**The project here includes how to**

1. **Send payment to terminal**
2. How to do refund from terminal

**MID :**

A list of all the CVC and that’s it here

How would we do any debuggin down here

A screenshot of a computer

Description automatically generated

1. **MAID**

This is the equivalent of First American’s Merchant Id. It is a business entity that exists within a MID to process payments.

What are the usecases of using the POS terminal?

POS Use Case

1. Send a transaction to POS device (Reader)
2. Cancel a transaction
3. Reverse a previous transaction
4. Refund from previous transaction
5. Blind Refund
6. Reverse refund transaction

What does the transaction look like?

A diagram of a diagram

Description automatically generated with medium confidence

1. As you can see above this is what it looks like

**API design**

**How many new endpoints do you need?**

1. POST /sale

Sends sale transaction request to reader

1. GET /{txId}

Get transaction details here

1. PATCH /update/{tx}

Used to update the transaction

**Braintree API:** [**requestChargeFromInStoreReader**](https://graphql.braintreepayments.com/reference/#Mutation--requestChargeFromInStoreReader)

Input: inStoreTransactionInput

1. Amount
2. orderID

**Output:**

[**InStoreContextPayload**](https://graphql.braintreepayments.com/reference/#object--instorecontextpayload)

Top-level fields returned when requesting a state change on an in-store reader.

*fields:*

* **clientMutationId:** [**String**](https://graphql.braintreepayments.com/reference/#scalar--string)

An identifier used to reconcile requests and responses. 255 characters maximum.

This one is important as it will be used as the idempotency key here

* **id:** [**ID**](https://graphql.braintreepayments.com/reference/#scalar--id)**!**

A unique ID for this in-store context request.

* **reader:** [**InStoreReader**](https://graphql.braintreepayments.com/reference/#object--instorereader)

The reader associated with the in-store request.

* **status:** [**InStoreContextStatus**](https://graphql.braintreepayments.com/reference/#enum--instorecontextstatus)**!**

The status of the context created.

And the status is the 5 statuses that we discussed before here

**How does the happy path work on a high level?**

**Steps**

1. Finance-service sends a requestChargeFromInStoreReader request to the reader and receive RequestChargeInStoreContext.id
2. Store the transaction details in the DB.
3. Start polling job (or this can be achieved via pub/sub) to check the transaction status every 2 secs and check RequestChargeInStoreContext.status.
4. Update DB every time transactionstatus is changed and notify subscriber for UI to update.
5. Stop polling when status is COMPLETE, store complete transaction details in the DB and notify subscriber.

In other words

1. Finance-service will consume the CREATED message from Pub/Sub and call RequestChargeFromInStoreReader mutation to send the Tx to the Reader.
2. Retry will be handled by Pub/Sub. If Reader sends the response back then update the Tx in the DB with the following details.

updateTransactionRequest{

modificaitonDate

lastPing

brainTreeContextId

statusId

rawTxData

}

1. If reader sends response back then, at this point, transaction has been picked up by the reader and waiting for customer interaction. Finance-service has sent a message with status as PENDING to Pub/Sub and has sent another message to notify user. How UI subscription works will be a separate design d
2. Finance-service will consume the PENDING message, call BT node query, update Transaction and notify UI. Exact details of how retry/timeout works are in polling design

At this point, transaction has been completed successfully by the reader and payment status is SUBMITTED\_FOR\_SETTLEMENT

1. Finance-service sends message ACK that removes the PENDING message from the broker, update Tx and notify UI.

**What do we do in the case of errors for example gateway rejected?**

1. Finance-service sends requestChargeFromInStoreReader to the reader.
2. Receives transaction.status = GATEWAY\_REJECTED. This means most likely there is some configuration issue and Braintree is rejecting all transaction coming from Rhapsody.
3. Retry transaction by passing the same Idempotency-Key in the header to avoid duplicate transaction.
4. If we receive transaction.status = GATEWAY\_REJECTED again, then store the result in the DB and notify subscriber with the failure reason.

Database changes

1. **Create a new transaction table similar to existing pos\_pay\_transaction table.**
2. Create the table static\_transaction\_status to store transaction statuses
3. **static\_transaction\_failure\_reason**

**How does the transaction status get propagated to other services using this? Using the pub/sub?**

A diagram of a service

Description automatically generated

**Design walk-through**

Creating 3 separate topics as the configuration for each topic will differ   
Please see step 2 on how these steps are different

**Topic 1**: Messages in CREATED state. ACK deadline is 60 secs (Braintree API timeout)

**Topic 2:** Messages in PENDING state. ACK deadline is 180 secs (Reader timeout waiting on customer’s action)

**Topic 3:** Messages in COMPLETED, FAILED or CANCELLED state. Used to update UI. ACK deadline can be decided later.

Let’s review the message queue configuration before we dive deep into message processing

* **Delivery type:** There are 2 ways messages can be delivered to the subscriber. Pull or Push. Pull is recommended
  + **Push:** Message broker will send the message to the subscriber endpoint as soon as the message is available in the queue. Ideal if you have a reliable, always-available endpoint ready to process messages immediately.
  + **Pull (recommended):** Subscriber will pull the messages from the broker on-demand. Messages can be consumed in batches depending upon subscriber throughput. Ideal, if you need flexibility in processing rates and do not want to expose any public endpoint. There are multiple pull strategies:
    - **Continuous Pulling (recommended):** Continuously pull messages from Pub/Sub, with a short or no delay between pulls.
    - **Interval-Based Pulling:** Pull messages at fixed intervals.
    - **Event-Driven Pulling:** Trigger the pull operation based on specific events or conditions, such as reaching a message count threshold or resource availability.
* **Acknowledgement deadline:** How long Pub/Sub waits for the subscriber to acknowledge receipt before resending the message. Min/Default is 10 secs and Max is 600 secs.  
    
   **Braintree API timeout is 60 secs and Reader timeout is 180 secs**. Our subscriber should not timeout before Braintree API timeout.
  + **Topic 1**: Configure ACK deadline to be 10 secs with 3 retries and exponential back-off strategy.   
      
    If broker does not receives ACK after Tx with CREATED state message was sent to the subscriber within 10 secs, then message is put back to the queue and gets visible to the subscribers to consume.   
      
    After multiple retries, the message will move to DLQ which can be analyzed separately. The total\_timeout for a message to be processed should be >= 60 secs.
  + **Topic 2:** This is an interesting case when message in PENDING state is consumed and we need to call node query multiple times. There can be multiple approaches:
    - **ACK deadline is 180 secs**: Once the message is delivered to subscriber, it’s subscriber responsibility to keep calling node query every 2-3 secs until transaction gets COMPLETED or times out (180 secs). There is no point of setting retry policy as after 180 secs, transaction is marked as TIMEOUTand UI needs to send a new transaction.
      * **Cons** - If subscriber dies, the message will be lost as retry policy is 0.
    - **ACK deadline is 60 secs:** In this scenario, retry policy can be 3 retries with exponential back-off strategy. If broker does not receives ACK after Tx with PENDING state message was sent to the subscriber within 60 secs, then message is put back to the queue and gets visible to the subscribers to consume. After multiple retries, the message will move to DLQ which can be analyzed separately.
      * **Pros -** Even if subscriber dies while processing the message, the message will be visible again after 60 secs for subscribers.
    - **ACK deadline is 3 secs:** With such a short ACK deadline, we are relying on broker retry policy and making subscriber super light weight and not worrying about retry logic on the subscriber side. Messages in PENDING state will be delivered to the subscriber, subscriber makes a node query call with timeout of 2-3 secs and sends ACK if Tx is COMPLETED or FAILED else does nothing. Message becomes visible again after 3 secs for subscriber and another node query call will be made.

Other questions such as

**What are the subscription filter, message rentention duration and all that?**

* **Subscription filter:** Subscribers will only receive messages that match the filter. If we are going with separate topics then we don’t need the filters.
* **Subscription message retention duration:** Messages will be retained for up to the message retention duration of the topic or the subscription, whichever is greater.
  + Can be configured to 7 days. Messages will live on the topic even if finance-service is down for 7-days. Ideally, messages will move to DLQ post retry failure.
* **Message ordering:** Messages tagged with the same ordering key will be received in the order they are published. We can enable this feature to make sure Txs are processed in FIFO manner.

**Deep dive the above is on the high level, here is a deep dive**

**Step 1: Send new transaction**

* Portal UI sends a new transaction to public-api which is further sent to Finance-Service.

@Transactional

* Finance-Service creates a new transaction with status as CREATED and writes it to DB.
* Writes a message to pubsub (topic 1) for subscriber to process the transaction.

Sample payload

{

id: UUID // Tx Id

status: CREATED

creationDate: Instant

businessId: Integer

...

}

**Step 2: Process CREATED messages**

* Finance-Service will consume the messages from the topic-1 with the filter on status field.   
  The status should be either CREATED. Messages with any other status will remain in queue and then eventually be dropped after retention duration.

@Transactional

* Call BT with API timeout of 10 secs, here the mutation we are calling is the

requestChargeFromInStoreReader mutation

* If successful
  + Update DB Tx
  + Send ACK for consumed message
  + Publish another message with status as PENDING to topic-2
  + Publish message to topic-3 for UI to update
* If timeout
  + Broker will send the message again with retry policy until max timeout of 60 secs.

**Step 3: Process PENDING messages**

* Finance-Service will consume the messages from the topic-2 with the filter on status field. The status should be either PENDING.   
    
  Messages with any other status will remain in queue and then eventually be dropped after retention duration.

@Transactional

* Call BT with API timeout of X secs.
* If successful or failed
  + Update DB Tx
  + Send ACK for consumed message
  + Publish COMPLETED, FAILED or CANCELLED message to topic-3 for UI to update
* If timeout
  + Broker will send the message again with retry policy until max timeout of 180 secs (reader timeout)

**What’s the acknoledgment in pub/sub?**

In Google Cloud Pub/Sub, once a message is acknowledged by a subscriber, it is considered successfully processed and is removed from the subscription's queue.

**What happens when another transaction sent to a reader that is in the middle of another transaction.**

AFAIK: If reader is in Pending state, then new transaction will override the previous transaction.

If reader is in the middle of transaction then it will reject the new request.

@Rachit Puri confirm with Braintree. We might not need to track the reader status in the Rhapsody side if it’s already take care on the Braintree side.

**What happens if customer doesn’t pay or tap after a while**

* 1. Braintree sends a timeout status when customer do not tap pay within the timeframe

**What’s needed to cancel the transaction?**

1. Need to pass the inStoreContextIds
2. The mutation looks like below

mutation RequestCancelFromInStoreReader(

$input: RequestCancelFromInStoreReaderInput!): InStoreContextPayload

{

input: {

clientMutationId: String,

inStoreContextId: ID!

}

}

What about refund?

mutation refundTransaction($input: RefundTransactionInput!): RefundTransactionPayload {

input {

clientMutationId: String,

transactionId: ID!

refund {

id: ID! -> Unique identifier,

amount {

value: 10.00

currency: USD

}

}

}

}

New endpoint for this

Purpose

Sends refund transaction request to reader

Request path: POST/Refund

**How to achieve idempotency (Avoiiding duplicate charges)**

To provide idempotency on the request charge mutation, you must also include the HTTP header Idempotency-Key with a unique value. UUIDv4 is recommended. (Braintree recommendation)

Idempotency key in this case is the token that you get back when first requesting a transaction here

For example, if you request a charge from the reader, the customer completes the charge on the reader, but for some reason, the POS does not get back the transaction result before timing out. In this scenario, the POS could recover the original transaction using the same Idempotency-Key without accidentally creating a duplicate charge.

BrainTree M2B documentation part 2 here

**What’s Braintree singup flow**

For each Practice’s Pet Parent signing up for a membership plan, Rhapsody will setup a Braintree subscription plan. Rhapsody will send key membership data to Braintree, including but not limited to, charge amount, billing frequency, Pet Parent’s email, and a Rhapsody id linking the Braintree subscription to the customer’s membership plan.

Once the Braintree subscription is created, Rhapsody, near instantly, will send a payment charge authorization request to Braintree. When Rhapsody receives a successful payment charge authorization message back from Braintree, the membership will be activated. If the payment charge authorization is not successful, then the Rhapsody does not activate the subscription plan.

**Endpoint change**

1. POST /registration/signup/{token}/braintree

Pbt-service-wplans

We will send the token,

When signup, we get a synchonrous success/failure response here   
right away, with no delay pretty much

**Database Changes**  
  
1. We need a wp\_scheduled\_job here   
  
 that has the following fileds, braintree\_subscription\_id, wplog\_id  
 creation\_date, modification\_date, scheduled\_date, job\_category, job\_detail

2. class RechargeJobDetail extends WpJobDetail here   
  
 patientId, subscriptionId, retryCount, retryLimit, wplanLogId, retryLimit   
  
3. Wp\_scheduled\_job\_log   
  
 to log all the results of a job   
  
 DbWplanScheduledJobLog then has column   
 - id, jobId, executedDate, succeeded, result

Class RechargeJobResult extends WpJobResult{  
  
 transactionId, processorCode, Map<String, String> errors

How is pub-sub linked up with the subsciber (**BraintreeUnsuccessfulChargeSubscriber)** for example?

Step 1:

Create the topc and subscriber in terraform repo  
  
Step 2:

In pbt-service-wplans:

In the application-k8s.yml file

braintree:

pub-sub:

canceled-subscription-subscriber:

rhapsody.v1.finance.braintree\_webhook.rhapsody.wplans.subscription\_canceled

subscription-charged-unsuccessfully-subscriber:

rhapsody.v1.finance.braintree\_webhook.rhapsody.wplan.subscription-charged-unsuccessfully

**What are ths components of retry and decline in pub/sub**

0. Webhook section here

1. class BraintreeWebhookMessage<T>   
  
 used to indicate the BraintreeWebhookMessage   
 - Long id; String kind; Instant timestamp T data

2. **BraintreeMessageHandleService**

- handle the message that we receive from braintree if sucecesful or unsuccessful. The kind here can be SUBSCRIPTION\_CHARGED\_UNSUCCESSFULLY

3. **BraintreeUnsuccessfulChargeSubscriber**

- this extends PubSub, and handles BraintreeWebhookMessage

- if not successful then we pass this to the messageHandleService (point 2)

3.5 BraintreeSubscriptionCancelledSubscriber

This is for subscription cancelled

**4. Subscription**   
  
 should be created for rhapsody.v1.finance.braintree\_webhook

**5. WpLanCommonSchedulerJob** @Profile(“job”)

- which runs a job here based on certain condition

6. Clean up the job here

Scheduled jobs for recharge and notification should be cleaned up

1. when payment method is updated.
2. Whenever SubscriptionWentActive webhook notification is received.
3. Recharge is successful

Basically this call this.wplanScheduledJobCRUD.terminateAllBySubscriptionId(wPlanLog.getBraintreeSubscriptionId());

When updatePaymentMethod() is called

**What happens if it’s unsuccessful flow?**  
  
 1. Sync the state of the plan

2. Send payment fix notification

3. Check if we need to rescedule a recharge job here

4. And the condition for this is

1. should recharge if the latest transaction is:   
  
 - retryable and no existing scheduled jobs

- retryable for the updated payment method

- retryable for a different error code

Flow for create scheduler to execute recharge jobs

1. fetch scheduled jobs that are ready to recharge from table wp\_scheduled\_jobs.
2. Execute recharge sequentially for each scheduled job by calling Braintree retry charge API.
3. When a recharge request is sent to braintree , update scheduled job and create job log
   1. if recharge succeeded, update scheduled job state = TERMINATED, job log execution SUCCEEDED = true
   2. if recharge failed (error returned from braintree), scheduled job state = SUBMITTED, job log SUCCEEDED = false
4. If retryCount >= retryLimit and not 3.a,
   1. set the scheduled job state as EXHAUSTED
   2. save a new job log into wp\_scheduled\_job\_log table where SUCCEEDED = false
   3. save WpJobResult as Json bytes in wp\_scheduled\_job\_log
   4. send a notification to client for payment method update.
   5. schedule a notification job in 10 days for payment method update.

**Why use terraform for pubsub?**

0. In our repo this lives in pbt-terraform

1. Google Cloud Pub/Sub is a messaging service that allows applications to communicate asynchronously. Using Terraform, you can automate the creation of topics and subscriptions,

2. We create the subscription in the pubsub based on a topic here

For example here we have the module

module "rhapsody\_v1\_finance\_braintree\_webhook\_rhapsody\_wplan\_subscription-went-active\_dl" {

project\_id = var.project\_id

topic = "rhapsody.v1.finance.braintree\_webhook"

grant\_token\_creator = false

And in pbt-service-finance  
 We configure in the app.yml file   
 webhook-log-topic:

webhook-log-topic: rhapsody.v1.finance.braintree\_webhook

webhook-log-subscription: rhapsody.v1.finance.braintree\_webhook.rhapsody.finance

Due to a number of foreign keys revolving around the wellness plans structure, we will need to add columns to existing tables to help differentiate the Braintree and Stripe wellness plans.

**wp\_customer**

// Add columns

braintree\_customer\_id // text

wp\_log

// Add columns

braintree\_subscription\_id // text

braintree\_state // text

wp\_payment

// Add columns

braintree\_transaction\_id // text

braintree\_subscription\_id // text

wp\_plan

// Add columns

braintree\_plan\_id // text

Payment update flow   
  
The memebership renews every month, so when it renew fails, we need to schedule a job to retry the recharge **once a day for 8 days**. After 8 days,  
   
 After retry attempts have been exhausted, a message needs to be sent to the Pet Parent. The message needs to be sent per the Pet Parent’s preferred commation (SMS or email) set in Rhapsody with a link to update here

What are components involved here?

1. Breaintree plan id   
2. Patient id

3. Braintree subscription id

4. An ema  
  
  
  
This is important please pay attentino here

1. We have a job exectuor   
  
to send payment method update message, when a recharge attempt fials   
we sendnotification of payment fix-up and schedule a delayed notification to be sent in x days.

**Retry flow**

When a Braintree decline code is received with processorResponseType = “soft\_declined” AND cardType = “Visa” AND Column E in the BT Decline Code Tab (Visa Category) <> 1, then automatically retry payment authorization **once a day for 8 days**. After 8 days,