Secure Model of Web APIs development in the Cloud

João Miguel Correia de Sousa Silva Santos
joao.c.santos@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal
November 2018

Abstract

Banks store important client information, making this one of the most sensitive types of information that currently exists. With the needs of today, specific 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

1. Introduction

Recent changes in European legislation, namely Second Payment Services Directive (PSD2) and General Data Protection Regulation (GDPR), are forcing Banks to be more open and careful with how they share and process information regarding third parties. This has made Banks to look into solutions to provide PSD2 compliance while still having a secure system with Authentication and Authorisation mechanisms in place. This means that Banks and companies that provide Banks with Internet Banking products are the more interested parties in these changes.

A company such as this is Link Consulting, since they have an Internet Banking product called BankOnBox, which now needs to provide PSD2 compliance. Requiring an analysis of PSD2 and any provided standards or other documentation is necessary.

Henceforth bringing to the conclusion that the main 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 BankOnBox as a fully PSD2 compliant product. 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 Application Programming Interface (APIs), exchanged information (request-response messages), and so on.

- After the two previous analyses, a gap analysis between Bank On Box and PSD2 is to be done 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 to make it fully compliant 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.

2. Background
The experience provided by the community around this subject, introduced in Section 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 Financial Technology (FinTech) industry, API Management and Security, as well as ongoing projects regarding the subject in question.

2.1. PSD2
The main focus of this Thesis, therefore, remains PSD2, because it concerns Banks in, or that provide services within, the E.U. PSD2 is the revised Payment Services Directive which came to fruition due to the needs of changing, more rapidly, how Institutions, Bank or FinTech company can turn out a solution to the problem at hand.

Regulatory Technical Standard (RTS) [8] associated with it, which mandates that all Banks have to provide a dedicated set of interfaces to give the needed openness, and to do so in a secure manner. It also states that Users have to authenticate before the Bank with at least two out of three types of Multi-Factor Authentication: Knowledge, Possession, Inherence. Aside from these requirements, no Standards or more specific information is given.

With PSD2 there is a relatively significant 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.

Regarding European legislations on the matter, there is more to it than just PSD2. There is also another set of important regulations called GDPR [6] which demands to provide the customers with information about how their data is saved, where it is processed and by who, and to securely protect it.

2.2. API – Application Programming Interface
APIs are of the utmost importance, because that is the way in which the BankOnBox internal system will be able to communicate with the third parties.

PSD2 regulations ask for the development of dedicated interfaces that Third Party Provider (TPP) 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.

Since the interfaces that are asked to be developed per PSD2 legislation are to be used in an online 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). There have been found different ways on how to secure Web APIs, whether during information exchange as well as protecting access to the APIs and the information behind.

Authentication and Authorisation are essential for a client->server communication and 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.

OAuth2.0 and OpenID Connect (OIDC) are protocols that provide Authentication and Authorisation, with the former [OAuth] being just an Authorisation framework, but the latter [OIDC] providing Authentication on top it (since OIDC works on top of OAuth2.0).

OAuth2.0 has four different types of Grants: Authorisation Code, Implicit, Password and Client Credentials. The one that will be explained is considered to be the most secure one, Authorisation Code Grant. Not only because of it having an extra step to exchange a code for an access token, but because it requires that the application first launches a browser window to begin the flow. It has been considered best practice to use Authorisation Code Grant along with an extension of OAuth2.0 called Proof Key for Code Exchange (PKCE) [7], which is a technique to secure public Clients.

An example of the OAuth authorisation code grant can be seen in the Figures 1, 2, 3 and 4 along
with a brief description. This was based in the Open Bank Project [4] use of the OAuth2 implementation.

![OAuth2 Authorization Flow request](image1)

Figure 1: OAuth2 Authorization Flow request token

![OAuth2 Authorization Flow consent](image2)

Figure 2: OAuth2 Authorization Flow request consent

The Authorisation Code is done by having the Client contact the Authorisation Server requesting access to information of the End User (Figure 1). 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 [7]) and “state” parameters in the Uniform Resource Locator (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. Afterward, 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. This “state” needs to be the same as the one received in the initial request to prevent Cross-Site Request Forgery (CSRF) attacks (Figure 2). Afterward, 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 3). 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 information within its scope of consent (Figure 4). In every exchanged message there is also a “nonce” parameter to prevent Replay Attacks.

The mentioned PKCE [7] 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 with the rest of the parameters on the 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. If the Authorisation Code is intercepted it will not be useful since the token request relies on the initial “code challenge” secret. This is used to prevent the Authorisation Code Interception Attack.

Since OAuth2.0 only provides Authorisation, there is the need for OpenID Connect to provide Authentication. OIDC works by delivering Identity and Access Tokens to the Applications. These Identity Tokens provide the Authentication because the End User has to perform a login logic with the Bank for the Application to get this Token. At the same time it receives the Identity Token, it also gets the Access Token that provides Authorisation to access the User’s Bank Account information. This information is only accessible if the User has given his consent to do so.

To better provide these mechanisms, instead of manually implementing the OIDC protocol, some libraries help on it, one being IdentityServer4. These libraries take care of the underlying mechanics of creating and validating the tokens, as well as performing the flow of the protocol.

2.3. BankOnBox
Knowing about BankOnBox was also needed. In that note, BankOnBox is a solution, for Financial Institutions in order 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 5 is the currently used one and was developed using Microsoft technology (APS.Net Framework 4.5, Microsoft SQL Server).

As can be seen in Figure 5, 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 eXtensible Markup Language (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 in relation to 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).

2.4. Gap Analysis
Since there is no Standard, researching for other existing solutions provided some needed insight. With this, we came upon OpenBankProject and Apigee, each with their own implementation of a PSD2 compliant platform. Both 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), as well as which interfaces are to be implemented. The ones pertinent to this Thesis are related to Accessing a Bank Account’s Details, Balance, Movements, as well as seeing a List of Accessible Accounts and Initiating Payments.

2.5. Cloud Service Provider
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 Cloud Service Provider (CSP) is also nothing new, especially considering that Cloud Computing has seen an increasing growth over the last few years (Fig-
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 [1]). 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 community, provided documentation, tutorials, guides, as well as how easy it is to integrate with other products from the CSP.

3. Architecture and Implementation

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. It was defined which components the system has and a database model where needed information is saved.

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. 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.

Following is the list of features to be provided: Authentication of the User towards the Bank: 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 in order to Initiate Payments.

3.1. Architecture Decisions

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.

OAuth2.0 and OIDC were already described in Section 2.2 and as was noted, each has their own specific quirks, 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.

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 Representational State Transfer (REST)ful manner using JavaScript Object Notation (JSON) formatted messages. Due to it being easier, simpler and lighter on both applications and servers.

3.2. Architecture Components

Each component is crucial for the whole system to work and to provide the needed security and functionality to the system (see Figure 7).

The API Endpoint 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

![Figure 6: Cloud Computing Growth](image)

![Figure 7: System Architecture](image)
and in the end a response is returned. The connection between the API Management Platform and API Endpoint is made with “Mutual TLS” (also known as “Mutual Authentication”).

In the OIDC Database 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 OpenID Endpoint 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 Authorization. It also offers the possibility of external logins, i.e. performing a login with an external identity provider like Google or Facebook by making an association between their Internet Banking Account and External Provider Account.

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 Service Level Agreement (SLA), depending on the chosen Tier, and an increased amount of requests per second.

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.

3.3. Implementation Decisions
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

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 conventions related to these two elements (REST and JSON). By putting these conventions into practice, it forces applications to follow them as well.

The OIDC protocol Flow decided to use was the Authorisation Code Flow, even though 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.

Tokens can turn out to have a big payload, therefore increasing the data going through the network and the size of the URL. In attempts to keep it light and shorter for all entities, the Access Tokens are of Reference type. This also makes it easier to have more fine-grained control over them since Self-Reference JSON Web Token (JWT) are only invalid when they expire, but Reference Tokens are invalid from the moment they are deleted from the Database as well.

In regards to the BankOnBox component, its implementation only lets a user perform requests for 30 minutes after login. Therefore it was needed to implement a way 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 in the response after Authentication was successfully performed. Afterward, this Session Identifier would need to be saved in the Database, such that it could only 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.

3.4. Implementation Components
The OpenID Endpoint 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. 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 8) 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.

```csharp
 demonstrations at its servers. User: Where (x -> x.Name.Equals(name)).FirstOrDefault()?
```

Figure 8: Example of LINQ query for Database

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 authorization 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.

The API Endpoint is an 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. 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”.

A set of policies was defined in the API Management Platform, which are as follow: A policy to check if an Access Token is present; A 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; Throttles the amount of calls permitted by 5 for every 10 seconds, specifically for the calling IP Address; Throttle for a maximum amount of calls per day, per IP Address, by 200; Logging of 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 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. 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.

The difference between using ASP.NET Core in OpenID Endpoint and ASP.NET Framework in API Endpoint relied on the fact that IdentityServer4 is a library specifically for ASP.NET Core. Although there is IdentityServer3, developed for ASP.NET Framework, it is no longer under development nor being maintained, while IS4 is under
constant development and maintenance.

4. 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 OpenID 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 [5].

There were three different types of tests for the system, related to functionality and security. The first set is composed of correct requests, in order to obtain how much time it takes, on average, to perform the intended actions. The second set is pertinent to erroneous requests, i.e. from sending wrong login credentials to performing a request with the wrong Bank Account identifier. The third set of tests is about 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.

4.1. 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 1.

Table 1: Table of Performed Correct Tests. Measured in seconds.

<table>
<thead>
<tr>
<th>API</th>
<th>Login</th>
<th>AcctDets</th>
<th>AcctBal</th>
<th>AcctList</th>
<th>AcctTrans</th>
<th>InitPay</th>
<th>ConcPay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg.</td>
<td>1301.5</td>
<td>130.9</td>
<td>130.9</td>
<td>239.9</td>
<td>135.1</td>
<td>133.8</td>
<td>161.8</td>
</tr>
</tbody>
</table>

AcctList has a parameter, “withBalance” which was set to “true”. This means that for every account to list, it had to get that account’s balance.

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 [?]. It was also taken into account that the APIs being tested are of the Financial type.

In the end, all of the responses gave the expected results.

4.2. 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 confirm if the system correctly handles erroneous requests. To verify this, a few cases were tested.

These were done directly to the OpenID Endpoint: Connecting to OpenID Endpoint with incorrect credentials (Client Application credentials are incorrect); End User Login credentials are wrong; Two-Factor Authentication Code is erroneous; No permissions were chosen in Consent page; Choosing a scope but not a bank account in Consent page; Consent a bank account but not choosing a scope in Consent page.

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 result being what was expected in the beginning.

The following tests were done by making requests to the API Management Platform: Without subscription key; or an invalid one; Without an Access Token present in the request; Without a valid Transaction-ID header; Without Bank Account identifier; With wrongly formed Bank
Account identifier; With Payment body message badly formatted; Without Payment body message; Without Payment product for completing payment; With wrong Payment product for completing payment; Without code for completing payment; With wrong code for completing payment.

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.

All of these tests returned the expected results. Which were to state that: “Access denied”; “401 Unauthorized”; “Valid Token Required”; “Resource Not Found”; “Wrong Payment Information”; “Wrong Code”.

There were two tests in this set that had their times taken: Wrong Bank Account Identifier; Wrong code for completing payment. As with the Correct Tests, there were 10 requests made from which their times were taken. The “Wrong Bank Account Identifier” test averaged on 88.4ms per request. The “Wrong code for completing payment” test averaged on 136.2ms per request. These times are lower than if the requests were correctly made because an error was encountered. This is due to the Bank Account Identifier translation 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.

4.3. 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 thousands upon thousands of requests per second, like Facebook or Google do every day.

When too many requests are made via the API Management Platform, they start being actively refused. This is due to the existence of a policy that does not permit to have too many requests in a short amount of time per IP Address. Not to mention the existence of a 2nd policy that also limits the number of requests made per day, also per IP Address.

Four 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 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 make multiple requests per second. At 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 taken and requests made per second.

The first two load tests put the system under a higher load, as was to be expected, resulting on a higher average time taken to complete a request. 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. Although, 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 to bring the definite conclusion that more requests bring a higher load which take more time to conclude.

The third test was performed by making calls directly to the API Endpoint while having “Mutual Authentication” turned on. From this, it is only reasonable that the requests take time as low as 40 milliseconds, 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.

The fourth test 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, leading to similar results as the requests had been made through the Management Platform.

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 at the beginning of this Chapter.

5. 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 in the name of the End Users.

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 important 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 performs within expectations. With this it means that even though it takes longer to respond it still does so correctly, assuring that the system is correct, reliable and consistent in its responses.

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

- 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 Multi-Factor Authentication. The third element is about Inherence (something the user is), for example, a Fingerprint or Facial Recognition.

Acknowledgements
I would like to thank Professor José Alves Marques for giving me the chance to work on this interesting and innovative Project, as well as 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 also thank my parents and close friends for all their strength and support in the last couple years.

References