4. Balances architecture

Merchant balances

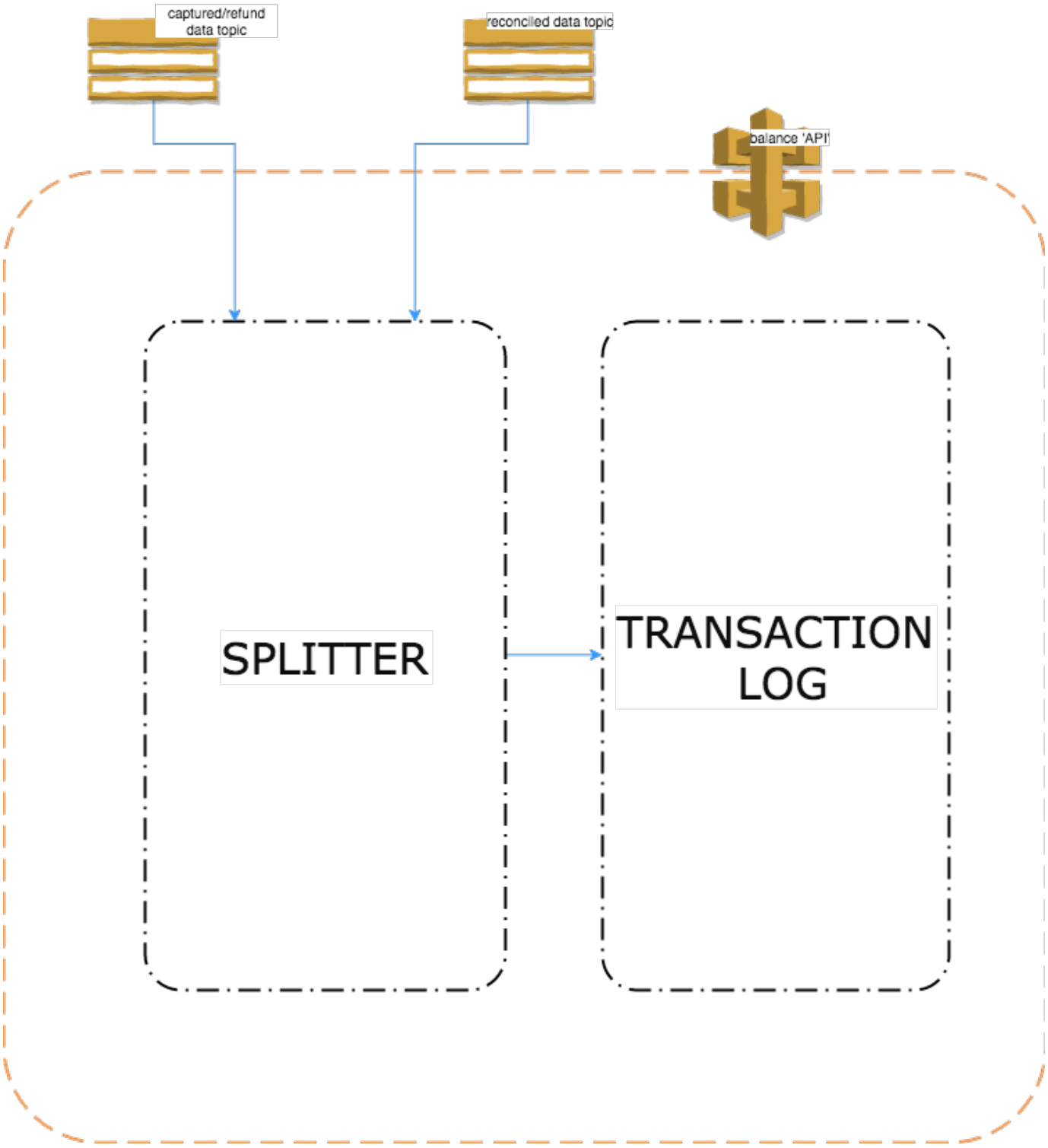
Falcon module that holds merchant balance (Falcon-B).

Falcon glossary

name & short name	description
Falcon Balance (F-B)	Falcon module that holds merchant balance
Falcon Merchant (F-M)	Falcon module that caches merchant context; it will basically mirror MC but only store, by internal representation of its choosing, the required merchant data that enable Falcon (payout configuration, etc)
Falcon Core (F-C)	Falcon module that holds all Falcon-related streams and topics (pub/sub for internal modules)
Payment Service Provider (PSP)	PayU runtime, usually country specific, that provide merchants with online payments. The available payment methods vary across providers or countries
Merchant context (MC)	Represents the global representation of a merchant that signed with either global PayU legal entity, or a PSP legal entity, in a particular country. This context is a small but crucial functionality, responsible for translating local merchant to global merchant
Confirmed Operation (CO)	Confirmed operation represents a capture or refund, performed in (local) PSP. The current API accepts local PSP Confirmed Operations, and Falcon is responsible for converting this to a global representation
Reconciled Operation (RO)	Reconciled operation represents a confirmation that PayU can trace the money for a particular CO in its bank accounts.
Balance Operation (BO)	Representation of a Confirmed operation, internal for Falcon Balances module

Intro

Conceptually, the module is represented by 2 components, as shown in the diagram below:



Splitter - while not strictly related to balances, and possibly moved to its own module later, the splitter is responsible to accept CO or RO, determine global merchant, based on MC, and generate BOs

Transaction log - balance component that records BOs. This component is used to answer all questions related to balances (how much money a particular merchant has, block the balance, end-of-month turnover, etc). Transaction log is the main component in the balances module and must adhere to the following restrictions:

- system of record for balances (merchants and other systems can reliably use F-B for balance-related questions)

- read-only component (no changes can be done to inserted data)
- auditability (provides the means to explain to both merchants and auditors source of money / movement of money)
- replay (provide the means to recreate any balance, from the beginning, by iterating through all records)

Implementation flow

F-B is interested in couple of events that are already routed by F-C: CO, RO & CreatePayoutRequest.

Although flows are similar across events coming from different topics (and possibly lambda functions could be reused), each of these flows is described below:

Consuming a confirmed operation:

- CO is published in a specific SNS topic `falcon-core-${env}-confirmed-operation-${region}`
- Splitter reacts to that as the initial step function of an entire process. As input, it gets the CO and as output it gives a list of BO of certain types (capture, refund, psp fee, etc). Each operation belongs to a specific global merchant and includes details of the local PSP merchant and shop where it was generated. For instance, considering a non-marketplace purchase with 1% psp fee, a capture of 100€ will result in two Operations: a +100€ of type capture and a -1€ of type psp fee, both for the same merchant
- The second step function is a lambda to persist the Operations in our Transaction Log (S3).
- The last step function will publish the result in a Kinesis stream so that we can build a balances cache
- There is yet another lambda that will consume from the Kinesis and reduce the balances information every 15 minutes (to start with), storing the information in a DynamoDB. This mechanism will work as a fast cache for sync operations that demand fast response, such as UI or sync refund authorization (async refunds or high-risk merchants can also use the transaction log for the same purpose)

Consuming a reconciled operation:

- RO is published in a specific SNS topic `falcon-core-${env}-reconciled-operation-${region}`
- Splitter reacts to that as the initial step function of the process - although similar to CO, this step function is simpler. As input, it gets the RO and as output it gives a list of BO of certain types (capture, refund, psp fee, etc).
- The second step function is a lambda to persist the Operations in our Transaction Log (S3), using different key than CO.
- The last step function will publish the result in a Kinesis stream so that we can build a balances cache
- There is yet another lambda that will consume from the Kinesis and reduce the balances information every 15 minutes (to start with), storing the information in a DynamoDB. This mechanism will work as a fast cache for sync operations that demand fast response, such as UI or sync refund authorization (async refunds or high-risk merchants can also use the transaction log for the same purpose)

As you can see, besides storing RO, the step functions are very similar to CO

Balance operation details

Each balance operation (BO) has the following structure (one CSV line for each operation):

column name	description
uuid	F-B unique identifier for this operation (generated when operation is stored)
external_reference	id for the external operation (psp reference, payout id, etc)
amount	operation amount
currency	operation currency
type	capture, refund, withdrawal, cashback, etc
falcon_operation_id	unique identifier of the operation inside Falcon system. This is very useful in order to track different state changes for the same PSP operation

operation_date	
local_merchant_id	id of the merchant in the local psp
shop_id	id for the shop with type local psp (defined in MC)
balance_shop_id	logical association between a shop and a currency (defined in MC)
psp_id	unique internal id for the psp that processed the transaction
merchant_id	id of the merchant (defined in MC). *This is directly stored in the S3 partition, hence not directly visible on *operation line, but still part of operation! **

Examples:

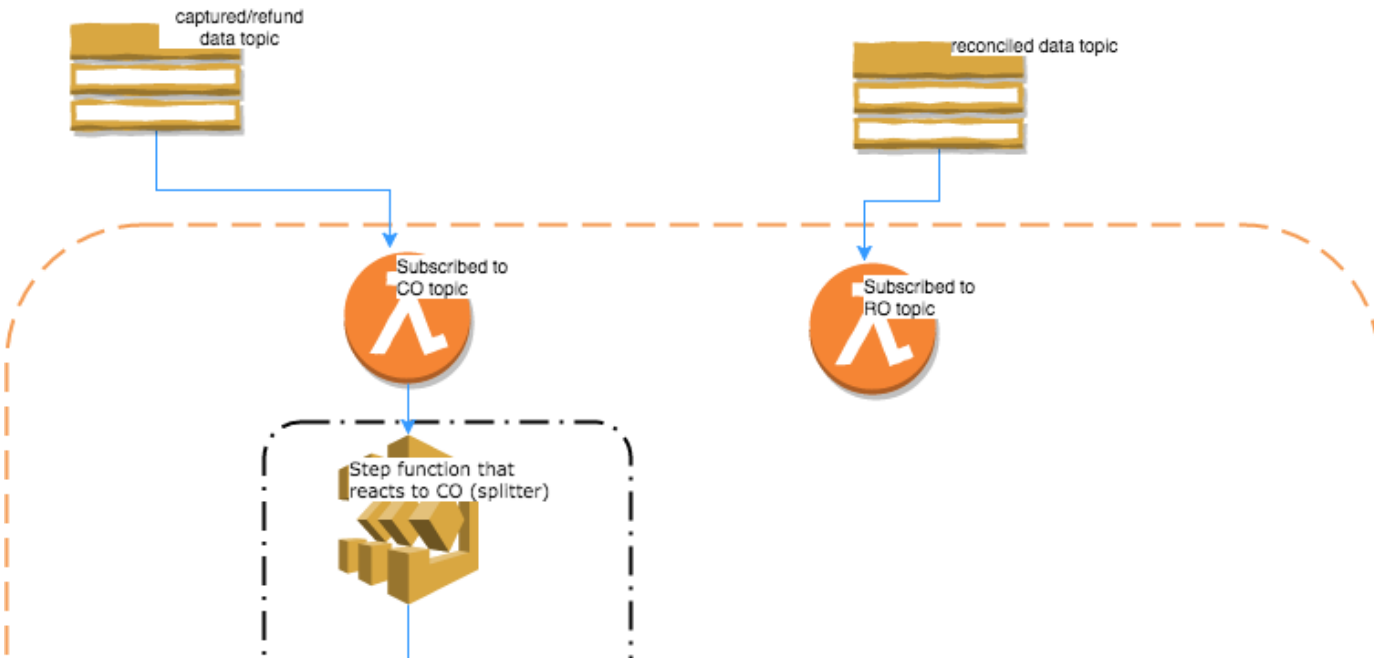
operation without fee:

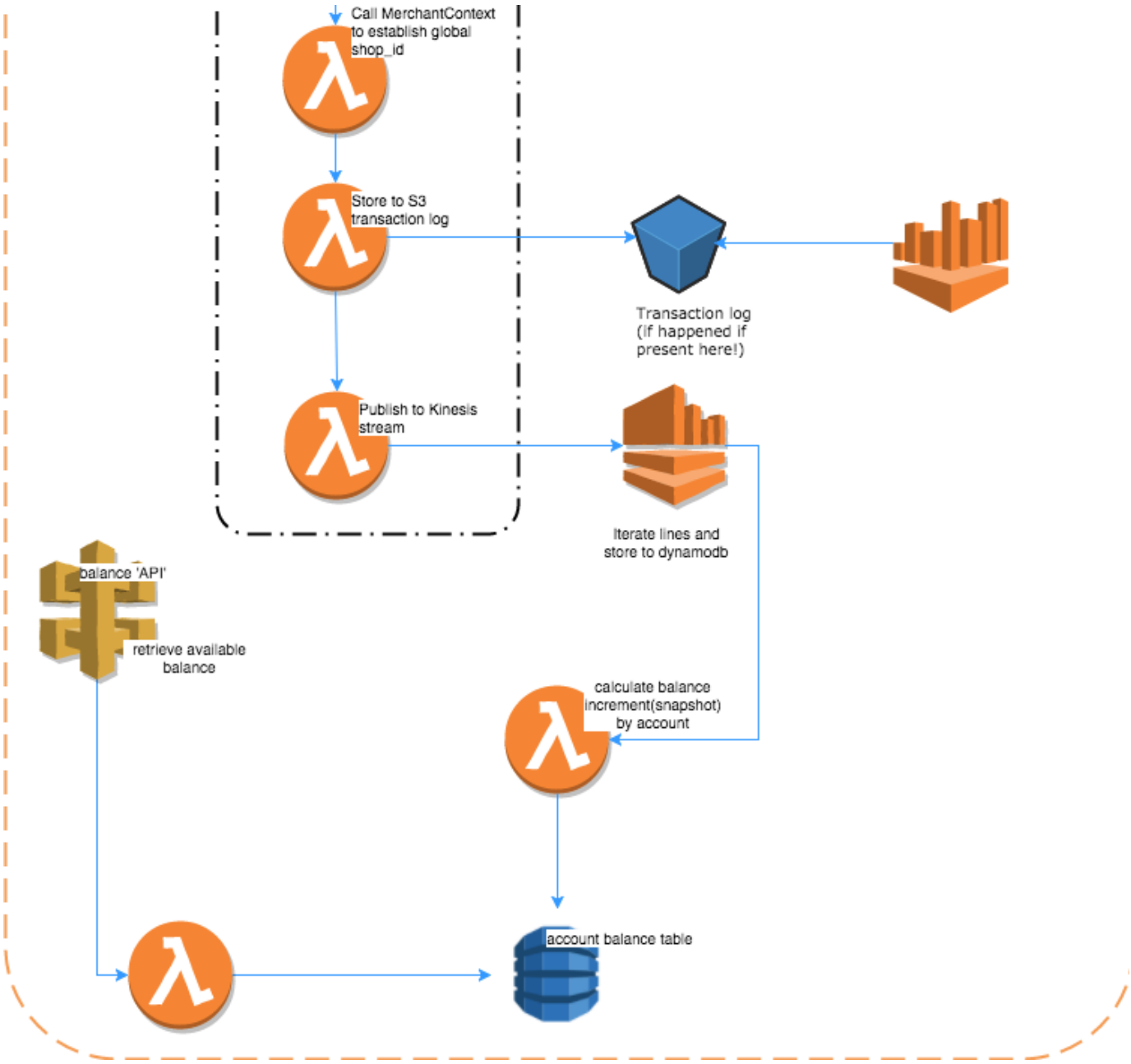
uuid	external_reference	amount	currency	type	falcon_operation_id	operation_date	local_merchant_id	shop_id
1234	asl8jk23er-234-agw	100	EUR	capture	1234	02/02/2018	5553	1cccaa4b-47ed-474a-a51f-59d0f8

operation with fee 1%:

uuid	external_reference	amount	currency	type	falcon_operation_id	operation_date	local_merchant_id	shop_id
1234	asl8jk23er-234-agw	100	EUR	capture	1234	02/02/2018	5553	1cccaa4b-47ed-474a-a51f-59d0f83a0bd1
1235	asl8jk23er-234-agw	-1	EUR	psp_fee	1234	02/02/2018	5553	1cccaa4b-47ed-474a-a51f-59d0f83a0bd1

More detailed diagram:




Call MerchantContext to establish global shop_id
Store to S3 transaction log
Transaction log (if happened if present here!)
Publish to Kinesis stream
Iterate lines and store to dynamodb
balance 'API'
retrieve available balance
calculate balance increment(snapshot) by account
account balance table