API_REF - 8E357AD6-6E69-4A5A-A486-E566F5D3A6EC

See BNF Format for API specification format being used

This is a proposed API for PSD2 XS2A that provides the types of features needed by the new AISP and PISP companies to develop compelling digital solutions.

It has been defined to create discussion and to try and influence the Open Banking UK and Berlin Group who are working on the official APIs

API Definition

/ Complete list of methods within the API*/*
psd2Message :=
    getSessionToken | getSupportedAPI | getAccountList | getAccountActivity |
    getAccountBalance | makeDomesticPayment | getPaymentStatus

getSupportedAPI := SESSION_TOKEN, apiID_List → API_GUIID

apiID_List :- { API_GUIID } *
/ apiID_List - List of APIs in order of preference*

getSupportedAPI returns the first API in the list that is supported.

getSupportedAPI provides a way to evolve APIs. This API that I am defining now has API_GUIID=8E357AD6-6E69-4A5A-A486-E566F5D3A6EC but someone could come along and extend it with a new API_GUIID for it.
/

```
getSessionToken :=
      requestLogonToken → returnLogonToken →
      signedLogonRequest → logonResult

requestLogonToken := NULL   // No parameters needed for this method call
returnLogonToken := LOGON_TOKEN | logonFailReason
logonResult := SESSION_TOKEN | logonFailReason
```

```
/*
SESSION_TOKEN is a cryptographically secure token.
- It is used in all other API calls. So you need to perform the getSessionToken handshake
before you can do anything else with this API.
- Security countermeasures should be in place within the API implementation to prevent
brute-force guessing of valid SESSION_TOKEN
*/

logonFailReason := … // TBD - List of reason codes for failure

signedLogonRequest :=
      digitalSign(
            TPP_PRIVATE_KEY, TPP_PUBLIC_KEY,
            LOGON_TOKEN, CUSTOMER_ACCESS_TOKEN)

/* CUSTOMER_ACCESS_TOKEN allocated to AISP/PISP by the ASPSP as part of the
registration process. Although we should keep CUSTOMER_ACCESS_TOKEN secret lose of it
is not critical. Only valid for a given AISP/PISP via their registered TPP_PUBLIC_KEY.

signedLogonRequest is a digitally Signed message. Message Content consists of
TPP_PUBLIC_KEY + LOGON_TOKEN + CUSTOMER_ACCESS_TOKEN and it is signed using
the TPP's TPP_PRIVATE_KEY which never leaves the TPP site and should be protected via
highest levels of security countermeasures.

The ASPSP will only accept logon's for active LOGON_TOKEN that it has assigned.

NOTE: In this API the ASPSP will not have any responsibility for the customer authentication.
Strong Customer Authentication (as mandated by PSD2) is the sole responsibility of the TPP.

22 Dec 2017 - At the current time this approach to authentication is not officially part of PSD2.
This type of delegated authentication responsibility to the TPP is allowed but only via a bilateral
agreement between TPPs and individual ASPSPs. Hopefully this will change.

*/

getAccountList := SESSION_TOKEN → {accountDetail}*
// Return zero of more accounts that the CUSTOMER_ACCESS_TOKEN has access to

acountDetail :=
      accountID, [accountType],
      [accountPermissions], [[accountAttributes]]

/* accountID will be different for UK compared with EURO countries. */
```

```
accountID := accountID-UK-BACS | accountID-EU-SEPA | …
accountID-UK-BACS := ukSortCode, ukAccountNumber
sortCode = BACS_SORT_CODE
ukAccountNumber = BACS_ACCOUNT_NUMBER
/* Other non-EUR countries will have their own domestic account format */

accountPermissions := READ_ONLY | TRANSACTIONAL
// Transactional means you can make payments from the account
// As APIs evolve to be more complex these permissions will get more complicate

accountAttributes := … // TBD - attributes associated with the account: Name etc


getAccountActivity :=
      (SESSION_TOKEN, accountID, [afterEventIdentifier]) →
      getAcountActivityResponse

afterEventIdentifier := afterEventID | afterEventToken
/*
afterEventID are passed back in transactionHistoryList.
afterEventToken supports paging through a single result.
Passing afterEventToken will not increment your API call count. Passing a afterEventID will.

If no  afterEventIdentifier is passed then getAccountActivity will return recent history.
How much determined by the ASPSP.

For accounts with very large volumes of transactions passing afterEventID to see history from a
previous point may not be supported. You may only be able to see recent transaction history.
*/

getAcountActivityResponse :=
      (transactionHistoryList, [mayBeMore], [afterEventToken]) |
      getActivityFailReason

getActivityFailReason :=
      INVALID_SESSION, TOO_MUCH_HISTORY, … //TBD - define complete list

transactionHistoryList := {transactionDetail} *

transationDetail :=
      basicTransactionDetail, [extendedTransactionDetail]
```

```
basicTransactionDetail :=
      EVENT_ID,
      TRANSACTION_ID,
      Amount,
      transactionAttributes,
      transactionDateList,
      [transactionState]

/*
A number of unique events with seperate EVENT_IDs can be associated with a single
TRANSACTION_ID.

When an update on a TRANSACTION_ID is received TPP should replace existing details for
that TRANSACTION_ID

NOTE: EVENT_ID(s) can be passed back into getAccountActivity(...) to return history from the
previous point.
*/

transactionState := BOOKED| ACCRUAL | CLEARED | HOLD | RELEASE

transactionDateList := { transactionDate }+
/* When in CLEARED state the list will contain the dates for all previous states */

transactionDate := transactionState, DATE
/* DATE - YYYYMMDD-[HHMMSS]] */

transactionAttributes :=
      transactionReference,
      transactionDescription,
      [merchantDetails | payeeDetails]

merchantDetails := … //TBD
/* This is critical  information for the AISP to be able to segment transactions into categories
Automatically */

extendedTransactionDetail :=
      [swiftDetails], [chequeImage], [payeeAdvice], [payerAdvice]

swiftDetails := … //TBD
chequeImage := ... //TBD
```

```
payeeAdvice := … //TBD
payerAdvice := … //TBD

/* NOTE: Unlikely we will get anything like this extendedTransactionDetail in the initial XS2A API
*/

afterEventToken := PAGING_TOKEN
/*
ASPSP should return transaction history in pages.
If there is more information available then afterEventToken returns a paging token.
When afterEventToken is passed into getAccountActivity(...) you will get the next page worth of
results

*/

mayBeMore := BOOLEAN

/*
mayBeMore is an optional attribute to return as part of getAccountActivity.
By supporting it however the ASPSP can avoid unnecessary calls to its systems by telling TPP
that there is no more results.
If mayBeMore is not passed back then the TPP should keep calling getAccountActivity(...)
passing in afterEventToken until an empty transaction list is returned
*/

getAccountBalance := SESSION_TOKEN, accountID → getBalanceResponse
getBalanceResponse :=
     ([accountCurrency], [supportedBalances], {balanceDetail}*) |
     getBalanceError

getBalanceError := … //TBD

accountCurrency := ISO_CURRENCY_CODE // If not passed assume domestic currency

supportedBalances := {balanceType}1*6
balanceType :=
     BOOKED | AVAILABLE | CLEARED | ACCRUING| UNCLEARED| HELD

/* If supportedBalances is not returned then you will have to wait and see what you get.

ASPSP should return all known balances at that point in time for the account
*/
```

```
makeDomesticPayment :=
      (SESSION_TOKEN,
      accountID,
      payerDetails,
      payeeDetails,
      Amount) → domesticPaymentResult

/* Payments will be made immediately or next available business day (if out-of-hours) */

domesticPaymentResult :=
      paymentResult, [paymentID], [domesticPaymentFailReason],
      [paymentStatus]

paymentResult := OK | FAILED

domesticPaymentFailReason = … //TBD

paymentID := UNIQUE_PAYMENT_ID
/* To be unique across all ASPSPs should be a GUIID */


getPaymentStatus :=
      SESSION_TOKEN,
      paymentIDList → (paymentStatusList | getPaymentStatusError)

paymentIDList := {paymentID}1*MAX_PAY_STATUS_BATCH
/*
List of previous paymentID to return status of.
For efficiency we should support batching up multiple calls.
MAX_PAY_STATUS_BATCH is the maximum batch size allowed
*/
MAX_PAY_STATUS_BATCH = 1000

getPaymentStatusError := … //TBD

paymentStatusList := { (paymentID, paymentStatus) }*

paymentStatus := UNKNOWNID, PENDING, SUCCESS, PENDING, UNKNOWN_PAYEE

payeeDetails :=
      payeeType,
      [intermediaryAccount],
      [payeeAccount],
```

```
      [payeeDetails],
      [payeeReference],
      [merchantDetails]

/*
One or both of intermediaryAccount payeeAccount should be passed.

TPPs should pass as much information to the ASPSP as possible else the transaction history
they get back will be poor
*/

payeeType := COMMERCIAL | PERSONAL | COMMERCIAL_FEE_EXEMPT

/* COMMERCIAL payments will have fees associated with them.
Similar to debit card fees i.e. approx 0.7% per transaction.

PISP should handle the fees and will send intermediaryAccount to identify its account that the
ASPSP should pay to.

PISP should not lie about payeeType to avoid the fees. The ASPSP will be checking.

PISP will deduct fees from amounts paid to merchants and will split the fees between PISP and
ASPSP.  ASPSP will receive the same fees that they get for domestic debit card transactions.
The PISP may have to sub-divide its fees further.

The PISP will batch up fees and send on a periodic basis to ASPSP daily or when
MIN_FEE_AMOUNT is accumulated
*/

intermediaryAccount := accountID
/* This is an account belonging to the PISP */

payeeAccount := accountID
/* If intermediaryAccount is passed then you do not necessarily have to identify the
payeeAccount. Although the regulators may insist on this to allow the ASPSP to perform AML
checking*/

payeeDetails := Name, [Address]

Name = [title], [firstName], [lastName]

Address := Address_UK
Address /= … // Address structure in other countries will be different
```

```
Address_UK :=
      {AddressLine}1*3,
      [County],
      [Country],
      [PostCode]
```

/ NOTE: Because of GDPR ASPSP should only keep payee address details for a limited time And just use for fraud detection */*

```
Amount : = [LocalAmount] , [FxAmount]
```

/ for getAccountActivity both LocalAmount and FxAmount can be returned. If transaction was billed in non-domestic currency.*

For makeDomesticPayment the Amount must be in LocalAmount

/

```
LocalAmount = AMOUNT_DECIMAL  //With correct decimal places for local currency

FxAmount = ISO_CURRENCY, AMOUNT_DECIMAL, [FxRate]

FxRate = fromCurrency, toCurrency, rate
fromCurrency = ISO_CURRENCY
toCurrency = toCurrency
Rate = FX_RATE_DECIMAL
```