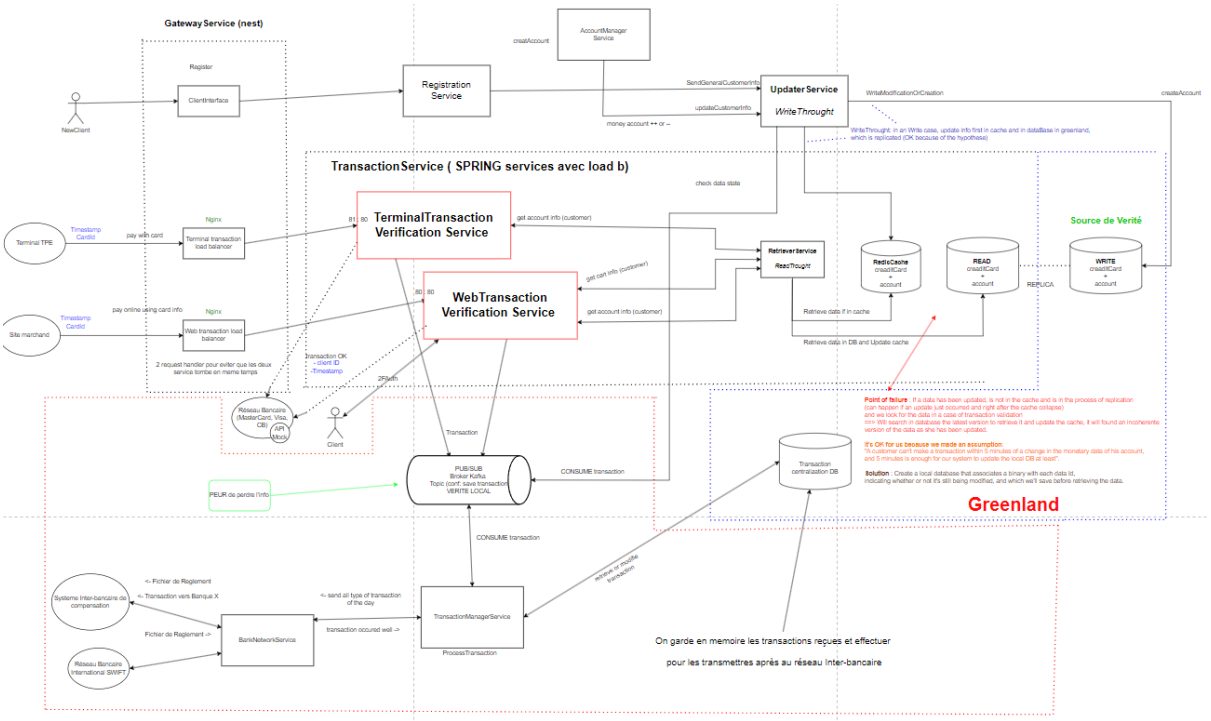
Follow-up Architecture Report
AL Néo-Bank

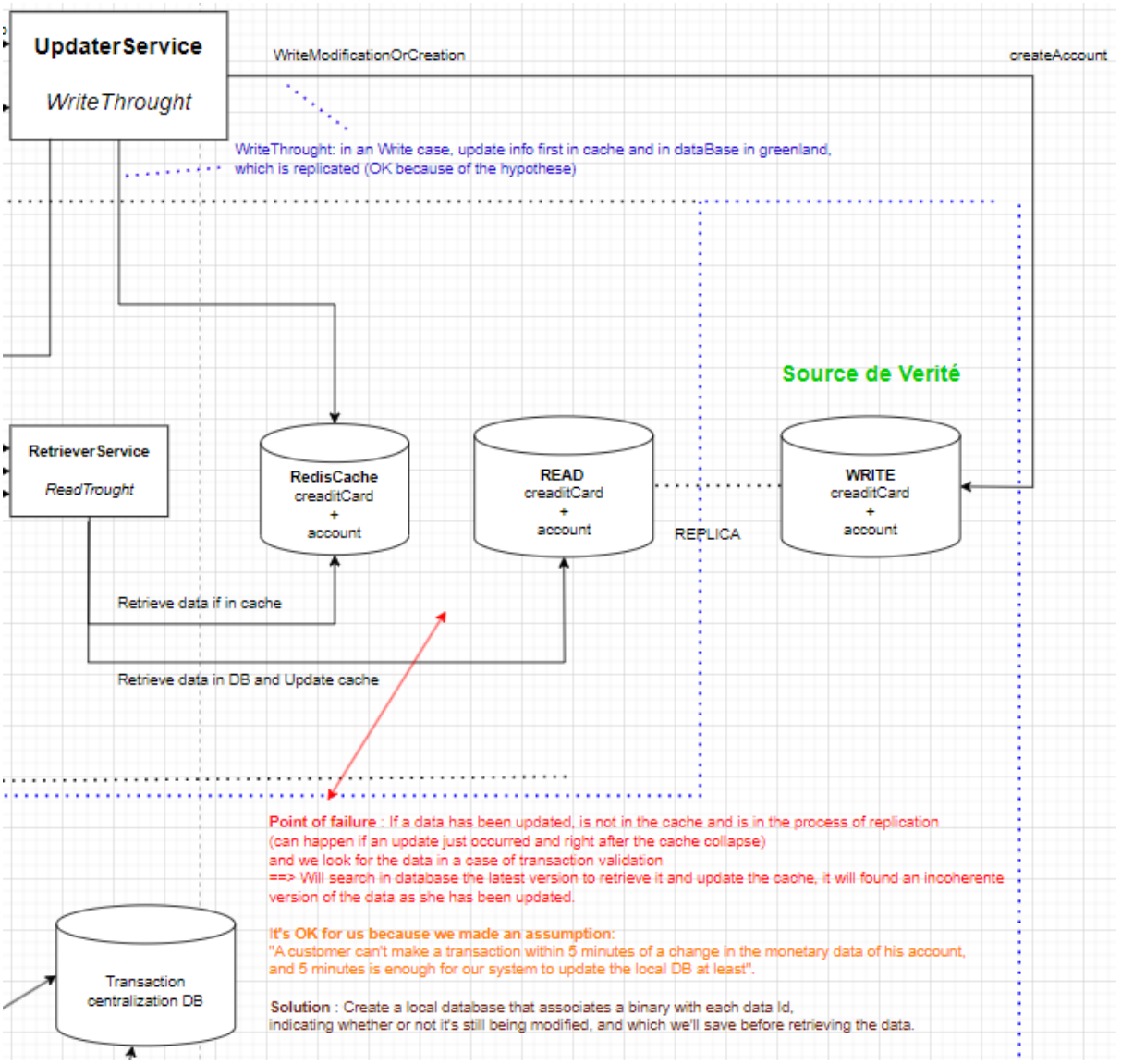
Week 44 (after vacation) :

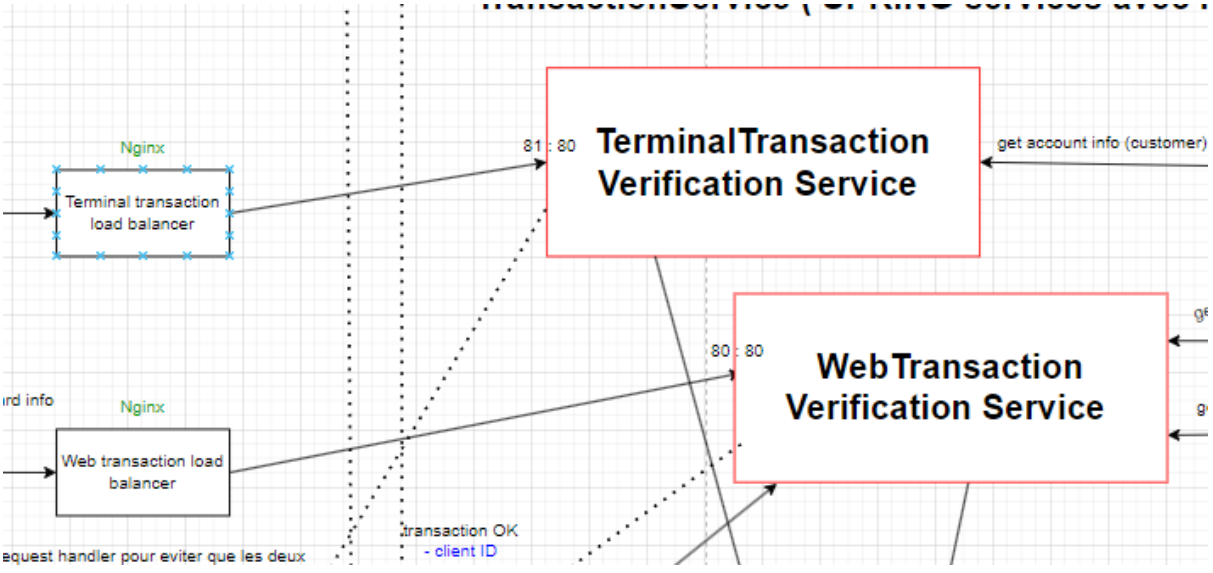
Current Structure of the services architecture :



The latest changes were made to the database, following the various studies carried out on the previous models, implemented or not. We deduced a data model that combines data consistency, high availability and speed.
In effect, we've decided to read and write to the database via a cache, so that the data is replicated on the cache, which will respond rapidly to the request. If the data is present in the cache, we retrieve it, if not, we retrieve it from the database and duplicate it on the cache. We only duplicate exactly what we need in our context of use, not all the values. We can do this because it's through a service that we'll perform persistence on the cache.

We'll have two services, one for writes and one for reads. This way, these services can respond quickly and perform CRUD. In fact, we've decided to separate the CRUD logic from our request verification services in order to make them more accountable. In the context of this POC, our resources are limited, so we can't perform a perfect verification, but this one embeds more complexity, especially as this CRUD deporter logic could also be used by other persistent services in our application. (See image below)


UpdaterService
WriteThrought
WriteModificationOrCreation
createAccount
WriteThrought: in an Write case, update info first in cache and in dataBase in greenland,
which is replicated (OK because of the hypothese)
Source de Verité
RetrieverService
ReadTrought
RedisCache
creaditCard
+
account
READ
creaditCard
+
account
REPLICA
WRITE
creaditCard
+
account
Retrieve data if in cache
Retrieve data in DB and Update cache
Point of failure : If a data has been updated, is not in the cache and is in the process of replication
(can happen if an update just occurred and right after the cache collapse)
and we look for the data in a case of transaction validation
==> Will search in database the latest version to retrieve it and update the cache, it will found an incoherente
version of the data as she has been updated.
It's OK for us because we made an assumption:
"A customer can't make a transaction within 5 minutes of a change in the monetary data of his account,
and 5 minutes is enough for our system to update the local DB at least".
Solution : Create a local database that associates a binary with each data Id,
indicating whether or not it's still being modified, and which we'll save before retrieving the data.
Transaction
centralization DB


Nginx
Terminal transaction
load balancer
81 : 80
TerminalTransaction
Verification Service
get account info (customer)
80 : 80
WebTransaction
Verification Service
Nginx
Web transaction load
balancer
transaction OK
- client ID

We've also decided to add load balancers via Ngix before the transaction verification services, as this allows loads to be distributed between several instances, thus managing greater demand. To do this, we use a Round Robin algorithm, so we don't need any further optimization. (See image above).

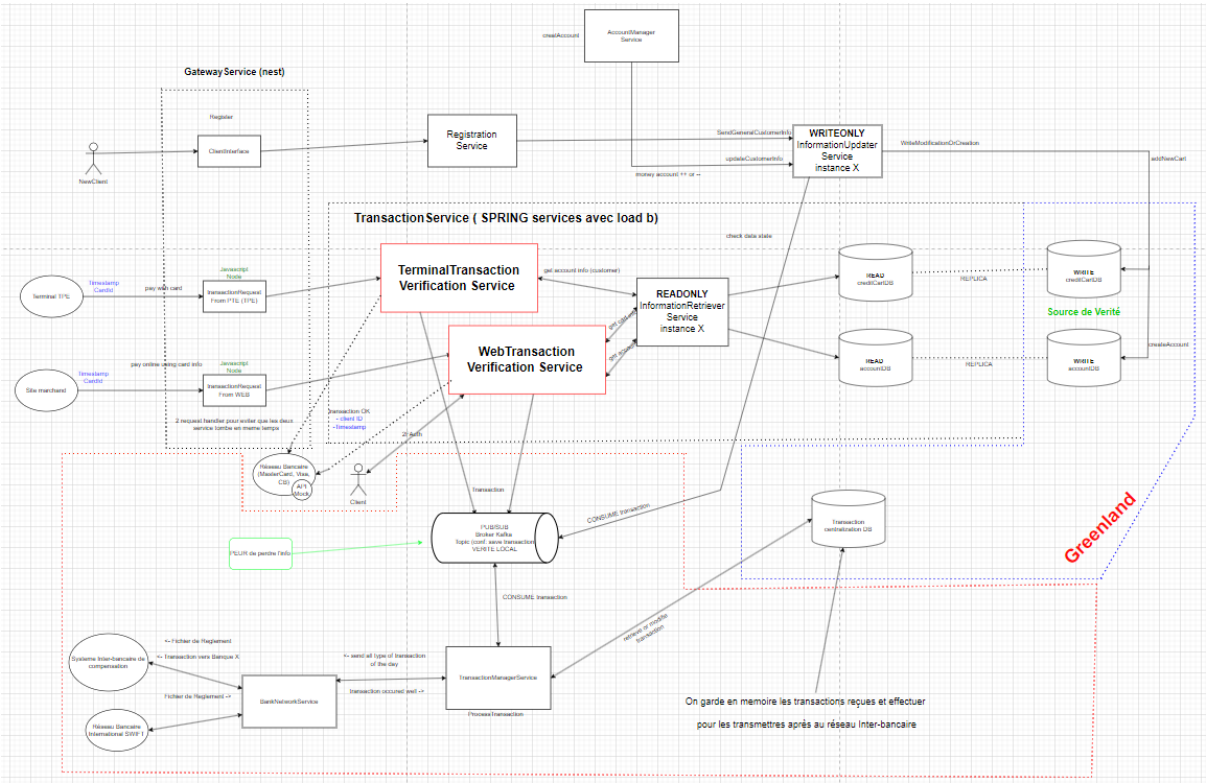
We're thinking of adding a load balancer in front of multiple instances of our new solution's crud services, but since we're only doing CRUD, we're thinking that using Spring Web Flux for asynchronous and concurrent code, for example, might enable us to fill the equivalent demand at service level. In fact, time constraints are holding us back, as the configuration steps are very time-consuming.

Scenario of the week :

To test all the first transaction validation par using everything but the caches, indeed time remaining to configure cache may take time, and we want to have the main flow before.

Week 42 :

Current Structure of the services architecture :



- Some change at the level of the databases has been made. The following hypothesis has been put forward: *a customer who is in the process of making a paying transaction (which means a transaction involving a change to his money in his bank account) cannot at the same time make a major change to his account, at least in the category linked to his money in the bank*.
  With these hypotheses in mind, we decided to reduce the complexity of the architecture at the database level. In effect, we would have a CQRS architecture with databases in master script at the Greenland level, which replicated their data at the level of the local Slavs read only in the region.

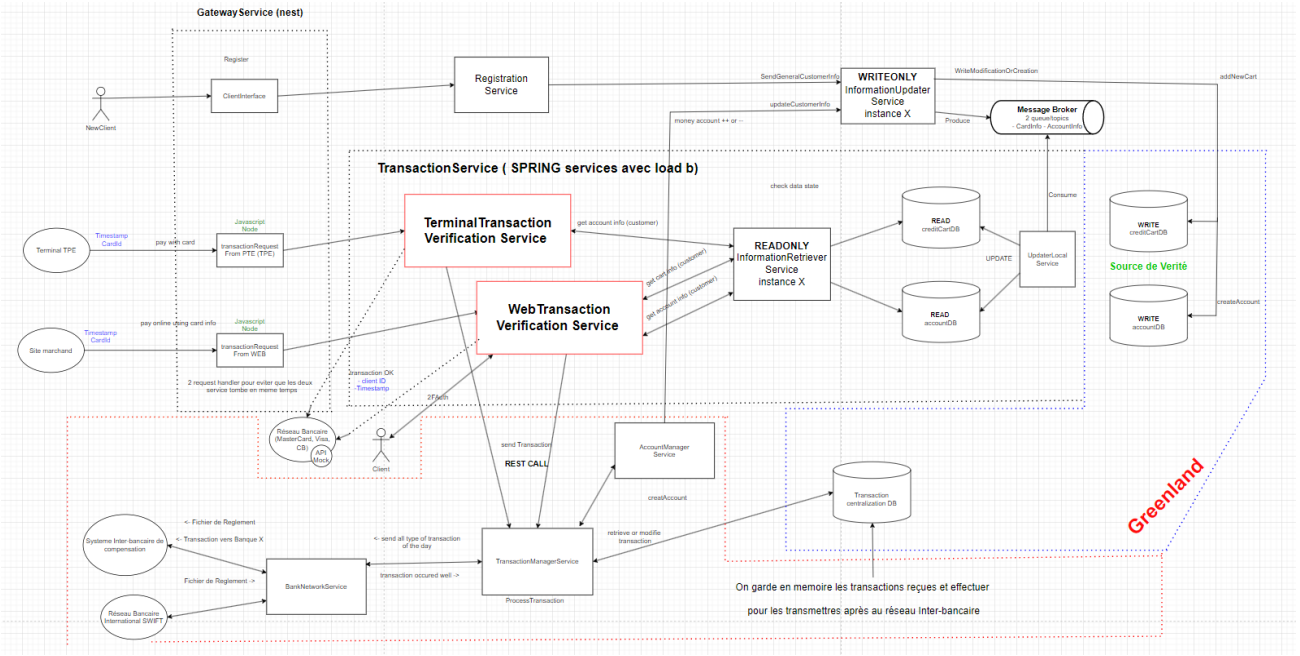
- Also at the level of resilience of a validated transaction data that has to go and modify the database. We've found that it doesn't pass through too many services, which increases the risk of losing the data in the event of a breakdown or other failure of a service that doesn't have much interest in being present in the flow. That's why we've removed the AccoutManager service, and set up a data bus to work more on an event-driven basis, with a "transaction validated" event that will be stored in memory on the bus, then consumed by our database write service.

Scenario of the Week :

We're still in the same scenario as last week, but in parallel we're also developing a load balancing solution on several instances of our transaction services. Indeed, in the case of a basic implementation, with the service retrieving the values to be checked from the database, and those in a case of large demand, the services may struggle, hence this initial implementation to see how it would work with a load balancer.

Week 41 :

Current Structure of the services architecture :



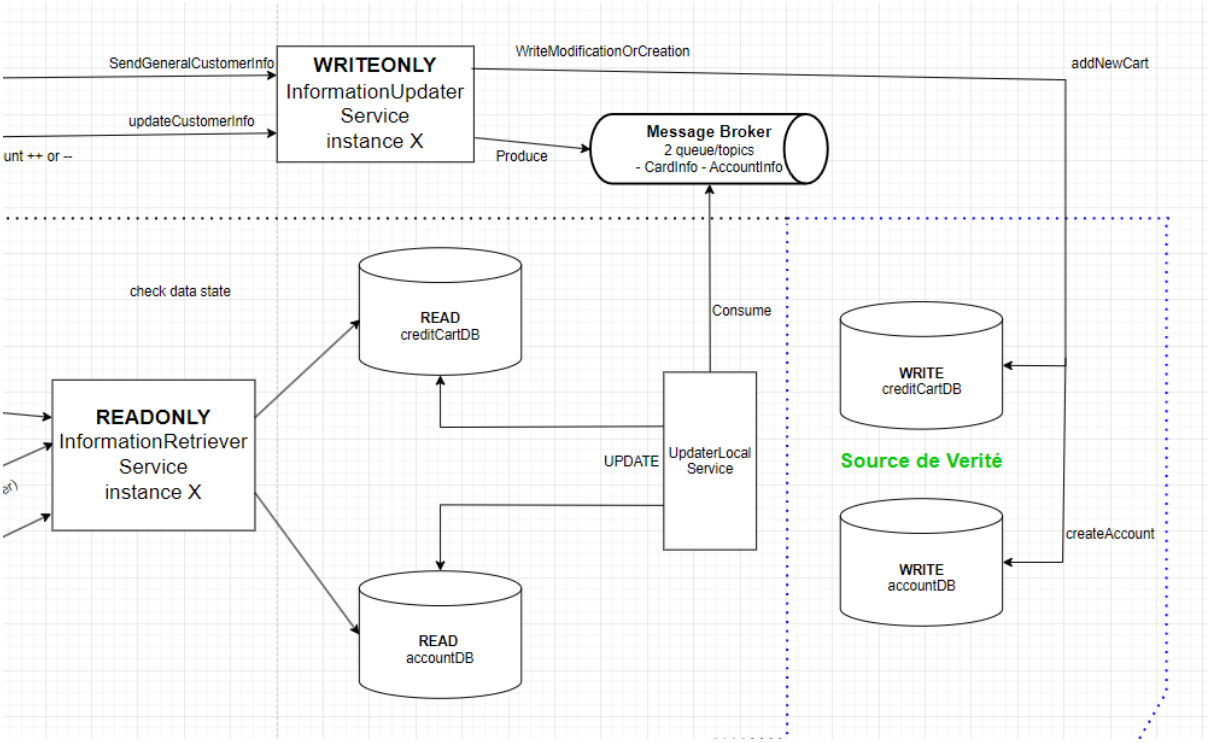
In our new service architecture, we haven't really changed the logic in terms of how transactions work or anything else - everything is still faithful to what we did the week before. But we did look at what we were going to do about the databases.

The challenge for us was not only to reduce database costs, but also to be able to retrieve data as quickly as possible. So we absolutely had to have databases that were always available and with consistent data, since the account information from which we validated a transaction had to be the same after updating. In order to avoid verification on biased data.

So, put in perspective with the CAP theorem, here we would give priority to consistency and availability. For partitioning tolerance, we assume that each time data is modified or added, it is duplicated between the Greenland database and the local database. Local databases are read-only, but in the event that they are no longer available, following a 50x server-side error for example, we'll send the request to Greenland so that we can still access the data.

The bottom line is that: the validation of a banking transaction must, in as many cases as possible, return a rapid response, whether or not it has been validated; a read transaction to a database must always be able to do this, and must return data consistent with the latest state of the data.

Architecture ZOOM :



Here is the part of the architecture that has really changed within a week. Indeed for this one, we have implemented a solution based on the CQRS model. We would have two services, one to write and the other to read. We would also have two databases in Greenland, and two others locally depending on where we are deploying. We would have a Broker that will notify the update service of the incoming modification sent to Greenland.

We thought of this solution only for this part without including the transaction database one because for now, we don't feel that we will need it as the real fast part is where, and in the context of validation of a bank transaction, we want it to be fast. For what comes next, it can be slower.

New Service Serving CQRS :

ReadOnly service : this service is here to retrieve data from a local database for each type of data that we would want. If an update and read action occurs at the same time, the update one occurs first. This service is the one that will increase its instance in case of a big scale request.

WriteOnly service : This service is here to modify any type of information. It accesses the Greenland database and through the broker sends the update to the UpdateService which only Job is to perform an update of the data in the readOnlyDatabase.

Why CQRS :

We thought of the CQRS pattern because in our case, we, from the beginning, were doing things on two different levels. The first one being action that occurs but only needs to read things from the database, the second one being all other cases that will involve account management.

The second reason for this choice was that, in terms of data recovery, we needed a model that would enable us to recover data quickly. We couldn't make a request all the way to the greenland, as this would potentially take too long, and could quickly load the databases, knowing that other actions would be taking place.

So this model allows us to distinguish between reading and writing, and if other functionality involving fast reading arises, the solution would be to add an uptade service on the duplicate of a new database.

The last point is that, based on our current knowledge, this solution seems more suitable and general, rather than having databases with a master-slave relationship of data replication. But this approach can be used as an alternative depending on our results.

Persona (no change):

- Fabrice : technician at the new neo-bank "IgorBanque", his job is to keep BNP IgorBanque's servers up and running.
- Philippe : A merchant who recently opened a cookie-shop in Nice, he offers customers a means of paying by card, or on the internet via click and collect.

User Stories (no change):

- As Fabrice, I need to know from the cardNetwork which kind of transaction is coming
- As Phillipe, I need to know from my TPE terminal if the client transaction have succeeded
- As Phillipe, I need to be notified if a transaction have succeeded if the purchase of the click and collect have been made on my website
- As Fabrice, I need to know in the bank systeme if any kind of transaction have succeeded

Scenario for this week :

Phillipe second scenario, but with one customer (phillipe), and multiple customers, in order to explore what could happen if in the case of a naive approach we try to handle a really large amount of bank transaction.

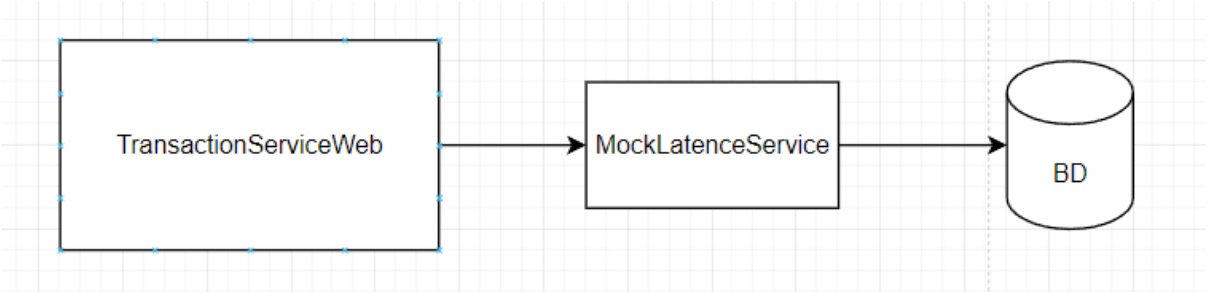


Image of what could be the first naive implementation

In this scenario, we will suppose that our service in requesting data directly from greenland, and to do so, we will have a service that will retrieve data with latency to explore what happens when :

- the transaction service request data from database
- the transaction service request data from database with latency

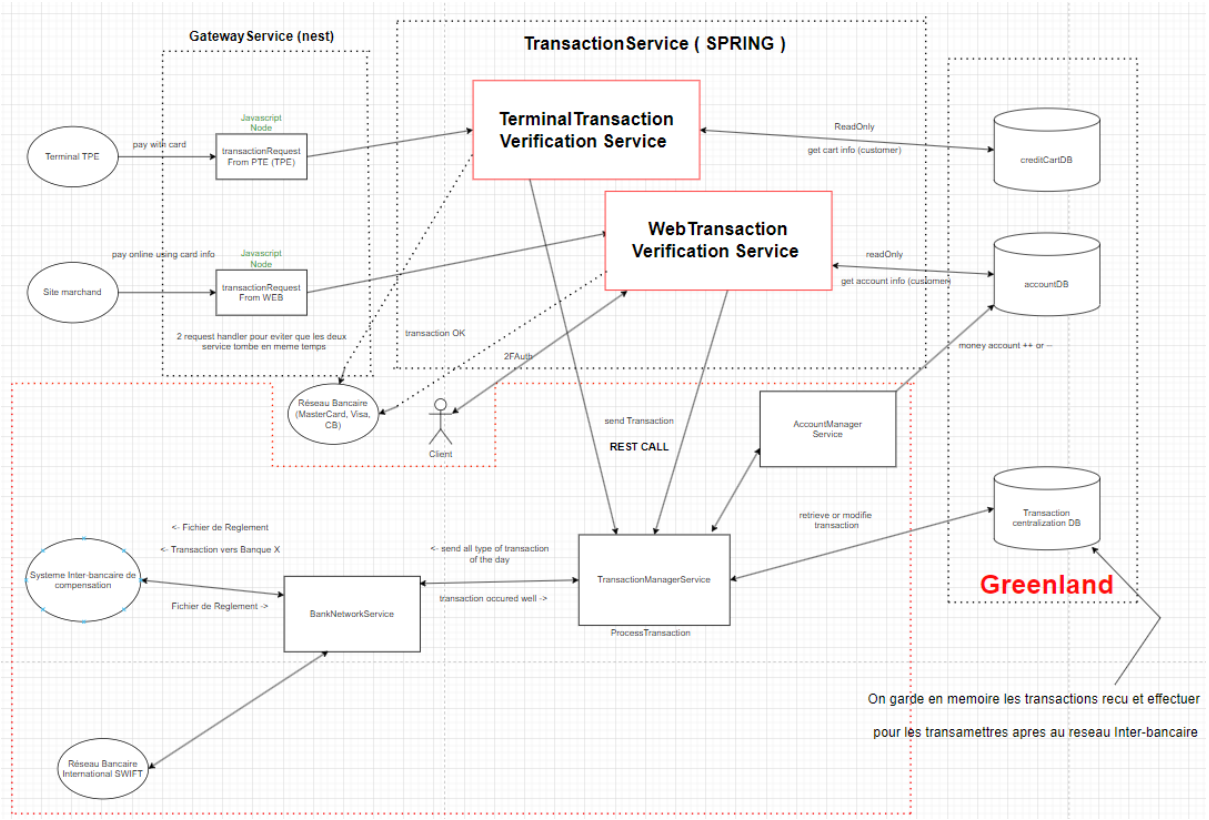
this two scenario will have multiple test with different amount of data to treat

For the intermediar latency service, it will be implemented using some optimisation so it don't crash while simulating, indeed we just want to simulate what could happens to the transaction service if a too big amount of transaction object are created to validate it, in the case where we could wait 0 second or 2-3 second.

So we can test resiliency and also the performance of this naive solution (it also give us a first implementation of our main service for this iteration)

Week 40 :

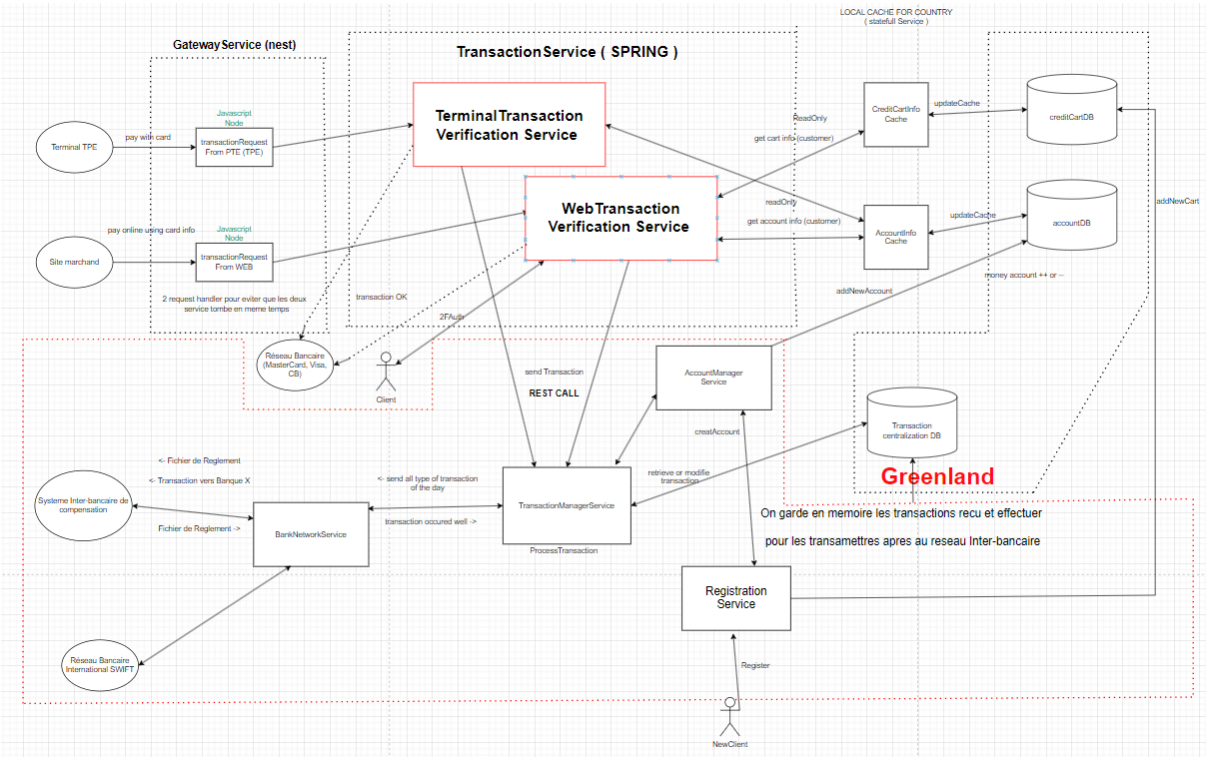
Current Structure of the services architecture :



Our architecture has been designed to meet a primary need: the scalability of our banking system. Indeed, the first type of service provided will be the banking transaction, so we decided to focus on this one.

In this diagram, we've separated the former large transactionService block into two services, one responsible for transactions issued from a TER and the other for transactions issued from another type of terminal, in our case a CB, Visa or other web-based banking transaction service. We have decided to leave bank transfers aside, as they are not a priority for the moment. This separation allows us to better manage a large flow of transaction requests. There will indeed be a certain amount of repetition in the verification process, but this will enable us to have a faster verification service in the TER case..

As far as the databases are concerned, at first we thought of a read-only solution. But given that we'll have to process a large number of requests in several countries in the future. With databases located in Greenland, this can be very slow. So we've come up with a kind of caching solution that would be set up on the servers in each country. Each time the databases undergo an update via a modification, they notify the caches that change their data. This solution would be even more feasible for credit cards, given that this information changes very little. Here's what the diagram would look like:



The parts circled in red are part of the transaction system (apart from customer registration, which is there to populate the databases). In fact, the most important thing is to create a simple transaction schema with the various rules and verifications linked to it. Then we'll add the system for notifying the banking network of successful transactions, and the processing of settlement files. These are logs of all the transactions carried out during the day, and it is on the basis of this file that we add or withdraw money from the various customers.

Note also that in the real world, a transaction comes from the credit card network and is then processed, so we've chosen to distinguish two types of transaction, rather than abstracting one from the credit card network.

Services :

TransactionRequestFromTPE:

- His role is to process the transaction for a TPE on client side, and to send only needed info to the service

TransactionRequestFromWEB:

- His role is to process the transaction coming from an NetworkCard API, such as Master, Visa or CB, and to send only needed info to the service

TransactionManagerService:

- Its role is to process a transaction when this one is validated by the validators services, it sends the transaction to the AccountManagerService so this service can process the impact of the recent transaction on account. He also added the transaction to the list of transactions of the day in the TransactionCentralisationDB. When the daily cycle has finished, he takes the list of transactions that occurred during this time and sends them to the BankNetwork. He can also be notified that transaction coming from other bank have arrived and he notify the AccountManager (in future implementation, will store those information so client can seed an historique of transaction)

AccountManagerService:

- Manage client's account. Create a new account for clients. Depending on the invoked action, can modify a new client balance depending on the transaction regarding him.
- Scaling point : this service is invoked only if needed, and for scale needs, can have just before a bus that content all the action that he have to archive (like add or retire money), those action can be done in the background as they don't have an immediate impact

BankNetworkService:

- Service that is linked to the outside, when a transaction comes from the systeme, it changes it into a reglementation file understandable by the banking network.
  When a file comes from the outside, I change it into a transactional JSON understandable by the TransactionManagerService.

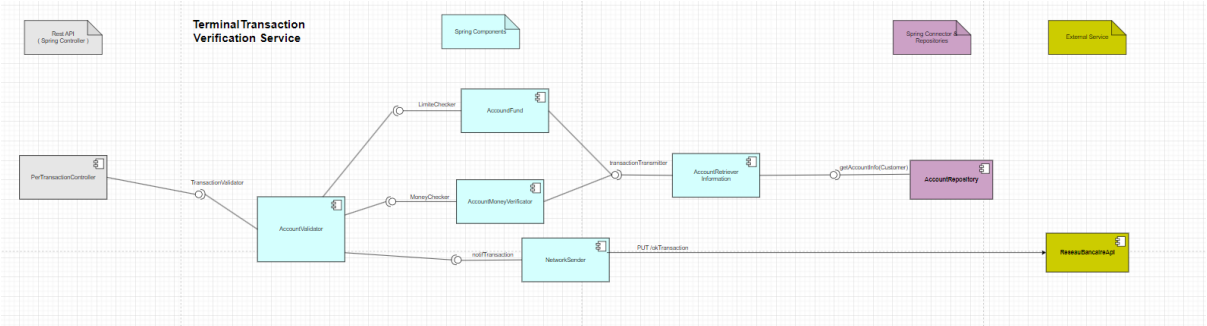
RegistrationService:

- Not importante service in this scope, here to fill DB. This service is here to manage the step in a new client registration. Treat a JSON with all client info and create his account and generate him a new neo-card.

TerminalTransactionVerificationService :

- Its role is to check if a wanted transaction can be done. It then sends a REST Call to the transactionManagerService so it continues the process.

Component Diagram :



Components :

ComponentAccountValidator **:**

- receives a transaction and validates it or not

AccountFund **:**

- manages the verification linked to the account ceiling
  - checks if the account has reached the payment limit

AccountMoneyVerificator **:**

- handles verification related to account money, check if the account has enough money.If false: checks whether the account can or may still go into the red

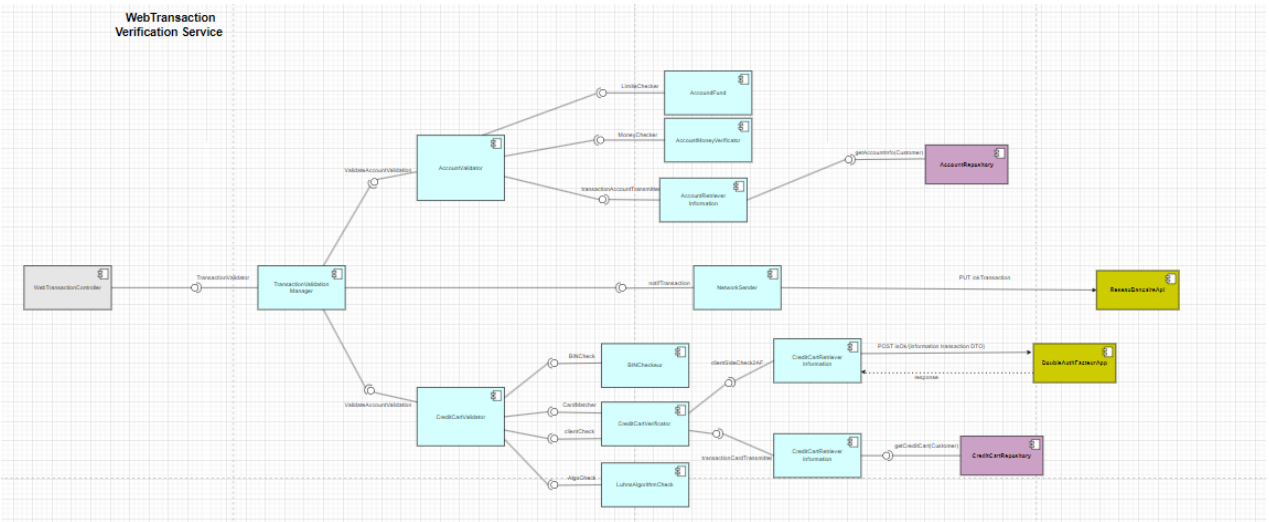
NetworkSender **:**

- if transaction is marked as successful, notifies the card network

TerminalTransactionVerificationService :

- its role is to check if a wanted transaction can be done. it then send a REST call to the transactionManagerService so it continue the process.

Component Diagram :



Components :

TransactionValidationManager:

- checks that all verifications have been carried out correctly, and is responsible for the order in which transactions are carried out.

CreditCartValidator:

- performs verification of credit card information

BINChecker:

- verifies regex format of card information submitted for verification

LuhnsAlgorithmCheck:

- checks the 16-digit code provided to verify the validity of the number

CreditCartVerificator:

- verifies that the information given by the customer corresponds to the object in BD

Persona :

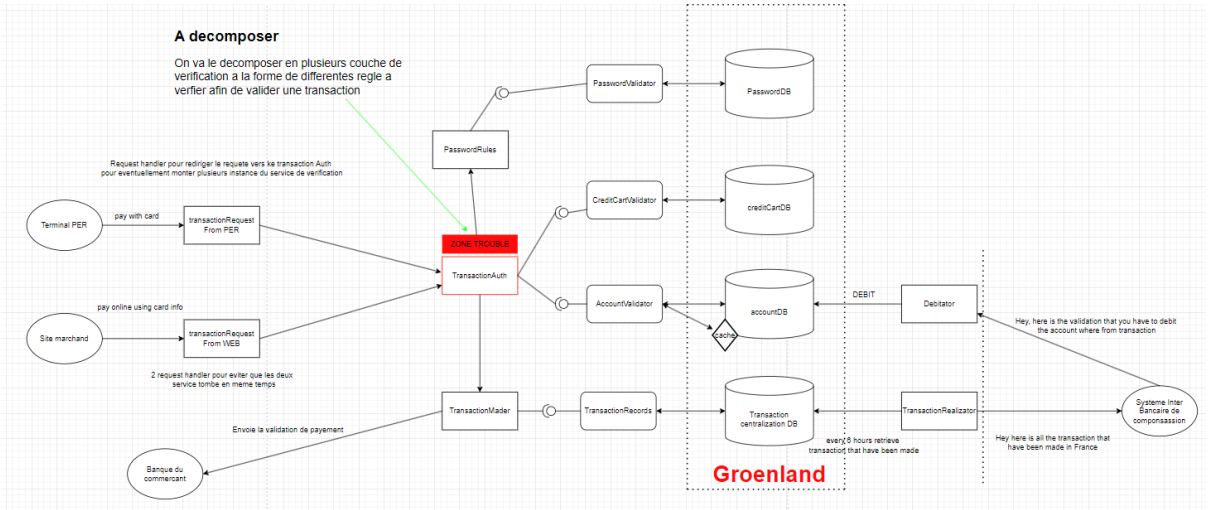
- Fabrice : technician at the new neo-bank "IgorBanque", his job is to keep BNP IgorBanque's servers up and running.
- Philippe : A merchant who recently opened a cookie-shop in Nice, he offers customers a means of paying by card, or on the internet via click and collect.

User Stories :

- As Fabrice, I need to know from the cardNetwork which kind of transaction is coming
- As Phillipe, I need to know from my TPE terminal if the client transaction have succeeded
- As Phillipe, I need to be notified if a transaction have succeeded if the purchase of the click and collect have been made on my website
- As Fabrice, I need to know in the bank systeme if any kind of transaction have succeeded

Week 39 :

Current Structure of the architecture :



In our current architecture, we only support transactions for the moment, but we'd like to concentrate on them so that we can consider applications of version 7 of the bank, i.e. large-scale scaling.

In our schema, we have the following external services: the PER terminal, a merchant's site, his bank and the interbank clearing system, which manages money exchanges between banks.

We then have a verification layer which retransmits transaction requests to the central transaction verification system. We've imagined it this way in order to potentially isolate the transaction verification service, which could be the part most in demand during a scale-up, so we can raise several instances of this service.

In our verification service, we'd have a set of components that verify each part of the request, which must comply with a set of established rules (ceiling limit, sufficient balance, valid credit card, valid transmitted information, correct passwords, etc.). Most of the information will be stored in Greenland databases, with which we will interact to retrieve this information. A caching solution could also be deployed to speed up retrieval in the event of scaling.

A broadcast component will then be called if the transaction can go through. The money is then frozen on the account, and the new transaction is added to all those already processed. The customer's bank, if external, is notified of the pending transfer.

Every 6 hours (according to the Internet), the SIT will retrieve all the transactions carried out and, once all the transfers have been made, will tell the bank to withdraw the frozen money from the accounts. Given that the action of debiting the money happens every 6 hours in our conception of the thing, the debiting component may not be asynchronous in the immediate future, but in a scaling perspective, it probably will be.

In the end, we'll have 4 layers: transaction reception, verification, communication with external services and databases.

Ideally, what would be most feasible would be to group external management into a single service, which would also include functionality not required for transactions, and to output transaction verification processing in a single service.

Sequence Diagram Flow :