BIAN
Banking Industry
Architecture Network

BIAN – PNC Open APIs for Banking

Capstone Project at Carnegie Mellon University

Mark Grobaker, Arashdeep Kaur, Chaitanya Kommuru
Wenting Tao, Pallavi Thakur

10th May 2017

Contents

Executive Summary

This was a joint project between masters-level graduate students at Carnegie Mellon University's Heinz College, and stakeholders at PNC Bank, BIAN (Banking Industry Architecture Network), and IFX (International Financial eXchange).

We demonstrate a working proof of concept for open APIs in banking, in compliance with the European Commission's PSD2 financial regulation document.

Other groups can build on these efforts to ensure PSD2 compliance at their respective financial institutions.

Acknowledgements

We would like to thank all those who contributed to the success of this project. CMU Professor Mike McCarthy was our adviser, and guided us in our approach to the problem. Our client teams also provided valuable information and feedback as we progressed through the project. Our thanks go to: Hans Tesselaar and Guy Rackham of BIAN; Rich Urban of IFX; Chad Ballard, Mike Downs, Laura Ritz, and Elesha Schulze of PNC; and Ganeshji Marwaha, Chamindra De Silva, Pubudu Welagedara, and Chinthaka Dharmasiri of Virtusa Polaris, consultants to PNC.

Project Objectives

Objective 1: Comply with PSD2

The first goal of the project was to create a working proof of concept for open APIs for banking, in compliance with PSD2. PSD2 is a financial regulation document that applies to banks and financial institutions in the European Union. This regulation was published on January 13, 2016, and will go into effect for banks on January 13, 2018.

The reason PSD2 seeks to have banks create these APIs is so that third parties can use them to interact easily with the bank. There are two primary use cases required by PSD2. Banks should enable third parties to: (1) submit peer-to-peer payments to the bank, and (2) check account balances. Both of these capabilities are to be enabled via openly accessible APIs.

Per PSD2 requirements, no fees should be charged to third parties for these services. Further details on PSD2 can be found in a whitepaper published by Deutsche Bank [1].

Objective 2: Demonstrate a solution built on BIAN, IFX, and PNC

The second goal of the project was to use principles from BIAN and IFX to build a solution. This solution was built to interact with PNC, but could be adapted to any bank.

BIAN provides architectural principles designed to guide technology implementations at financial institutions. IFX provides a messaging standard, again, designed specifically for financial institutions. BIAN and IFX were interested in producing a prototype to show how their standards, designed specifically for financial applications, could be combined to produce a functioning product.

1. http://cib.db.com/insights-and-initiatives/flow/Payment_Services_Directive_2.html

PNC is a top ten US bank and a partner with BIAN in several other initiatives. Because of their interest in furthering the work of BIAN, they offered to make a test environment accessible to the CMU development team.

We were to use BIAN frameworks at a high level to guide the implementation of a message exchange. The messages themselves would be structured according to IFX format. We chose the scenarios we wanted to model based on the requirements of PSD2, which will be discussed further below.

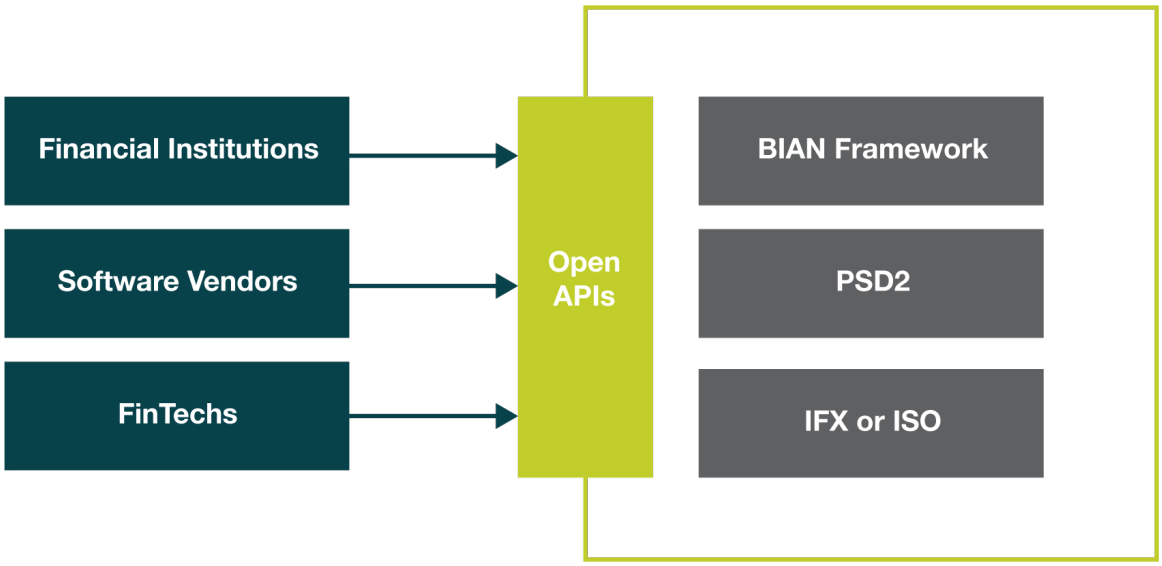


Figure 1. Left: Sample parties interacting with open APIs. Right: standards used to structure the messages to be returned from the bank, in this case, PNC Bank.

Project Methodology

PSD2 Use Cases

We begin by examining the use cases of PSD2. Per PSD2, banks should enable third parties to: (1) submit peer-to-peer payments to the bank, and (2) check account balances. We produced the below diagrams to model these use cases.

In figure 2, we model a peer-to-peer payment. In this case, Ben Roethlisberger wants to make a payment to Amazon. (Since we were working from Pittsburgh, we made examples involving star players from the Pittsburgh Steelers!) Ben has an account at PNC, and Amazon has an account at Chase. In order to initiate the payment, Ben fills out a form on his third party provider (TPP) to request a payment to be sent. In this case, we have displayed Venmo as an example TPP. Venmo then sends a message to PNC requesting the transfer. PNC transfers the money to Chase, and the funds are now available for Amazon to access.

Payment could just as easily have been from Ben to another consumer, rather than from Ben to a business. Either one would qualify for this PSD2 use case of sending a payment.

(As a side note, services like Venmo do not currently exist in Europe for payments between EU countries. Even in the US, Venmo works by using the ACH system, which takes up to 3-5 days for processing. Using open APIs would enable instantaneous transfers.)

The area in the red dotted box is the one we will be focusing on for our prototype; that is, we developed the messaging between the TPP and the bank.

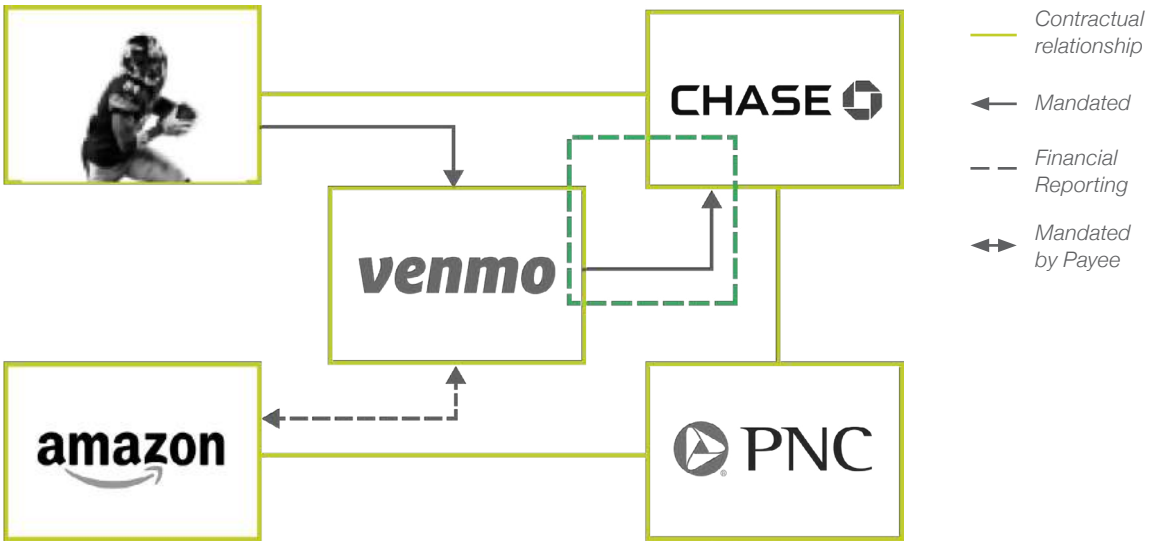


Figure 2. Peer-to-peer payment, the first use case specified in PSD2. Ben Roethlisberger sends payment to Amazon.

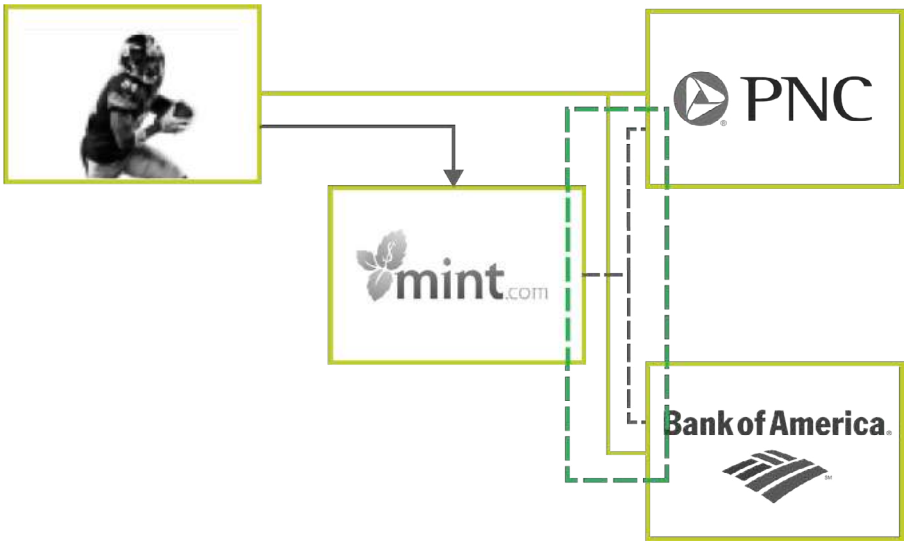


Figure 3. Check account balances, second use case specified in PSD2. Antonio Brown gets his balances from PNC and Bank of America.

The above figure shows a model of checking account balances across multiple banks. In this case, Antonio Brown requests his TPP, Mint, to monitor balances from his two accounts, PNC and Bank of America. Mint would automatically generate balance requests to PNC and Bank of America on a regular basis (daily or more frequently, depending on the TPP configuration).The banks would then respond to this message by providing their respective balances to Mint. (Services such as Mint do not currently exist in Europe.)

Again, we wanted to build just the part boxed in red: the communication between the TPP and the banks.

BIAN's Contribution

We had a number of calls with Guy Rackham, BIAN Lead Architect. He explained to us the work that his organization has done. In particular, he referred us to the BIAN Semantic API How-To Guide, an architectural document he helped produce.

One of the main components of this document is the Semantic API Selection Framework, which is shown below. The framework helps the developer or architect to "ask all right questions" that need to be asked before developing a solution. (Full explanations of how to use this framework are available in the BIAN Semantic API How-To Guide and are not replicated here.) This framework would prove helpful in structuring our solution to the needs of PSD2.

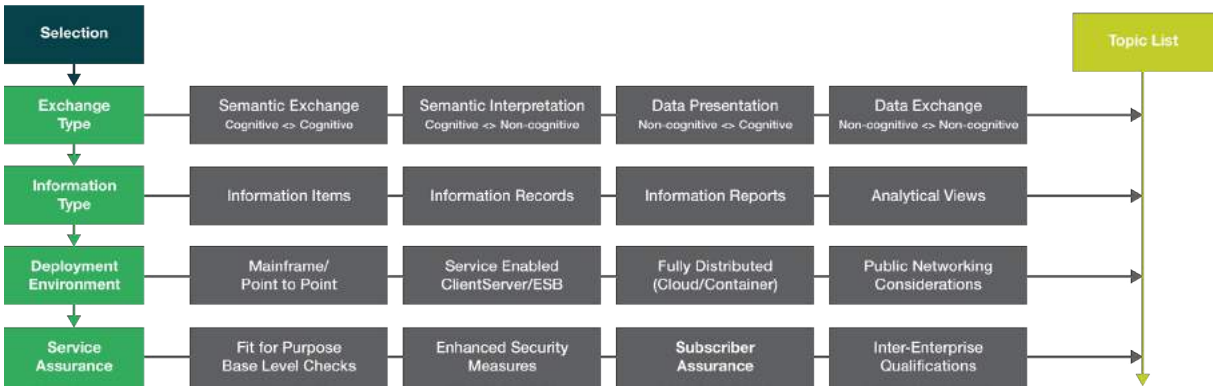


Figure 4. BIAN's Semantic API Selection Framework

After this introduction, Guy provided us with PSD2-specific guidance, by giving us an overview of the different steps he saw as necessary to carry out the use cases of PSD2: send payment and check balance. We took those steps and reformulated them into the diagram shown below in figure 6.

This sort of guidance showed the kind of value-add that BIAN can bring. IFX and other messaging standards bodies are more concerned with the messages themselves, not the business use cases. A group like BIAN was helping in explaining what all the different steps needed to be for the use cases. Then we were able to implement some of these steps using IFX in our solution.

As shown in the steps in the figure below, in the send payment case, the consumer (PSU, the payment service user) first registers the TPP (third party provider). Then he requests the TPP to send a payment. The TPP authenticates itself with the bank, and finally instructs the bank to make the payment.

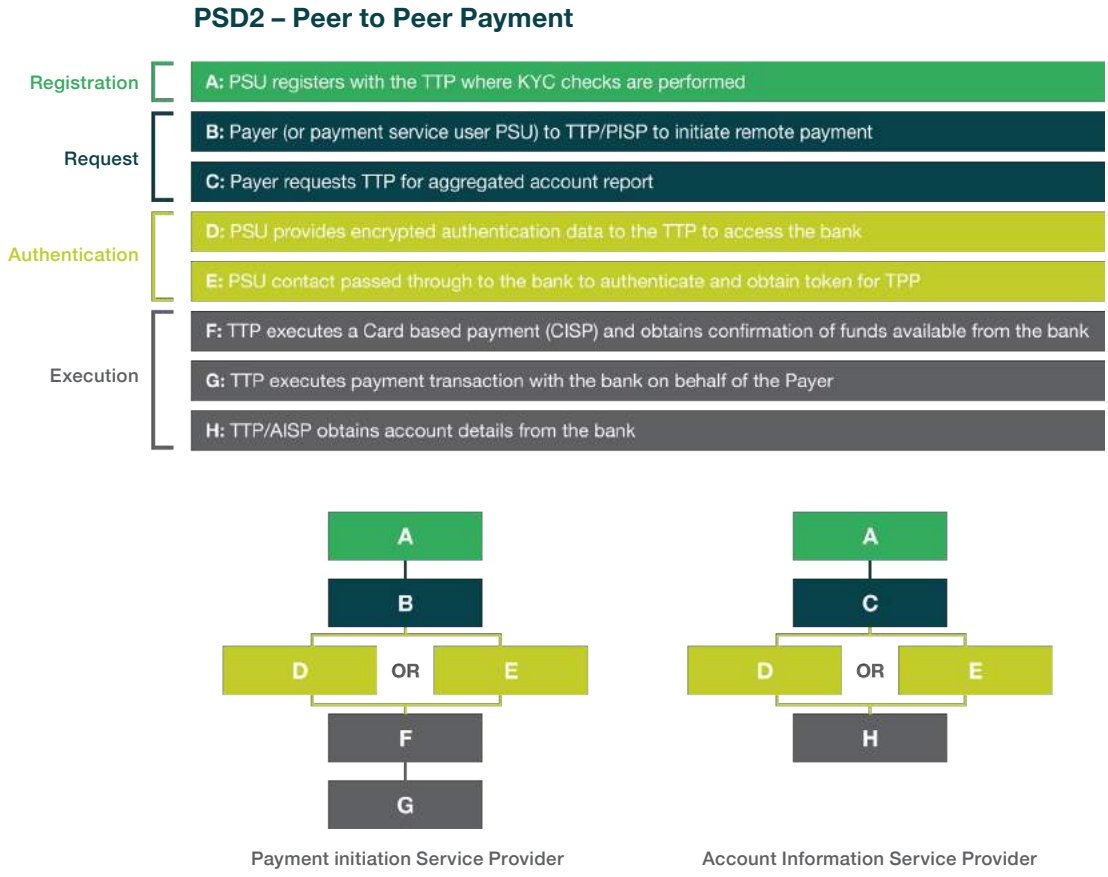


Figure 5. PSD2 use cases: peer to peer payment (left), and check balances (right)

In the check balances case, once again the PSU first registers with the TPP, and then asks the TPP to check balances. After an authentication step, the TPP retrieves the balances from the bank

Of these steps, we needed to determine which would be in or out of scope for our project. The registration would be out of scope, as that pertains to user registration for the TPP only (your login credentials for Venmo, Mint, etc). However, the request, authentication, and execution would all relate to our project and were potentially in scope. Our work on these steps is discussed further below.

Furthermore, we mapped all these steps from PSD2 using the BIAN Semantic API How-To Guide. The results of this exercise can be found in the appendix of this report.

At this point, we now needed to learn more about the kind of messages would send, especially in the execution step. We turned to our partners at IFX to learn more about the messaging format they could offer...

IFX Messages for PSD2

Rich Urban, president of IFX, provided an introduction to IFX over a number of phone calls with the CMU team. In particular, he indicated the right IFX message formats that would apply to each of the use cases we had in mind. For sending a payment, he indicated that we should use the IFX message called PmtSendRq, which would be acknowledged by the PmtSendRs message. For checking balance, we were to use the BalInqRq, which was acknowledged by BalInqRs.

All IFX documentation was available online at www.ifxforum.org. The most important part of the site is BMS, which is short for “business messaging specification”. The BMS section of the site can be searched for thousands of available message specifications. In our case, there were only four messages that we needed to use, as listed above.

We were also able to download the JSON for these messages from a Swagger utility on the website. There were a number of optional fields contained in each message, and we chose not to use them in our implementation. To remove these optional fields, we had to manually go through the JSON and strip them out. As a small feedback to IFX, we would recommend enabling JSON exports that have the optional fields already removed.

We used all (or most all) of the required fields for each of these messages. To review field level detail at which we implemented the IFX messages, we recommend reviewing our code. Screenshots of the message format specifications of the four messages we selected are included in figures 7-10.

8.7.1.1 — Payment Add Request <PmtAddRq>

The Payment Add message allows a client to schedule a single payment, where the amount is input by the customer or from a presented bill. The Payment Add message may reference an existing payee or add a new one, by specifying the information within <RemitInfo> <PayeeInfo>. If the Payment service provider supports <CustPayeeId>, the client must specify an existing <CustPayeeId> or include the <CustPayeeInfo> aggregate, but not both. Whether or not the Payment service provider supports <CustPayeeId>, the <CustPayeeInfo> aggregate may specify an existing standard payee or create a new fully specified or transfer payee. It is not possible to modify an existing payee within an Add Payment message. The customer may modify a payee via <CustPayeeModRq>.

See the matching response message PmtAddRs

Datatype: **Aggregate**

Tag	Type	Usage	Description/Context notes
begin Aggregate			
begin-block		Required	
RqUID	UUID	Required	Request Identifier. Sent by a client as a universally unique identifier for the message. Used to correlate responses with requests.
MsgRqHdr	Aggregate	Optional	Message Request Header aggregate.
AsyncRqUID	UUID	Optional	Asynchronous Request Identifier. Sent by a client to retrieve a response that was asynchronously generated by a server, generally in the case where the response would have taken too long to build and be able to be sent synchronously. For more information, see **Status**.
CustId	Aggregate	Optional	Customer Identification Aggregate. This is the identifier of the user for whom the request is being issued. This element is required if the owner of the object(s) specified in the request is not the user specified in <SignonRq>. For example, if a CSR or SP issues the request on behalf of the user, then <CustId> is required, and must contain the value of the user whose request is being issued.
end-block			
begin-block		Required	
PmtInfo	Aggregate	Required	Payment Information Aggregate.
DupChkOverride	Boolean	Optional	Duplicate Check Override Flag. When set to *True*, requests that the server not perform duplicate checking if any is normally performed. The client is affirming that this is a new payment being added.
end-block			
end Aggregate			

Figure 6. PmtAddRq screenshot from IFX BMS website.

8.7.1.2 — Payment Add Response <PmtAddRs>

The <PmtAddRs> message is used to provide an acknowledgement to a customer-initiated <PmtAddRq>. It is also used in the Payment Audit Response <PmtAudRs> and Payment Synchronization Response <PmtSyncRs> to communicate to the client that payments have been added by the customer using <PmtAddRq> and by the Pay provider using the customer's Recurring Payment Models.

See the matching request message PmtAddRq

Datatype: **Aggregate**

Tag	Type	Usage	Description/Context notes
begin Aggregate			
begin-block		Required	
Status	Aggregate	Optional	Response Status Aggregate. If this aggregate is absent, <StatusCode> defaults to 0 (zero).
RqUID	UUID	Required	The Identifier of the Request that resulted in this response.
MsgRqHdr	Aggregate	Optional Echoed	Message Request Header aggregate.
MsgRsHdr	Aggregate	Optional	Message Response Header aggregate.
AsyncRqUID	UUID	Optional Echoed	Asynchronous Request Identifier. Sent by a client to retrieve a response that was asynchronously generated by a server, generally in the case where the response would have taken too long to build and be able to be sent synchronously. For more information, see **Status**.
CustId	Aggregate	Optional Echoed	Customer Identification Aggregate. This is the identifier of the user for whom the request is being issued. This element is required if the owner of the object(s) specified in the request is not the user specified in <SignonRq>. For example, if a CSR or SP issues the request on behalf of the user, then <CustId> is required, and must contain the value of the user whose request is being issued.
end-block			
begin-block		Optional but see Description	required if message is successful
PmtInfo	Aggregate	Required Echoed	Payment Information Aggregate.
DupChkOverride	Boolean	Optional Echoed	Duplicate Check Override Flag.
PmtRec	Aggregate	Required	Payment Record Aggregate.
CSPRefId	Identifier	Optional	Customer Service Provider Reference Identifier. <CSPRefId> is used to inquire about a transaction corresponding to a confirmation number that was returned to the client when the transaction was added or modified. When a transaction has been modified, only the <CSPRefId> received in the most recent PMPMODRS is valid. The use of an <CSPRefId> from an earlier response is likely to result in a "transaction not found" response.
SPRefId	Identifier	Optional	Service Provider Reference Identifier. Same usage as <CSPRefId>.

Figure 7. PmtAddRs screenshot from IFX BMS website.

7.4.1.1 — Balance Inquiry Request <BalInqRq>

Allows client to obtain the balance of an account. The client specifies only the account for which to retrieve balances. The effective date of the balance is also returned.

See the matching response message BalInqRs

Datatype: **Aggregate**

Tag	Type	Usage	Description/Context notes
begin Aggregate			
begin-block		Required	
RqUID	UUID	Required	Request Identifier.
MsgRqHdr	Aggregate	Optional	Message Request Header aggregate.
AsyncRqUID	UUID	Optional	Asynchronous Request Identifier. Sent by a client to retrieve a response that was asynchronously generated by a server, generally in the case where the response would have taken too long to build and be able to be sent synchronously. For more information, see **Status**.
CustId	Aggregate	Optional	Customer Identification Aggregate. This is the identifier of the user for whom the request is being issued. This element is required if the owner of the object(s) specified in the request is not the user specified in <SignonRq>. For example, if a CSR or SP issues the request on behalf of the user, then <CustId> is required, and must contain the value of the user whose request is being issued.
end-block			
begin-block		Required	
begin-xor		Required	
DepAcctId	Aggregate	Required XOR	Deposit Account Identification Aggregate.
CardAcctId	Aggregate	Required XOR	Card Account Identification Aggregate.
LoanAcctId	Aggregate	Required XOR	Loan Account Identification Aggregate.
end-xor			
InclExtBal	Boolean	Optional	Include Extended Balances Indicator. If True, the response should also include the <ExtAcctBal> aggregate and return all available balances for the type of account. If False or omitted, the response should only include the standard balances for the account in <AcctBal>.
DeliveryMethod	Open Enum	Optional Profiled values	Delivery Method. Default is Channel. Value must be supported in Service Profile.
end-block			

Figure 8. BalInqRq screenshot from IFX BMS website.

7.4.1.2 — Balance Inquiry Response <BalInqRs>

Allows client to obtain the balance of an account. The effective date of the balance is also returned.

See the matching request message BalInqRq

Datatype: **Aggregate**

Tag	Type	Usage	Description/Context notes
begin Aggregate			
begin-block		Required	
Status	Aggregate	Optional	Response Status Aggregate. If this aggregate is absent, <StatusCode> defaults to 0 (zero).
RqUID	UUID	Required	The Identifier of the Request that resulted in this response.
MsgRqHdr	Aggregate	Optional Echoed	Message Request Header aggregate.
MsgRsHdr	Aggregate	Optional	Message Response Header aggregate.
AsyncRqUID	UUID	Optional Echoed	Asynchronous Request Identifier.
CustId	Aggregate	Optional Echoed	Customer Identification Aggregate. This is the identifier of the user for whom the request is being issued. This element is required if the owner of the object(s) specified in the request is not the user specified in <SignonRq>. For example, if a CSR or SP issues the request on behalf of the user, then <CustId> is required, and must contain the value of the user whose request is being issued.
end-block			
begin-block		Optional but see Description	required if message is successful
begin-xor		Required	
DepAcctId	Aggregate	Required XOR Echoed	Deposit Account Identification Aggregate.
CardAcctId	Aggregate	Required XOR Echoed	Card Account Identification Aggregate.
LoanAcctId	Aggregate	Required XOR Echoed	Loan Account Identification Aggregate.
end-xor			
InclExtBal	Boolean	Optional Echoed	Include Extended Balances Indicator.
DeliveryMethod	Open Enum	Optional Echoed	Delivery Method.
AcctBal	Aggregate	Required Repeating	Account Balance Aggregate.
ExtAcctBal	Aggregate	Optional Repeating	Extended Account Balance Aggregate.
MktgInfo	C-255	Optional	Marketing Information.
end-block			

Figure 9. BalInqRs screenshot from IFX BMS website.

Comparing IFX and ISO

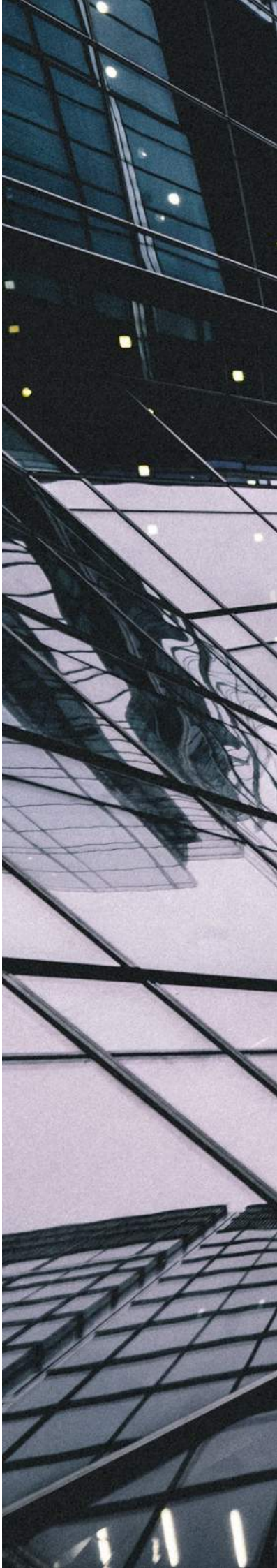
As part of the project, we compared and contrasted IFX and ISO. Both are standards to be used by financial institutions for sending messages, most commonly payments messages.

ISO's 20022 standard is currently used worldwide with prominent contributors like SWIFT and VISA. ISO 20022 includes eight parts: ISO 20022-1 through ISO 20022-8. These parts describe the metamodel, UML profile, XML schema etc. for the messages.

IFX promotes messages that are sent in XML or JSON. There is also built-in capability to generate a swagger document for these message structures. IFX can contain ISO elements, if desired. Based on our experience in this project, IFX messages are defined and organized in a way which is easy to read/understand.

Both these standard bodies promote the idea of "interoperability" across financial institutions. ISO is currently in broad use, while IFX has support but is yet to see broad adoption. Both are acceptable standards for sending financial messages.

Resources for further reading on IFX and ISO can be found in the appendix.



Interacting with PNC

After working with BIAN and IFX to understand their frameworks, standards, and message formats, we began to collaborate more closely with PNC. In order to test out the APIs we were developing, we needed to be able to simulate sending and receiving messages to/from other systems in the bank.

PNC had already set up an environment, called the API store, which had some open APIs. These open APIs did not generate IFX compliant messages. Rather, they returned data in a flat format as part of a RESTful exchange. For example, a request to

http://apimanager.pncapix.com:8280/SmartBank-API-Services/V2.0/card/findByCardNumber/{cardNumber}

Will return details about that card number, if it exists in the data.

We were able to interact with and test out these APIs by using the freely available tool "Postman." Postman enabled us to create HTTP requests to APIs in the PNC API store. (Note that the API store is not used in daily business operations yet at the PNC Bank. Right now it is a sandbox area where PNC is exploring how it could deploy open API solutions in production.)

Solution architecture

As shown earlier, in figures 2 and 3, we wanted to focus our work on the interaction between the TPP and the bank. If we “zoom in” on this relationship, there are a few steps that need to happen in the communication.

As shown in this figure, first a third party provider (TPP) would initiate a payment using the RESTful API that we have built (“API” in the figure). This could be a form available at an endpoint such as **www.pnc.com/sendpayment**. This should only be available after logging in to your account.

Once the payment is submitted to the API, the API first looks up additional data from other PNC systems (steps 2 and 3). Once that data has been retrieved, it makes a call to the Bank Payment System to make the payment.

Finally, the bank payment system returns a response to the API (step 5). The API, in turn, reads this message, and uses it to generate a JSON IFX compliant message (step 6). The message will be the PmtSendRs or the BalInqRs.

In our implementation, note that we ended up not using PmtSendRq and BalInqRq. The bank might want to use these messages when communicating with internal payment systems. However, as we did not have that system available to us, we did not need to send messages to it.

However, PmtSendRs and BalInqRs messages are used. The appropriate message is generated and sent back to the TPP. What we gain by doing this is that now, the TPP can expect a consistently formatted response message when it interacts with any bank. Furthermore, the bank can also choose to archive these response messages, which may be useful for historical or reporting purposes.

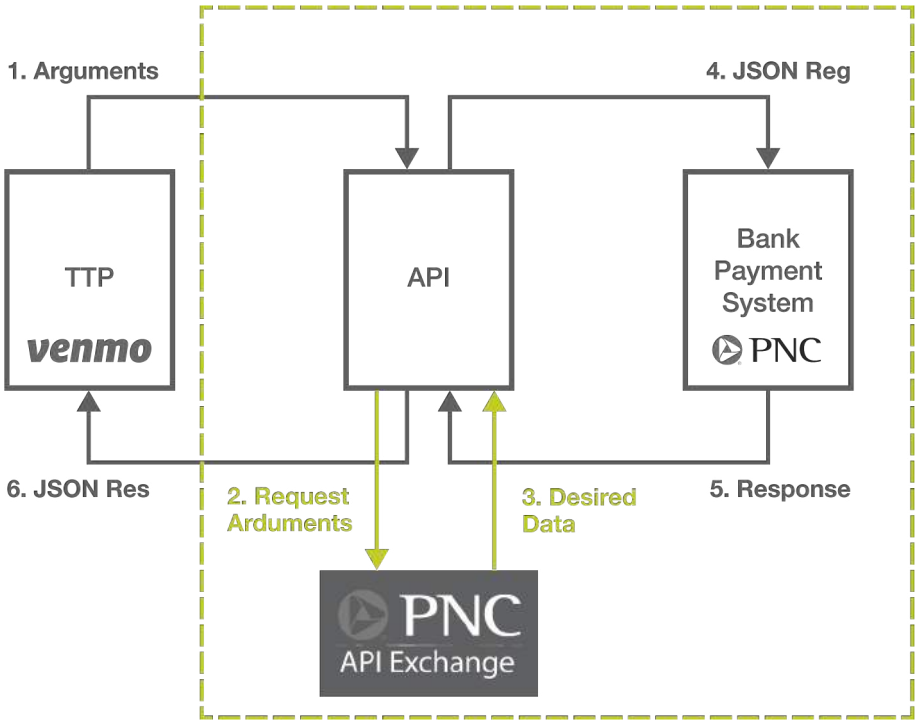


Figure 10. Message Architecture of TPP, API, and Bank Systems

Implementation

Introduction and file description

All code is available on Git at this link: https://github.com/chinthakadd/cmu-bian-starter

We developed and deployed the API on our local machines during the development phase of the project. The PNC team's Virtusa consultants helped us get set up with the proper IDE and development environment to work with the PNC API store.

In this section, we describe the different files in the Java solution that we developed. All files are listed in the figure at right, a screenshot from IntelliJ IDE.

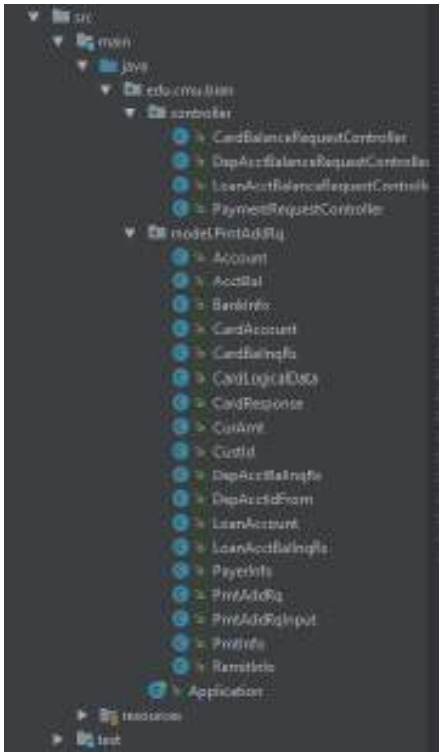


Figure 11. Java files used in our implementation.

Reading from the "controller" folder, we have two types of controllers – PaymentRequestController, and Card/Dep/Loan AcctBalanceRequest Controller. The first controller has the logic for the send payment operation. The second set of controllers are very similar to each other, and all handle for the check balance operation. There is slightly different logic depending on whether a card, deposit, or loan is being queried.

You will also notice a number of files listed under the model section. These are used to structure the IFX response message. For example, a message might consist of Account, AcctBal, and BankInfo objects. To create this message, we could create an object that contains all three of these objects. These will be described further below.

```
public class BankInfo {

    private String bankIdType;
    private String bankId;
    private String name;
    private String branchId;
    private String branchName;
```

Figure 12. Fields of the BankInfo class.

One advantage of this design is that BankInfo, for example, can be used consistently in many different contexts. Several different IFX messages may contain BankInfo. By having a BankInfo object, we can enforce that it must contain the same five fields everywhere it is used. These fields are shown below. Now, wherever BankInfo is used, it must have this consistent definition.

BankInfo and other objects are then converted into the appropriate JSON structure when they are used.

Choice of HTTP method for RESTful API

For our APIs, we needed to choose the appropriate HTTP method to correspond to the operation. BIAN's API How-To Guide was a useful resource for this task. The figure below shows a mapping between action terms and the corresponding HTTP method. We used this to perform mappings for the two use cases.

BIAN How-To Guide Semantic APIs

		ACTION TERMS	Description	Applicable REST Verbs
Initialise & Register	Actions to set-up, establish a new control record instance	Register	Record the details of a newly identified entity	PUT; POST
		Initiate	Begin an action including any required initialization tasks	PUT; POST
		Activate	Commence/open an operational or administrative service	PUT; POST
		Create	Manufacture and distribute an item	PUT, POST
Invocation & Execution	Actions to access/update/influence an established instance	Configure	Change the operating parameters for an operational capability	PUT; PATCH
		Update	Change the value of some (control record) properties	PUT; PATCH
		Record	Capture transaction or event details against managed activity	PUT; PATCH
		Execute	Execute a task or action on an established facility	PUT; PATCH
		Evaluate	Perform a check, trial or evaluation	GET
		Provide	Assign or allocate resources or facilities	PUT; PATCH
		Authorise	Allow the execution of a transaction/activity	PUT; POST
		Request	Request the provision of some service	GET
		Terminate	Conclude, complete activity	DELETE
Maintain & Analyze (de egation only – n action terms apply)				
Report & Notify	Actions to extract details & subscribe to updates	Notify	Provide details against a predefined notification agreement	Managed using event frameworks
		Retrieve	Return information/report as requested	Managed using event frameworks

Figure 13. Action term to HTTP verb mapping.

Mapping for PSD2 use cases:

- Send payment: For this, we need to "Create" a message to send a payment, so we use the HTTPPUT operation.

- Check balance: For this operation, we need to "Request" a balance, so we use the HTTP GEToperation.

Code Walkthrough

Here, we copy the code and comment on the functionality of a representative set of the files. First, the **PaymentRequestController:**

```
//Import Statements
package edu.cmu.bian.controller;
import edu.cmu.bian.model.PmtAddRq.*;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.util.LinkedMultiValueMap;
import org.springframework.util.MultiValueMap;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.client.RestTemplate;
import java.util.HashMap;
import java.util.Map;

/*
Business Use Case: This caters to the payment initiation service
provider (PISP) requirement of the PSD2.
Objective: The Payment Request Controller provides the end point
for the TPP to Access and Initiate a payment.
           It looks up the PNC API Exchange to fill a few
additional fields based on the parameters sent by the TPP.
           An IFX Compliant JSON response is built and echoed
back.
Created By: CMU BIAN-PNC Capstone Team
Last Update Date: 4/24/2017
*/

@RestController
public class PaymentRequestController {
//PUT request for initiating payment
    @PutMapping("/pmtAddRq")
    public PmtAddRq paymentAddRequest(@RequestBody PmtAddRqInput
pmtAddRqInput) {

        //Objects required to build the IFX compliant response
        PmtAddRq pmtAddRqResponse = new PmtAddRq();
        PayerInfo payerInfo = new PayerInfo();
        CurAmt curAmt = new CurAmt();
        BankInfo bankInfo = new BankInfo();
        RemitInfo remitInfo = new RemitInfo();
        RestTemplate restTemplate = new RestTemplate();
        DepAcctIdFrom depAcctIdFrom = new DepAcctIdFrom();
        CustId custId = new CustId();
        PmtInfo pmtInfo = new PmtInfo();
        CardLogicalData cardLogicalData = new CardLogicalData();

        //Code to access the PNC API Exchange to retrieve details
based on Card Number
        String url =
"http://apimanager.pncapix.com:8280/SmartBank-API-
Services/V2.0/card/findByCardNumber/{cardNumber}";
        //Map to store the Parameters that are to be passed
        Map<String, Object> paramMap = new HashMap<>();
        paramMap.put("cardNumber",
pmtAddRqInput.getCardEmbossNum());

        //Map to store the headers required for the API Access
        MultiValueMap<String, Object> headers = new
LinkedMultiValueMap<>();
        headers.add("Accept", "application/json");
        headers.add("Authorization", "Bearer 16cd1907-6dfd-33ab-
b196-9d8419333f3d");
        HttpEntity httpEntity = new HttpEntity(headers);

        // Sending the request to the PNC API Exchange
        ResponseEntity<CardResponse[]> cardResponseEntity =
                restTemplate.exchange(url, HttpMethod.GET,
httpEntity, CardResponse[].class, paramMap);

       // Response stored in an object built as per the PNC API
Response body JSON Structure
        CardResponse[] cardResponse =
cardResponseEntity.getBody();


       // Building the target response structure which is IFX
Compliant
        curAmt.setAmt(pmtAddRqInput.getAmt());
        curAmt.setCurCode(pmtAddRqInput.getCurCode());
```

This will take a PUT request, based on the input in the RequestBody

Create objects for the response message

Prepare elements needed to make call to API Store

Make call and store the response

```
        remitInfo.setCurAmt(curAmt);
        bankInfo.setBankId(pmtAddRqInput.getBankId());
        bankInfo.setBankIdType(pmtAddRqInput.getBankIdType());
        bankInfo.setBranchId(pmtAddRqInput.getBranchId());
        bankInfo.setBranchName(pmtAddRqInput.getBranchName());
        bankInfo.setName(pmtAddRqInput.getBankName());
        depAcctIdFrom.setAcctCur(pmtAddRqInput.getPayeeAcctCur());

depAcctIdFrom.setAcctType(pmtAddRqInput.getPayeeAcctType());
        depAcctIdFrom.setBankInfo(bankInfo);
        depAcctIdFrom.setAcctId(pmtAddRqInput.getPayeeAcctId());

payerInfo.setPayerAcctId(cardResponse[0].getAccount().getAccountId
());
        pmtInfo.setDepAcctIdFrom(depAcctIdFrom);
        pmtInfo.setPayerInfo(payerInfo);
        pmtInfo.setRemitInfo(remitInfo);
        pmtInfo.setDueDt(pmtAddRqInput.getDueDt());
        pmtInfo.setPrcDt(pmtAddRqInput.getPrcDt());
        cardLogicalData.setBrand(cardResponse[0].getGateway());

cardLogicalData.setCardEmbossNum(pmtAddRqInput.getCardEmbossNum())
;
        cardLogicalData.setExpDt(pmtAddRqInput.getExpDt());

cardLogicalData.setName(pmtAddRqInput.getCardEmbossName());
        custId.setCardLogicalData(cardLogicalData);
        custId.setCustLoginId(cardResponse[0].getPartyId()); //
need to change this to Cust Login ID
        pmtAddRqResponse.setPmtInfo(pmtInfo);
        pmtAddRqResponse.setCustId(custId);
        pmtAddRqResponse.setRqUID(pmtAddRqInput.getRqUID());

        //Return the JSON response
        return pmtAddRqResponse;
    }

}
```

Format response into IFX format, using objects created above

Note that formatting of this message (eg which objects to include) is based on IFX message specification

CardBalanceRequestController:

```
//Import Statements
package edu.cmu.bian.controller;
import edu.cmu.bian.model.PmtAddRq.*;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.util.LinkedMultiValueMap;
import org.springframework.util.MultiValueMap;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;
import java.util.HashMap;
import java.util.Map;

/*
Business Use Case: This caters to the account information service
provider (ATSP) requirement of the PSD2.
Objective: The Card Balance Request Controller provides the end
point for the TPP to get Balance of a Card Account.
          It looks up the PNC API Exchange to fill a few
```

```
additional fields based on the parameters sent by the TPP.
          An IFX Compliant JSON response is built and echoed
back.
Created By: CMU BIAN-PNC Capstone Team
Last Update Date: 4/24/2017
*/


@RestController
public class CardBalanceRequestController {
    //GET request for retrieving balance

@GetMapping("/cardBalRq/{rqUID}/{cardEmbossNum}/{cardEmbossName}/{
expDt}")
    public CardBalInqRs
CardAcctBalanceRequest(@PathVariable("rqUID") String rqUID,

@PathVariable("cardEmbossNum") String cardEmbossNum,

@PathVariable("cardEmbossName") String cardEmbossName,

@PathVariable("expDt") String expDt) {

        //Objects required to build the IFX compliant response
        CardBalInqRs cardBalInqRs = new CardBalInqRs();
        CardLogicalData cardLogicalData = new CardLogicalData();
        CardAccount cardAccount = new CardAccount();
        Account account = new Account();
        CurAmt curAmt = new CurAmt();
        AcctBal acctBal = new AcctBal();
        cardLogicalData.setName(cardEmbossName);
        cardLogicalData.setCardEmbossNum(cardEmbossNum);
        cardLogicalData.setExpDt(expDt);
        RestTemplate restTemplate = new RestTemplate();

        //Code to access the PNC API Exchange to retrieve details
based on Card Number
        String url =
"http://apimanager.pncapix.com:8280/SmartBank-API-
Services/V2.0/card/findByCardNumber/[cardNumber]";
        //Map to store the Parameters that are to be passed
        Map<String, Object> paramMap = new HashMap<>();
        paramMap.put("cardNumber", cardEmbossNum);

        //Map to store the headers required for the API Access
        MultiValueMap<String, Object> headers = new
LinkedMultiValueMap<>();
        headers.add("Accept", "application/json");
        headers.add("Authorization", "Bearer 16cd1907-6dfd-33ab-
b196-9d8419333f3d");
        HttpEntity httpEntity = new HttpEntity(headers);

        // Sending the request to the PNC API Exchange
        ResponseEntity<CardResponse[]> cardResponseEntity =
                restTemplate.exchange(url, HttpMethod.GET,
httpEntity, CardResponse[].class, paramMap);

        // Response stored in an object built as per the PNC API
Response body JSON Structure
        CardResponse[] cardResponse =
cardResponseEntity.getBody();
        cardLogicalData.setBrand(cardResponse[0].getGateway());
```

Responds to a GET request, and reads in the variables in the URL path

Create objects for the response message

Prepare elements needed to make call to API Store

Make call and store the response

```
        // Building the target response structure which is IFX
Compliant
        cardAccount.setCardLogicalData(cardLogicalData);
        cardAccount.setAcctType("Credit");
        cardBalInqRs.setCardAccount(cardAccount);
        cardBalInqRs.setRqUID(rqUID);
        account = cardResponse[0].getAccount();
        curAmt.setAmt(account.getBalance());
        curAmt.setCurCode("USD");
        acctBal.setCurAmt(curAmt);
        acctBal.setBalType("CreditHeld");
        cardBalInqRs.setBalance(acctBal);

        //Return the JSON response
        return(cardBalInqRs);
    }

}
```

Format response into IFX format, using objects created above.

Note that formatting of this message (eg which objects to include) is based on IFX message specification


BIAN

Using our APIs

In this section we walk through how one can interact with the APIs. In the screenshots below, we show the APIs running locally and how to interact with them via the Swagger UI. Our code automatically generated a Swagger UI, since it was built with the Spring framework. The user can submit data to each of these endpoints and examine the response messages. Of course, a TPP would also be able to send data to these endpoints without having to use the Swagger UI.

Our API was also deployed to the PNC API store, where the interface is somewhat different. However, the functionality is the same. These screenshots can be found in the appendix section.

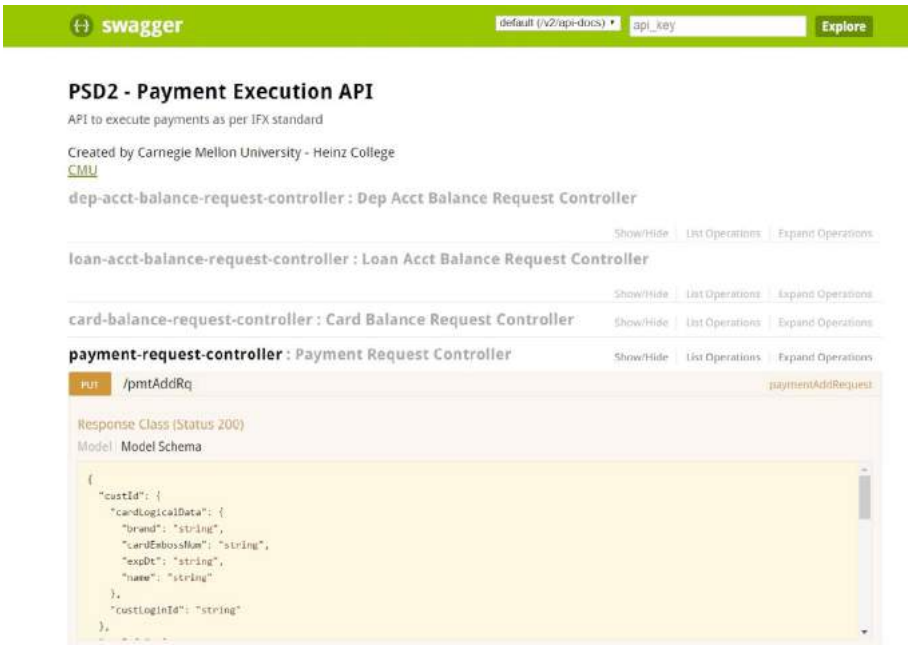


Figure 14. Swagger UI in our local deployment of the API. As shown, an API to send a payment is selected. Equivalently, the API can be deployed to the bank's API store.

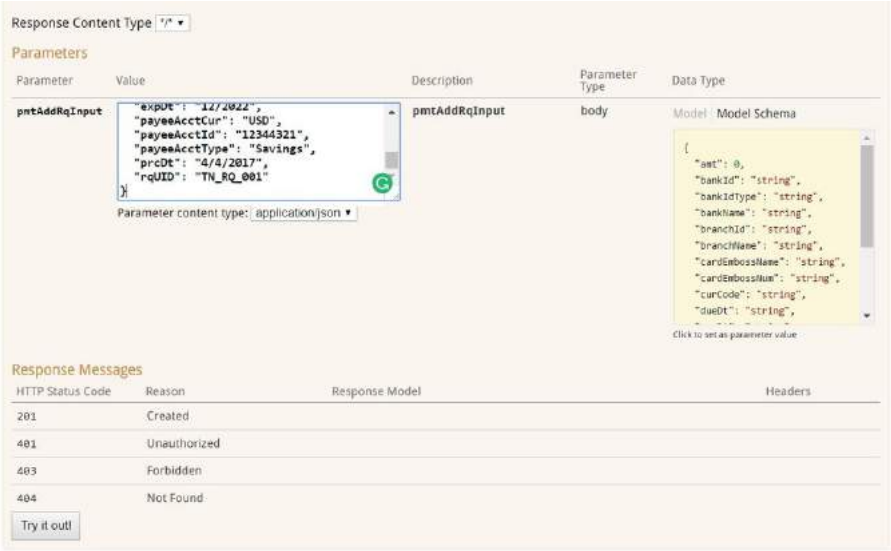


Figure 15. As shown, we submit data to the API as parameters in the body of the HTTP request.

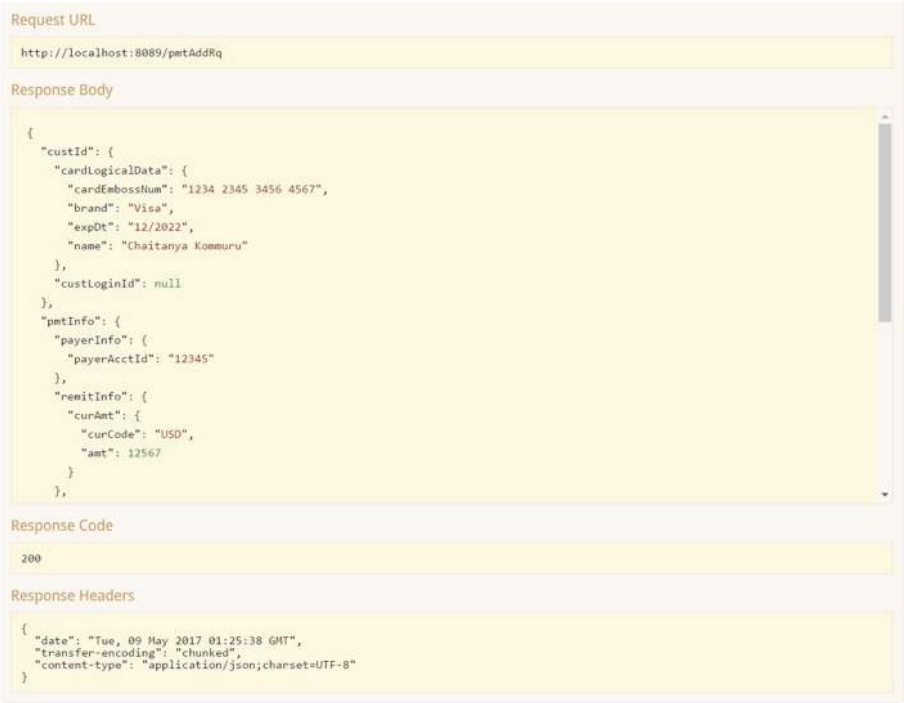


Figure 16. The API returns data in the IFX-compliant format. (More data could be seen if the developer were to scroll down.)

Conclusions and Recommendations

In this project, we have shown that the BIAN and IFX frameworks can be applied successfully to comply with PSD2 requirements in order to enable third parties to send payments and check balances.

As the banking industry will be required to build new functionality (open APIs) to comply with PSD2 requirements, now is an ideal time to coalesce around a standard method for doing so. Messages with the IFX format are well-suited to send response messages to the TPP, as well as messages from one bank to another. When a TPP submits a send payment request, it could receive back a standard IFX compliant message from whichever bank it interacts with. The bank, in turn, could send an IFX message from the payee bank to the payer bank to provide details for the money transfer.

We recommend that industry partners continue to explore these opportunities by partnering with BIAN and IFX when developing their solutions.

Lessons Learned

This project required us to collect information from different stakeholders, and determine the type of solution they were asking for. In a fairly short time, we also had to become acquainted with much previous work that had been done by other standards bodies such as BIAN, IFX, and the European Commission (which produced PSD2).

Suggestions for Future Work – Standards and Frameworks

1. **Standardize which IFX fields the TPP should be required to submit**

   In our current solution, for simplicity the TPP is required to submit many of the fields needed to produce the IFX response message. In reality, a number of these fields can and should be retrieved from within the bank's own systems. (For example – branch name. The TPP would not know this, but the bank would. However, not all of this data was easily available from the API Store made available to us.)

   We suggest that industry leaders determine which fields should be required from the TPP, and which fields can be supplied by the bank. If feasible, this classification could even be included with the IFX standard itself.

2. **Clarify which parties should create an IFX message (TPP, bank, or both)**

   Industry leaders should also clarify which parties need to create an IFX message. Here is our current understanding, which should be reviewed.

   It is currently not clear if the TPP would be required to submit an IFX-formatted message, or if just the banks should communicate using this standard. We believe the TPP could be asked to format its message in a specific way, but there may also be concern about putting that burden on the TPPs.

   Furthermore, the industry should define whether banks should use IFX for messages to other systems within the same bank (eg PNC to PNC) and to other banks (eg PNC to Chase).

3. **Consider developing context-specific guidelines for message formats**

   Consider that there are three distinct messaging contexts: TPP-bank, intra-bank, and bank-to-bank. To use real names as examples: Venmo-PNC, PNC-PNC, and PNC-Chase. As we worked on this project, it became clear that each of these contexts would have different expectations in terms of what information would be sent or received. We recommend incorporating this distinction into either BIAN or IFX standards as appropriate. “Message context” seems to be an important topic that a PSD2 solution much take into account.

4. **Review required/optional fields in IFX**

   In its documentation, IFX has marked fields in the messages as either required or optional. Based on the needs discovered during further implementation work, these required/optional fields should be updated as appropriate. For example, if a field currently marked as optional is discovered through discussion and implementation to be required, then this should be updated in IFX documentation.

5. **Clarify the BIAN How-To Guide sections on deployment environment and service assurance**

   When reading the BIAN How-To Guide, we were not able to understand the framework’s guidance around deployment environment and service assurance. We talked through it on the phone and were eventually able to understand. However, perhaps some additional wording in this section would be helpful to future users of the guide.

Suggestions for Future Work – Development Work

1. Develop process to lookup necessary fields within bank

The banks will need to develop a process to look up the relevant information from within their systems to populate IFX fields not provided by the TPP.

We also note that, in this project, we would have used other APIs to communicate within the bank. However, depending on the implementation, the bank in question may just be able to look up necessary information from a database table. In such a case, interaction with an API may not be required since the data is internally available.

2. Build solution for messages between banks

In this project, we have not worked on the messaging from one bank to another. However, if there is to be payment between users of different banks, such a message would be required. We recommend formatting this message in an IFX format. The PmtSendRq messagemay be the appropriate message to use in this case.

Another possibility would be to send data between banks in the message body of an HTTP PUT or GET (a RESTful API). However, since banks will likely be familiar with IFX, and also since this message body may become rather long (longer than what we would expect from a TPP), it may be preferable to send messages between banks in IFX formats.

3. Implement authentication step of PSD2 (OAuth)

We also have not implemented the authentication step required in PSD2. Through our conversations with PNC, we determined that an OAuth implementation would eventually be necessary to implement PSD2. As the work required would be fairly intensive, it was deemed out of scope for our project.

We did create a prototype of what the OAuth interaction might look like, and we included this as part of the UI shown in our demo. However, we opted not to include this in our code submission as it was just for simple demonstration purposes.

Appendix

Mapping of PSD2 steps to BIAN's Semantic API Selection Framework

Figure 17. PSD2 use case: send payment.

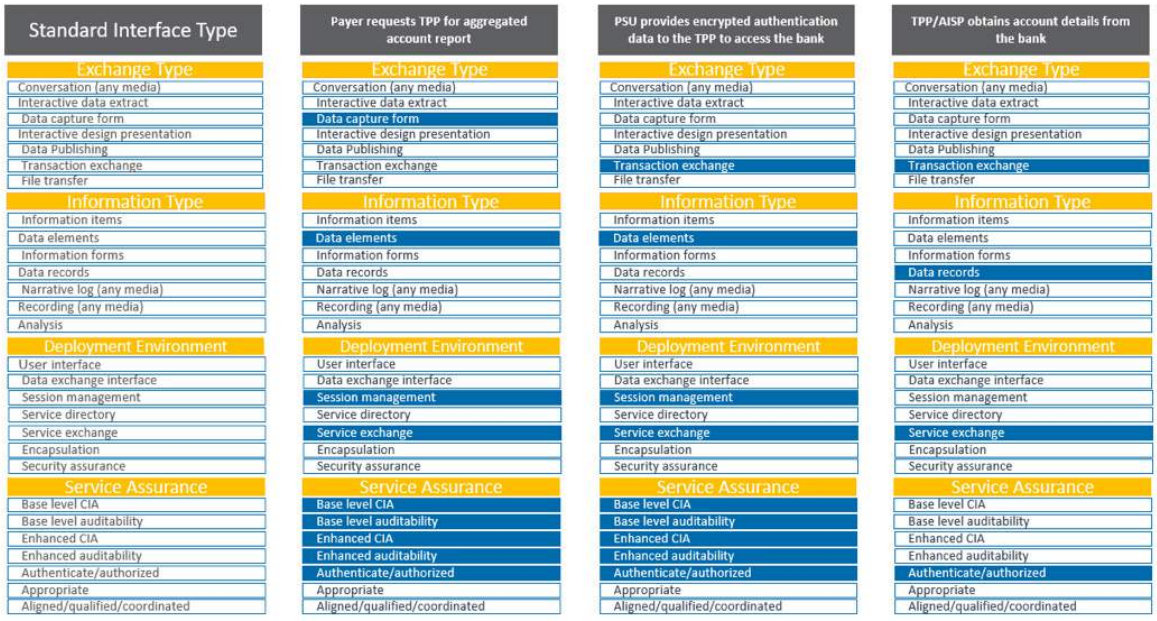


Figure 18. PSD2 use case: check account balances.

Process mapping for API Generation

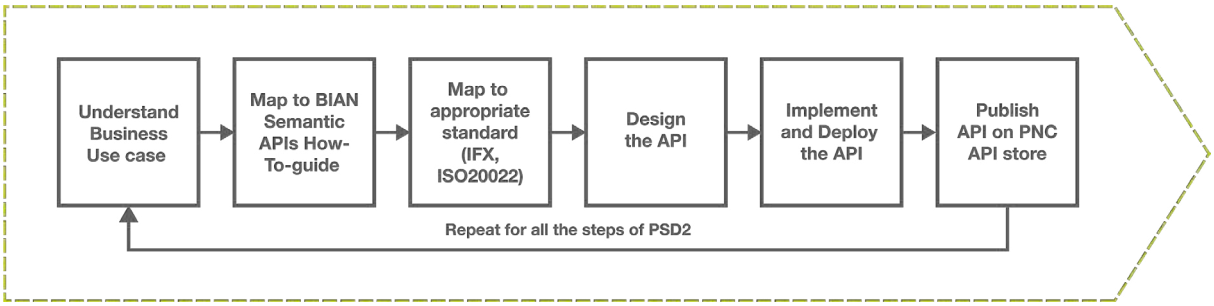


Figure 19. Business process for building solutions to the different steps of PSD2

API deployed in PNC API store

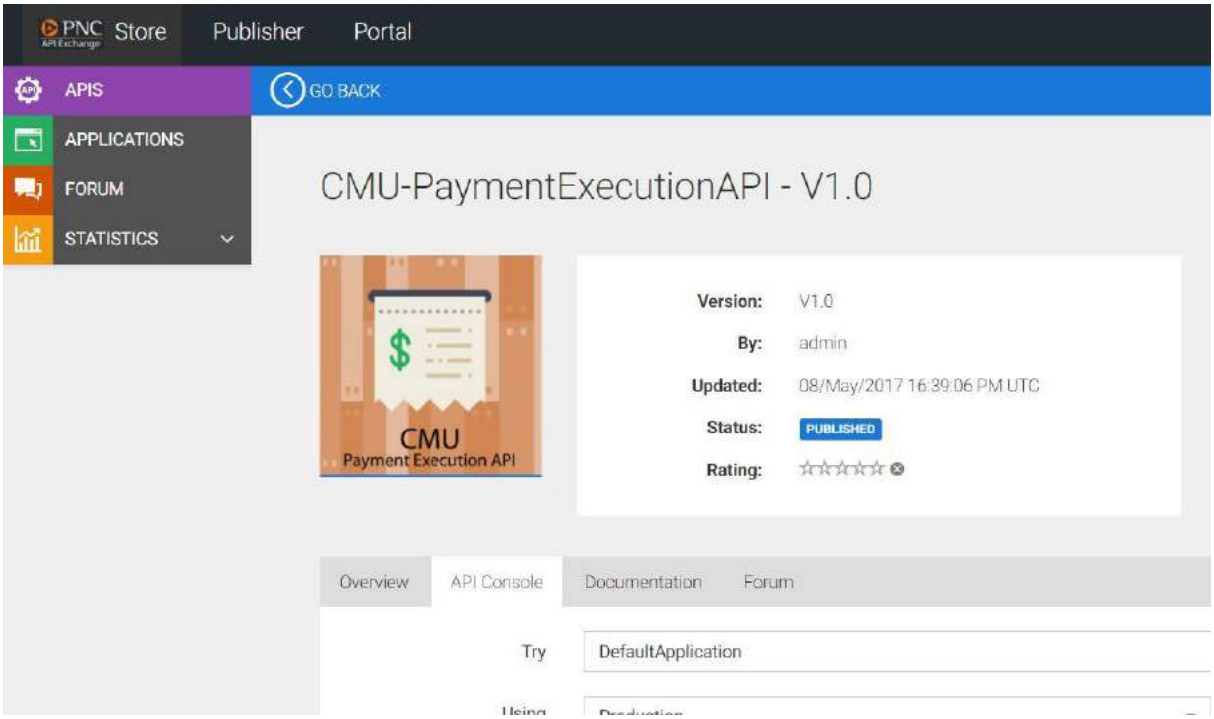


Figure 20. Home page for the payment execution API in the PNC open API store

Process mapping for API Generation

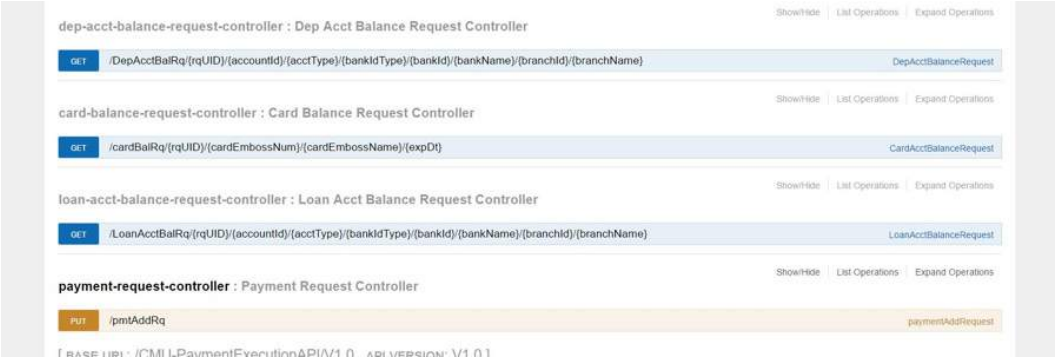


Figure 21. Interface in PNC API store. By clicking in to one of these endpoints you can then send a request with the appropriate parameters.

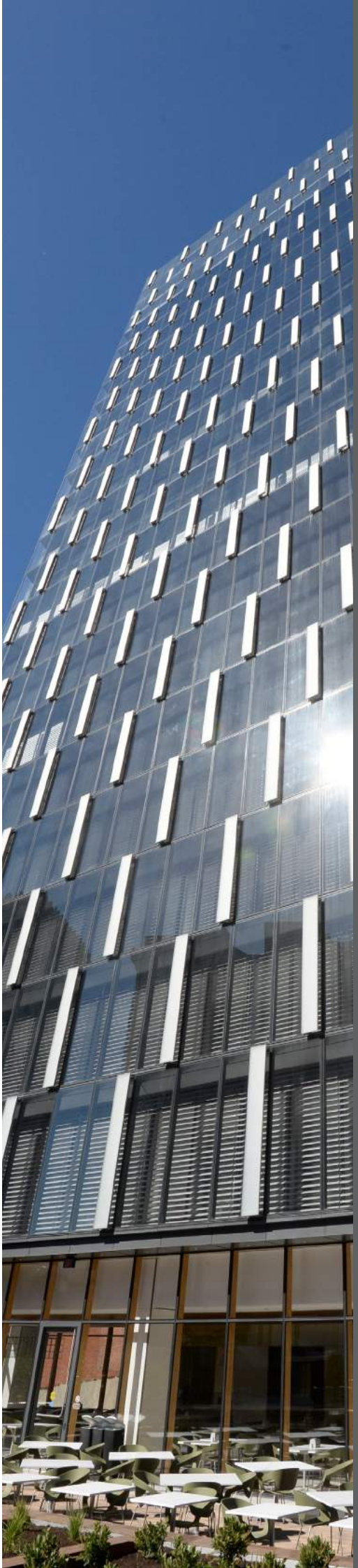
Sources and further reading on IFX and ISO comparison

https://www.gtnews.com/articles/adoption-of-iso-20022-messages-by-ifx-forum/]

https://www.iso20022.org/the_iso20022_standard.page

https://en.wikipedia.org/wiki/ISO_20022

https://en.wikipedia.org/wiki/Interactive_Financial_Exchange



https://bian.org