0 is an authorisation framework, while OIDC is an Identity Layer on top of the OAuth2.0 framework. While OAuth is only concerned with providing authorisation to resources that the user consented to, OIDC also ensures that the end user is authenticated with the server and application.

Both work by enabling third-party applications to obtain limited access to a service by means of an approval interaction between the owner of the resource (called end user) and the service. There are different entities involved in this interaction, and their names and definitions also change a little when either talking about OAuth or OIDC. To begin with, first a description of how OAuth works and then about how OIDC works on top of OAuth.

2.3.1 OAuth2

The entities involved in the OAuth interactions are: Resource Owner, or end user, which is who decides to authorise the Client to have access to his information that is saved in the Resource Provider; Client, or application, which is the entity that wants to gain access to the Resource Owner's information; Resource Provider, which is the server where all the information is securely stored at, and depending on the scope of authorisation granted to the Client it can provide information needed when asked for it; Authorisation Server, which is on the path of the validation of the Resource Owner and issues a Token to be used by the Client for accessing the information in the Resource Provider.

There are different implementations for authorising a Client to access the information in this framework. Each of these types is used for more specific situations, and they are:

- Authorisation Code Grant - After the Resource Owner logs in the Authorisation Server and gives consent to the Client, an authorisation code is passed on to the Client, which will afterwards be exchanged for an access token.

- Implicit Grant - Simplified Authorised Code, since it does not get a code to exchange, but rather receives the access token directly after consent was given.

- Password Grant - Involves the Client asking the user for their username and password, presenting it to the Authorisation Server and obtain an access token in exchange.

- Client Credentials Grant - When there is no user involved in the interaction, the Client obtains an access token by presenting its Credentials. Typically used to access resources about the Client itself.

Considering the aims of PSD2, the Authorisation Code Grant is deemed the most secure of the four. Not only because of the grant having an extra step to exchange a code for an access token, but also because it requires that the application first launches a browser window to begin the flow. Moreover, it has been considered best practice to use Authorisation Code Grant without using the Client secret, or to use an extension of OAuth2.0 called Proof Key for

Code Exchange (PKCE) [25], which is a technique to secure public Clients that do not use the client secret.

Before explaining what PKCE [25] is, here is an example of the OAuth authorization flow that can be seen in the Figures 2.2, 2.3, 2.4 and 2.5 along with a brief description. This was based in the Open Bank Project [26] use of the OAuth2 implementation.

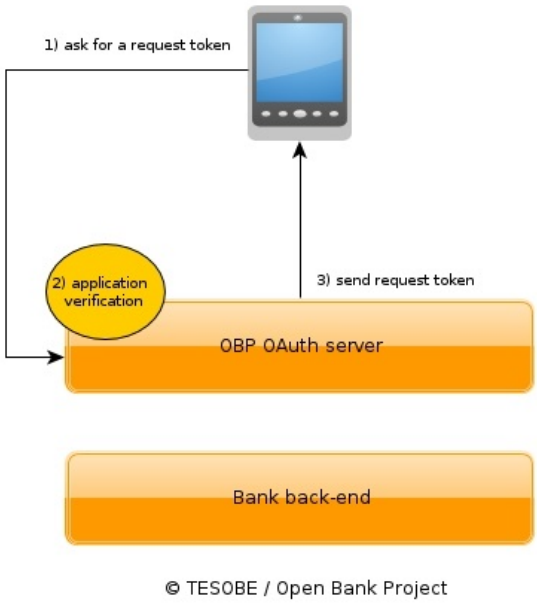

Figure 2.2: OAuth2 Authorization Flow request token[27][28]

The Authorisation Code is done by having the Client contact the Authorisation Server requesting access to information of the End User (Figure 2.2). To do so it usually provides its Client Credentials so that the Server knows which application is making the request, as well as including a "code challenge" (part of PKCE [25]) and "state" parameters in the URL request. Then, the Client, in the case of mobile applications, is required to open a browser window redirecting it to the Resource Provider's URL, which also includes the "state" parameter with it. This presents a login screen for the user to input his credentials and log in. Afterwards, the user chooses the scope of consent for the Client. After the End User gives this information, the Resource Provider will redirect back to the Client an Authorisation Code, along with the "state" parameter (Figure 2.3). This "state" needs to be the same as the one received in the initial request to prevent Cross-Site Request Forgery (CSRF) attacks. Afterwards, the Client sends the Authorisation Code and the "code verifier" to the Authorisation Server in exchange for the Access Token. This "code verifier", in the token exchange request, is used to match with the previously received "code challenge". If it is successfully validated, then an Access Token is sent back to the Client (Figure 2.4), otherwise an error occurs. Henceforth, the Client can make requests to the Resource Provider by sending the Access Token to be able to access the

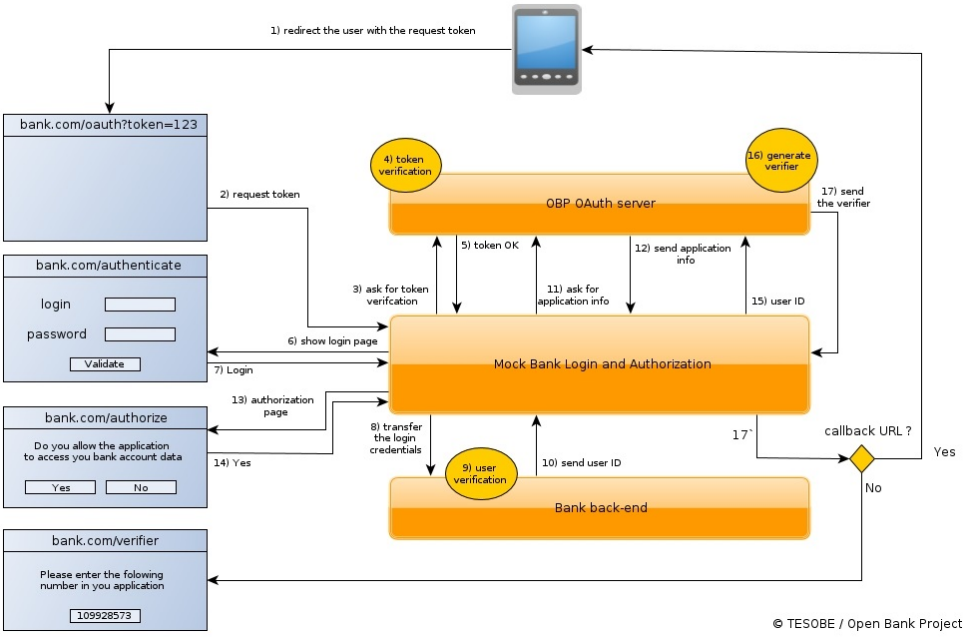

Figure 2.3: OAuth2 Authorization Flow request consent[27][28]

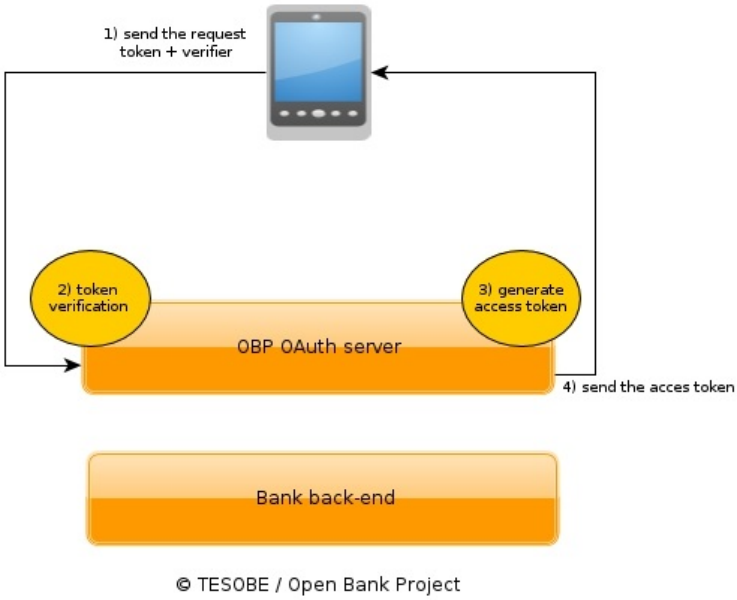

Figure 2.4: OAuth2 Authorization Flow getting access tokens[27][28]

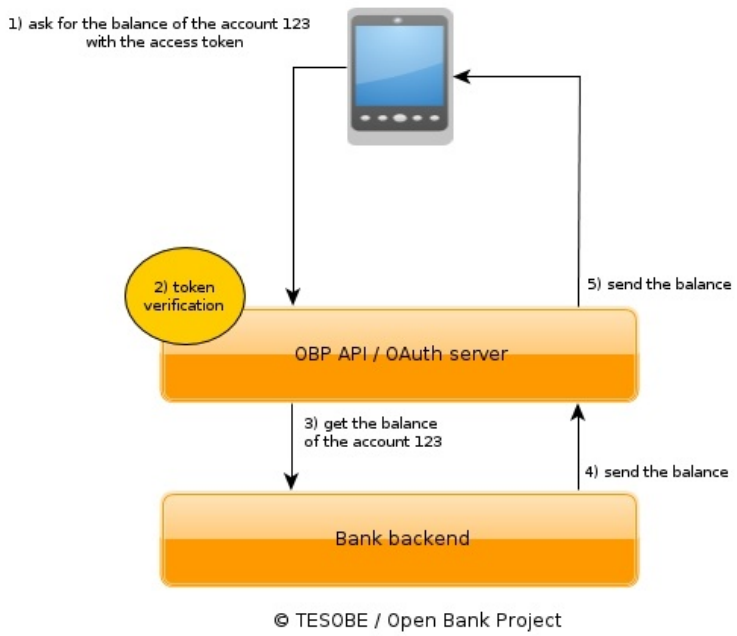

Figure 2.5: OAuth2 Authorization Flow accessing information[27][28]

information within its scope of consent (Figure 2.5). In every exchanged message there is also a "nonce" parameter to prevent Replay Attacks.

The mentioned PKCE [25] involves that the Client creates a secret, also called "code verifier", which is a cryptographically random string of characters. It is then used to create a "code challenge", which is usually a Base64 encoded string of a SHA256 hash of the "code verifier". Then, it includes this "code challenge" with the rest of the parameters on the initial authorisation request. When exchanging the Authorisation Code for an Access Token, it also sends the initial secret for the Authorisation Server to match the two codes. When the Authorisation Server receives this exchange request, it will compute the received secret to match with the previously received "code verifier". This is used to prevent the Authorisation Code Interception Attack, in case a malicious party intercepts the Code and tries to exchange it for tokens, since, as stated, the exchange request needs to pass along the "code verifier" as well.

This means that the Authorisation Code, using PKCE [25], is more secure as an Authorisation Flow than the other three types, but in the end it only provides *Authorisation*, not *Authentication*. Which is where OIDC, with its specifications and definitions plays an important role.

2.3.2 OpenID Connect

Although OAuth has four different types of grants, and OIDC is used on top of OAuth, there are only three types of flows in the OIDC protocol, not to mention that the entities involved in

the interactions have different names as well.

The different entities are called: Relying Party (RP), also known as the Client; End User, which is the user; OpenID Connect Provider (OP), is the server that takes care of Authentication and Authorisation. The Resource Provider from OAuth can still be called the same.

Regarding the three flows of this protocol, they are as follows:

- Authorisation Code Flow - Works mostly the same way as in OAuth, except that in OIDC the authorisation code is exchanged for an identity token and/or an access token.
- Implicit Flow - It requests tokens without explicit client authentication, and obtains the tokens in one round trip to the OP, making it the simplest to implement.
- Hybrid Flow - As the name implies, it is a combination of the previous two flows. It allows to immediately receive an Identity Token at the same time it receives the Authorisation Code. It then exchanges the code for an Access Token.

Since OIDC is built on top of OAuth, both the Authorisation Code and Hybrid Flows can make use of the PKCE [25] extension as well to provide additional security.

As was mentioned, these Flows supply an extra token, the identity token, which contains information about the End User, for example: it can contain the user's full name, phone number, e-mail, etc. It is signed by the OP, which can then be validated by the RP, proving that who made the request for the tokens and gave access permissions is in fact that authenticated user.

*It can be seen in Figure 2.6, the Authorisation Code Flow, that the interactions between involved parties follow the same pattern as in OAuth2. Even so, there are a few additional steps. For starters, the initial interaction is the same, but it includes in the requested scopes, that it wants to obtain the identity token. Then, after exchanging the authorisation code for the tokens, the RP receives an identity and access tokens. The RP validates this identity token before making any more calls. This is so that the RP is assured that the User has been properly authenticated. Afterwards, the RP can make calls to the API endpoints while sending the access token with the request. If the token is valid, then it will process the request, otherwise it returns a HTTP error 401 "Unauthorized".

There is also another token in both OAuth and OIDC, the Refresh Token. This is an optional token and is used to exchange for new, valid access tokens once the previous one has expired. This works by communicating with the endpoint from where the access token is additionally obtained, presenting the refresh token, and if it is valid then a new access token is given.

The tokens mentioned both in this Sub-section and the previous one can be of two types, Self-Referenced JWT or Reference tokens (example of each in Figures 2.7 and 2.8). The Identity

Token is **always** a Self-Referenced JWT, since it contains information pertinent to who issued the token, hashing algorithm, scopes, audience, at_hash and c_hash (in case of the Hybrid flow). The access token is the one that can either be self-referenced or a reference. Self-reference means that it contains information about the scopes that have been consented, and who has the token can see this. On the other hand the reference token is an opaque string that divulges no information whatsoever, i.e. it is used for the RP to validate it by communicating with the OP and asking if that string corresponds to a valid access token. The refresh token is of type Reference, meaning that it only shows to the RP as an opaque string, leaving the real data of it to be kept in the OP.

These three tokens have another thing in common, they all expire after a given time. For how long they are valid to be used depends on the configuration of the OP, which can mandate that all applications have the same base time to live on the tokens or can define different times for distinct RPs. When a token has expired it is immediately deemed as invalid. This is possible to know at any time if the token is of type JWT, but in case it is of type Reference then only by contacting the OP to validate it will the RP know. It is to note that, a Reference token is deemed invalid whether it has expired or if it really is an invalid token.

The two parameters in the identity token called "at_hash" and "c_hash" are used to ascertain, in the RP, if two other objects are valid. The "at_hash" is a hash of the access token. By using the hashing algorithm mentioned in the identity token it is possible for the RP to hash the received access token and then try to match it with the "at_hash" field. The "c_hash" follows the same idea but is the hash of the authorisation code, and this is only present in the Hybrid Flow since it receives both the identity token and authorisation code at the same time.

These tokens are cryptographically signed by using HS256, RS256 or ES256. The first character, "H", "R" and "E" refers to the cipher that was used while the "S256" is referent to the type of hash function. "H" means that the cipher used was "HMAC", "R" refers to "RSA" and "E" to "ECDSA". "HMAC" uses a symmetric secret to encrypt the token, while "RSA" and "ECDSA" use asymmetric encryption to sign the token. With this, "RSA" and "ECDSA" provide Non-Repudiation on top of Integrity and Authentication.

The tokens are also divided into three parts: Header, Payload and Signature. In the Header it states which algorithm was used (from "HS", "RS" or "ES") and what is the token's type (JWT). The Payload contains the claims, the information regarding the issuer of the token, to who it is intended, subject, scopes, etc. To have the Signature part, the Header and Payload need to be encoded, then hashed and signed. This last part is used to verify the token's Authenticity, Non-Repudiation and Integrity.

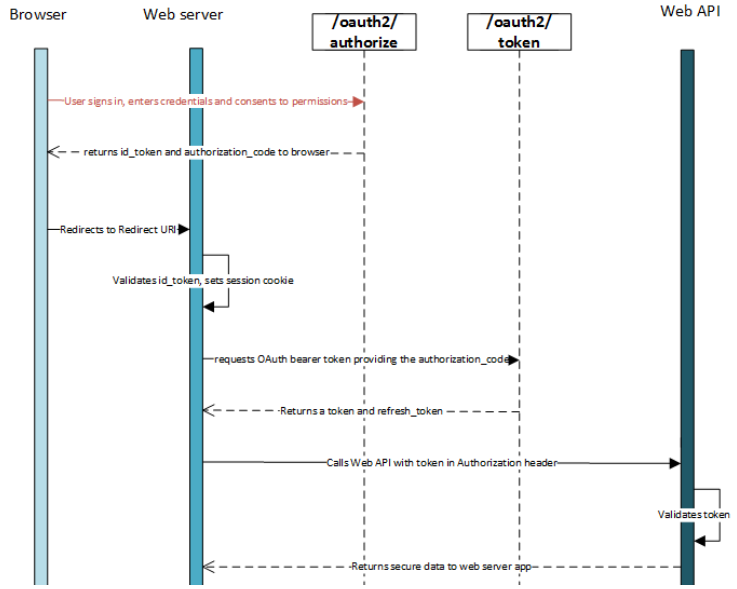

Figure 2.6: OIDC Hybrid Flow. Image provided by Microsoft.

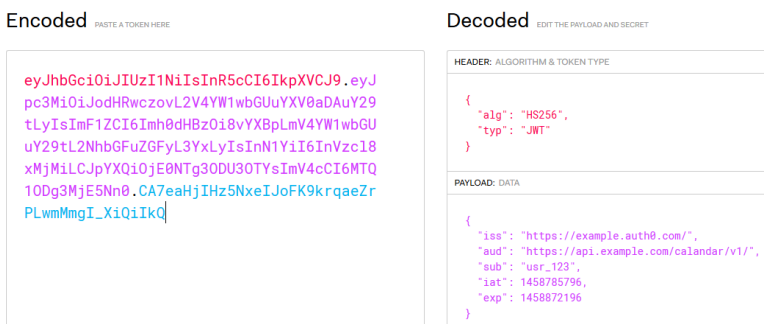

Figure 2.7: Example of a Self-Referenced JWT Access Token

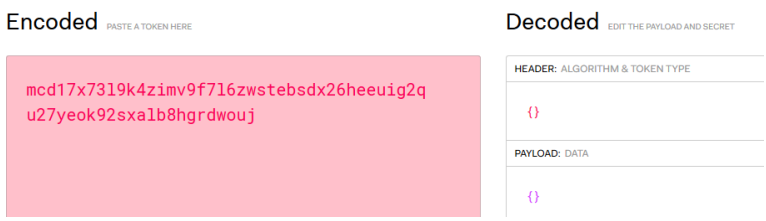

Figure 2.8: Example of a Reference Token

Of the three types of flows, even though the Authorisation Code is still deemed secure and the most commonly used, there are those who prefer to use the Hybrid Flow stating that it provides better security overall. The reason being that it provides a verifiable token before making additional round trips [29] and offering more flexibility as well. Since the Hybrid Flow follows the same pattern of interactions as the Authorisation Code, then it is also deemed to be within the aims of PSD2.

OIDC differs from OAuth2.0 in the part where it provides both Authentication and Authorisation mechanisms, instead of just Authorisation. It also defines cryptography policies for the tokens and token validation, something that is not explicitly defined in OAuth2.0. The JWT token is also a standard defined by OIDC, while OAuth2.0 gave an idea about what a token is and developers would implement however they wanted. Having these explicit definitions in OIDC allows for better interoperability between libraries and identity providers.

Libraries

There are many ongoing implementations of both OAuth and OIDC, but the OpenID Foundation keeps track and certifies any libraries that have correctly implemented OIDC [30]. These [libraries] provide an immense help when developers want to implement their own Authorisation Servers, because it means that the security, interactions, etc., have already been implemented and just need to call the functions of these libraries. There are libraries to help implement not only the Authorisation Server but also a Relying Party, in different programming languages.

An example of such libraries, which in some cases are called "middleware", are IdentityServer4, Connect2id Server, MITREid Connect, OidcClient2, oidc-client-js, pyoidc. The idea behind these libraries is that the developers who want to build an OpenID Provider, or a Relying Party, will get the library for the corresponding programming language and then customise accordingly. Customising means a developer can choose to have specific endpoints closed or open, use a database to save persistent data, provide a certificate for token signing, or extend provided interfaces, which ultimately allow storage of client and resource information.

2.4 Existing Applications

Upon research of Open Banking, PSD2 and APIs, some existing applications/solutions were found, whether already in use or still in development. These are a couple of them:

- Mint[1] – As already mentioned in Section 2.1, Mint is an application used in U.S.A. and Canada that started as an information aggregator that shows to the End user his various

bank account's information in one place. The customer (end user) can choose to budget his money, perform transactions, payments, keep track of his investments and credit score, etc. As previously mentioned, this application uses the "screen scraping" method.

- Monzo[31] –This is a bank located in the United Kingdom, founded in 2015, which works as a digital bank. This means that they do not have physical bank branches, and the only way to contact them, is through the application. Monzo started as a banking application and after getting the right certifications with the Financial Conduct Authority and Prudential Regulation Authority, in 2016, they were finally regarded as an official bank. Before officially becoming a bank, Monzo only provided customers with prepaid cards. Once certified the bank started providing credit cards that are usable worldwide, as well as manageable through the mobile application.

- Teller[32] – Similar to Mint, as in it aggregates information from various bank accounts for a determined user. But the way they found to access the banks APIs was to reverse engineer their applications to find out which API endpoints they use, which security requirements are met and so on (one could argue it is a type of "screen scraping"). Nevertheless, it was proven to be legal; no legal infringements were committed and Teller.io provides their own APIs to any interested third party developers.

- Revolut[33] – Is a banking application, like Monzo, which sends the credit card by mail and lets the user carry out various functionalities in the application, just like with Monzo. Monzo and Revolut are very similar, but their main difference lies in that they are different applications (owned by different companies). Nevertheless, both provide the same type of services and functionalities.

- Open Bank Project[26] – This project, created by TESOBE, brings about innovation with a big community. Started by developing Open APIs to have access to bank customers data in a more homogeneous environment, i.e. third parties can use these APIs without making changes to their code depending on the bank. Instead, they just need to state which bank they want to access and which account from the corresponding bank. These APIs began by being self-sustained but with the appearance of OBS and PSD2 solutions specific to be compliant with these legislations started being developed. Unlike with Monzo or Teller, OBP makes use of the OAuth framework to provide its services, while still using OAuth1 but already migrating to OAuth2.

- IdentityServer4 [34] – IS4 is a highly customisable OSS (Open Source Software) OIDC middleware library for the ASP.NET Core programming language and is officially certified

by the OpenID Foundation [30], as well as part of the .NET Foundation. It provides authentication as a service via a centralised login flow, providing access control to APIs that are to be accessed by applications. It is also possible to perform external authentication, i.e. login with an external identity provider like Google or Facebook. Anything that has been implemented in this middleware can be overridden, allowing the use of more customised logic when it comes to storing tokens, client information, API specific information, how to deal with the authenticated users, etc. This library has been well-documented and provided tutorials and quick-starts to learn how to use it.

- OidcClient2 [35] – Is a OIDC library to be used in native applications that want to access APIs protected by OIDC-based servers. This library is to be used in C#-based applications and takes care of the needed steps when interacting with the OpenID Connect Provider. Afterwards, the access token can be simply obtained and saved to be used when making API calls. Since the same creators of IS4 have developed this library, it also provides a set of quick-starts to learn how to use it.

2.5 BankOnBox

With some applications already developed, or on their way to be published, it means that companies have been attentive to the needs of both Banks and Customers.

BankOnBox is a solution, for Financial Institutions, to bridge technologies, needs of banks and their customers in a way to better engage and retain customers. In the context of this Thesis, the focus fell upon the functionalities of electronic channels (internet banking), which allow clients to perform the traditional functionalities of internet banking (consult balance, payments, transfers, etc.) The architecture represented in Figure 2.9 is the currently used one and was developed using Microsoft technology (APS.Net Framework 4.5, Microsoft SQL Server).

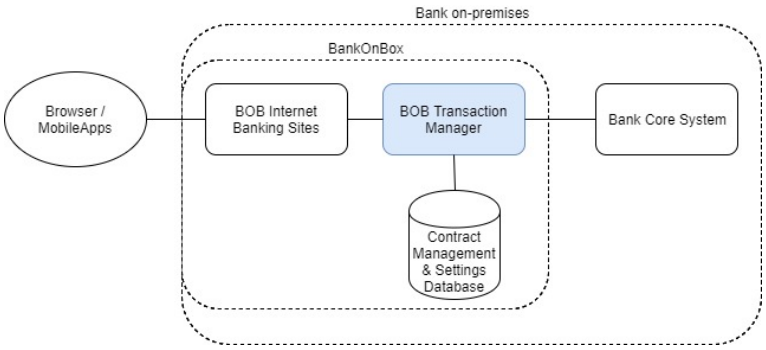

Figure 2.9: BankOnBox Architecture

As can be seen in Figure 2.9, the BankOnBox Transaction Manager is the business component that deals with the operations, authentication and validation of operations done by the users. This module exposes the native APIs in a XML-formatted SOAP protocol (Web service). These APIs are proprietary and the ones that provide the underlying services to be shared, thus bringing the needed openness to make it a PSD2 compliant platform. When it comes to authentication, the users have a first level which is by means of **username** and **password**, then letting users consult their accounts. This module also manages a second level authentication method which is a One-Time Password (OTP) SMS token, used when a user wants to perform payments or transfer money from his account.

Since PSD2 and the corresponding RTS define that the accounts to be accessible are payment accounts (credit, overdraft, etc.). The interfaces concerning this have to be able to access account details, account balance, to initiate payments and confirm them. The only information that cannot be passed, by any means, is information that is deemed sensitive and that can directly identify the user (phone number, address, e-mail).

2.6 Gap Analysis

After the analysis to PSD2, the corresponding RTS and BankOnBox, it came a time to do a Gap analysis to clearly define which interfaces are to be defined. Since there is no Standard, researching for other existing solutions provided some needed insight.

As was stated in Section 2.4, there is the Open Bank Project, which has defined some interfaces to be PSD2 compliant. It was also found that another company, called Apigee Corp, had been developing a PSD2 compliant solution to help accelerate Banks. From there, it was decided to do the gap analysis between Open Bank Project, Apigee and BankOnBox.

Table 2.1: API comparison

OBP API	Apigee API	BOB API
Get Accounts at Bank	GetAccounts	CCNT-Consultar Contas
Get Account by Id	GetAccount	CDOD-Consulta Detalhe de Conta à Ordem CDCC-Consulta Detalhe de Conta Caucionada
Get Accounts at all Banks	– – –	– – –
Get Transactions for Account	GetAccountTransactions	CMOV-Consulta Movimentos
Create Transaction Request	CreateSingleImmediatePayment	TSEP-Pedido de Transferência SEPA
Get Transaction Requests	GetSingleImmediatePayment	– – –

Presented in Table 2.1 are the existing interfaces provided by the PSD2 compliant solutions from Open Bank Project and Apigee, along with the currently provided interfaces of BankOnBox's internal system.

This Table shows that there are interfaces in BankOnBox's internal system that can provide the needed core functionalities. Then, this leads to defining that the changes pertinent to BOB will be the definition of endpoints similar to the ones presented by the two solutions (Open Bank Project and Apigee). How to make calls to them, how to provide an inquiry of funds when the only characteristic received is an account identifier (for example), the response to be provided considering the scope of consent from the end user to the requesting application, etc. This will be looked at in more detail in Chapter 4.

Table 2.2: OBP APIs

Interface	Request Data	Response Data
Get Accounts	URL - BANK_ID	(For each Account) Account Id Account Type Account Description
Get Account By Id	URL - BANK_ID and ACCOUNT_ID	Account Id Account Number Account Owner Account Type Account Balance
Get Transactions for Account	URL - BANK_ID and ACCOUNT_ID	(Per Transaction) Account Debtor and Number Account Creditor and Number Transaction Amount New Balance Date Posted Date Completed Description
Create Transaction Request	URL - BANK_ID and ACCOUNT_ID Body - Amount Creditor Account	Amount Debtor Account Creditor Account Transaction ID Status Start Date End Date

The interfaces analysed from the Open Bank Project and Apigee, or at least the available documentation, provided some knowledge into how to structure the URLs, to make them simple and what should be the required contents of the request and response messages (body and headers, success or error). Tables 2.2 and 2.3 point to the core information regarding the

interesting APIs. Another important piece of knowledge that was obtained was regarding the security involved in it, i.e. both of these entities (OBP and Apigee) make use of the OAuth framework.

Table 2.3: Apigee APIs

Interface	Request Data	Response Data
GetAccounts	– – –	(For each Account) Account Id Account Type Description Currency
GetAccount	URL - AccountId	Account ID Account Type Description Currency
GetAccountTransactions	URL - AccountId	(Per Transaction) Account Id Transaction Id Amount Status BookingDate
CreateSingleImmediatePayment	URL - AccountId Body - Amount Creditor Name Creditor Account	Payment Id Creation Date Amount Debtor Account Creditor Account

When comparing interfaces from BOB with these two groups, it was noticeable that the information in the messages exchanged was similar, i.e. request and response had the same core elements in the body, similar header parameters (some optional, others required). The URL structure to access a resource was also really simple and easy to understand, making it easier for developers to know how to access an API. Examples of requests and responses for the important interfaces are shown from Figures A.1 to A.16 in Annex A. Figures A.5 and A.8 show possible error message codes.

The API names used in Table 2.1 are the ones provided in the documentation of each party (OBP, Apigee and BOB). In the case of BOB it is a code followed by the corresponding operation name. It is noticeable that the interface "Get Account by Id" of OBP, and "GetAccount" of Apigee, correspond to two from BOB. These two provide the information that is within the compliance requested by PSD2, which means that developing a set of APIs to provide PSD2 compliance needed to make a connection with these two internal interfaces. There are also two APIs that have no correspondence to BOB, "Get Account at all Banks" and "Get Transaction Requests". The first one is due to BOB being installed on-premises of the Banks while OBP

has a homogeneous and inter-banking solution, the second one is due to BOB providing the transaction information in the movements list and not having a specific, shareable, identifier for every transaction.

2.7 CSP – Cloud Service Providers

After the gap analysis, this is supposed to be a mobile-centric and distributed solution, since mobile applications are supposed to make use of these APIs. Thinking about this, the thought of a CSP is also nothing new, especially considering that Cloud Computing has seen an increasing growth over the last few years (Figure 2.10). There are many CSPs nowadays that provide API Management capabilities within their product portfolio, either by having an in-house platform (like Azure) or by having acquired a dedicated platform to be part of their platform (like Google [36]).

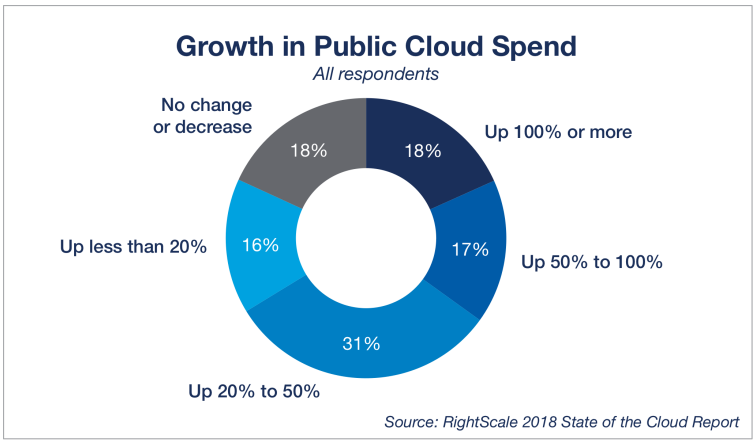

Figure 2.10: Cloud Computing Growth

CSPs are not only important because they have an API Management Platform, but also because they have a myriad of products for anyone to make use of. Data Storage, Virtual Machines, Chatbots, Machine Learning, Text-To-Speech, have been used for a long time or are on the rise (as is the case of Chatbots). Due to this growing portfolio of products CSPs have grown and evolved quite considerably. Initially there was one CSP in the lead, Amazon Web Services (AWS), but a couple of years later Microsoft and Google saw the potential in it and have been able to catch up, thus having them as the biggest three providers of CSP products.

When choosing an API Management Platform provided by a CSP there are a couple of aspects to take into consideration. For starters, how easy it is to use when it comes to deploying and controlling the APIs, how much support there is from the company and the overall community, provided documentation, tutorials, guides, etc. Since these three CSPs provide identical

products, it can also come down to another aspect, which of the companies that own the CSP in question have ongoing deals and businesses. For example, if the company Examp wanted to use an API Management Platform but was still undecided, it would ask itself "which one do we have acquired services from?". Then that would be the CSP to choose from of the big three (Amazon, Microsoft, Google).

2.8 Summary

In the end, after having analyzed PSD2 and its requirements, along with the solutions presented in Section 2.6, it was possible to notice that the important interfaces, that go along with what is requested in the PSD2 legislation, are the ones referred as "GetAccounts", "GetAccount", "GetAccountTransactions" and "CreateSingleImmediatePayment". These interfaces, or APIs, are bound to be exposed in an easier way by using an API Management Platform, which in turn is a service provided by a CSP.

Chapter 3

Solution

As was seen in the previous Chapter, namely Section 2.6, BankOnBox has the necessary interfaces in place to communicate with a set of PSD2 compliant APIs. There is the need to ensure that the exposure of this set of APIs is done securely and transparently, such that the system's attack surface is as small and resilient as possible while maintaining its functionality and availability. There was also a specification of the system's architecture by defining each of the needed components and models, i.e. it was defined what components the system has and a database model where needed information is saved. Why this database is necessary will be described in both Sections 3.3 and 3.4.

Since the regulatory entities that mandated PSD2 have not defined a standard, the Banks and other companies are left to design and develop however they see fit for their needs. At least in the RTS it is stated the minimum level of security requirements that are needed.

3.1 Requirements

One of the main requirements is to define and develop a set of PSD2 compliant APIs that serve as intermediary connecting TPP applications and BankOnBox internal system. This is to provide openness and compliance while maintaining the system secure. It is important to take into consideration the requirements presented in Section 2.1.2, therefore arose the need to guarantee the following security and functionality requirements:

- User Authentication
- Access to Information
- Payment Initiation
- Data Confidentiality

- Data Integrity
- Data Privacy
- System Availability

It was also possible to take from the gap analyses which set of APIs to work on and afterward define them better by designing some Use Cases. The set of PSD2 compliant interfaces are based on the information presented in Section 2.6. Taking this into account and also the requirements presented at the beginning of this Section, the following is the resulting list of features to be provided:

- Authentication of the User
- Consenting authorization to User's Bank Accounts for the calling Application
- Access to User's Bank Account Details
- Access to User's Bank Account Balance
- Access to User's Bank Account Movements
- Access to a List of the User's Accessible Bank Accounts
- Access to User's Bank Account to Initiate Payments

3.2 Relevant Decisions

In regards to the requirements, some essential or relevant decisions needed to be done about the components and overall architecture, these will be explained in more detail during the description of each component and the overall system architecture.

To start with, when deciding which API Management Platform to use, it was weighed about CSP provided solutions and out-of-the-box products as well. In the end, the chosen product was Azure's API Management Platform. The reasoning behind it is explained in Section 3.4.5

OAuth2.0 and OIDC were already described in Section 2.3 and, as was noted, each has their own specification, but since OIDC builds on top of OAuth2.0 and provides the extra Authentication layer, it was decided to implement a OIDC server. This is to take care of the Authentication and Authorisation needs of the system.

There is also a Mobile Application with the intention to show part of the system's functionality. More specifically, the application is a Proof of Concept (PoC) to demonstrate Authentication, Authorisation, Simple Access to Account and Initiating a Payment.

Since the communication between entities is made by using an internet connection, then to prevent, mitigate and provide both confidentiality and integrity of the messages, the communication **must** be done via HTTPS.

Still, about the communication, it was decided that it is to be done in a RESTful manner using JSON formatted messages. Due to it being easier, simpler and lighter on both applications and servers.

3.3 Architecture

It can be seen in Figure 3.1 the overall system Architecture that was come up with:

- Two endpoints, API and OpenID, connecting to Bank On Box internal system.
- API Endpoint – Receives the access to accounts and payment initiation requests.
- OpenId Endpoint – Takes care of Authenticating the End User with the Bank and knowing which Permissions the End User is granting to the Application performing the request.
- OIDC Database – Connected to the OpenID Endpoint. Keeps information about the Applications, API Resources, Tokens and User-specific information.
- Connection between API and OpenID Endpoints – For Access Token validation, when these are received in the API Endpoint. To get any User-specific information from the OIDC Database that is deemed needed to execute a request.
- API Management Platform – Used as a layer of access, to make API requests to the back-end (API Endpoint), by means of API Subscription Keys and Policies.
- Mobile Application – Is a PoC, used to demonstrate the system's functionality. Performs requests via the API Management Platform. Authenticates the End User and obtains Tokens via the OpenID Endpoint.

The interfaces, being the ones pertinent to the features referenced in Section 3.1, are implemented in the API Endpoint and exposed on-line with the help of the API Management Platform. Therefore having this extra layer of transparency from whence only the Management Platform knows which is the back-end's URL address and has direct access to it (will be explained further in Sections 3.4.2 and 3.4.5).

As was mentioned about the API Management Platform, an Application needs to be registered to get a Subscription Key so that it can make API calls to the Platform, which in turn will forward these calls to the back-end, i.e. API Endpoint. As shown in Figure 3.2, even though

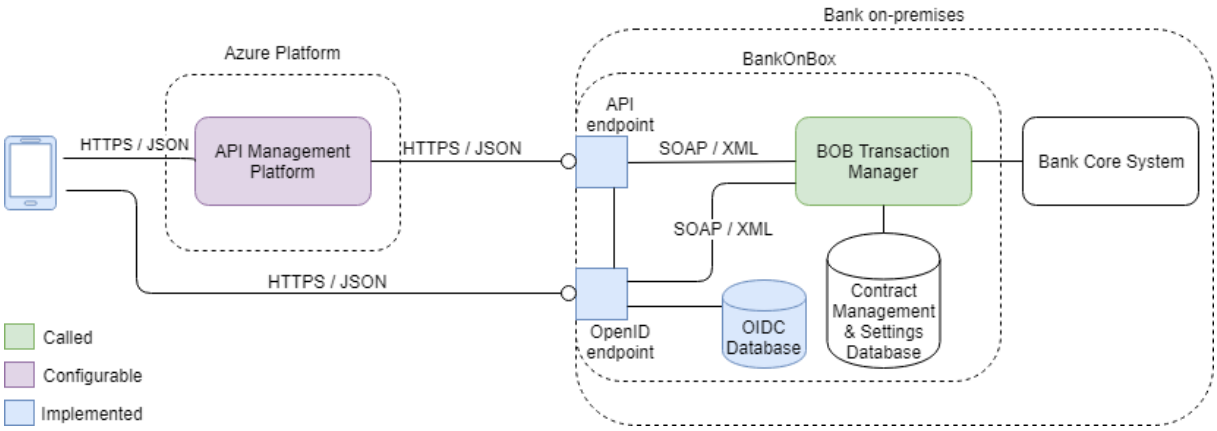

Figure 3.1: System Architecture

the Application might be registered, it still needs to go through another step in order to make Authorized calls to the back-end. This means that the Application **must** communicate with the OpenID Endpoint to let the End User authenticate itself before the Bank and provide consent for the Application. Afterward, the Application will get an Identity Token, Access Token and a Refresh Token. The Access Token grants access to the authorized resources. Only then can the Application effectively make a successful access to account request.

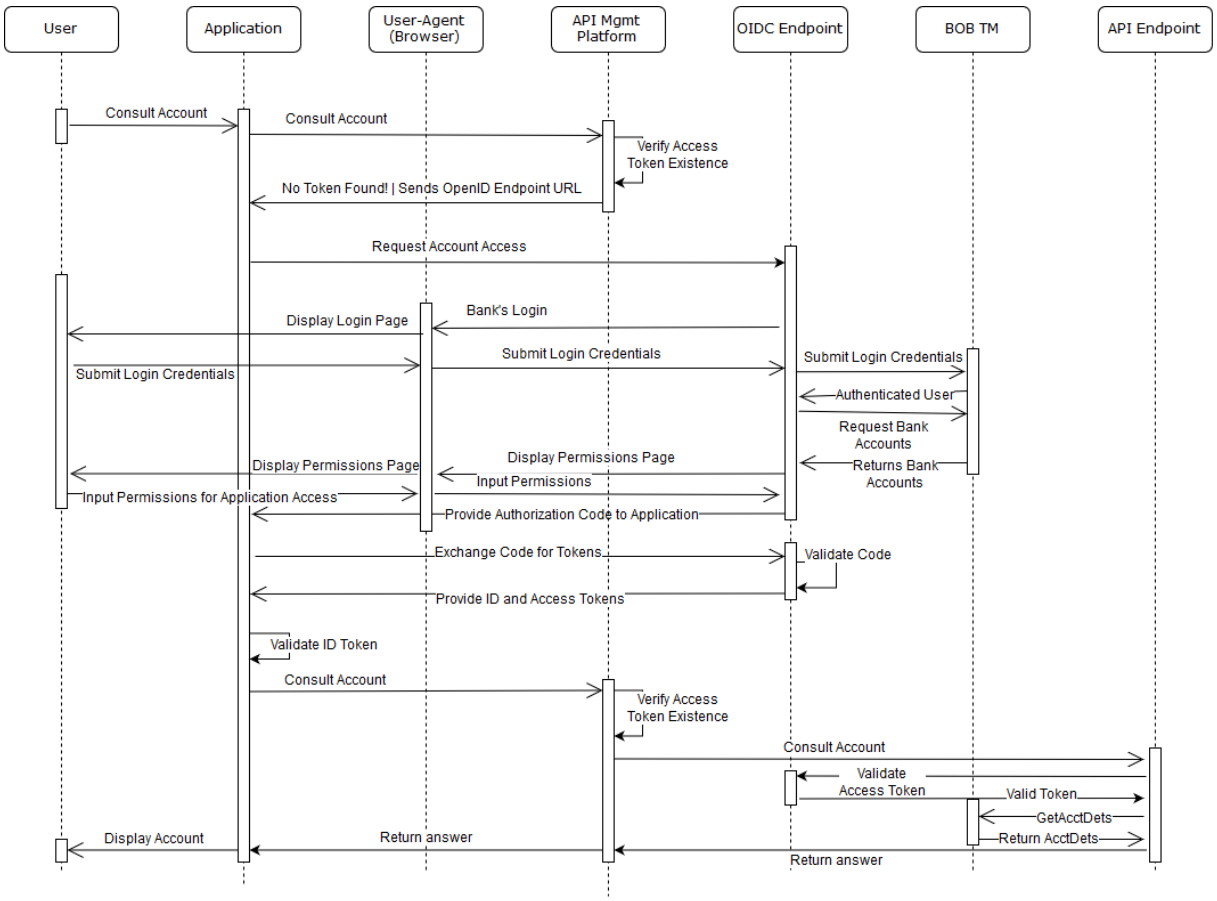

Figure 3.2: Use Case Authentication and Authorization

As shown in Figure 3.3 the Application will not need to make any more requests directed towards the OpenID Endpoint, since it has the Access Token. This Token grants authorisation to access the bank accounts permitted by the End User. There is an exception, as can be seen in Figure 3.4, if the Access Token happens to be revoked or invalid. Then, the Application needs to make a request directed to the OpenID Endpoint to refresh the Access Token. It does this by sending the Refresh Token and in turn receiving a refreshed Access Token in the response. This second situation is also helpful when, for some reason, the End User decides to change the permissions associated with that Application, thus making the corresponding Access Token invalid and needing to be refreshed.

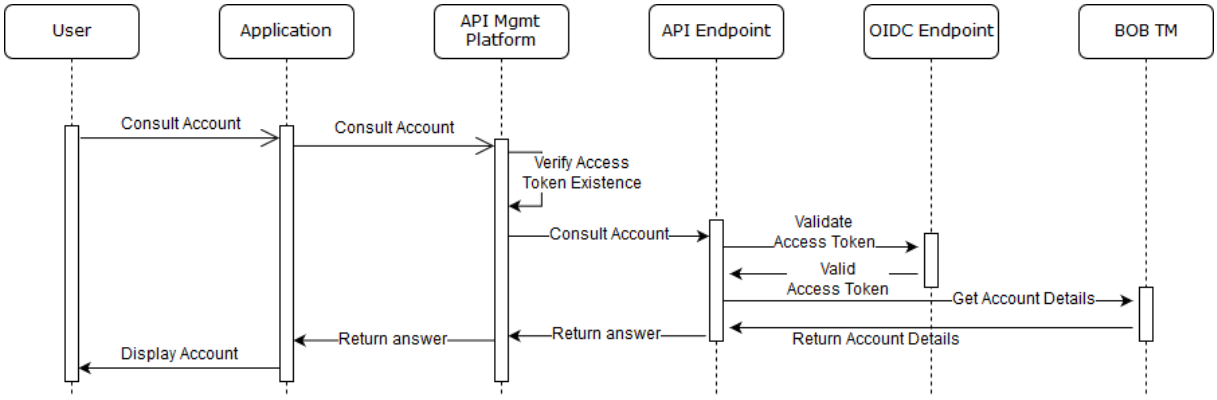

Figure 3.3: Use Case Consult OK

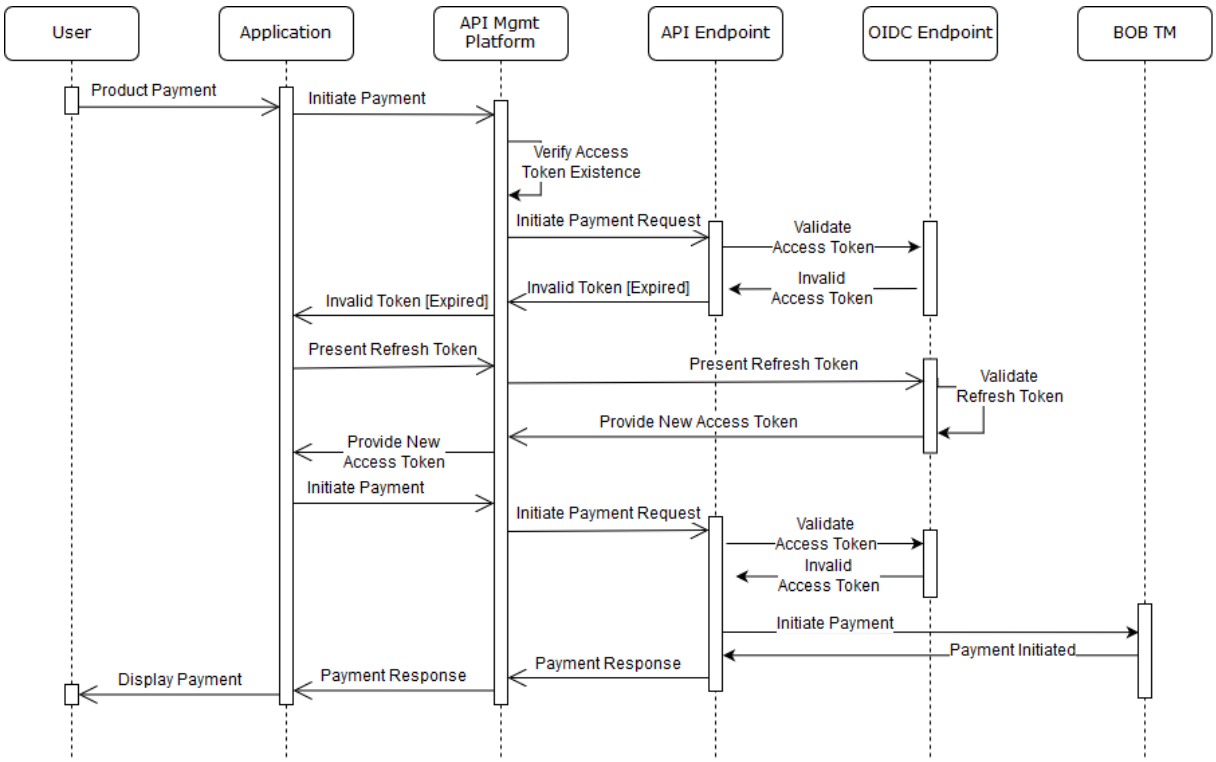

Figure 3.4: Use Case Token Not OK

With the Application performing all the previously mentioned steps, it is worth recalling from Section 2.3.2 the types of Flows in the OIDC protocol, with the Authorization Code and Hybrid flows being considered the most secure. Both of these Flows are to be used when it comes to Native/Mobile Applications, which are the main target, as well as Browser-based Applications. In this case, the Flow that was decided upon to use is the Authorisation Code flow, which can be seen in action in Figure 3.2.

Regarding the OpenID Endpoint, it was decided to develop it by using a configurable middleware called IdentityServer4, i.e. middleware that can be configured to suit the needs of the developer and/or system, which eases on the implementation of the OIDC framework. As was briefly mentioned in Section 2.4, by using this middleware it is possible to implement the OpenID Endpoint more easily since it already takes care of the underlying Authentication and Authorisation mechanisms by adding the OIDC specification compliant endpoints and mechanics to a server application (in this case, the OpenID Endpoint). This leaves to the developer the configuration of the framework to fit other needs of the system. In the case of this project, the needed configurations are as follows:

- Providing a certificate to sign the Tokens with.
- Performing Login with the Bank by making the connection between the Endpoint and BOB Transaction Manager.
- Allowing Login from external identity providers, e.g. Facebook
- Getting Bank Accounts of the End User to give consent to and save this information.
- Identity and API scopes for the End User to consent to in regards to his Bank Accounts.
- Choosing to save important information in a Database (or In-memory) and connecting to it.
- Using a logging library to save in a file the request and response messages, as well as unexpected errors and warnings.

In regards to the OIDC Database, it was decided to implement it due to PSD2 and GDPR legislations, as wells as to save needed and important information in a persistent storage. Regarding the legislations, since user identifying data is not to be divulged there is a need to perform "Pseudonymisation" by providing a unique identifier instead. This unique identifier corresponds directly to an item in the Database, whether it be a User or a Bank Account. Both Users and Bank Accounts pertinent data are kept in this database to be able to perform said

"Pseudonymisation". The provided unique identifier is then used to obtain the needed information about the User or Bank Account when there is a need to perform internal system calls. For example, the API Endpoint receives the unique identifier of a Bank Account and the request is to get the account's details. The unique identifier is given in order to find the matching Bank Account in the Database. After getting the Bank Account number from the Database it is possible to proceed with the internal system call to get the account's details. When sending the response, the account number is not to be shared.

It was also decided that communication between all components of the architecture is done via HTTPS, in which the Server authenticates itself to the caller. This is to provide confidentiality and integrity, to prevent and mitigate Man-in-the-Middle and other possible attacks.

A simple Threat Model was performed, in order to ascertain which Threats could affect the system. The information presented in Tables 3.1, 3.2 and 3.3 show the Threats that were found in regards to the OpenID Endpoint, API Endpoint and API Management Platform, respectively.

Threat	Mitigation
Tampering with data during communication	Since the communications between entities is done using HTTPS, then this Threat is mitigated
Trying to Login with another User's Credentials	To mitigate this Threat, the solution lies in Multi-Factor Authentication
Seeing the Tokens being provided in the communication	To be able to perform this attack, the communication has to be done in HTTP, but since it was already stated that the interactions are performed in HTTPS, then this attack is mitigated
Replay Attack	By using HTTPS, a Replay Attack is not possible. This is due to the generation of a random value every time a TLS handshake is performed
Cross-Site Request Forgery	As required by the OIDC specification, Servers have to echo back the "state" parameter. It then, falls upon the Application developers to verify if this "state" parameter is the same one that was sent
Cross-Site Scripting	To prevent against this Threat, the response for the authorize request is always encoded by the Server before returning it
Application trying to obtain permissions the User did not grant	Since it is required that an external browser window, to which the Application has no access, is used for the Authentication and Consent, then this Threat is mitigated
Trying to perform the OIDC protocol as another Client Application	If another Application has access to the Client Credentials that are passed in the beginning of the OIDC protocol flow, then it is possible. To mitigate against this, enforcing "Mutual Authentication" is a solution
Distributed Denial of Service	To prevent a DDoS: a Firewall rule can be established to throttle the amount of calls being received Providing redundancy to the system by having more running Servers

Table 3.1: OpenID Endpoint Threats

Threat	Mitigation
Tampering with data during communication	Since the communications between entities is done using HTTPS, then this Threat is mitigated
Trying to Login with another User's Credentials	To mitigate this Threat, the solution lies in Multi-Factor Authentication
Obtaining information from other accounts	To mitigate against this Threat, the solution lies in keeping information about which Client Application has access to which Bank Account from each End User
Performing requests with a different Access Token	In case a Client Application is able to obtain a different Access Token and perform requests, these are all logged in the Server. Meaning that, even though the attack is possible, there is evidence of it happening. This also mitigates against Repudiation
Distributed Denial of Service	To prevent a DDoS: a Firewall rule can be established to throttle the amount of calls being received Providing redundancy to the system by having more running Servers

Table 3.2: API Endpoint Threats

Threat	Mitigation
Tampering with data during communication	Since the communications between entities is done using HTTPS, then this Threat is mitigated
Trying to Login with another User's Credentials	To mitigate this Threat, the solution lies in Multi-Factor Authentication
Seeing the Tokens being provided in the communication	To be able to perform this attack, the communication has to be done in HTTP, but since it was already stated that the interactions are performed in HTTPS, then this attack is mitigated
Replay Attack	By using HTTPS, a Replay Attack is not possible. This is due to the generation of a random value every time a TLS handshake is performed
Using a different Client Application's Credentials	If an Application is able to perform requests with another Application's credentials, then it can do all the requests it wants. How to mitigate this Threat is to enforce every Application to provide a Client Certificate, which is then validated by means of a Policy expression

Table 3.3: API Management Platform Threats

3.4 Components

Each component is essential for the whole system to work and provide the needed security and functionality to the system (see Figure 3.1). Having defined the system's architecture and decided which components are necessary to build it, then specifying and describing each one is important. Following is the description of each of the unique components of the system.

3.4.1 BankOnBox Transaction Manager

This is a core component of the architecture. As was explained in Section 2.5, it provides the needed connections to the internal system in order to be able to perform the necessary

operations, such as Login, Accessing Bank Account Details, Initiating Payments, etc.

3.4.2 API Endpoint

This component is where the developed interfaces are exposed from. These are PSD2 compliant APIs that make the connection between the requests and internal services. For all the incoming requests there is a first phase of Access Token verification to validate it. This token is sent, internally, to the OpenID Endpoint for introspection which returns an answer stating if the token is valid or invalid. In case of being invalid, this endpoint returns the response "Unauthorized", in case it is valid, the request is processed. Any necessary calls to the BOB Transaction Manager are henceforth made, and in the end a response is returned.

Both requests and responses to this Endpoint are done via REST in JSON or XML format. When a request is received there is the need to construct a XML message to make a call to BOB Transaction Manager in order to get account information. This message uses information that was received from the JSON/XML request message. Since what is received in the request is an identifier of the account to access, there is an extra step to communicate with the OpenId Endpoint. This is to get the account number correspondent to that identifier. When asking the OpenID Endpoint for the account number, it also asks for the Session Identifier pertinent to that user and application making the request. This information regarding the accounts and users is saved in the OIDC Database, where a translation from a unique identifier to the bank account number, or to the user, is saved at.

In regards to the BankOnBox component, its implementation only lets a user perform requests for 30 minutes after login. Therefore it was needed to come up with a solution to overcome that time limit. This brought the idea of Session Identifiers, which would correspond to a pair of End User and Client Application. The idea was to have this Session Identifier generated in BankOnBox and given, to the OpenID Endpoint, in the response after Authentication was performed successfully. Afterward, this Session Identifier would need to be saved in the Database, such that it only could be used internally when making calls to BankOnBox interfaces by including it in the request messages. With this Session Identifier, the time constraint was no longer existent (the implementation of it will be explained in Section 4.1).

Considering the last paragraph, that is the main reason why there is no transformation, or XML message construction being made, in the API Management Platform. The purpose of doing the translations internally (in the OpenID Endpoint) is to follow with GDPR and PSD2 specifications regarding user identifying and sensitive data. Therefore, providing any of the core information concerning users and their accounts to the outside of the system would go against it.

This would also force extra round trips of on-line communication to get the needed information from the Database.

The connection between the API Management Platform and API Endpoint is made, not by using simple HTTPS but rather "Mutual TLS" (also known as "Mutual Authentication"). This is performed by having both of the Certificates (Client and Server) as Self-Signed. By doing so, it prevents that any outside entity besides the Management Platform can interact with this Endpoint and perform potentially malicious requests.

3.4.3 OIDC Database

The OIDC Database, as was mentioned in the previous Section 3.4.2, is where information regarding Users and their Bank Accounts is saved at, as well as information regarding the API Resources, Client Applications and Tokens (namely Access Tokens of type Reference and Refresh Tokens). Each User and Bank Account has a unique identifier associated with it. This was decided as to share as little information about Users and Bank Accounts to outside of the system. Therefore there was the need to perform what is called "Pseudonymisation".

The information pertinent to the Client Applications are regarding their configurations. This is, particular specifications can be made for different Clients, for example, a Client can be configured to only request to have access to APIs related to account information while another Client can request to only have access to APIs regarding payments. Each Client Application also has a pair of "id" and "secret" which are passed to the OpenID Endpoint when requests are made, directly, to that Endpoint. Each Client can also have an Access Token with different base time duration, i.e. an Access Token for Client A can only be used for 30 minutes while Client B receives Tokens that can only be used for 6 hours. A model of the Database is shown in Chapter 4 and explained the reasoning behind it.

3.4.4 OpenID Endpoint

This component is one of the core parts of the system because it is the one that takes care of the Authentication and Authorization mechanisms. To be able to provide Authentication it communicates with the internal BankOnBox Transaction Manager component for the End User to perform the Login with its credentials. Afterward, it receives the consent information, chosen by the End User, and creates an Identity Token, Access Token, and optionally a Refresh Token. These are sent back to the Application. Whenever a Token needs to be validated, it must be done so in this component, which is where the Tokens received on the API Endpoint are validated at. In this component, there is also a connection to the OpenID Database, which

keeps information regarding Client Applications, API Resources, Users and Bank Accounts.

Having this component implementing the OIDC protocol instead of OAuth2.0 is due to it providing Authentication capabilities on top of Authorisation. It provides better and more reliable security, as was seen in Section 2.3.2. This component also provides the possibility of external logins, i.e. performing a login with an external identity provider like Google or Facebook. For this to be successful, the End User must have its account associated with these external providers. Otherwise, it can be associated by performing a Login with the Bank's Internet Banking credentials, therefore automatically associating the accounts.

Client Applications connect directly to this Endpoint, but only after making an initial call to the API via the Management Platform. This call is intended to verify if there is an Access Token present in the request. In case there is none, then the Client Application receives in the response the URL for the OpenID Endpoint in order to perform the needed interactions afterward.

It was also decided that, although Identity Tokens are, always, of Self-Reference type, the Access Tokens should be of Reference type instead. This is due to an Access Token having a variable size which can render requests, in which it is contained, to be too large. This can be an issue due to possible size restrictions in the API Management Platform regarding request-response URL passing. In order to prevent it from happening, and to also prevent against any malicious third parties who could obtain the Access Token, it was decided to use the Reference type instead of Self-Referenced JWT. This means that there is an extra round of communication when the API Endpoint receives an Access Token and wants to validate it with the OpenID Endpoint, but this is considered negligible in comparison to the potential risks.

3.4.5 API Management Platform

Looking back at what was mentioned during the Background analyses (more specifically, Section 2.2), it was noticed that using an API Management Platform provided by a CSP would be beneficial since it increases the availability, scalability and reliability of the overall system, as well as bringing a more direct and faster access to any of their [CSP] other services. With an API Management Platform it is also possible to make use of specific policies, as was mentioned in Section 2.2.5.

The Platform that was decided upon to use is from Microsoft (Azure's API Management Platform) and it is due to simple facts. The services provided by each of the CSPs is very similar, thus what weighed more was the fact that companies which make use of other services from a specific CSP tend to stick with that one. Considering that in Link Consulting there

are certain products and technologies from Microsoft (e.g. Azure Active Directory, ASP.NET programming language), it was close to a given that Azure would be the chosen platform. This also helps on having a more direct connection between Azure's API Management Platform and other products (e.g. there are configurations in the platform to better connect with other Azure products).

The idea of using this platform is to provide additional features to the system. For example: Throttle requests by restricting how many calls can be made per minute and day; Checking if the Client Application has an API subscription key; Authenticate the Clients by checking for a Certificate they have sent; Checking if a custom Header parameter is included or not; Logging to Event Hub, which logs request and/or response messages to a log; Sending to the back-end a Header parameter which states the IP Address of the originating request. The Platform can also provide better scalability to the system by having more running units, a better SLA, depending on the chosen Tier, and an increased amount of requests per second 3.5.

Purpose	Non-production use cases and evaluations	Entry-level production use cases	Medium-volume production use cases	High-volume or Enterprise production use cases
Price (per unit)	€0.06/hour	€0.18/hour	€0.80/hour	€3.23/hour
Cache (per unit)	10 MB	50 MB	1 GB	5 GB
Scale-out (units)	1	2	4	10 per region (call support to add more)
SLA	No	99.9%	99.9%	99.95% [1]
Azure Active Directory integration	Yes	No	Yes	Yes
Virtual Network support	Yes	No	No	Yes
Multi-region deployment	No	No	No	Yes
Estimated Maximum Throughput[2] (per unit)	500 requests/sec	1,000 requests/sec	2,500 requests/sec	4,000 requests/sec

Figure 3.5: Azure's API Management Platform Pricing

This component also performs "Mutual Authentication" with the API endpoint by sending a Client Certificate in a Header parameter. This Certificate was uploaded in the Management Platform and a policy defined to send it along to the API Endpoint when a connection is made.

3.4.6 Mobile Application

The importance of this component is to demonstrate, in the end, how the whole system works from a functional point of view, i.e. to show the overall functionality of the system. To do so, the Mobile Application is able to perform a Login, provide consent to accessing Bank Accounts, receive Tokens from the OpenID Endpoint, access Bank Accounts to obtain details and to initiate payments.

It connects, via HTTPS, to the API Management Platform. Only when it receives the URL, from the Platform, to connect with the OpenID Endpoint, is it able to interact with it in order to perform the needed Authentication and Authorisation steps to receive the Tokens.

3.4.7 Summary

By the end of this Chapter, a Solution that follows the motivation and objectives stated in Chapter 1 has been described, stating the importance of having an API Management Platform to help ease on the exposure of the APIs, and the needed components, technology, protocols and interactions between components for the system to work. From here on, in the next Chapter, how this system was implemented will be described, stating important decisions and how specific parts of each component work.

Chapter 4

Implementation

This Chapter presents the Implementation in regards to the Solution that was defined in Chapter 3. Decisions made during Implementation are also stated and shown. These decisions were taken in order to have a system that corresponds to Business needs, legislation compliance, functionality and security requirements presented throughout this document. Some of these decisions are component specific, i.e. they affect a given component of the architecture particularly, while others involve the whole system.

4.1 Architecture Implementation

Regarding the presented architecture in the previous Section 3.3, some implementation decisions that affected the whole system needed to be taken.

For starters, it was required to use HTTPS in every communication between the entities. Since the Endpoints are hosted in Internet Information Services (IIS), HTTPS was turned on in it by binding each Endpoint to a specific port and providing the Certificate. To provide this functionality, a Certificate had to be generated, which was obtained via an outside party (SSL for free). The API Management Platform has a configuration field where it can be chosen for the incoming connections to be HTTP, HTTPS or both. This was set to HTTPS. It was also decided to implement "Mutual Authentication" in the connection between API Management Platform and API Endpoint (further explained in Sections 4.4 and 4.5).

Apart from HTTPS in the connections, it was also decided to follow the REST architectural style with JSON formatted messages. It was implemented by following the specifications related to these two elements (REST and JSON). By putting these specification requirements into practice, it forces applications to follow them as well.

As was previously mentioned, in Section 3.4.4, Tokens can turn out to have a big payload,

therefore increasing the data going through the network and the URL length. In attempts to keep it light and shorter for all entities, the Access Tokens are configured to be of Reference type. This also makes it easier to have more fine-grained control over them since Self-Reference JWT are only invalid when they expire, but Reference Tokens are invalid from the moment they are deleted from the Database as well.

The OIDC protocol Flow decided to use was the Authorisation Code Flow. Even though, as was previously mentioned in Section 2.3.2, both this Flow and Hybrid are the most secure. Choosing the Authorisation Code Flow relied more on the fact that it is currently the most commonly used, since it works in the same way as its OAuth2.0 counterpart.

As was stated in Section 3.4.2, there is the existence of Session Identifiers. These are generated in the BankOnBox Transaction Manager component when a User Authentication has been successfully performed. The Identifier is given to the OpenID Endpoint, which in turn saves it in the OIDC Database, and associates it to a pair of User Identifier and Client Identifier (Application). When the API Endpoint asks for an Identifier translation of the Account Number, it also obtains, in the response, this Session Identifier. In the API Endpoint, a XML message is constructed and, in it, the Session Identifier included, in order to be able to perform requests to the BOB Transaction Manager component without having the 30 minute constraint impeding it from accessing the account information or initiating payments.

As was mentioned in Section 3.4, each component of the system is very important for it to work as a whole. These following Sections present implementation decisions taken for each component in regards to their particularities.

4.2 OpenID Endpoint

This component was implemented in ASP.NET Core 2.0, using the IdentityServer4 middleware and all the libraries related to it that were needed (IdentityModel and EntityFramework). Another library, Serilog, is used for logging purposes. It also runs on IIS, which manages the ports for communication with this Endpoint, as well as requiring the use of HTTPS.

It was implemented in ASP.NET Core instead of ASP.NET Framework due to constraints in regards to a previous version of IdentityServer, namely IdentityServer3. IdentityServer3 was developed for ASP.NET Framework, but it is no longer under development nor being maintained, while IS4 is under constant development and maintenance.

As was mentioned in Section 3.4.4, this component is the one that provides the main Authentication and Authorisation mechanisms.

4.2.1 IdentityServer4

It is in this Endpoint, by using the OSS library IS4, that the OIDC protocol is implemented, therefore obtaining a fully working OpenID Server to provide the mentioned mechanisms (Authentication and Authorisation).

As was stated in Section 3.4.4, there can be different configurations for each Client. Although some of these configurations can differ, there are others that were decided to maintain as "static" (i.e. to keep them constant between all Clients). The configurations that can differ from Client to Client depend on internal decisions (made by the bank) or an agreement between the developers and the bank.

Following are the "static" configuration decisions for all Client Applications:

- The type of Access Token to be used is Reference, due to the following reasons: Provides a more fixed-size string instead of variable size, which can make the URL to be too long; It is an opaque string from which no information is given, unlike with Self-Referenced JWT in which it can be read by anyone who has the Token.

- Access Token's lifetime is of 1 hour, to prevent and mitigate any malicious use of it in case another entity tries to.

- Authorisation Code Flow using PKCE [25] extension. Considered the most secure flow alongside Hybrid, and also the most commonly used.

- Identity Token including only the "name" field in it as to help provide a unique user experience in the application, in case third-party developers want to do so. Apart from the "name" field, the other fields (as were seen in Section 2.3.2) are all required. The "sub" field is an identifier of the End User, in this case, a GUID defined in the OpenID Endpoint and then saved in the Database.

- A Consent page is shown every time the End User performs a login. This is to remind them of their consents and that it can only be changed via the Bank's on-line portal. Since PSD2 and GDPR state not to share User identifiable information unless the User consents to it, the same can choose to share his name or not in this Consent page.

- Authorisation Code lifetime is of 5 minutes. This is to prevent any attempts at using one maliciously. Which in turn is being prevented by using the PKCE extension.

- When Refresh Tokens are issued, they all have a maximum lifetime of 30 days before expiring.

- Requiring the Client to send its Credentials (Client ID and Secret) when attempting to initiate the OIDC protocol flow. These Credentials are saved in the Database.

Apart from previous configurations to all of the Clients, there are some that are Client specific:

- ClientId – A unique identifier of the Client. This element is a string, and when information is saved in the database it has a corresponding integer to it.
- ClientName – The name given to the Application in question.
- ClientSecret – The "password" of the Client, when used in conjunction with ClientId, towards the OpenID Endpoint, it is validated and afterward permitted to proceed with the OIDC protocol.
- RedirectUri – The URI to which the OpenID Endpoint will send back the Authorisation Code when after consent by the End User was given.
- AllowedScopes – The scopes of access that the Application is given consent to. This list of scopes specifically states what particular access the application will have. For example, if the list only has the scope "ais.consacctbal", it can only perform balance consultations on the chosen bank accounts.
- AllowOfflineAccess – Regarding Refresh Tokens. If set to true, then a Refresh Token is sent during the exchange of the Authorisation Code for the Tokens. Otherwise, no Refresh Tokens are issued for this Client.

There are more configuration fields, but they all have a default value set by IdentityServer4. These were all analysed and deemed valid for the purposes at hand. Example of a possible Client configuration is shown in Figure 4.1.

With the Clients having been correctly configured, the OpenID Server also needs to save information regarding the API and Identity Resources. The latter are only the "sub" and "name", pertinent to the End User. The former are pertinent to the APIs to be accessed on the API Endpoint. Each API has a name, secret and list of scopes (shown in Figure 4.2). The "secret" field is needed for when an Access Token of Reference Type needs to be validated, i.e. the request asking to validate the Access Token **must** contain both the API "name" and "secret" to validate it. Otherwise, an error response is returned.

In regards to this project, the names are "pis" and "ais", Payment Initiation Services and Account Information Services, respectively. The scopes of each are the ones regarding the interfaces, i.e. "pis" has the scope of "pis.initpay" which pertains to Initiating Payments while "ais"

has “ais.consacctbal”, “ais.consaccttrans”, “ais.consacctlist” and “ais.consacct” which pertain to consulting balance, movements, list of accessible accounts and account details.

```
new ApiResource
{
    Name="ais",
    DisplayName="Account Information Services",
    ApiSecrets = { new Secret("secret".Sha256()) },
    Scopes =
    {
        new Scope()
        {
            Name="ais.consacctlist",
            DisplayName="Access to an account list",
        },
        new Scope()
        {
            Name="ais.consacct",
            DisplayName="Access to a specific account details",
        },
        new Scope()
        {
            Name="ais.consacctbal",
            DisplayName="Access to a specific account balances",
        },
        new Scope()
        {
            Name="ais.consaccttrans",
            DisplayName="Access to a specific account transactions"
        }
    }
}
```

Figure 4.1: Client Configuration Example

```
new Client
{
    ClientId = "mobile.application",
    ClientName = "Mobile Application",
    AllowedGrantTypes = GrantTypes.CodeAndClientCredentials,
    AccessTokenType = AccessTokenType.Reference,
    ClientSecrets =
    {
        new Secret("secretas".Sha256())
    },
    RedirectUris = { "mobileapplications://callback" },
    PostLogoutRedirectUris = { "mobileapplications://callback" },
    AlwaysIncludeUserClaimsInIdToken = true,
    AllowedScopes =
    {
        IdentityServerConstants.StandardScopes.OpenId,
        "user.name",
        "ais.consacctbal",
        "ais.consacct",
        "ais.consaccttrans",
        "ais.consacctlist",
        "pis.initpay"
    },
    AllowOfflineAccess = true,
    RequireConsent = true,
    UpdateAccessTokenClaimsOnRefresh = true,
    RequirePkce = true,
}
```

Figure 4.2: API Resource Configuration Example

4.2.2 Database

The interactions between this Endpoint and the OIDC Database are made by using the EntityFramework library. Therefore it is not needed to implement specific Database language queries but use lambda expressions instead (Example in Figure 4.3) through the method of LINQ Querying. This provides better security since LINQ queries are not susceptible to traditional SQL injection attacks, due to using "SqlParameter" in queries, which means that input is turned into parameter values.

```
var user = ctx.Users.Where(x => x.Name.Equals(name)).FirstOrDefault();
```

Figure 4.3: Example of LINQ query for Database

4.2.3 Login Flow

In regards to Authentication, a login work-flow with Two-Factor Authentication Login and Consent pages was implemented, in Razor Pages (Microsoft technology to create dynamic web pages, similar to HTML). After the Two-Factor Authentication has been correctly performed, a cookie is issued and managed by the cookie authentication handler from ASP.NET Core. This type of cookie is a ticket containing specified claims and properties that were afterward serialised and encrypted. This cookie is then passed on to the Client, who sends it back with each request until the login flow is over. Which is why during Login, the methods in OpenID Endpoint use the tag "ValidateAntiForgeryToken" to validate this encrypted cookie (shown in Figure 4.4).

```
[HttpPost]
[ValidateAntiForgeryToken]
2 references | João Santos, 7 days ago | 2 authors, 14 changes | 0 requests | 0 exceptions
public async Task<IActionResult> LoginTwoFactor(LoginTwoFactorModel model, string button)
```

Figure 4.4: Tag – Validate Anti Forgery Token

This login flow, as can be seen in Figure 4.5, is built with redirects until the Authorisation Code is given to the Client. This means that when an authorisation request is made to the OpenID Endpoint, it will redirect the Client Application to begin the login logic. This opens up a login page, in which, if the credentials are valid, redirects to the second step of the two-factor authentication. After validating the code of 2FA, it redirects to the consent page. After consent has been given it redirects to the return URL, which leads the Client Application it back to the authorisation endpoint to finish the logic. In every part of these redirects, validating the return URL is important in order to mitigate against open-redirect attacks. This validation is performed by calling a method in the IdentityServer4 library (shown in Figure 4.6).

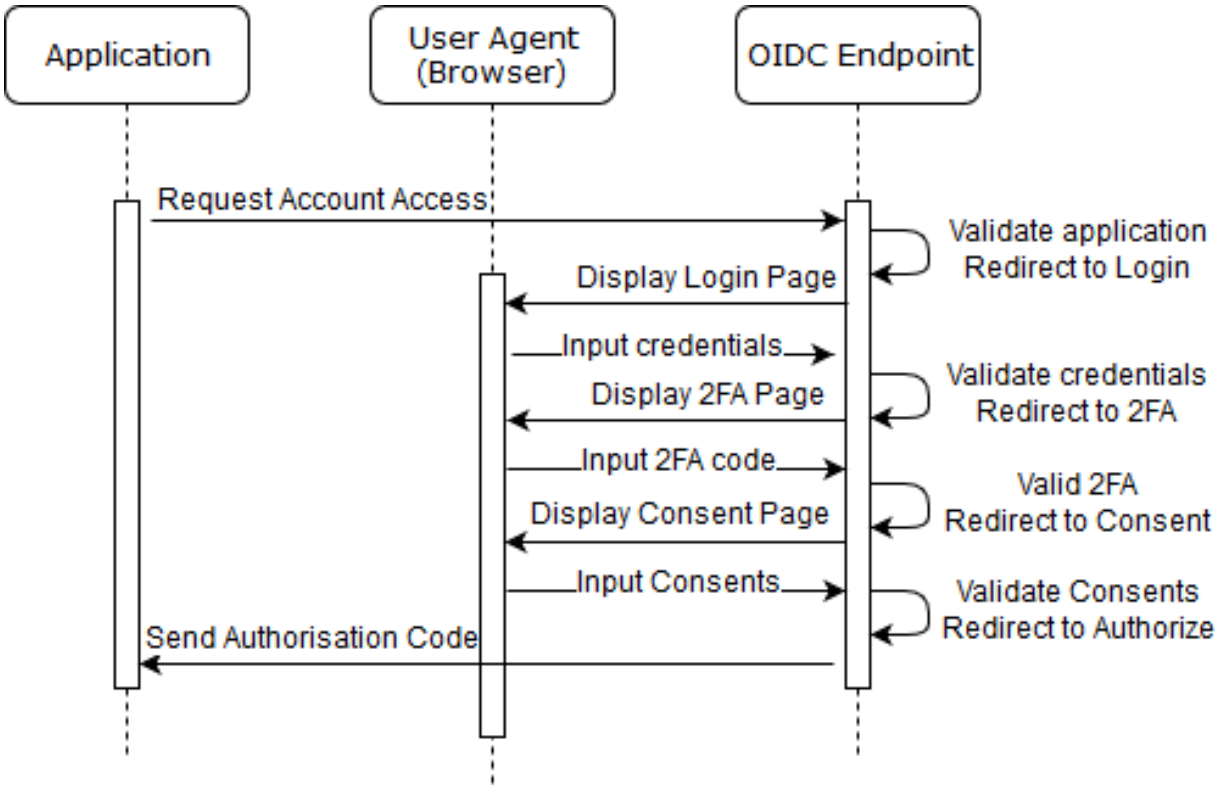

Figure 4.5: Login Redirects

```
if (_interaction.IsValidReturnUrl(model.ReturnUrl) || Url.IsLocalUrl(model.ReturnUrl))
{
    return Redirect(model.ReturnUrl);
}
```

Figure 4.6: URL Validation

How the Authorisation Code, Identity Token, Access Token and Refresh Token are generated and validated is part of the underlying implementation of IdentityServer4.

It is also possible to perform an external login by using a Facebook account. If the Facebook account has not been associated with a User, then the User has to perform a login and it will immediately make the association. Otherwise, it will perform the login automatically. A User can only have one Facebook account associated. In case other external identity providers are added later, the same applies.

All of the interactions during the OIDC protocol flow are logged, by using Serilog. Token Validation and requests for translations by the API Endpoint are also logged (example is shown in Figure 4.7).

```
2018-10-13 23:31:45.812 +01:00 [INF] Invoking IdentityServer endpoint:
IdentityServer4.Endpoints.IntrospectionEndpoint for /connect/introspect
2018-10-13 23:31:48.485 +01:00 [INF] Success token introspection.
Token active: false, for API name: ais
2018-10-13 23:32:15.328 +01:00 [INF] Clearing 1 tokens
```

Figure 4.7: Logging Example

4.2.4 User Related

End User information is also processed in this Endpoint in order to save it in the Database and make the necessary "Pseudonymisation" translations. When the API Endpoint needs to obtain a bank account number or a user's username, to perform calls to the BankOnBox Transaction Manager, it contacts this endpoint first. The API Endpoint provides the unique identifier and states which information it needs. The OpenID connects with the Database to do the translation and sends back the needed data. Part of this data is the Session Identifier pertinent to the End User and Client Application making the request to the API Endpoint.

Before performing the mentioned translations, there is a previous interaction from the API Endpoint. This interaction happens when it receives an Access Token and needs to validate it. This validation checks if the Token includes the permissions to access the requested interface. The OpenID Endpoint returns a response stating if the Token is valid or invalid (both in case of being expired, malformed or no permissions were granted).

4.3 OIDC Database

One of the early decisions, after being decided to implement the OIDC protocol in the OpenID Endpoint, was to model the Database, i.e. to define a model for it, which entities it would have and how they would connect. The OIDC protocol does not define a database schema, this was decided to do in order to save important information in a persistent storage component, instead of in-memory. The model for this database can be seen in Figures 4.8 and 4.9. This Database is hosted in a SQL Server.

As can be noticed, and was stated in the previous Section 4.2, information in regards to the Applications is needed, as well as for the API Resources and End User-specific information. Briefly explained:

- Clients – The tables that are pertinent to the Applications, it has saved up all the needed configuration information in regards to each Application, from which type of Access Token they are to obtain to how long said Token is valid (this is per Application), as well as knowing which Flow each Application is using and also the URI to where the OpenID Endpoint

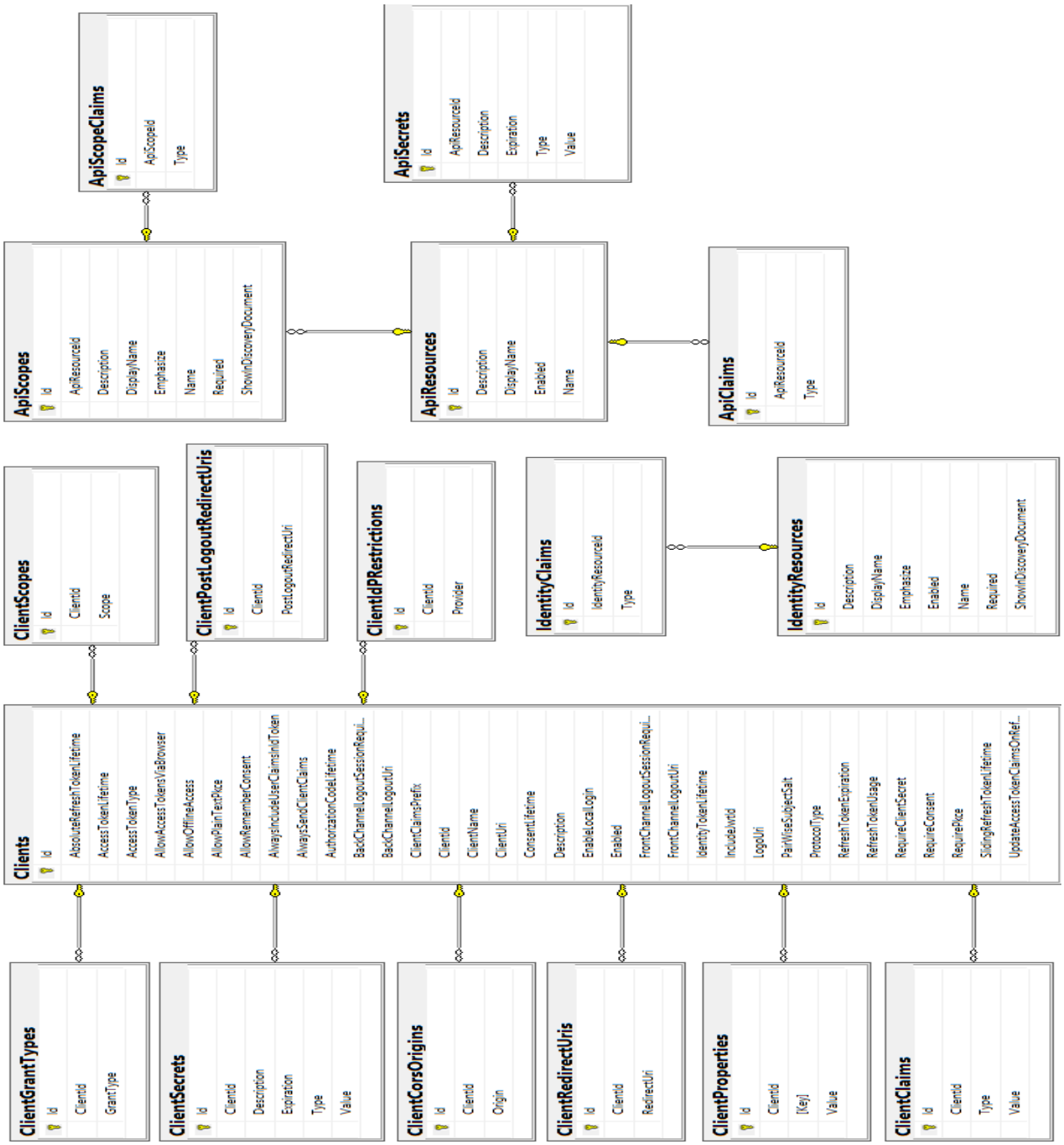

Figure 4.8: Database Model - Configuration Context

will redirect to when the End User Authentication flow is over and the authorization code is to be sent to the Application (this URI is where to send the code).

- API – The tables regarding the API Resources information, scopes of each API, i.e. each API Resource can have multiple scopes, e.g. "api1" can have "api1.read" and "api1.write" scopes and these are the ones that go into the Access Token, not "api1". It also needs to have a secret such that when a verification of the Access Token validity is needed, the API Endpoint needs to send this secret along with the Token, which helps to identify which API it is that the API Endpoint wants to validate the Token towards to.

- Identity – These 2 tables “IdentityClaims” and “IdentityResources” are the ones regarding configurable information (non-explicitly required) to be included in the Identity Token, in the case of this project this has two entries, one for the “sub” and one for the “name” claims, as was already specified in the previous Section.

- PersistedGrants – This is the table in which the Access Tokens (Reference type), Refresh Tokens and Consent information are saved at, indicating as well which Application has which Token and to which User it refers to, making it easier to verify if the Token might have been misused or not. When a token is to be validated, the received token will be compared to the one kept in this table and see if they are equal to each other.

- User Information – The tables regarding the End User are really important in this, even though they have saved the name, username, bank account number, this is for purposes of “Pseudonymisation”, i.e. since User’s sensitive information is not to be shared (except the name in case the User consents to it, for UX-purposes), especially the login information and bank account number, there is a need to give a unique identifier to each End User and each Bank Account. These are the identifiers that the Application will be receiving, and therefore when an Application wants to access bank account information, the API Endpoint has to ask the OpenID Endpoint for this information by providing it with the Bank Account and User identifiers.

This database is divided into three Contexts. One for the Configuration data, which pertains to Clients and API Resources, since these tend to be more static and there are mostly requests to read rather than writing. One for Persistent Storage, which is the “PersistedGrants” table, where the tokens and consent information are written into and read from. The last one is for the User data, into which the User specific and pertinent information is saved on and read from. This is where the data regarding the User’s username (for BankOnBox calls), Bank Accounts and their permissions, External Provider Identifier and Session Identifiers are kept. The “BankAcctPerms” table was implemented to provide more fine-grained control over the permissions regarding Bank Accounts. This is due to the Access Tokens containing information about which scopes the Client Application received permission to access, but not in regards to the Bank Accounts. Therefore, this table provides that needed control over which accounts are permitted for each scope, per Client Application.

It was decided to implement it this way in order to separate different concerns inside the same Database, separating each other into its own self-contained area.

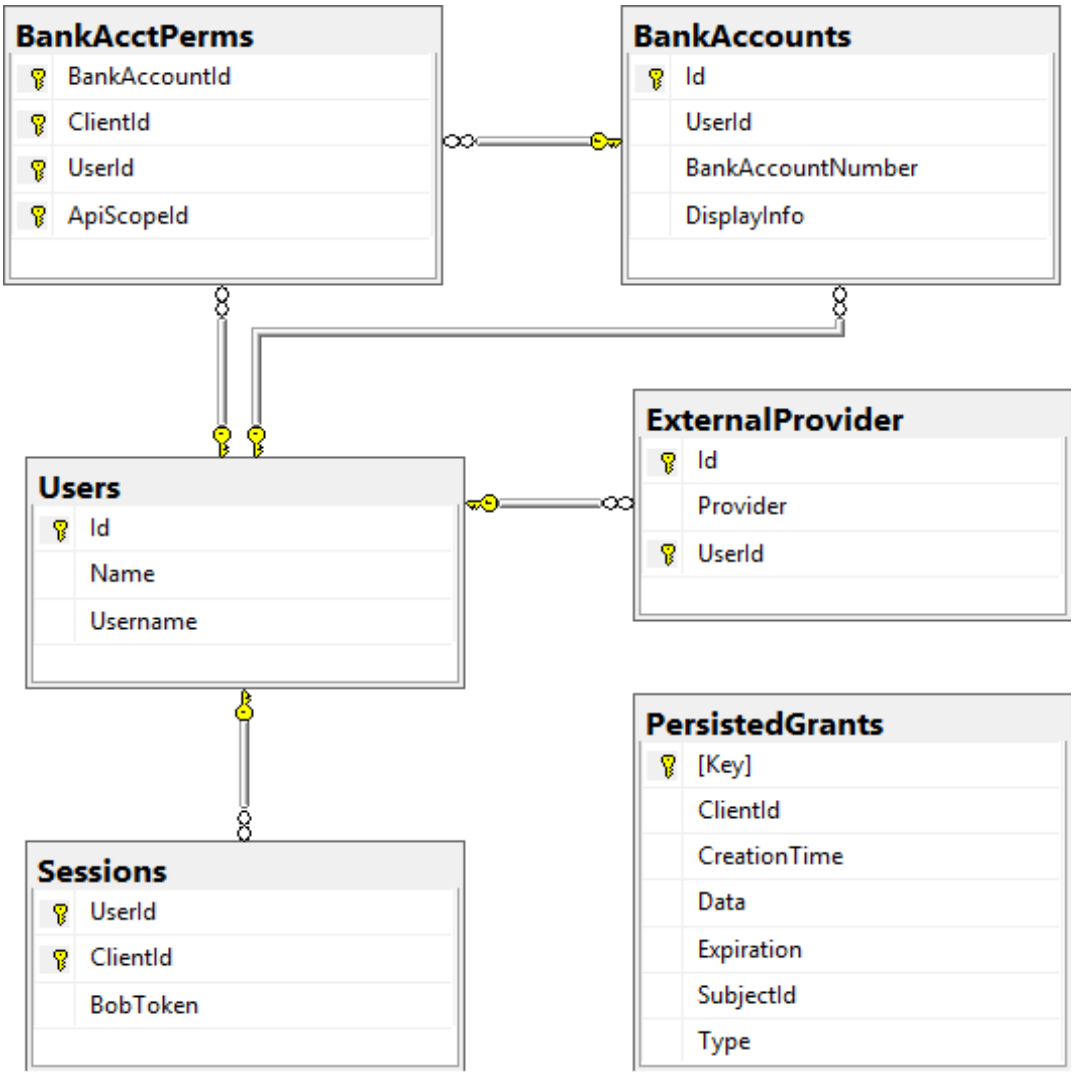

Figure 4.9: Database Model - Persistent Storage and User Contexts

4.4 API Endpoint

The API Endpoint is a ASP.NET Web API implemented using ASP.NET Framework 4.6.2 and using the libraries IdentityModel, IdentityServer3.Contrib.AccessTokenValidation and Serilog. As with the OpenID Endpoint, this component also runs on IIS and requires HTTPS to be used.

In case of this Endpoint, IIS was configured to force any incoming towards its "port" to provide a specific Certificate. This ensures the existence of a "Mutual Authentication" connection between this Endpoint and entities that have the needed Certificate. The API Management Platform includes said Certificate in all its requests.

All of the requests that are made to this Endpoint are logged, by saving it in a file with information about who made the request, which was the called interface, the originating IP Address of the request, the identifier of the end user and the identifier of the request (shown in Figure 4.10).

```
Log.Logger.Information("Transaction [{0}] requested by client [{1}] from IP-Address [{2}]" +
    "in name of [{3}]: Performing a [{4}] operation",
    requestIdValue, clientIdClaim, requestIpValue, userIdClaim, ACCTDETS);
```

Figure 4.10: Logging string

The TokenValidation library is the one that leads this Endpoint to connect towards the OpenID Endpoint and validate the Access Tokens. In case the response from OpenID Endpoint states that it is invalid, then it returns a response to the caller with a HTTP code 401 "Unauthorized". To be able to perform this Token Validation request, Figure 4.11 shows the needed implementation.

```
app.MapWhen(ctx => ctx.Request.Path.StartsWithSegments(new PathString("/pis")), authz =>
{
    authz.UseIdentityServerBearerTokenAuthentication(new IdentityServerBearerTokenAuthenticationOptions
    {
        Authority = "https://localhost",
        ValidationMode = ValidationMode.ValidationEndpoint,

        RequiredScopes = new[] { "pis.initpay" },
        ClientSecret = "secret",
        ClientId = "pis"
    });
    Log.Logger.Information("APIEndpointASPNET finished booting up PIS Route Mapping.");
    authz.UseWebApi(config);
});
app.MapWhen(ctx => ctx.Request.Path.StartsWithSegments(new PathString("/ais")), authz => {
    authz.UseIdentityServerBearerTokenAuthentication(new IdentityServerBearerTokenAuthenticationOptions
    {
        Authority = "https://localhost",
        ValidationMode = ValidationMode.ValidationEndpoint,

        RequiredScopes = new [] { "ais.consacctlist", "ais.consacct", "ais.consacctbal", "ais.consaccttrans" },
        ClientSecret = "secret",
        ClientId = "ais"
    });
    Log.Logger.Information("APIEndpointASPNET finished booting up AIS Route Mapping.");
    authz.UseWebApi(config);
});
```

Figure 4.11: Implementation – Token Validation Request

4.4.1 Messages

As was stated previously, this Endpoint is the one which provides PSD2 compliance regarding the interfaces exposed for third-party developers to make use of. An example is shown in Figure 4.12, where it is also seen the existence of the "AuthorizationConfig" tag. Since an Access Token contains the information about its scopes, the API Endpoint knows if it is valid for a certain API Resource or not, but the underlying scope field is not validated during Token Validation. Therefore, a custom verification needed to be made by obtaining the scopes and matching it with the API that was requested to execute. This is possible because when an Access Token is successfully validated, information contained in the Token is passed to the API Endpoint.

To better provide documentation to the developers, a JSON Swagger document is generated in accordance with the implemented interfaces, their request and response parameters (header fields and body message). Even though it was mentioned that the connections are to be made in REST using JSON, with the way ASP.NET Web API works it lets this Endpoint accept requests and return responses that are in XML format as well. This is due to Web API providing built-in support for JSON and XML request and response messages. This means that it will automatically de-serialize or serialize the message it receives or sends. By doing this, it makes the system more inclusive to other third parties that do not use JSON, but rather use XML.

```
[Route("pis/initpay/{accountid}")]
[Authorize]
[AuthorizationConfig(ClaimType = "scope", ClaimValue = INITPAY)]
0 references | Kharmico, 7 days ago | 2 authors, 13 changes | 0 requests | 0 exceptions
public ResponseMessageResult InitPayPost(string accountid, [FromBody] PaymentModel payment)

public class PaymentModel
{
    ///
    1 reference | João Santos, 33 days ago | 1 author, 2 changes | 0 exceptions
    public string Amount { get; }
    ///
    1 reference | João Santos, 39 days ago | 1 author, 1 change | 0 exceptions
    public string CreditorName { get; }
    ///
    1 reference | João Santos, 39 days ago | 1 author, 1 change | 0 exceptions
    public string CreditorAccount { get; }
}
```

Figure 4.12: Implementation – PIS Example

Since the calls to BankOnBox Transaction Manager are done in XML via Simple Object Access Protocol (SOAP) (as was mentioned in Section 2.5) with specific parameters, the construction of a XML message is performed before every call. When a request to the API Endpoint is made, it asks the OpenID Endpoint for the "username", "bank account number" and Session Identifier. The "username" and "bank account number" elements are obtained by providing their unique identifiers, which in turn lead to an internal translation in the OpenID Endpoint. By having these three elements, the XML message is then constructed and requests are made (to get account details, list of movements or initiate payments). For simplicity, when requesting a list of movements, only the last 20 movements made are returned.

These transformations, or XML message construction, is done in the API Endpoint instead of the API Management Platform for three simple reasons: There is no defined WSDL which could be used to perform these changes automatically (either on the Management Platform with policies or the Endpoint); There is no direct translation from the received JSON messages to that XML construction; There is information kept in an on-premises Database (the OIDC

Database), which should not be sent or processed on the outside, regardless of security (by GDPR specifications, unless the User consented to, and Bank's concerns on keeping that information locally stored).

4.5 API Management Platform

The API Management Platform, even though the JSON to XML transformations are not present in the set policies, it still provides more capabilities to take advantage of.

There are six types of Policies, as was mentioned in Section 2.2.5, but not all were used. The policies that were used in this implementation were:

- It has a policy to check if an Access Token is present, in case it isn't then it immediately sends back a response stating this fact and sending a Header with the URL for the OpenID Endpoint.
- Policy verifying if the caller sent a Header with a GUID and is well-formed. This Transaction-ID is used to have a unique identifier for each request made to the API Endpoint and call made to the BankOnBox Transaction Manager.
- Throttles the number of calls permitted by 5 for every 10 seconds, specifically for the calling IP Address. This is to deny DDoS attempts towards the overall system.
- Regarding the throttle, there is another policy to set the maximum amount of calls per day, per IP Address, by 200. With the same reasoning behind it. If calls could still be made in 10second intervals non-stop, it would still be possible to succeed with a DDoS attack.
- There is a policy to log the requests that are made to the Platform, keeping information about who made the request and IP Address from where it came.
- Policy to send a certificate to the back-end for "Mutual Authentication" between the Platform and API Endpoint.
- Existence of API Subscription Keys to enforce that only registered Applications can make these calls.

The first policy on the list was made in regards to not forcing the third party developers the need to have the OpenID Endpoint's URL hard-coded in their application. It was also made to force the Applications to perform the Authentication and Authorisation mechanics to

get the Access Token without previously knowing the URL and only being able to get it if the Application was previously registered to access the API Management Platform.

4.6 Mobile Application

The Mobile Application was implemented in Xamarin.Forms, using IdentityModel.OidcClient2 as the library for the OIDC protocol interactions with the OpenID Endpoint. Having a Xamarin.Forms application lets developers make use of shared libraries and implement one application that works in Android, iOS and Windows Phone Operating Systems (OS). Some parts are still OS specific, but the core of the implementation lies solely in the Xamarin.Forms application.

This application was developed thinking of implementing, mainly, an Android application as a Proof of Concept. The library, OidcClient2, that was used was implemented in Xamarin. Since it uses a browser window to perform the login flow of the OIDC protocol, it was implemented in Android how to use this window. Apart from this, many of the needed steps for the protocol interaction are performed by this library. In the back, it implements the needed verifications and message building to be passed on to the OpenID Endpoint. In the end what developers see are the received tokens.

In this PoC, three different interfaces are called: List Bank Accounts, Get Bank Account Details and Performing a Payment. The first two are simple GET methods, while the third is a POST method. To perform the payment, a set Amount of money, a Creditor Name and Creditor Amount are needed, these values were randomly generated and hard-coded.

Chapter 5

Results

To evaluate the system's performance, functionality and security, a BankOnBox Sandbox was deployed in a Virtual Machine running in Microsoft Azure. This Sandbox was connected to both the API and OpendID Endpoints. These two Endpoints were compiled in "Release" mode and deployed in IIS. During the tests, no other programs or applications, other than the necessary minimum, were running in the Virtual Machine.

The Virtual Machine has the following specifications: 4GB of RAM, 128GB SSD, 2 vCPUs (Virtual CPUs), 1600 Maximum IOPs (Input/Outputs per second). The computer from where the requests were made has the following specifications: 8GB of RAM, 320GB HDD, i5-3320M 2.60GHz CPU. The internet connection used had a latency of 46ms, 115Mbps Download and 90Mbps Upload.

Part of these tests were executed via the API Management Platform, which had a SLA of 99.9%, scaling to a maximum of 2 active units and estimated maximum throughput of 1000 requests per second. It also has active policies to: Limit the rate of calls being made per second and day; Ensure that an Access Token is present; Sending a Client Certificate when connecting to the back-end to ensure "Mutual Authentication".

To perform the needed tests, a Mobile Application, Desktop MVC Application and API Testing tool were used. The MVC Application was used in making requests that were not implemented in the Mobile Application due to it being a PoC. The API Testing tool used was SoapUI [37].

There were three different types of tests for the system, related to functionality and security. The first type was composed of correctly made requests, to obtain how much time it took, on average, to correctly perform the intended actions. The second type is pertinent to erroneous requests, i.e. sending wrong login credentials, performing a request with a wrong bank account identifier, etc. The third type of tests were Load tests, i.e. executing multiple requests per

second. These Load Tests were performed via the API Management Platform, as well as directly to both API and OpenID Endpoints. This was to ascertain how well the system performs against a higher load of requests.

5.1 Correct Tests

This set, tests the system for when the given input is correct and well-formed. There was a total of 10 requests made per test with each having had its time taken and the average calculated. The times were measured in milliseconds (ms) and are presented in Table 5.1.

Each one of the tests corresponds to one of the features presented in Section 3.1. Following are the mentioned features, along with the expected response and what was obtained:

- Normal Two-Factor Authentication Login (time to input the login credentials, code and consent was not measured). Expected to successfully login and provide, for the application, access to accounts. See Figure A.17 for response.
- Consulting a given Account's Details. Expected to receive bank account details. See Figure A.18 for received response.
- Consulting a given Account's Balance. Expected to receive bank account balance. See Figure A.19 for received response.
- Consulting a given Account's Movements. Expected to receive list of bank account transactions. See Figure A.20 for received response.
- Consulting a List of Accessible Accounts. Expected to receive list of accessible accounts. See Figure A.21 for received response.
- Initiating a Payment. Expected to initiate a payment. See Figure A.22 for received response.
- Concluding a Payment. Expected to conclude a specific payment. See Figure A.22 for received response.

In the end, all of the responses gave the expected results, as is possible to see in the Figures presented in Annex A.

In Table 5.1, there are two instances of AccList. In AcctList(1) the request is made with the query parameter "withBalance" set to "false", whilst in AcctList(2) this parameter is set to "true". The difference in time is observable by noticing that it takes, on average, 140ms more to

API \ Req.	1	2	3	4	5	6	7	8	9	10	Avg
Login	1361	1313	1293	1258	1361	1268	1212	1348	1215	1386	1301.5
AcctDets	121	123	124	129	157	127	118	126	158	126	130.9
AcctBal	120	130	128	132	166	126	134	127	126	120	130.9
AcctTrans	128	130	131	162	128	131	133	143	137	128	135.1
AcctList(1)	126	88	148	93	95	84	102	89	90	84	99.9
AcctList(2)	247	229	249	233	227	233	240	249	244	248	239.9
InitPay	134	136	137	143	132	128	124	145	127	132	133.8
ConcPay	150	161	170	148	184	156	162	157	162	168	161.8

Table 5.1: Table of Performed Correct Tests. Measured in milliseconds.

execute AcctList(2). This difference is due to executing BankOnBox calls to obtain the Balance of each separate account.

The Login operation also takes longer to execute due to the amount of calls that are made to perform the Login flow. There are also calls being made to the BankOnBox Transaction Manager component during the Login flow, as was stated before in Section 3.3. The need to authenticate the End User, generating and validating the Two-Factor code, fetching the User's Bank Accounts and then save the consent information in the OIDC Database, it all weighs, in the end. Considering that Authentication and Authorisation are highly essential parts of this Thesis, then executing the Login flow in, an average of, 1301.5 milliseconds is most definitely acceptable.

The difference between the times of Initiating Payments and Concluding them is a result of when Initiating a Payment the information being kept in a BankOnBox component. Only when the given code to authorise such payment is valid will the data be passed on to the internal system and registered to be completed by the Bank.

With the presented times and taking into consideration the fact that internal translations, in regards to "Pseudonymisation", and calls to the BankOnBox Transaction Management component are done, then these response times are deemed acceptable. This conclusion came to be by verifying that the usual response times of, for example, Facebook APIs range from 80 to 150ms [38]. It was also taken into account that the APIs being tested are of the Financial type.

5.2 Erroneous Tests

After performing the previous tests, in which the system shows it is working correctly, some other tests were needed. This set is to show if the system correctly handles erroneous requests. To verify this, a few cases were tested.

The following list is from tests of requests made to the OpenID Endpoint during End User Authentication, along with the expected result and received response:

- Connecting to OpenID Endpoint with incorrect credentials, i.e. each Client Application has a "username" and "secret" which must be sent to the OpenID Endpoint when attempting to make a connection. Expected "invalid_client", received "invalid_client".

- End User Login credentials are incorrect. Expected "Invalid Credentials", Figure A.23 shows the received response (this tested by trying SQL Injection).

- Two-Factor Authentication Code is incorrect. Expected "Invalid Code", Figure A.24 shows the received response.

- No permissions chosen in Consent page. Expected "Must pick at least one permission", Figure A.25 shows the received response.

- Choosing a scope but not a bank account in Consent page. Expected "Must pick at least one permission", Figure A.25 shows the received response.

- Consent a bank account but not choosing a scope in Consent page. Expected "Must pick at least one permission", Figure A.25 shows the received response.

When performing the penultimate test an unexpected result happened. When choosing a Bank Account but not having chosen a scope, it would still validate the request. The problem was that the Access Token was not valid because it had no scope information in it. This was already fixed and tested again with the end result being what was expected in the beginning.

There was another test, aside from these, by making requests directly to the API Endpoint. Since it needs "Mutual Authentication" and no certificates were provided, the expected response was "403 Forbidden: Access is Denied", which is precisely what was obtained.

The following tests were done by making requests to the API Management Platform, along with the expected result and received response:

- Without subscription key, or an invalid one. Expected "Access denied", Figure A.26 shows the received response.

- With expired Access Token. Expected “401 Unauthorized”, Figure A.27 shows the received response.
- Without an Access Token present in the request. Expected “Valid Token Required”, Figure A.28 shows the received response (this is a policy in the Platform).
- Without a valid Transaction-ID header. Expected “Header Transaction-ID is missing or invalid”, which is exactly what was obtained (this is a policy in the Platform).
- Without Bank Account identifier. Expected “Resource Not Found”, which is the obtained response.
- Without Payment body message. Expected “Wrong Payment Information”, Figure A.29 shows the received response.
- Without Payment product for Completing Payment. Expected “Resource Not Found”, which is the obtained response.
- With wrong Payment product for Completing Payment. Expected “Wrong Payment Product”, Figure A.30 shows the received response.
- Without code for Completing Payment. Expected “Resource Not Found”, which is what was obtained.

In the end, there were two other tests in this set. These had their execution times taken and their average calculated. The tests were about: When an invalid Bank Account Identifier is given; When a wrong Code for completing a payment is provided. Both of these tests had the expected results, “Invalid Account Number” and “Wrong Code” respectively. The obtained results are show in Figures 5.1 and 5.2, respectively.

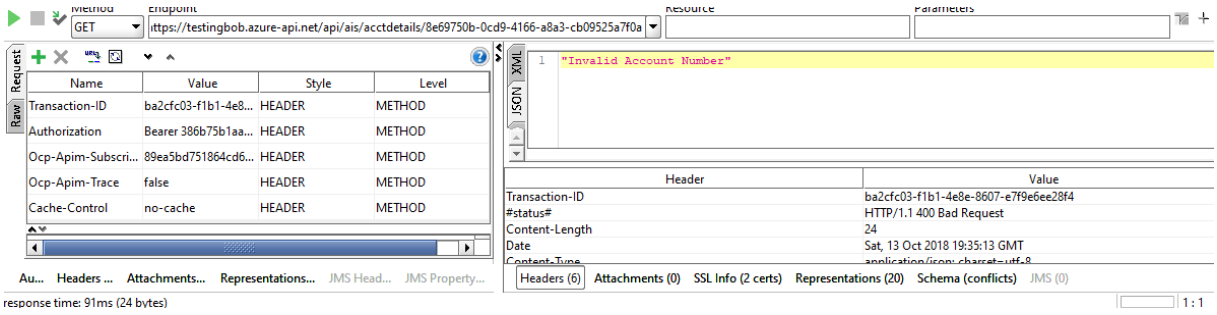

Figure 5.1: Erroneous Test – Invalid Account Number

The times and average, presented in Table 5.2 are lower than if the requests were correctly made because an error was encountered. This is due to the Bank Account Identifier translation

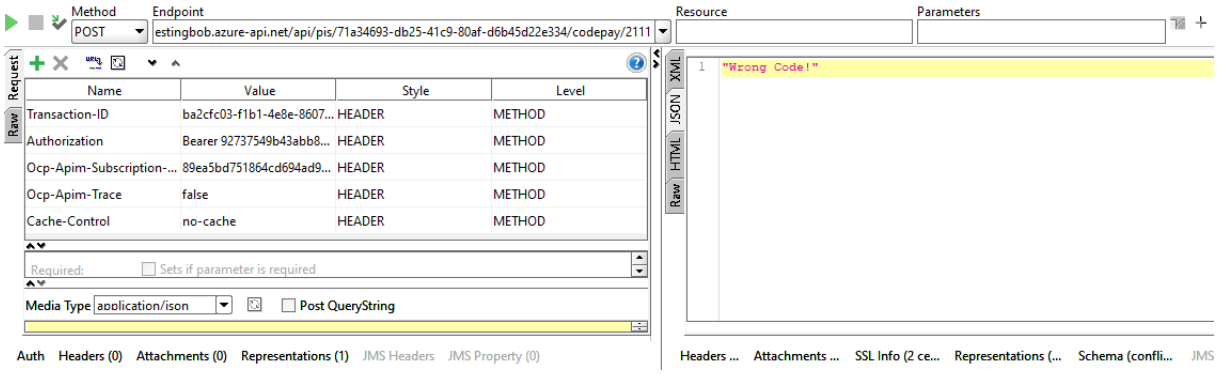

Figure 5.2: Erroneous Test – Wrong Payment Code

API \ Req.	1	2	3	4	5	6	7	8	9	10	Avg
WrongAcct	77	86	93	77	83	81	126	82	96	83	88.4
WrongCode	141	136	148	125	136	128	150	132	141	125	136.2

Table 5.2: Table of Performed Erroneous Tests. Measured in seconds.

not finding the corresponding Bank Account number (it does not exist), and with the Code being invalid it means that the payment is not processed.

5.3 Load Tests

A set of Load Tests was also performed to be sure if the system could handle a higher number of requests and how long it took to respond. The requests made on these tests were to Get Bank Account Details. It is worthy of remembering that the device where the system was running is not considered high-end to be able to process a high amount of requests per second, such as Facebook or Google.

When too many requests are made via the Platform, they start being actively refused. This is due to the existence of a policy, as was mentioned in Section 4.5, that does not permit to have too many requests in a short amount of time per IP Address (shown in action in Figures 5.3 and A.31). Not to mention the existence of a 2nd policy that also limits the amount of requests made per day, also per IP Address (shown in action in Figure A.32).

For this reason, during the rest of Load testing phase, these policies were turned off and only resumed after the tests were finished. Otherwise, Load Testing would have been impossible after having reached the daily quota of requests.

Apart from testing to be assured that the policies worked, four more Load Tests were executed. The first two were by performing requests to the API Management Platform. The second one by performing these requests directly to the API Endpoint. The third test was the same

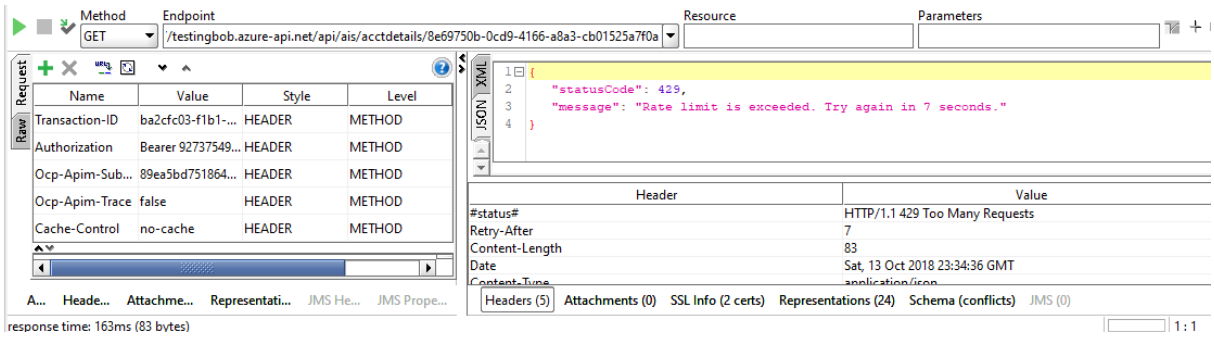

Figure 5.3: Erroneous Test – Exceeded Normal Rate

as the second one with the exception that "Mutual Authentication" on the API Endpoint was turned off.

These four Load tests were made by using SoapUI to do multiple requests per second. In the end of each Load Test it was possible to see the total of requests that got an answer, the minimum and maximum time in milliseconds of the test, along with the average of time (in milliseconds) taken and requests made per second.

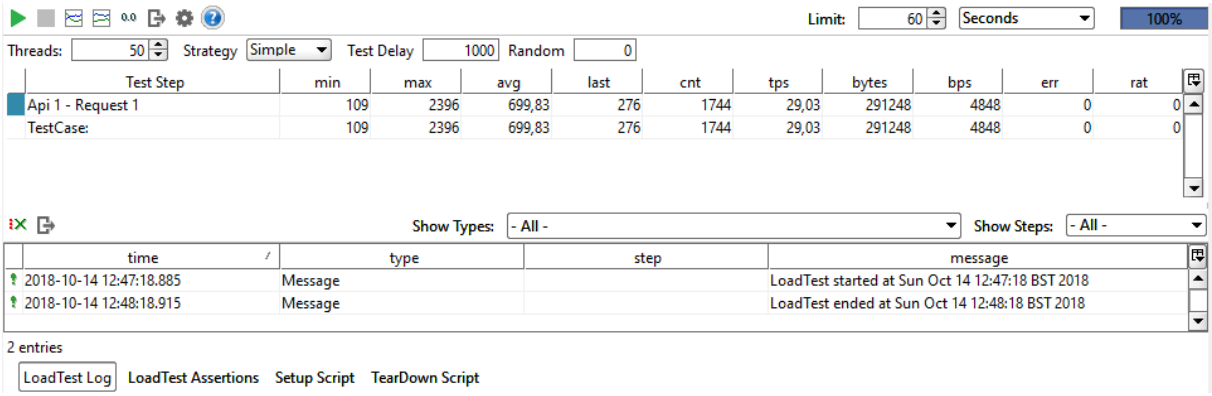

Figure 5.4: Load Test – Load Test 1

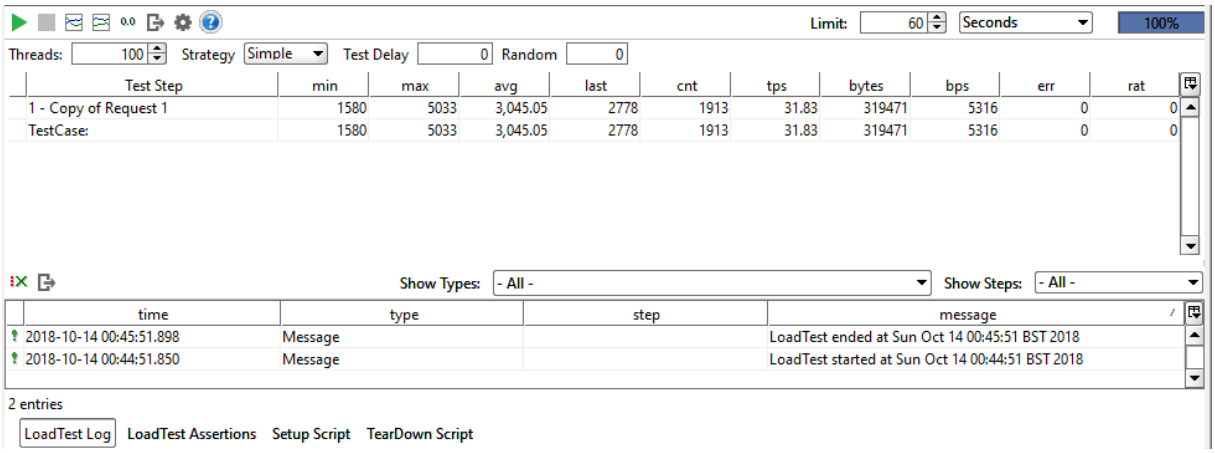

Figure 5.5: Load Test – Load Test 2

Figures 5.4 and 5.5 show the execution of two Load tests made to the API Management Platform. Since the system was under a higher load, it is shown, as was to be expected, that the average time taken to complete a request was higher. With both of these tests it was noticeable that the maximum time it could take for a request to complete was 5 seconds, with the highest average being 3 seconds, both from the 2nd test. From the 1st test these times were of 2.3 seconds and 699 milliseconds, respectively. But as can also be seen, the amount of requests per second in both tests is different as well. In the 1st it amounted to 29.03 requests, while in the 2nd it was 31.83, which helps bringing the definite conclusion that more requests bring a higher load which take more time to conclude.

This 3rd test, shown in Figure A.33, shows the execution being made directly to the API Endpoint while "Mutual Authentication" is still turned on. From this, it is only reasonable that the requests take time as low as 40milliseconds, with the maximum reaching 1.5 seconds, considering that in total 41430 requests were completed in 60 seconds (leading to 689 requests per second). This was due to the connections being immediately rejected.

This 4th test, shown in Figure A.34, was also executed directly to the API Endpoint, but in this case "Mutual Authentication" was turned off. From here, it is possible to see the stress that a higher load of requests does put directly to the system, since there is no connection to the API Management Platform. These appointed times lead to the same conclusions that were made from Load Tests 1 and 2.

All the times taken from these tests are, in our opinion, considered acceptable for the type of API that it is (Financial API) when under higher stress and in the environment described in the beginning of this Chapter.

Chapter 6

Conclusions

Changes to the E.U. legislation, namely GDPR and PSD2, have made Banks to start looking for ways to provide compliance with these new regulations. To address these changes, Banks have to expose their interfaces to interested third parties. Otherwise, there is a need to develop a set of APIs that provide said compliance by serving as an intermediary between the internal interfaces and TPPs. Banks are not the only entities looking for solutions, companies that offer Internet Banking products are doing the same. Such a company is Link Consulting, with their Internet Banking product, BankOnBox.

The idea behind this Thesis was to develop the set of APIs that provide compliance towards PSD2 and take into account GDPR requirements at the same time. After developing this set of APIs, it came to connect them with BankOnBox Transaction Manager to be able to perform the requests and applications access customer's bank accounts.

Since Authentication and Authorisation are a big concern in this type of systems, implementing the OIDC protocol to take care of these needs was important. It was possible, and easier, to implement this protocol by relying upon IdentityServer4 and its other related libraries (IdentityModel, OidcClient). With this protocol in place, it was possible to provide Authentication of the End User more reliably, with Applications being able to be assured that the User is duly authenticated. It was also possible to provide Authorisation by the use of Access Tokens which have information about the End User's consents in regards to the Client Application it is or was using.

All of the vital information is kept in an in-premises Database, where "Pseudonymisation" is also performed. With this in place, no sensitive or identifying information is provided to the outside. This Database is only accessible by the OpenID Endpoint.

By ensuring that all communications are made via HTTPS, it assures that the server authenticates itself, also providing integrity and confidentiality in the exchange of messages. In

the case between API Management Platform and API Endpoint, “Mutual Authentication” is in place to ensure that no malicious third parties can make a direct connection to the API Endpoint. Therefore only calls made via the API Management Platform are passed on to the Endpoint.

When the system was tested, it was seen that even when under a higher load of stress it still performed within expectations. Therefore meaning that even when it takes more time to reply, it does so correctly. This assured us that the system was correct, reliable and consistent in its responses.

6.1 Future Work

Even though the system was implemented, there are still improvements that can be made:

- Client Certificates in Client Applications. With this, the Applications can pass their certificates to the API Management Platform to authenticate themselves. Since certificates can be kept in the Platform, authenticating the application is easily done by using a simple policy expression. With the Client Certificate, it would also be made possible to provide “Mutual Authentication” when interacting with the OpenID Endpoint, therefore only allowing Applications with valid certificates to communicate with the Endpoint. With this it immediately rejects unauthenticated requests, providing better stability, reliability and higher availability of the system.

- Increasing the number of developed interfaces to augment the portfolio of provided services to be used by TPPs.

- Introducing a third element of Authentication in the Login flow, providing Three-Factor Authentication. The third element is about Inherence (something the user is), for example, a Fingerprint or Facial Recognition.

Bibliography

[1] "Mint," accessed: 14-10-2018. [Online]. Available: https://www.mint.com

[2] "Final Report - Draft Regulatory Technical Standards on Strong Customer Authentication and common and secure communication under Article 98 of Directive 2015/2366 (PSD2)," 2017.

[3] "CMA publishes updated issues statement in banking market investigation," accessed: 14-10-2018. [Online]. Available: https://www.gov.uk/government/news/cma-publishes-updated-issues-statement-in-banking-market-investigation

[4] European Union, "Regulation 2016/679 of the European parliament and the Council of the European Union," *Official Journal of the European Communities*, vol. 2014, no. March 2014, pp. 1–88, 2016.

[5] "Press Release – Open Banking revolution moves closer," accessed: 14-10-2018. [Online]. Available: https://www.gov.uk/government/news/open-banking-revolution-moves-closer

[6] "HM Treasury publishes report of the Open Banking Working Group," accessed: 14-10-2018. [Online]. Available: https://www.techuk.org/insights/news/item/7451-open-standard-application-programming-interface-api-in-banking

[7] "PSD2, CMA and OBWG – UK Open Banking today," accessed: 14-10-2018. [Online]. Available: http://digitalbaobab.com/psd2-cma-and-obwg-uk-open-banking-today/

[8] "PSD2 and OBS timeline comparison," accessed: 14-10-2018. [Online]. Available: http://digitalbaobab.com/wp-content/uploads/2016/09/openbankingtime_14152447_f4bbb7dcfeaf441089bce07a8ab5aba3a93296b0.pdf

[9]

[10] Office for Official Publications of the European Communities, "Payment Service Directive," *Official Journal of the European Union*, vol. 50, no. 319, pp. 1–36, 2007. [Online].

Available: http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2007:319:0001:0036:EN:PDF

[11] M. Cortet, T. Rijks, and S. Nijland, "PSD2: The digital transformation accelerator for banks," *Journal of Payments Strategy & Systems Volume 10 Number 1*, pp. 13–26, 2016.

[12] The European Commission, "Regulation - RTS SCA and Communication," *Official Journal of the European Union*, no. November 2017, 2018. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32018R0389{&}from=EN

[13] FinTech Manifesto, "Manifesto for the impact of PSD2 on the future of European Fintech," pp. 1–8, 2017. [Online]. Available: http://www.paymentscardsandmobile.com/wp-content/uploads/2017/05/Manifesto-for-the-impact-of-PSD2-on-the-future-of-European-Fintech.pdf

[14] "Fastest Growing API Categories since 2014," accessed: 14-10-2018. [Online]. Available: https://www.programmableweb.com/news/data-financial-analytics-api-categories-see-big-growth/research/2018/05/21

[15] "Most clicked APIs in 2017," accessed: 14-10-2018. [Online]. Available: https://www.programmableweb.com/news/programmablewebs-most-popular-apis-2017/research/2018/02/21

[16] "API Growth, 2005 to 2017," accessed: 14-10-2018. [Online]. Available: https://www.programmableweb.com/news/programmableweb-api-directory-eclipses-17000-api-economy-continues-surge/research/2017/03/13

[17] H. Hamad, M. Saad, and R. Abed, "Performance evaluation of restful web services for mobile devices," *Int. Arab J. e-Technol.*, vol. 1, pp. 72–78, 2010.

[18] "Swagger," accessed: 14-10-2018. [Online]. Available: https://swagger.io

[19] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," 2000. [Online]. Available: http://www.ics.uci.edu/{~}fielding/pubs/dissertation/top.htm

[20] M. Jones, D. Hardt, and Microsoft, "The OAuth 2.0 Authorization Framework: Bearer Token Usage," Tech. Rep. 6750, 2012. [Online]. Available: https://tools.ietf.org/html/rfc6750

[21] D. Hardt, "The OAuth 2.0 Authorization Framework," Tech. Rep. 6749, 2012. [Online]. Available: https://tools.ietf.org/html/rfc6749

[22] OpenID Foundation, "OpenID Connect Discovery 1.0 incorporating errata set 1," p. 311376, 2014. [Online]. Available: http://openid.net/specs/openid-connect-discovery-1{_}0.html

[23] "Truth behind the hubbuh over screen scraping," accessed: 14-10-2018. [Online]. Available: https://www.americanbanker.com/news/the-truth-behind-the-hubbub-over-screen-scraping

[24] "Fintech's defense of screen scraping is shortsighted," accessed: 14-10-2018. [Online]. Available: https://www.americanbanker.com/opinion/fintechs-defense-of-screen-scraping-is-shortsighted

[25] N. Sakimura and J. Bradley, "Proof Key for Code Exchange by OAuth Public Clients," Tech. Rep. 7636, 2015. [Online]. Available: https://tools.ietf.org/html/rfc7636

[26] "OpenBankProject," accessed: 14-10-2018. [Online]. Available: https://openbankproject.com

[27] "OAuth2 Authorization Flow description," accessed: 14-10-2018. [Online]. Available: https://github.com/OpenBankProject/OBP-API/wiki/OAuth-1.0-Server

[28] "OAuth2 Authorization Flow illustration," accessed: 14-10-2018. [Online]. Available: https://github.com/OpenBankProject/OBP-API/wiki/OAuth-Integration-Illustration

[29] "Which OpenID Connect/OAuth 2.0 Flow is the right One?" accessed: 14-10-2018. [Online]. Available: https://leastprivilege.com/2016/01/17/which-openid-connectoauth-2-o-flow-is-the-right-one/

[30] "Certified OpenID Connect Implementations," accessed: 14-10-2018. [Online]. Available: https://openid.net/developers/certified/

[31] "Monzo," accessed: 14-10-2018. [Online]. Available: https://monzo.com

[32] "Teller," accessed: 14-10-2018. [Online]. Available: https://teller.io

[33] "Revolut," accessed: 14-10-2018. [Online]. Available: https://www.revolut.com/pt/

[34] "IdentityServer4," accessed: 14-10-2018. [Online]. Available: http://docs.identityserver.io/en/release/

[35] "OpenID Connect Client Library for native mobile/desktop Applications," accessed: 14-10-2018. [Online]. Available: https://github.com/IdentityModel/IdentityModel.OidcClient2

[36] "Apigee is now part of Google," accessed: 14-10-2018. [Online]. Available: https://apigee.com/about/blog/digital-business/apigee-now-part-google

[37] "SoapUI," accessed: 14-10-2018. [Online]. Available: https://www.soapui.org/

[38] "Facebook - Platform Status," accessed: 14-10-2018. [Online]. Available: https://developers.facebook.com/status/dashboard/

Appendix A

Annex A: Figures

/obp/v3.1.0/banks/BANK_ID/accounts/private GET

Typical Successful Response:


```
{
  "accounts":[{
    "id":"5995d6a2-01b3-423c-a173-5481df49bdaf",
    "label":"String",
    "bank_id":"gh.29.uk",
    "account_type":"330",
    "account_routings":[{
      "scheme":"AccountNumber",
      "address":"4930396"
    }],
    "views":[{
      "id":"123",
      "short_name":"short_name",
      "description":"description",
      "is_public":false
    }]
  }]
}
```


Figure A.1: Gap Analysis – OBP - GetAccountsAtBank

/obp/v3.1.0/my/banks/BANK_ID/accounts/ACCOUNT_ID/account GET

Typical Successful Response:


```
{
  "id":"5995d6a2-01b3-423c-a173-5481df49bdaf",
  "bank_id":"String",
  "label":"String",
  "number":"String",
  "owners":[{
    "id":"5995d6a2-01b3-423c-a173-5481df49bdaf",
    "provider":"OBP",
    "display_name":"OBP"
  }],
  "type":"String",
  "balance":{
    "currency":"EUR",
    "amount":"10"
  },
  "account_routings":[{
    "scheme":"AccountNumber",
    "address":"4930396"
  }],
```


Figure A.2: Gap Analysis – OBP - GetAccountById

/obp/v3.1.0/my/banks/BANK_ID/accounts/ACCOUNT_ID/transactions GET

Typical Successful Response:


```
{
  "transactions":[{
    "id":"5995d6a2-01b3-423c-a173-5481df49bdaf",
    "this_account":{
      "id":"String",
      "bank_routing":{
        "scheme":"Bank_ID",
        "address":"gh.29.uk"
      },
      "account_routings":[{
        "scheme":"AccountNumber",
        "address":"4930396"
      }],
      "holders":[{
        "name":"OBP",
        "is_alias":true
      }]
    },
    "other_account":{
      "id":"5995d6a2-01b3-423c-a173-5481df49bdaf",
      "holder":{
        "name":"OBP",
        "is_alias":true
      },
      "bank_routing":{
        "scheme":"Bank_ID",
        "address":"gh.29.uk"
      },
      "account_routings":[{
        "scheme":"AccountNumber",
        "address":"4930396"
      }]
    },
    "details":{
      "type":"AC",
      "description":"OBP",
      "posted":"2017-09-19T00:00:00Z",
      "completed":"2017-09-19T00:00:00Z",
      "new_balance":{
        "currency":"EUR",
        "amount":"10"
      },
      "value":{
        "currency":"EUR",
        "amount":"10"
      }
    }
  }]
```


Figure A.3: Gap Analysis – OBP - GetTransactionsForAccount

/obp/v3.1.0/banks/BANK_ID/accounts/ACCOUNT_ID/VIEW_ID/transaction-request-type

{ "value":{ "currency":"EUR", "amount":"10" }, "to":{ "iban":"123" }, "description":" POST

Typical Successful Response:


```
{
  "id":"4050046c-63b3-4868-8a22-14b4181d33a6",
  "type":"SANDBOX_TAN",
  "from":{
    "bank_id":"gh.29.uk",
    "account_id":"8ca8a7e4-6d02-48e3-a029-0b2bf89de9f0"
  },
  "details":{
    "to_sandbox_tan":{
      "bank_id":"String",
      "account_id":"String"
    },
    "to_sepa":{
      "iban":"String"
    },
    "to_counterparty":{
      "counterparty_id":"String"
    "value":{
      "currency":"EUR",
      "amount":"100"
    },
    "description":"String"
  },
  "transaction_ids":["902ba3bb-dedd-45e7-9319-2fd3f2cd98a1"],
  "status":"COMPLETED",
  "start_date":"2017-09-19T00:00:00Z",
  "end_date":"2017-09-19T00:00:00Z",
  "challenge":{
    "id":"be1a183d-b301-4b83-b855-5eeffdd3526f",
    "allowed_attempts":3,
    "challenge_type":"SANDBOX_TAN"
  },
  "charge":{
    "summary":"Rent the flat",
    "value":{
      "currency":"EUR",
      "amount":"10"
    }
  }
}
```


Figure A.4: Gap Analysis – OBP - CreateTransactionRequest

Possible Errors:

- OBP-10023: obp_sort_direction parameter can only take two values: DESC or ASC!
- OBP-10024: wrong value for obp_offset parameter. Please send a positive integer (=>0)!
- OBP-10025: wrong value for obp_limit parameter. Please send a positive integer (=>1)!
- OBP-10026: Failed to parse date string. Please use this format yyyy-MM-dd'T'HH:mm:ss.SSS'Z'!
- OBP-20001: User not logged in. Authentication is required!
- OBP-30018: Bank Account not found. Please specify valid values for BANK_ID and ACCOUNT_ID.
- OBP-30005: View not found for Account. Please specify a valid value for VIEW_ID
- OBP-50000: Unknown Error.

Figure A.5: Gap Analysis – OBP - Possible Errors

GET GetAccount

Resource URL

Environment	URL
Sandbox	https://apis-bank-test.apigee.net/ais/open-banking/v1.0.1/accounts/123459876 ✔

Figure A.6: Gap Analysis – Apigee - GetAccountRequest

Resource Summary

Security	OAuth 2.0

Response | Errors

Property name	Type	Description
Data	array	
Links	object	Links relevant to the payload
Meta	object	Meta Data relevant to the payload


```
{
  "Data": {
    "type": "{string}",
    "title": "{string}",
    "items": {
      "type": "{string}",
      "description": "{string}",
      "title": "{string}",
      "properties": {
        "AccountId": {
          "type": "{string}",
          "description": "{string}",
          "minLength": "{number}",
          "maxLength": "{number}"
        },
        "Currency": {
          "type": "{string}",
          "description": "{string}",
          "pattern": "{string}"
        },
```


Figure A.7: Gap Analysis – Apigee - GetAccountResponse

Sample Error Messages & Error Codes.

Property	Type	Description
messages	array	Contains an array of JSON objects with error codes and description as shown above

HTTP Status Codes

Status Code	Description
200,201,202	Created Transaction request - OK.
400 Bad Request	Input Request is invalid or incorrect.
401 Unauthorized	Invalid API Key and Token.
404 Not Found	The requested resource does not exist.
500, 502, 503, 504 Server errors	Server Error at end.

Figure A.8: Gap Analysis – Apigee - Error Messages and Codes

Header Parameters

Name	Value	Description
x-fapi-financial-id *(required)*	123456789	The unique id of the ASPSP to which the request is issued. The unique id will be issued by OB.
x-fapi-customer-last-logged-time		The time when the PSU last logged in with the TPP.
x-fapi-customer-ip-address	10.20.30.40	The PSU's IP address if the PSU is currently logged in with the TPP.
x-fapi-interaction-id		An RFC4122 UID used as a correlation id.
Authorization		An Authorisation Token as per https://tools.ietf.org/html/rfc6750

Figure A.9: Gap Analysis – Apigee - Headers

POST CreateSingleImmediatePayment

Create a single immediate payment request

This API call requires **client_credentials access token** as Authorization header which can be obtained here.

All POST calls need a **x-jws-signature header** . Set the header by clicking "Create JWT button beside the x-jws-signature header field". This gets the payload from request body editor and creates the x-jws-signature for it.

Now you can try the API

Note the **PaymentId** which will be used to obtain access token for submitting the payment here.

Resource URL

Environment	URL
Sandbox	https://apis-bank-test.apigee.net/pis/open-banking/v1.0.1/payments ✔

Request Body

Value Description Schema


```
{
  "Data": {
    "Initiation": {
      "InstructionIdentification": "ACME412",
      "EndToEndIdentification": "FRESCO.21302.GFX.20",
      "InstructedAmount": {
        "Amount": "165.88",
        "Currency": "GBP"
      },
      "CreditorAccount": {
        "SchemeName": "SortCodeAccountNumber",
        "Identification": "08080021325698",
        "Name": "ACME Inc",
        "SecondaryIdentification": "0002"
      },
      "RemittanceInformation": {
        "Reference": "FRESCO-101",
        "Unstructured": "Internal ops code 5120101"
      }
    }
  },
  "Risk": {
    "PaymentContextCode": "EcommerceGoods",
    "MerchantCategoryCode": "5967",
    "MerchantCustomerIdentification": "053598653254",
    "DeliveryAddress": {
      "AddressLine": [
        "Flat 7",
        "Acacia Lodge"
      ],
      "StreetName": "Acacia Avenue",
      "BuildingNumber": "27",
      "PostCode": "GU31 2ZZ",
      "TownName": "Sparsholt",
      "CountySubDivision": [
        "Wessex"
      ],
      "Country": "UK"
```


Figure A.10: Gap Analysis – Apigee - CreateSingleImmediatePayment

CCNT	Pedido	
Campo	**Formato**	**Observações**
Header do pedido		
Classe de contas	A5	Se for recebido com valor, as contas devolvidas serão da classe recebida. **Ver anexo "Classes de Contas e Produtos"**
Grupo para consulta	A10	Define o grupo para o qual o utilizador pretende efectuar a consulta. Se não for recebido nenhum grupo serão enviadas todas as contas
Incluir saldos	A1	Especifica se na resposta devem ser devolvidos os saldos ou não. Assume os valores de N para não e S para sim. Esta transacção se for usada na personalização não precisa dos saldos, evitando assim um maior tempo de processamento, já que na Banca Remota residem as contas do utilizador (mas não os saldos se a aplicação estiver online)
Numero de ocorrências da página	N2,0	Numero de registos que se pretende receber até máximo de 20
Conta para paginação	N14,0	Conta a partir da qual se pretende nova página de contas.
Incluir contas saldadas	A1	N=Não inclui saldadas S=Inclui saldadas
Incluir contas encerradas	A1	N=Não inclui contas encerradas S=Inclui contas encerradas

Figure A.11: Gap Analysis – BOB - Consultar Contas Request

CCNT	Resposta certa	
Campo	**Formato**	**Observações**
Header da resposta		
Flag de paginação	N1,0	Indica se existem mais contas (para pedir nova página) 0=não existem mais contas, 1=existem mais contas
Numero de ocorrências	N2,0	Refere-se ao bloco seguinte. Até ao valor de página
Saldos incluídos	A1	Indica se foram incluídos saldos na resposta ou não
		Inicio de bloco
Conta14	N14,0	
Nome da conta	A35	Descrição que o utilizador associou à conta
Grupo da conta	A10	Grupo a que a conta pertence
Moeda da conta	A3	Moeda base da conta
Saldo contabilístico	N17,2	Saldo contabilístico da conta
Sinal do saldo contabilístico	A1	Sinal do valor. Pode ser "+" ou "-" significando credor ou devedor respectivamente
		Fim de bloco

Figure A.12: Gap Analysis – BOB - Consultar Contas Response

CMOV	Pedido	
Campo	**Formato**	**Observações**
Header do pedido		
Numero de conta	N14,0	Para canal SMS, se não for preenchida assume a conta por defeito.
Numero de ocorrências da página	N2,0	Numero de registos que se pretende receber até máximo de 20
Tipo de página	A1	Indica se se pretende a página seguinte ou a anterior em relação à chave de paginação recebida. Os valores possíveis são: + ← para página seguinte - ← para página anterior
Tipo de movimentos	A3	Tipo de movimentos que se pretendem. Pode assumir os valores: ALL ← todos os tipos
		BTL ← banca telefónica CHQ ← cheques CRE ← créditos DEB ← débitos INT ← Internet MB ← Multibanco SDD ← sistema débitos directos TFI ← transferências internas TFO ← transferências entre bancos
Data inicial de movimentos	N8,0	Data a partir da qual se pretendem os movimentos. Se não trouxer valor serão enviados os movimentos desde o inicio (dos que estão em disco online e que varia de banco para banco)
Numero de ordem do movimento	N7,0	Numero de movimento a partir do qual se pretendem os movimentos. Se este campo tiver valor a data inicial tem também de ter valor e servirá para enviar movimentos a partir do que aqui está referenciado
Data final de movimentos	N8,0	Data até à qual se pretendem os movimentos. Esta data apenas se aplica quando a página pedida é a seguinte (paginação para a frente)

Figure A.13: Gap Analysis – BOB - Consulta Movimentos Request

CMOV	Resposta certa	
Campo	Formato	Observações
Header da resposta		
Numero de conta	N14,0	
Saldo anterior	N17,2	
Sinal do saldo anterior	A1	
Saldo contabilistico	N17,2	
Sinal do saldo contabilistico	A1	
Saldo disponível	N17,2	
Sinal do saldo disponível	A1	
Moeda dos saldos	A3	
Flag de paginação	N1,0	Indica se existem mais movimentos (para pedir nova página) 0=Não existem + nem – movimentos 1=Existem + movimentos 2=Existem + e – movimentos 3=Existem – movimentos
Numero de ocorrências	N2,0	Refere-se ao bloco seguinte. Até ao valor de página
		Inicio de bloco
Data do movimento	N8,0	Data em que o movimento foi efectuado. Servirá para paginação
Numero de ordem do movimento	N7,0	Numero de ordem do movimento no dia. Servirá para paginação. Não deve ser afixado ao utilizador
Numero do documento	N9,0	Numero com que foi feito o lançamento. Este numero só adquire sentido para o utilizador se se tratar de um cheque.
Tipo de Documento	N3,0	Tipo de Documento é o elemento que agrega diversos códigos de operação.
Descrição do movimento	A40	
Valor da transacção	N15,2	Refere-se ao valor da transacção que o cliente efectuou e que pode ter sido numa divisa diferente da da conta. Por ex. Um levantamento em PTE sobre uma conta de EUR.
Sinal do valor	A1	
Divisa da transacção	A3	
Valor que afectou a conta	N15,2	Refere-se ao valor que afectou a conta e que pode não ter sido o valor da transacção do cliente se a conta tiver dual base currency
Sinal do valor	A1	
Divisa do movimento	A3	
Saldo após o movimento	N17,2	
Sinal do saldo	A1	
Data valor	N8,0	
Data de extracto	N8,0	Indica a data no caso do movimento já ter sido comunicado em extracto
Com imagem associada	A1	Indica se o movimento tem uma imagem associada (exemplo no debito de um cheque pode estar associada a imagem do cheque) . Esta imagem pode ser consultada através de transacção própria de consulta de imagem. N=Não S=Sim
Número de operação	N9,0	Numero de operação com que foi feito o lançamento. Este numero pode estar repetido em diversos movimentos caso pertençam à mesma operação.
		Fim de bloco

Figure A.14: Gap Analysis – BOB - Consulta Movimentos Response

TSEP	Pedido	
Campo	**Formato**	**Observações**
Header do pedido		
Conta a debitar	N14,0	Numero da conta à Ordem que pertence à entidade subscritora do contrato e que será debitada. Esta conta é uma das devolvidas na transacção de SignOn
Valor a debitar	N15,2	
Moeda do valor	A3	
Descrição do movimento na conta a debitar	A40	
BIC banco destinatário	A11	
IBAN da conta destinatário	A34	IBAN que identifica o numero de conta na outra instituição
Nome do destinatário	A70	Nome do titular da conta no outro banco
Informação adicional	A140	Este elemento é de caracter opcional e quando definido, pode ser preenchido com informação estruturada ou não estruturada, para a qual devem ser consultadas as normas SEPA.
Tipo de transferência	A1	Define o critério de execução da transferência. Pode ter os valores: E ¢ eventual, apenas é executada uma vez P ¢ permanente, é executada até uma data final ou cancelamento
Data da transferência	N8,0	Data em que a transferência será enviada ao outro banco. Se não for definida data, a transferência será enviada na próxima compensação
Data de vencimento	N8,0	Aplica-se para permanentes
Periodicidade da transferencia	A1	Aplica-se para permanentes
Email para notificação	A80	Endereço de email para o qual será enviada uma mensagem a avisar da execução da transferencia. O mail é enviado apenas para as transferencias com execução imediata.

Figure A.15: Gap Analysis – BOB - Pedido Transferência SEPA Request

TSEP	Resposta certa	
Campo	**Formato**	**Observações**
Header da resposta		
Numero de transferência atribuído	N9,0	Estes dados só estão presentes se a transacção não ficar sujeita a confirmação por parte do banco ou do cliente O campo STATUS do header de resposta indica o modo de finalização da transacção
Valor de despesas a debitar	N15,2	
Moeda das despesas	A3	

Figure A.16: Gap Analysis – BOB - Pedido Transferência SEPA Response

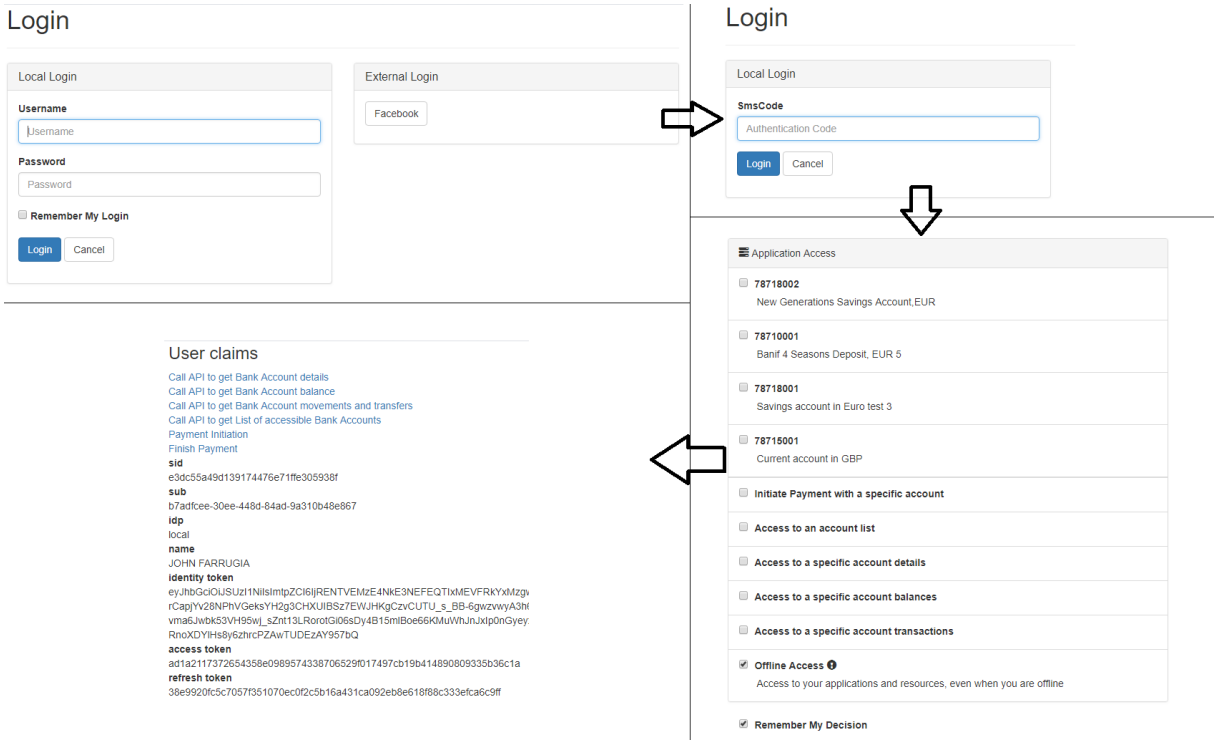

Figure A.17: Correct Test – Login Flow

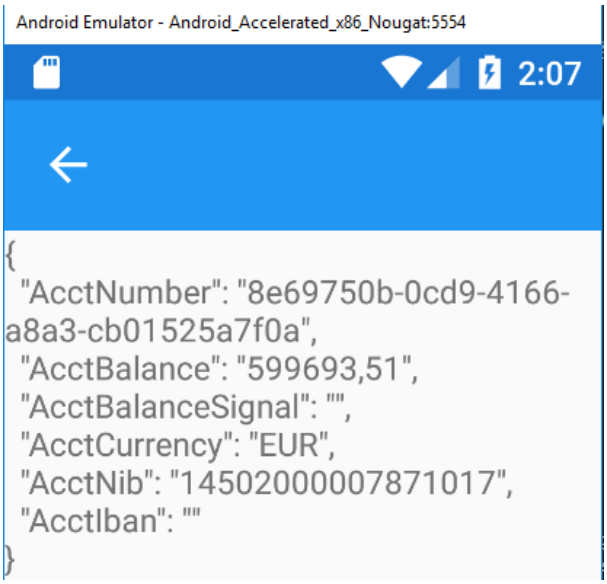

Figure A.18: Correct Test – Account Details

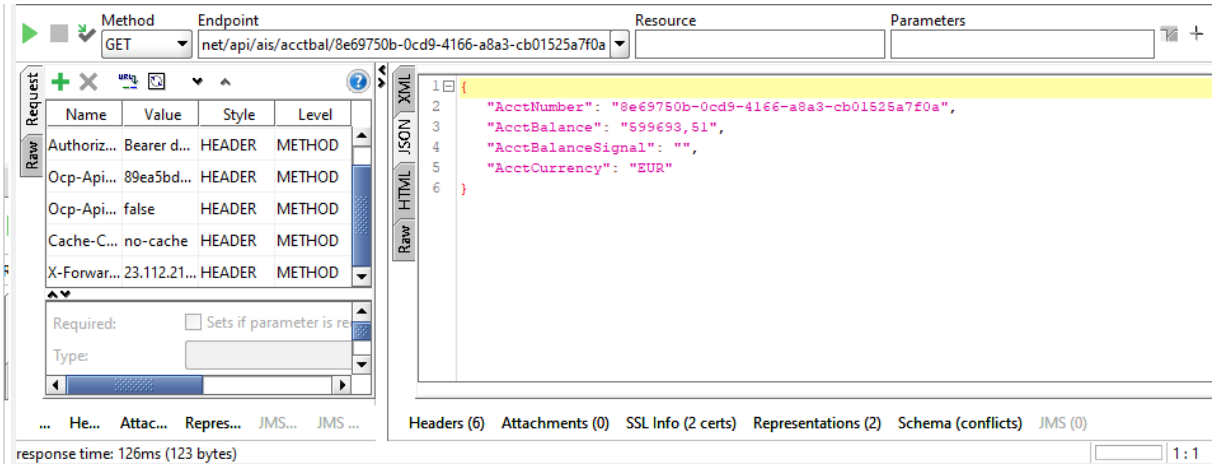

Figure A.19: Correct Test – Account Balance

Figure A.20: Correct Test – Account Transactions

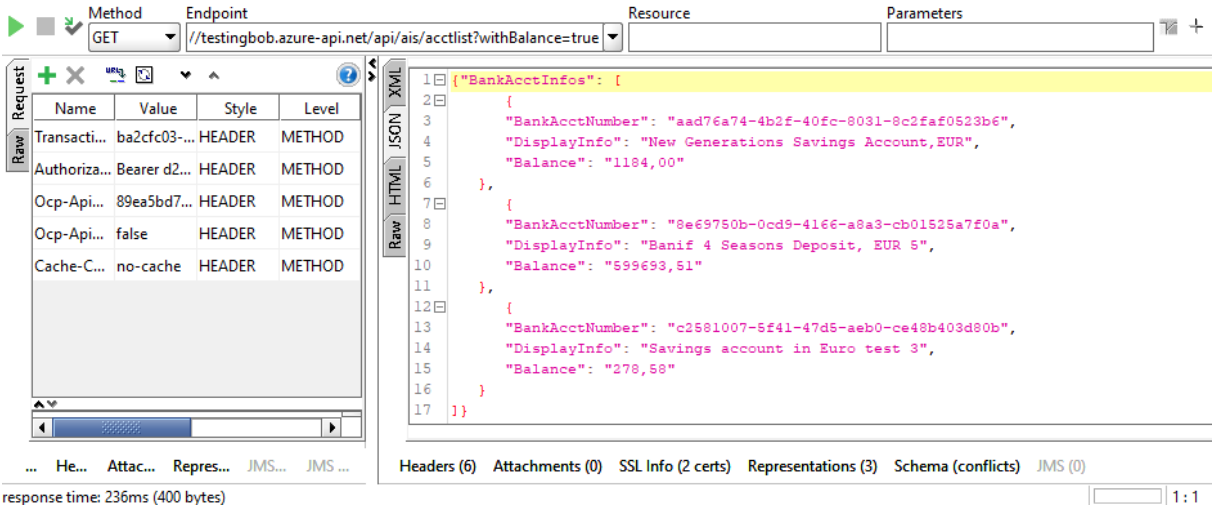

Figure A.21: Correct Test – Account List

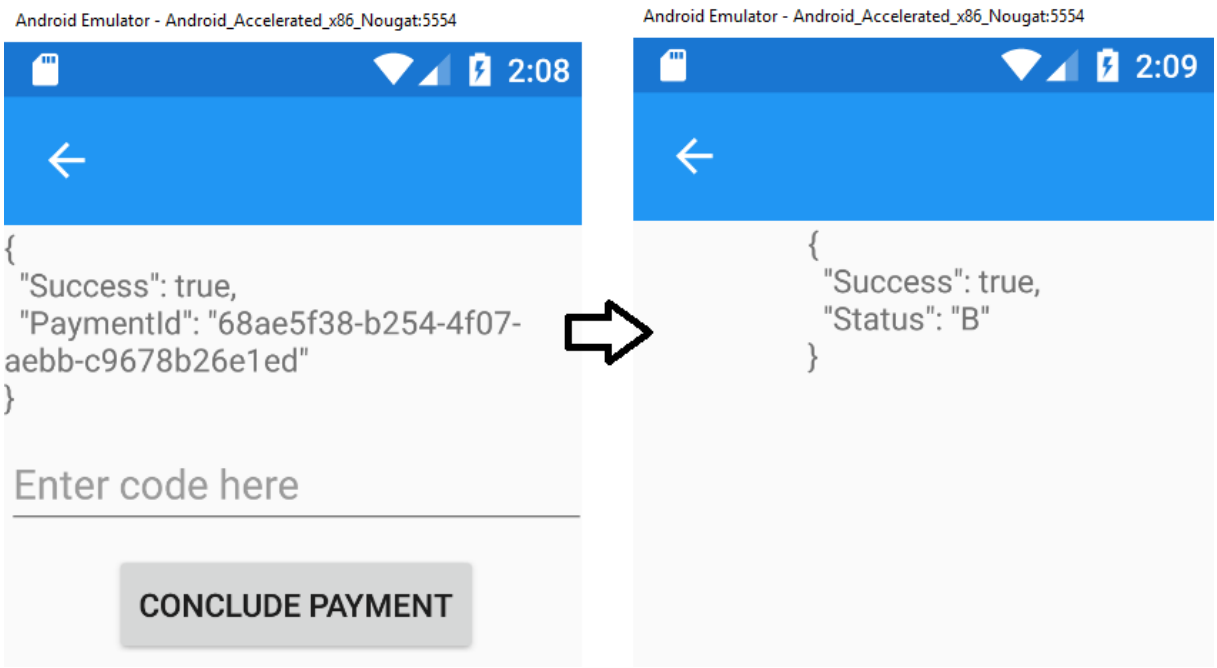

Figure A.22: Correct Test – Initiate Payment

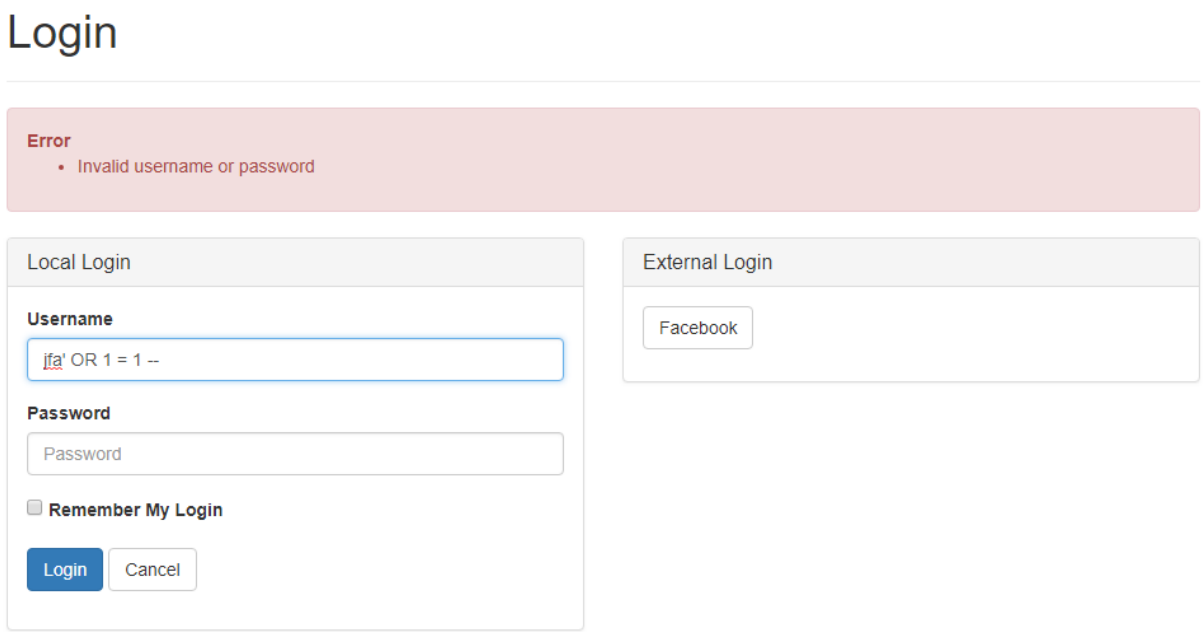

Figure A.23: Erroneous Test – Wrong credentials

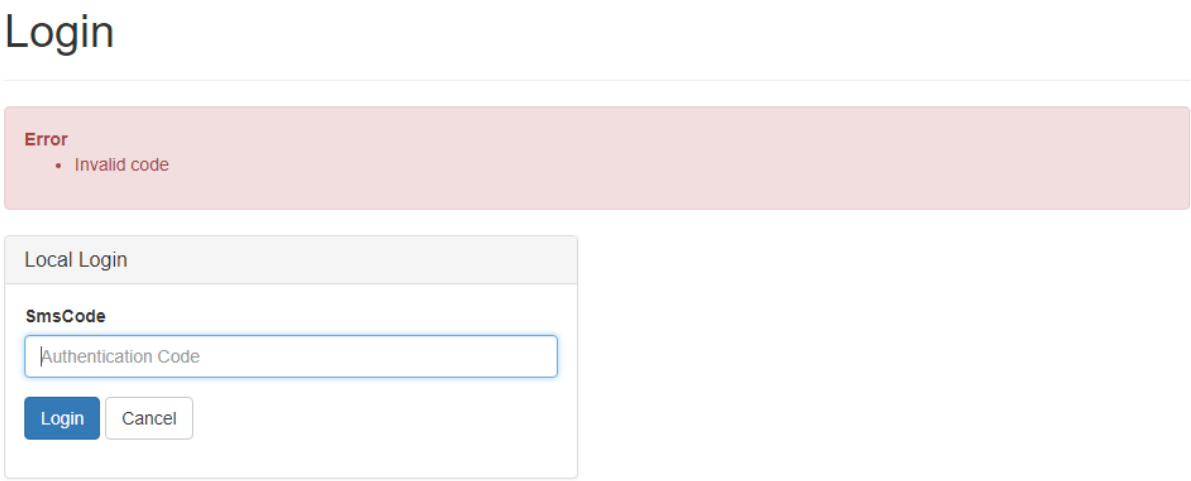

Figure A.24: Erroneous Test – Wrong login code

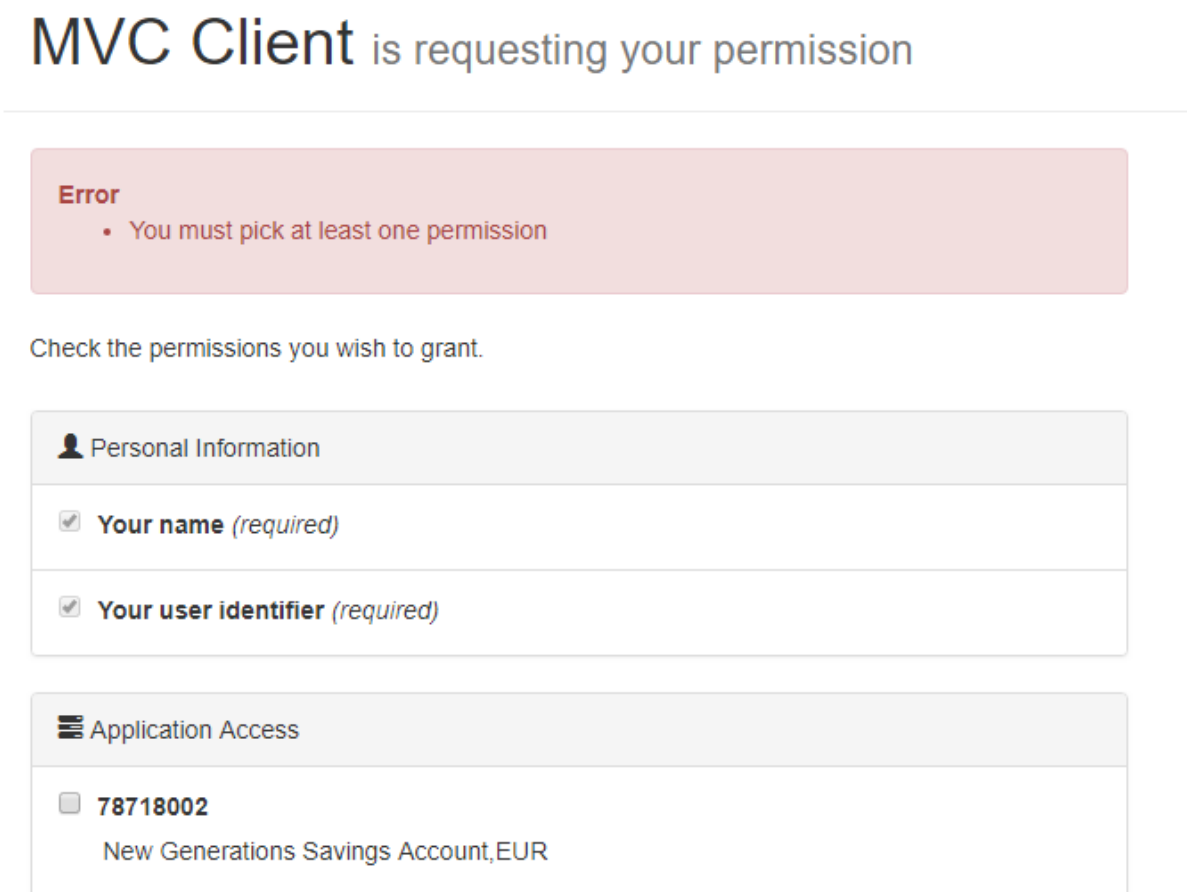

Figure A.25: Erroneous Test – Invalid Consent

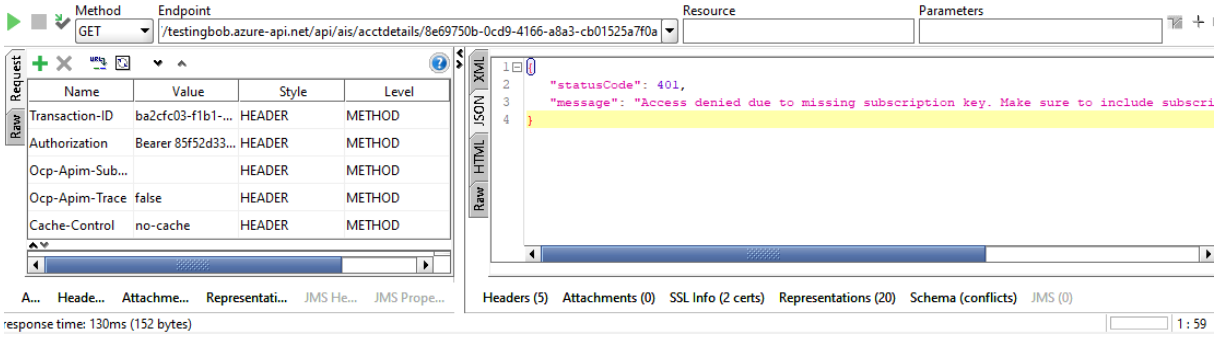

Figure A.26: Erroneous Test – Without subscription key

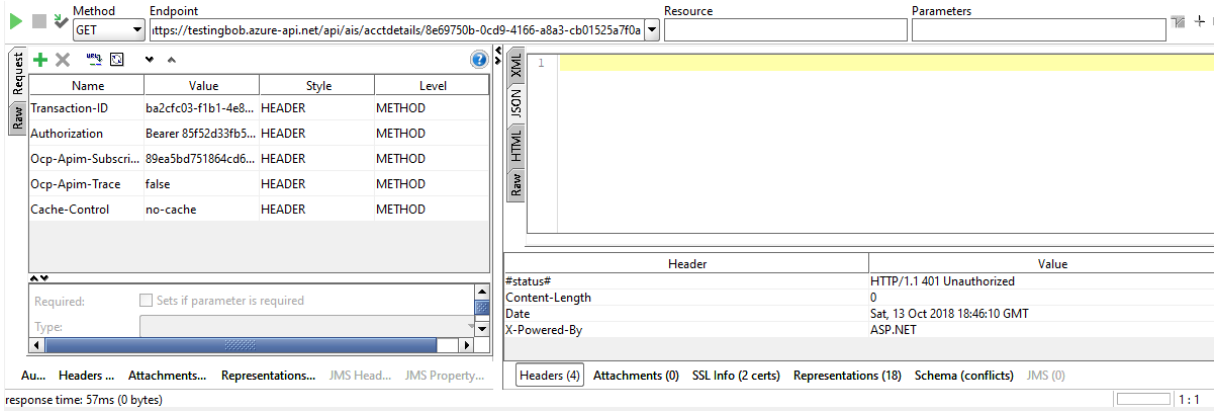

Figure A.27: Erroneous Test – Expired Token

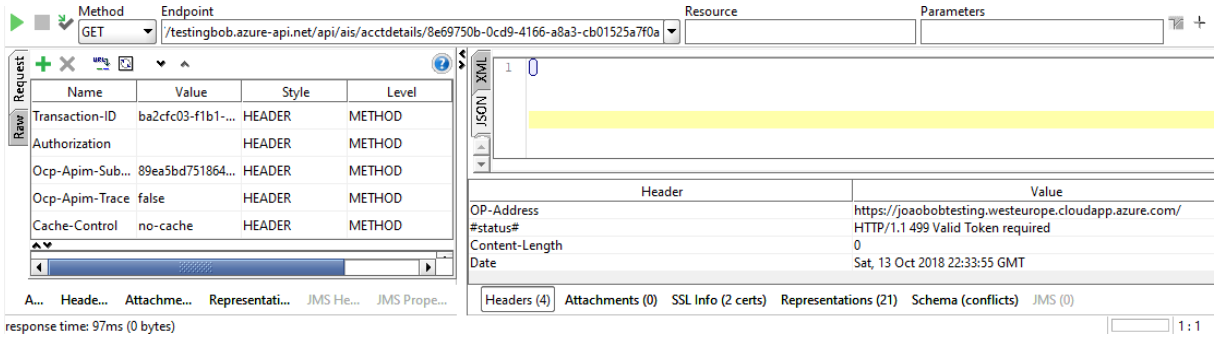

Figure A.28: Erroneous Test – No Token

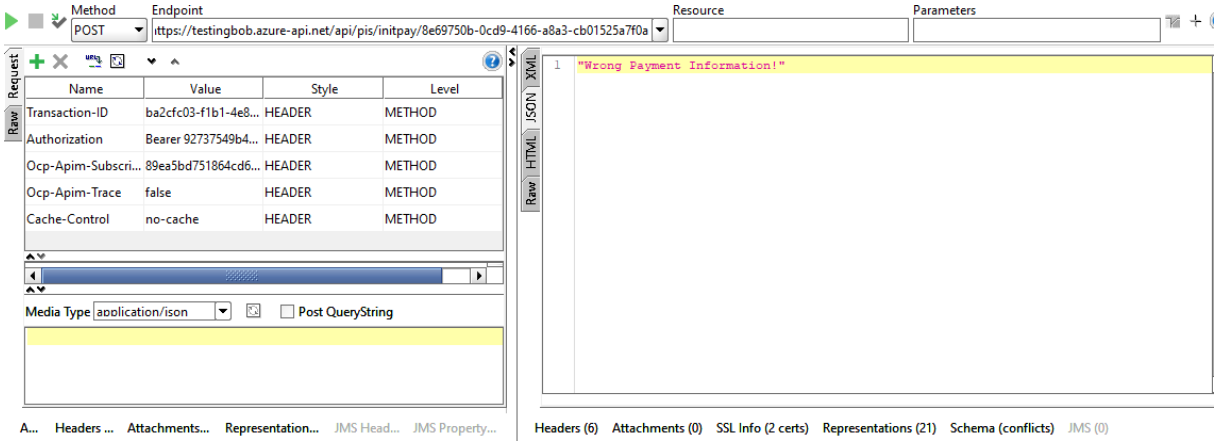

Figure A.29: Erroneous Test – Wrong Payment Information

Method POST
Endpoint testingbob.azure-api.net/api/pis/71a34693-db25-41c9-80af-d6b45d22e33/codepay/1111
Resource
Parameters

Name	Value	Style	Level
Transaction-ID	ba2cfc03-f1b1-4e8e-8607...	HEADER	METHOD
Authorization	Bearer 92737549b43abb8...	HEADER	METHOD
Ocp-Apim-Subscription-...	89ea5bd751864cd694ad9...	HEADER	METHOD
Ocp-Apim-Trace	false	HEADER	METHOD
Cache-Control	no-cache	HEADER	METHOD

Required: Sets if parameter is required
Media Type application/json Post QueryString

```
"Wrong Payment Product!"
```

Auth Headers (0) Attachments (0) Representations (1) JMS Headers JMS Property (0)
Headers ... Attachments ... SSL Info (2 ce... Representations (... Schema (confli... JMS

Figure A.30: Erroneous Test – Wrong Payment Product

Limit: 60 Seconds 100%
Threads: 100 Strategy Simple Test Delay 0 Random 0

Test Step	min	max	avg	last	cnt	tps	bytes	bps	err	rat
1 - Copy of Request 1	40	1599	98.97	109	41430	689.89	3440402	57289	0	0
TestCase:	40	1599	98.97	109	41430	689.89	3440402	57289	0	0

Show Types: - All - Show Steps: - All -

time	type	step	message
2018-10-14 00:36:42.226	Message		LoadTest started at Sun Oct 14 00:36:42 BST 2018
2018-10-14 00:37:42.228	Message		LoadTest ended at Sun Oct 14 00:37:42 BST 2018

2 entries
LoadTest Log LoadTest Assertions Setup Script TearDown Script

Figure A.31: Load Test – Quota Policy turned on

Method GET
Endpoint /testingbob.azure-api.net/api/ais/acctdetails/8e69750b-0cd9-4166-a8a3-cb01525a7f0a
Resource
Parameters

Name	Value	Style	Level
Transaction-ID	ba2cfc03-f1b1-...	HEADER	METHOD
Authorization	Bearer 08926a9a...	HEADER	METHOD
Ocp-Apim-Sub...	89ea5bd751864...	HEADER	METHOD
Ocp-Apim-Trace	false	HEADER	METHOD
Cache-Control	no-cache	HEADER	METHOD

Required: Sets if parameter is required

```
{
    "statusCode": 403,
    "message": "Out of call volume quota. Quota will be replenished in 00:17:18."
}
```

Header	Value
#status#	HTTP/1.1 403 Quota Exceeded
Retry-After	1038
Content-Length	100
Date	Sat, 13 Oct 2018 23:42:42 GMT
Content-Type	application/json

A... Heade... Attachme... Representati... JMS He... JMS Prope...
Headers (5) Attachments (0) SSL Info (2 certs) Representations (24) Schema (conflicts) JMS (0)
response time: 49ms (100 bytes) 1 : 1

Figure A.32: Load Test – Exceeded Daily Quota

Limit: 60 Seconds 100%

Threads: 100 Strategy Simple Test Delay 0 Random 0

Test Step	min	max	avg	last	cnt	tps	bytes	bps	err	rat
1 - Copy of Request 1	40	1599	98.97	109	41430	689.89	3440402	57289	0	0
TestCase:	40	1599	98.97	109	41430	689.89	3440402	57289	0	0

Show Types: - All - Show Steps: - All -

time	type	step	message
2018-10-14 00:36:42.226	Message		LoadTest started at Sun Oct 14 00:36:42 BST 2018
2018-10-14 00:37:42.228	Message		LoadTest ended at Sun Oct 14 00:37:42 BST 2018

2 entries

LoadTest Log LoadTest Assertions Setup Script TearDown Script

Figure A.33: Load Test – Load Test 3

Limit: 60 Seconds 100%

Threads: 50 Strategy Simple Test Delay 0 Random 0

Test Step	min	max	avg	last	cnt	tps	bytes	bps	err	rat
Api 1 - Request 1	490	3315	1.544,57	1274	1912	31,75	319304	5302	0	0
TestCase:	490	3315	1.544,57	1274	1912	31,75	319304	5302	0	0

Show Types: - All - Show Steps: - All -

time	type	step	message
2018-10-14 13:31:25.446	Message		LoadTest started at Sun Oct 14 13:31:25 BST 2018
2018-10-14 13:32:25.612	Message		LoadTest ended at Sun Oct 14 13:32:25 BST 2018

2 entries

LoadTest Log LoadTest Assertions Setup Script TearDown Script

Figure A.34: Load Test – Load Test 4