Document d'architecture

NewBank - V8 Cashback

Team A

(comme argent)

Antoine BUQUET

-

Benoit GAUDET

-

Ayoub IMAMI

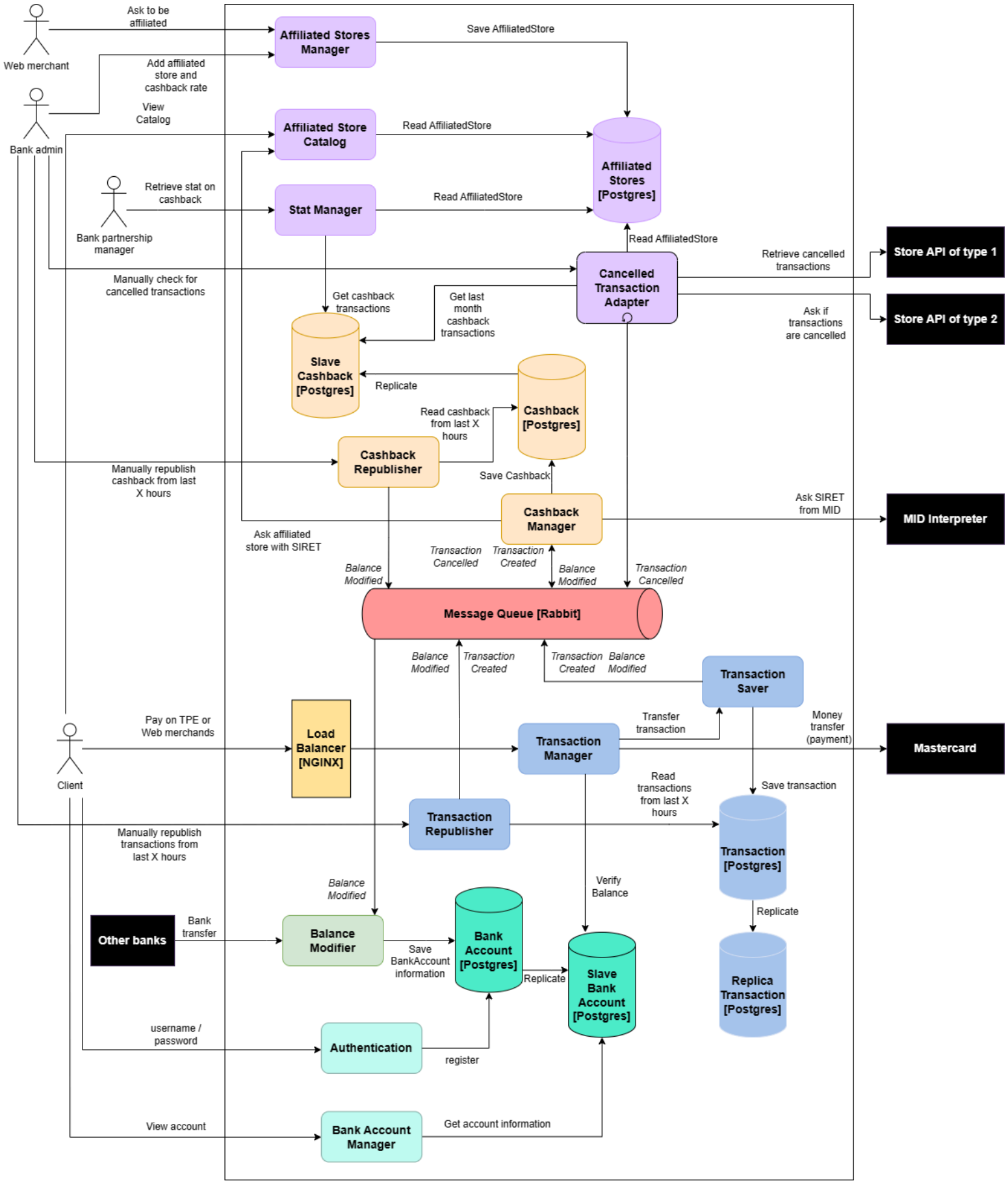
-

Mourad KARRAKCHOU

Table des matières :

I. État de l'architecture actuel

1. Diagramme d'architecture *(disponible ici en HD)*



Architecture composée de cinq services :

- Affiliated Store Service
- Cashback Service
- Transaction Service
- Balance Service
- Account Service

2. Justifications

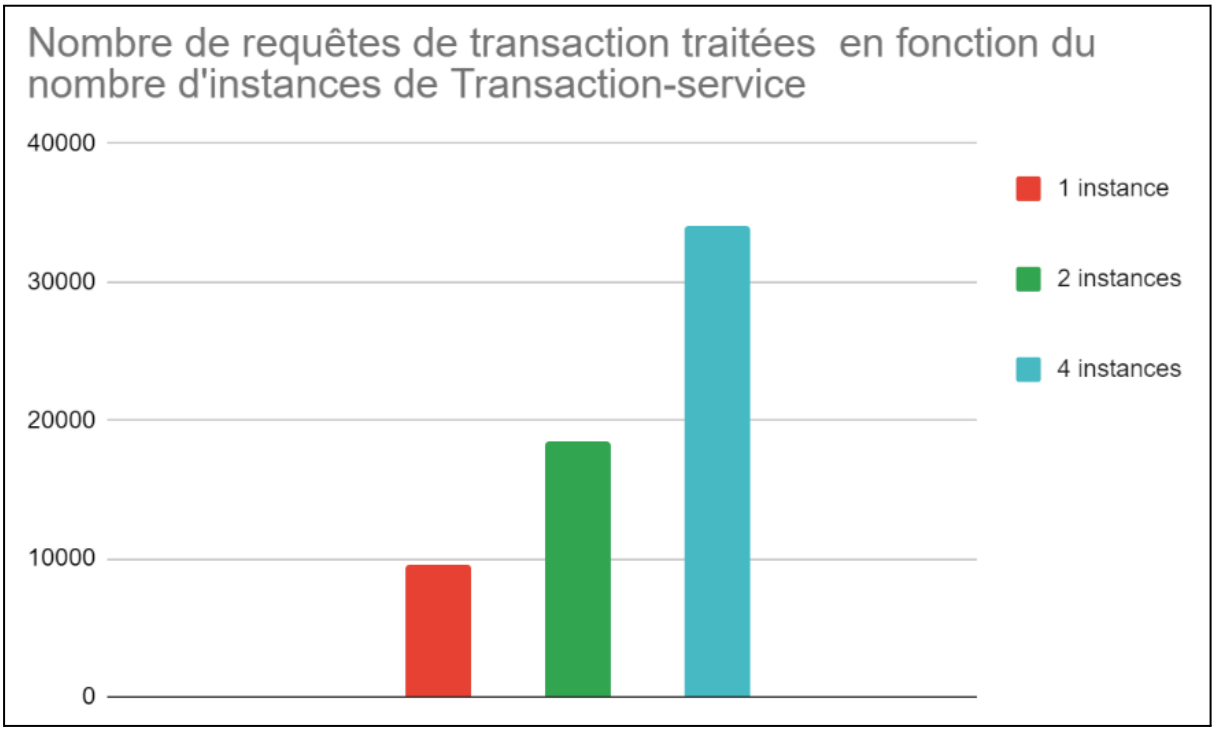
L'architecture de notre banque est basée sur une architecture en micro-services. Le premier service, représenté en bleu dans le diagramme, a pour objectif de gérer la partie des transactions. Le deuxième service, représenté en orange dans le diagramme, va quant à lui gérer la partie sur le cashback. Le service violet s'occupe des magasins partenaire ainsi que la connexion avec leurs services. En bleu ciel est représenté le service des comptes pour prendre en charge les comptes utilisateurs et enfin en vert, le service relatif aux comptes bancaire et aux différentes opérations possibles sur ces derniers.

Ce nouveau découpage en services plus précis comparé à la précédente version *(cf. architecture au 13/11/23)* permet à la banque d'être plus évolutive en séparant mieux les responsabilités, plus résiliente et également mieux mise à l'échelle afin de supporter une charge haute et constante. Le service bank a été séparé en un service gérant les transactions et un autre gérant les comptes bancaires des utilisateurs. Puis le service cashback a été séparé en un service qui s'occupe de la gestion des cashbacks et un autre qui s'occupe de la gestion des magasins partenaires. Ce changement était nécessaire, car nous avions remarqué que notre application ne supportait pas une charge élevée en raison d'un couplage important entre les différents services. Ce couplage important et la communication synchrone via l'API REST entre les deux services que nous avions (bank et cashback) entraînait d'importantes latences. En plus de cette latence, ces deux services devaient supporter des charges extrêmement élevées, causant ainsi des points de défaillance importants. La séparation des services de banque et de cashback permet de mieux répartir les responsabilités et mieux répartir la charge de l'application en facilitant la mise à l'échelle horizontale. Cependant, cela se fait au prix d'une plus grande complexité, avec plus de services à gérer et une communication plus complexe entre ces services.

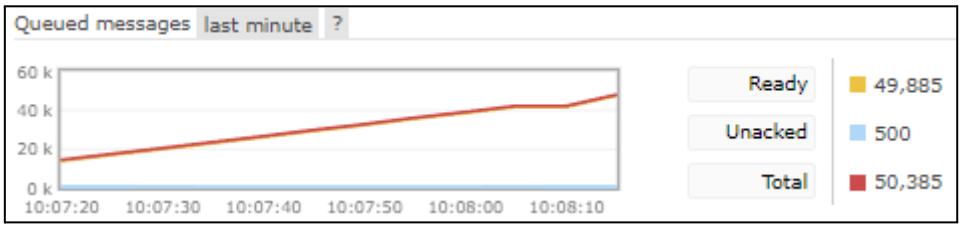
Nous avons choisi d'implémenter la spécialité de notre banque (le cashback) de cette manière en séparant le côté "transaction" de la partie cashback en elle-même, car nous estimons qu'il s'agit simplement d'une extension supplémentaire venant se greffer au fonctionnement d'une banque normale. Dans le cas d'une erreur dans le service de cashback ou des magasins partenaires, un utilisateur effectuant une transaction sans cashback ne doit pas être impacté.

Pour répondre à notre nouveau besoin de supporter une charge haute et constante, nous avons choisi de changer la façon dont les services communiquait entre eux en décidant d'ajouter une queue afin de permettre une communication asynchrone entre les différents services. Pour ce faire, nous avons choisi d'utiliser RabbitMQ qui est une queue de messages open source. Notre application dispose d'une base de données stockant déjà les transactions effectuées, nous n'avions donc pas besoin d'utiliser un event store comme Apache Kafka par exemple. L'envoi de messages asynchrones permet de réduire le couplage entre les différents services. De plus, si un service tombe en panne où est surchargé, les messages seront toujours envoyés et pourront être traités lorsque le service sera à nouveau disponible ou en rajoutant de nouvelles instances qui vont lire sur la queue, profitant ainsi de l'aspect stateless de nos services. Il y a donc une possibilité de faire scale notre application horizontalement.

Avec toutes les requêtes constantes que notre application devait maintenant être capable de supporter, nous avons pris la décision d'ajouter un load balancer devant le point d'entrée principale de notre banque, à savoir le service de transaction. Étant donné que notre service de transaction est stateless, nous pouvons désormais le dupliquer afin d'obtenir plusieurs instances et donc prendre en charge plus de requêtes simultanément. Le load balancer à ainsi pour rôle de rediriger le trafic entre les différentes instances avec une stratégie de round robin. Le choix de la stratégie du round robin pour le load balancer est lié aux opérations que le service de transaction effectue. Ici, étant donné que le service fait uniquement des transactions, chaque opération dure environ la même durée et donc le round robin était adapté.



Le principal intérêt d'avoir des services sans états pour notre architecture est de pouvoir créer de nouvelles instances de ces services pour pouvoir mettre à l'échelle horizontalement certaines parties de l'application. On remarque sur ce graphique que le nombre de requêtes de transaction traitées augmente en fonction du nombre d'instances de Transaction-service. Cette capacité de pouvoir faire une mise à l'échelle horizontale permet ici de supporter une quantité de transaction élevée.



Lorsqu'une grande quantité de transactions est effectuée, les services consommant les messages produits dans la file de messages ne les consomment pas forcément assez rapidement. Cela peut entraîner une augmentation du nombre de messages dans la file d'attente. Pour résoudre ce problème et tirer parti de l'aspect sans état de nos services, il est simplement possible de créer plusieurs instances de ces services qui iront lire les messages à traiter dans la file de messages.

Pour pouvoir répondre à nos extensions, nous avons dû renforcer la résilience. En effet, il était impératif d'avoir une application constamment disponible. Cette résilience a été rendue possible tout d'abord grâce à la queue et au load balancer, comme présenté précédemment. La problématique résidait dans le fait que la file d'attente devenait elle-même un point unique de défaillance (SPOF) et que, si elle venait à s'arrêter, tous les messages qu'elle contenait seraient perdus. Nous avons donc mis en place un système de rétablissement avec une route vers le TransactionRepublisher, permettant de renvoyer une partie des transactions enregistrées dans la file d'attente. Ainsi, ces transactions seront soit traitées, soit considérées comme ayant déjà été traitées par le service cashback et balance. Ceci permet, en cas de redémarrage de la file d'attente, de pouvoir continuer à gérer les transactions en les enregistrant simplement dans la base de données et de rétablir les messages perdus qui se trouvaient dans la file d'attente au moment du redémarrage. Les messages envoyés en double seront alors facilement considérés comme ayant déjà été traités.

Enfin, il y a quatre bases de données dans l'application. Une pour permettre le stockage des données sur les magasins affiliés, avec leur offre de cashback. Une autre pour le stockage des comptes bancaires. Une troisième pour sauvegarder toutes les transactions effectuées et enfin une dernière pour les cashback donné par la banque. Cette séparation permet d'isoler les données métier relatives à leurs services et évite par ailleurs une centralisation du stockage qui pourrait être problématique si l'unique base de données venait à cesser de fonctionner. Dans le cas présent, si la base de données du service de cashback n'était plus accessible, alors l'utilisateur pourrait tout de même effectuer une transaction. Nous avons également choisi de répliquer les bases de données les plus sensibles et utilisées afin d'assurer une meilleure résilience et scalabilité. En effet, comme il y avait beaucoup de lectures et d'écritures sur les bases de données du service de balance et de cashback, la gestion des requêtes était lente. Ajouter ce modèle de master-slave nous a permis d'avoir accès aux données des bases pendant que de nouvelles données sont écrites. Enfin, en cas de panne d'une base de données master, celle-ci pourrait utiliser la base de données slave afin de récupérer les données perdues. Dans le but de gagner encore plus en efficacité, les services ayant besoin d'effectuer des lectures dans les bases de données des autres services communiquent directement avec celles-ci sans passer par une API ou un composant.

3. Descriptions des composants du diagramme

1) Affiliated Store Service

Dans l'ancienne architecture, nous avions le service de cashback qui gérait aussi les affiliated store. Nous avons décidé de créer un service propre aux affiliated store pour plusieurs raisons. Tout d'abord, ce service n'est pas soumis à la même charge et donc n'a pas besoin des mêmes ressources que la partie cashback. De plus, pour répondre aux problématiques amenées avec les extensions et pouvoir permettre aux nouveaux petits marchants web de s'inscrire, il était pertinent de créer un service dédié.

- **Affiliated Stores Manager** : ce composant est responsable de la création des comptes magasins affiliée à notre banque avec leur taux de cashback offerts respectifs.

- **Affiliated Stores Catalog** : ce composant est responsable de fournir l'accès aux informations concernant tous les magasins affiliés comme leur SIRET ou bien leur taux de cashback.

- **Stat Manager** : ce composant est responsable de fournir des statistiques sur les cashback au responsable des partenariats de NewBank à des fins de monitorage. Un exemple de statistique est disponible en annexe *(cf. Figure 1).*

- **Cancelled Transactions Adapter** : ce composant a pour but de s'adapter aux différentes API en fonction du type du magasin afin de récupérer l'information des transactions cashback ayant été annulée ou remboursée. Dans le cas d'un magasin de type 1, il récupère auprès du magasin la liste des transactions annulé le dernier mois et les compare avec toutes les transactions ayant généré du cashback au cours des 30 derniers jours. Dans le cas d'une API d'un magasin de type 2, il transmet chacune des transactions et obtient une réponse booléenne indiquant si la transaction a été annulée ou remboursée. Cette vérification est déclenchée de manière régulière chaque mois, mais peut aussi être déclenchée manuellement par un administrateur de la banque.

2) Cashback Service

Le service de cashback est désormais beaucoup plus spécialisé et ne gère plus que les cashback comparé à l'ancienne architecture où il avait aussi la responsabilité des magasins partenaire comme évoqué précédemment. Il peut maintenant est mis à l'échelle plus facilement, notamment car son couplage avec le service de transaction (anciennement "bank") a été considérablement réduit avec l'ajout de la message queue.

- **Cashback Manager** : ce composant doit, pour chaque message Transaction Created consommé, communiquer au service externe MID Interpreter pour récupérer un numéro de SIRET exploitable alors par le service des magasins partenaire qui va lui fournir le taux de cashback associé au magasin disposant de ce SIRET. Si le magasin n'est pas un magasin partenaire, alors le taux retourné sera de 0 et donc le travail de ce composant s'arrête ici. Dans le cas où il s'agit d'un magasin partenaire, Cashback Manager doit alors calculer le montant à ajouter au solde de l'utilisateur en fonction du taux de cashback et de la valeur de la transaction. Une fois fait, il enregistre cette opération de cashback et émet un message Balance Modified sur la queue RabbitMQ. Lorsqu'il consomme un message de type Transaction Cancelled, il est responsable de retrouver l'opération de cashback associé à cette transaction afin de publier un nouveau message Balance Modified visant à retirer le cashback qui avait été octroyé à l'utilisateur.

- **Cashback Republisher** : Composant permettant de récupérer tous les cashback enregistrés depuis un certain nombre d'heures et de republier les messages BalanceModified associés dans la queue RabbitMQ. Cela est utile dans le cas où la queue RabbitMQ est tombé en panne et que des cashback n'ont pas été pris en compte par le composant Balance Modifier. Un ID de cashback est fourni afin que le composant Balance Modifier vérifie que l'opération n'avait pas en réalité déjà été prise en compte.

Les trois services suivants étaient à la base un unique service "bank". Ils ont été dissociés dans la nouvelle architecture dans le but de mieux responsabiliser les services et de pouvoir les mettre à l'échelle de manière indépendante. C'est principalement le cas pour le service de transaction qui doit supporter une charge importante et constante depuis nos nouvelles extensions, comparé au service de compte par exemple beaucoup moins utilisé.

3) Transaction Service

- **Transaction Manager** : ce composant est responsable de déclencher le paiement une fois les vérifications sur le solde ou la carte bancaire utilisée par exemple correctement effectuées. Après cette étape, il est responsable de communiquer avec le service externe de Mastercard afin d'effectuer la transaction. Une fois qu'il reçoit une réponse valide de la part de celui-ci, il transmet la transaction au composant Transaction Saver.

- **Transaction Saver** : ce composant est responsable de l'enregistrement de la transaction effectuée dans la base de donnée du service de transaction puis de publié un message dans la queue RabbitMQ indiquant la création d'une nouvelle transaction en fournissant des informations telles que le MID.

- **Transaction Republisher :** Composant permettant de récupérer toutes les transactions enregistrées depuis un certain nombre d'heures et de republier les messages BalanceModified et Transaction Created associés dans la queue RabbitMQ. Cela est utile dans le cas où la queue RabbitMQ est tombé en panne et que des transactions n'ont pas été prises en compte par le composant Balance Modifier et Cashback Manager. Un ID de transaction est fourni afin que le composant Balance Modifier vérifie que l'opération n'avait pas en réalité déjà été prise en compte.

4) Balance Service

- **Balance Modifier** : ce composant permet aux utilisateurs de recharger leur compte bancaire pour effectuer des achats avec leur carte NewBank. Pour ce faire, il est accédé par les banques externes pour transférer les fonds. C'est également lui qui consomme les messages BalanceModified et qui effectue la modification sur le solde de l'utilisateur si celle-ci n'a pas déjà été faite.

5) Account Service

- **Authentication** : ce composant est responsable de l'authentification des utilisateurs.

- **Bank Account Manager** : ce composant est responsable de la gestion des comptes utilisateurs et bancaires. Il permet de créer son compte et de visualiser ses informations telles que son solde et ses informations personnelles.

4. Descriptions des services externes

Nous avons différents services externes :

- **external-bank-mock-service** : il s'agit de la potentielle banque secondaire de l'utilisateur, il s'en sert pour faire des virements entre cette dernière et NewBank.

Nous avons pensé au fait qu'un article pouvait être rendu et le client remboursé. Il faut prendre cela en compte sur le cashback et retirer son montant du compte, sinon des clients pourrait abuser d'achat et de retour d'articles en conservant le cashback.

Pour cela, chaque mois, nous demandons aux magasins partenaire la liste des retours d'articles.

- **external-carrefour-mock-service** : il s'agit d'un simple mock permettant d'obtenir la liste des retours d'article.
- **external-decathlon-mock-service** : de même, mais la logique est différente, on donne une liste d'article et le service nous renvoie les éléments de cette liste qui ont été retournés au magasin.

Nous avons 2 exemples de logiques différentes, pour illustrer le fait que les magasins ne communique pas forcément de la même façon.

De plus, nous posons l'hypothèse que le client se fait rembourser directement en liquide ou en avantages tels que bon d'achats. Donc la banque ne rend pas l'argent dépensé dans la transaction, elle retire simplement le cashback octroyé.

Afin d'implémenter un modèle proche de la réalité, nous avons également implémenté ces services externes. D'après nos recherches, lorsqu'un paiement est fait, le TPE du commerçant envoi son identifiant MID à la banque. Après traitement de la transaction, la banque retire le montant du compte du client, qu'il redirige vers un autre service externe afin que le commerçant soit payé.

- **external-mastercard-mock-service** : il permet à la banque de rediriger l'argent du client vers le commerçant. Il renvoie à notre application le numéro de la transaction mastercard que nous pourrons utiliser pour reconnaître la transaction dans le cas d'un remboursement client.

Dans le cas d'une transaction avec cashback, il faut récupérer l'identifiant MID fourni par le TPE ou bien le site internet et l'envoyer dans un service externe pour obtenir le numéro de SIRET du commerçant car c'est avec le SIRET que l'application vérifie la présence du commerçant dans les magasins partenaire.

- **external-mid-interpreter-mock-service** : il permet d'obtenir le numéro de SIRET d'un commerçant à partir de son identifiant MID.

5. Descriptions des messages RabbitMQ

La communication entre les services de transaction, cashback et balance se fait à travers notre queue de message RabbitMQ. Les trois messages qui sont publiés et consommés, chacun sur sa propre queue, entre ces services sont les suivants :

- **Transaction Created** : Message publié par le composant Transaction Saver à destination du composant Cashback Manager dans le service cashback pour l'informer qu'une nouvelle transaction a été réalisée. Celui-ci utilise alors le MID contenu dans l'objet Transaction passé dans le message et le transmets au service externe MID Interpreter pour obtenir le SIRET associé au magasin et ainsi vérifier s'il s'agit d'un magasin partenaire ou non. Ce message est aussi publié par le composant Transaction Republisher dans le cas où des transactions ont été effectuées et que la queue était en panne à ce moment-là.

- **Transaction Cancelled** : Ce message est publié uniquement par le composant Cancelled Transaction Adapter du service des magasins partenaires. Il permet d'indiquer au service de cashback qui le consomme qu'une précédente transaction ayant généré du cashback a été annulé et qu'il faut ainsi récupérer le cashback.

- **Balance Modified** : Ce message est publié par le composant Transaction Saver après une transaction afin de débiter le solde de l'utilisateur. Le seul composant qui consomme ce message est Balance Modifier dans le service de balance. Le composant Cashback Manager publie également ce message lorsqu'il faut mettre à jour le solde de l'utilisateur en y ajoutant le cashback gagné après une transaction auprès d'un magasin partenaire. Mais il émet aussi ce même message après l'annulation du cashback suite au message *TransactionCancelled* pour retirer le cashback du solde de l'utilisateur. Ce message est également publié par les services Transaction Republisher et Cashback Republisher après que la queue soit tombé en panne et qu'il est nécessaire de renvoyer les messages de modification de balance suite à une transaction ou un cashback et qui n'ont pas été traités.

6. Fonctionnalités couvertes par notre architecture

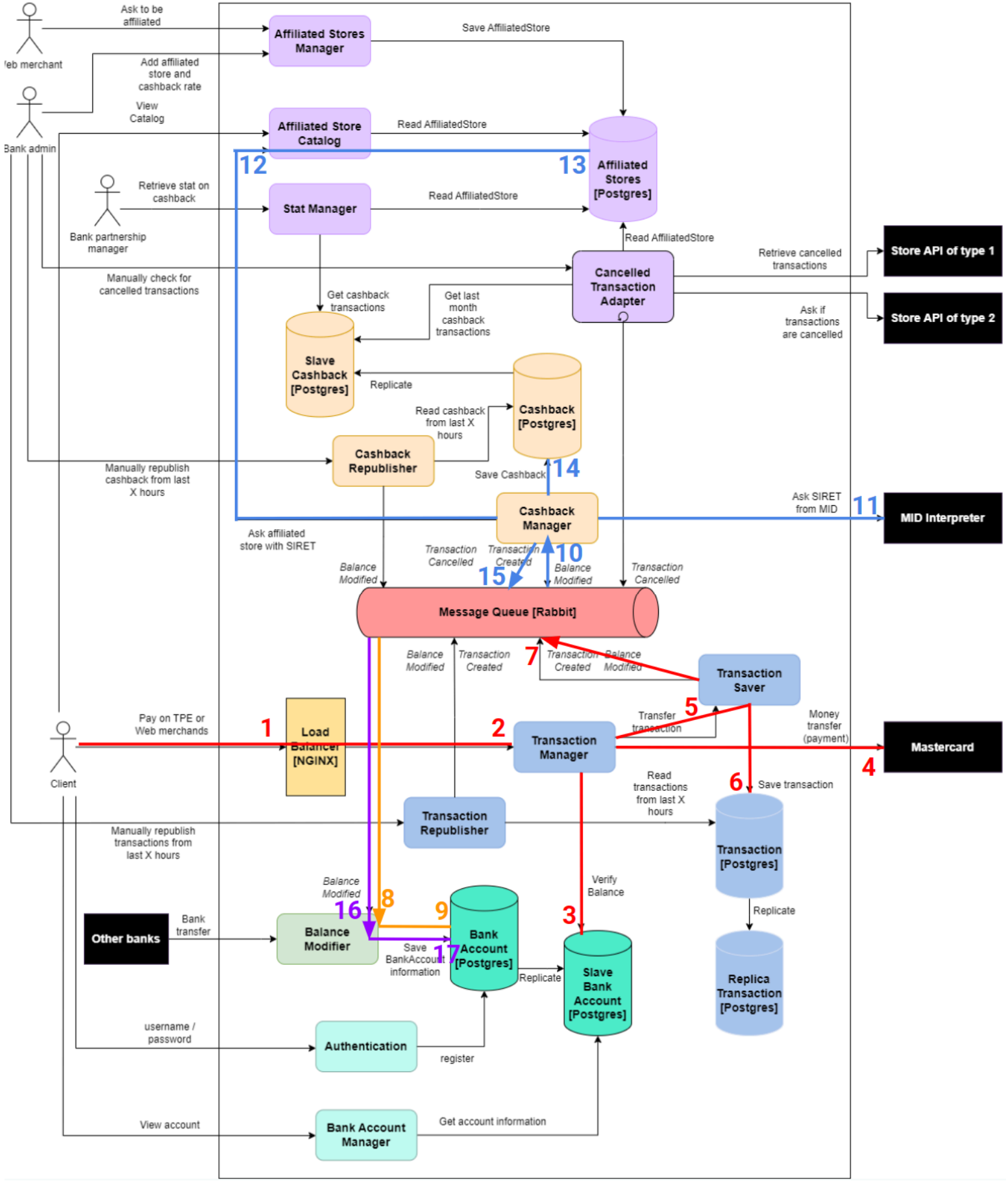
Notre architecture nous permet d'effectuer les différentes fonctionnalités suivantes :

- Pour l'utilisateur :
    - Créer un compte bancaire
    - Ajouter des fonds depuis une banque externe
    - Effectuer un paiement normal
    - Effectuer un paiement avec cashback
    - Visualiser des informations concernant son compte (solde, historique des transactions, historique du cashback…)

- Pour la banque :
    - Enregistrer un magasin partenaire
    - Générer des statistiques sur les cashback pour le manager des partenariats
    - Retirer le cashback des utilisateurs ayant annulé ou remboursé une transaction que leur en avait octroyé
    - Rejouer des opérations de transactions et de cashback en cas de panne

7. Scénario principal

Le scénario principal de notre application est celui d'un paiement auprès d'un magasin partenaire, ce qui aura pour effet de générer un cashback offert à l'utilisateur. Voici le chemin parcouru dans notre architecture lors de ce scénario :



Explication pas à pas

Remarque : en ce qui concerne les données qui transitent et les données de retour, nous avons décidé de décrire ci-dessous seulement les données les plus pertinentes, sans forcément les détailler, afin de conserver une bonne lisibilité du scénario. Par exemple, “CB” traduit {cardNumber, expirationDate, cvv}.

1 : Le client fait un paiement sur un TPE, *données qui transitent : CB, montant, MID*
2 : Le Load Balancer redirige la requête vers une des instances en Round Robin
3 : On vérifie la carte et le solde de l’utilisateur, *données qui transitent : id (carte → compte → id) et montant*
4 : On redirige l’argent en faisant appel au service externe Mastercard, *données qui transitent : Objet(MID + montant), retour de Mastercard : transaction id*
5 : *On crée un objet Transaction*
6 : On l’enregistre dans la base de données, *retour : le client reçoit une confirmation*
7 : On envoie 2 messages dans la Queue :

- les ids du compte et de la transaction et le montant → BalanceModifier
- Objet Transaction → CashbackManager

8 : On reçoit le message de mise à jour du solde
9 : Et on met à jour le solde dans la base de données

10 : En parallèle, on reçoit le message avec l’objet Transaction
11 : On contacte le service externe MID Interpreter pour obtenir le SIRET du magasin à l’aide du MID, obtenu depuis le TPE, contenu dans l’objet Transaction
12 et 13 : On recherche si le numéro de SIRET est dans la base de données des magasins partenaires, *données qui transitent : numéro de SIRET, retour : taux de cashback*
14 : la valeur du cashback est calculée et on enregistre un Objet Cashback dans la base de données
15 : On envoie un message dans la Queue : les ids du compte et de la transaction et le montant du cashback → BalanceModifier

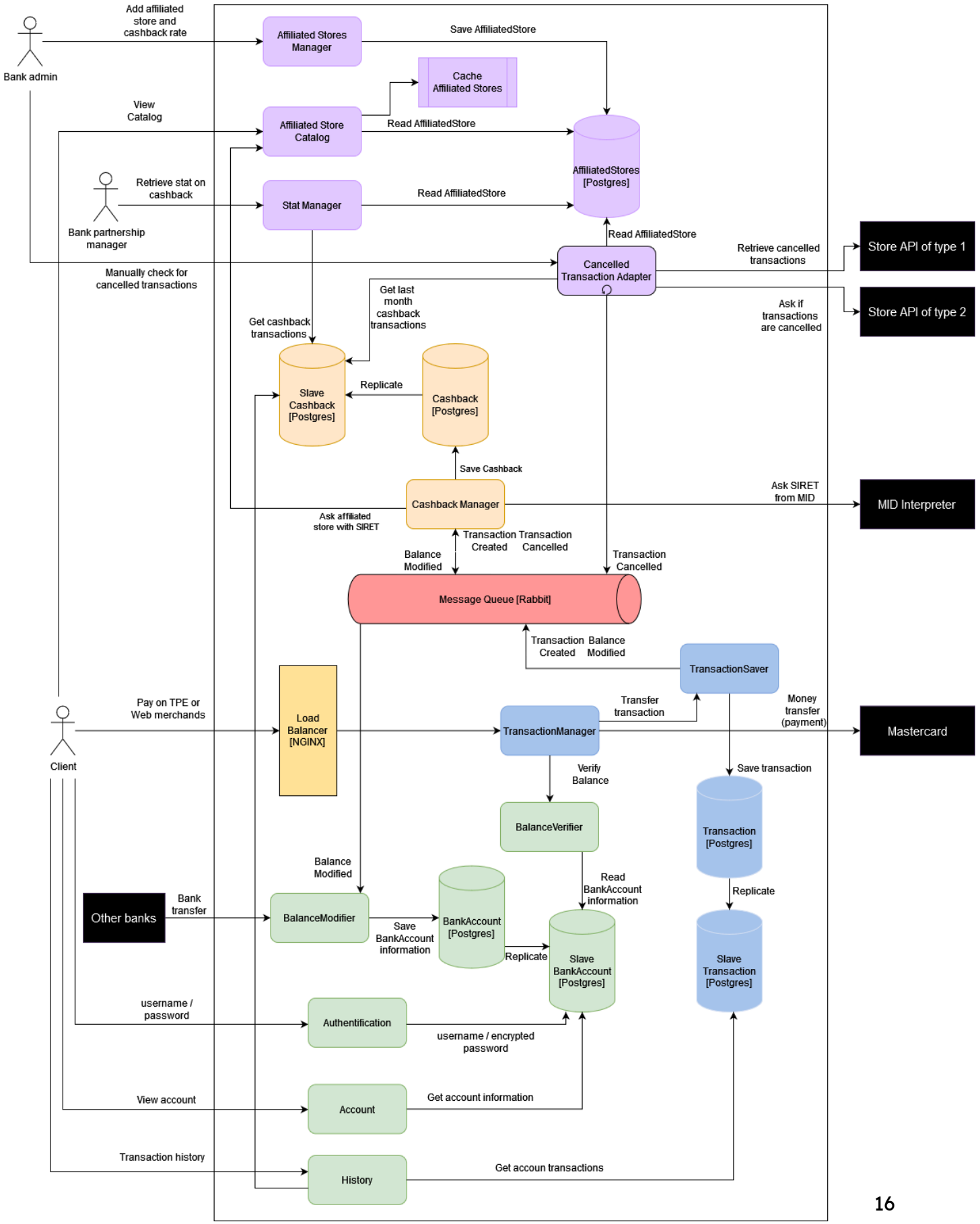
16 : On reçoit le message de mise à jour du solde
17 : Et on met à jour le solde dans la base de données

8. Faiblesses et perspectives futures

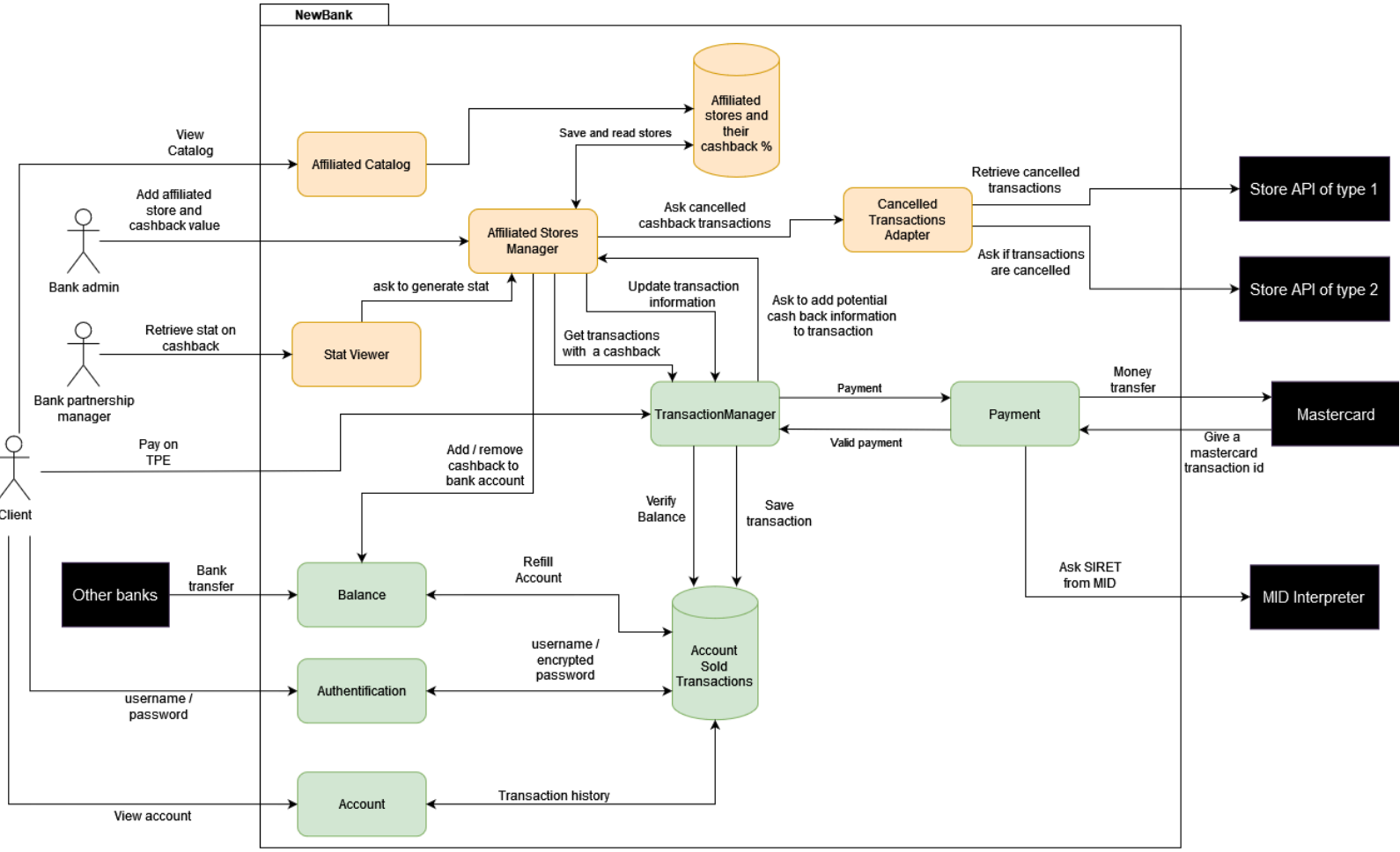
Nous pourrions envisager une migration de la base de donnée PostgreSQL du service de transaction en base de donnée NoSQL avec une approche Master/Master comme BigTable de Google ou bien Cassandra pour obtenir une meilleure résilience et utilisation dans le monde. Une autre faiblesse se trouve au niveau de notre système de rétablissement sur notre queue RabbitMQ. Une intervention humaine est pour le moment nécessaire pour renvoyer les messages des nouvelles transactions qui n'ont pas pu être traitées par la queue. Enfin, l'ajout d'un cache dans le service des magasins partenaires pourrait être pertinent. En effet, pour les magasins les plus importants et qui reçoivent le plus de transactions, on sait que leur taux de cashback ou leur présence dans le programme des magasins affilié ne va pas être souvent modifié, il n'est donc pas utile de communiquer à la base de donnée à chaque nouvelle transaction pour ce genre de magasins.

II. Historique de l'architecture

1. 25 novembre 2023

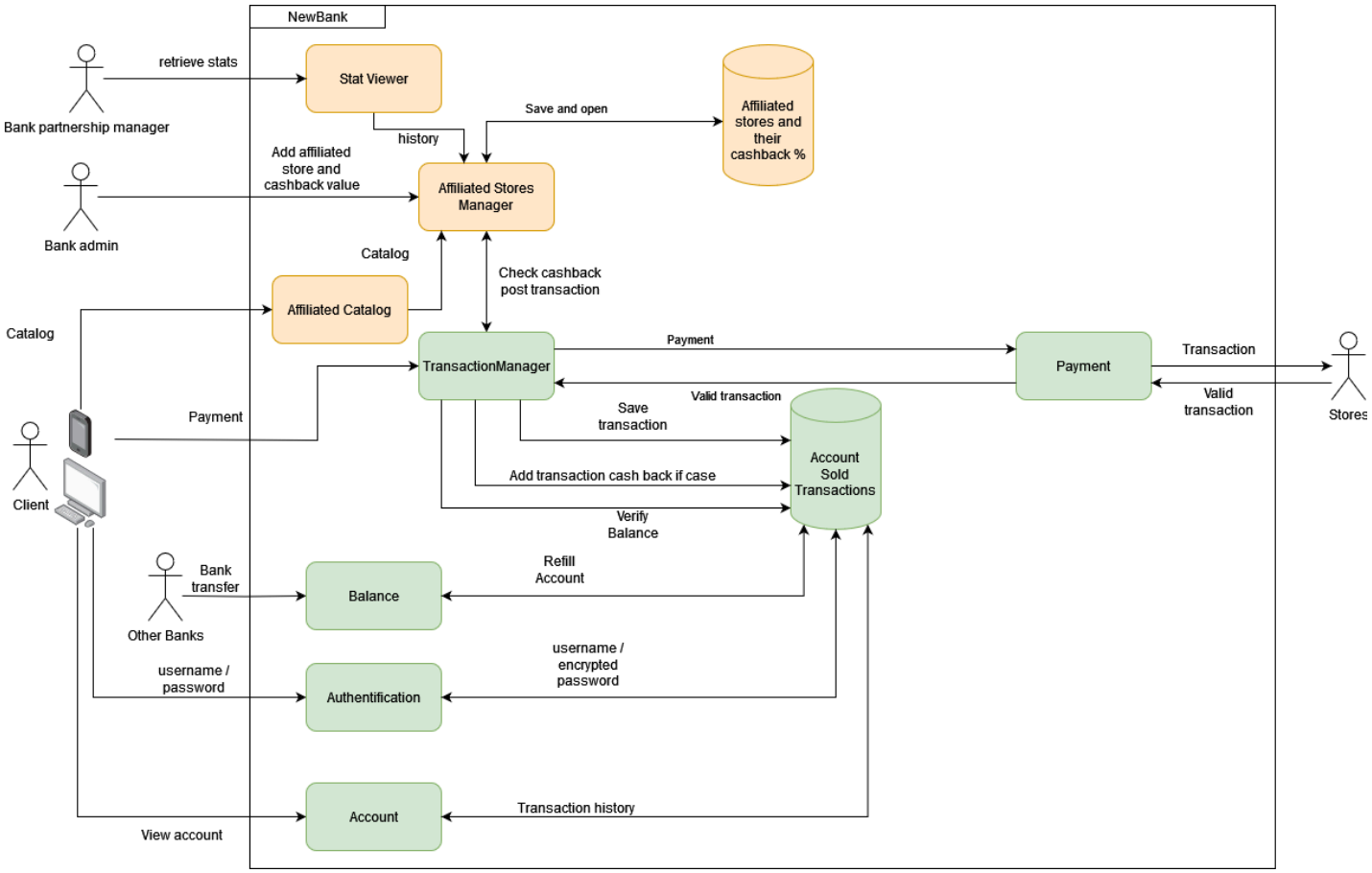


2. 13 novembre 2023



Voici notre architecture lorsqu'elle était composée seulement de deux services. Elle présentait plusieurs Single Point Of Failure, notamment le Transaction Manager et le Affiliated Stores Manager. Les responsabilités de chacun étaient trop importantes dû à un mauvais découpage. De plus, les base de données n'étaient pas répliquées non plus, impliquant alors une moins bonne résilience.

3. 20 octobre 2023

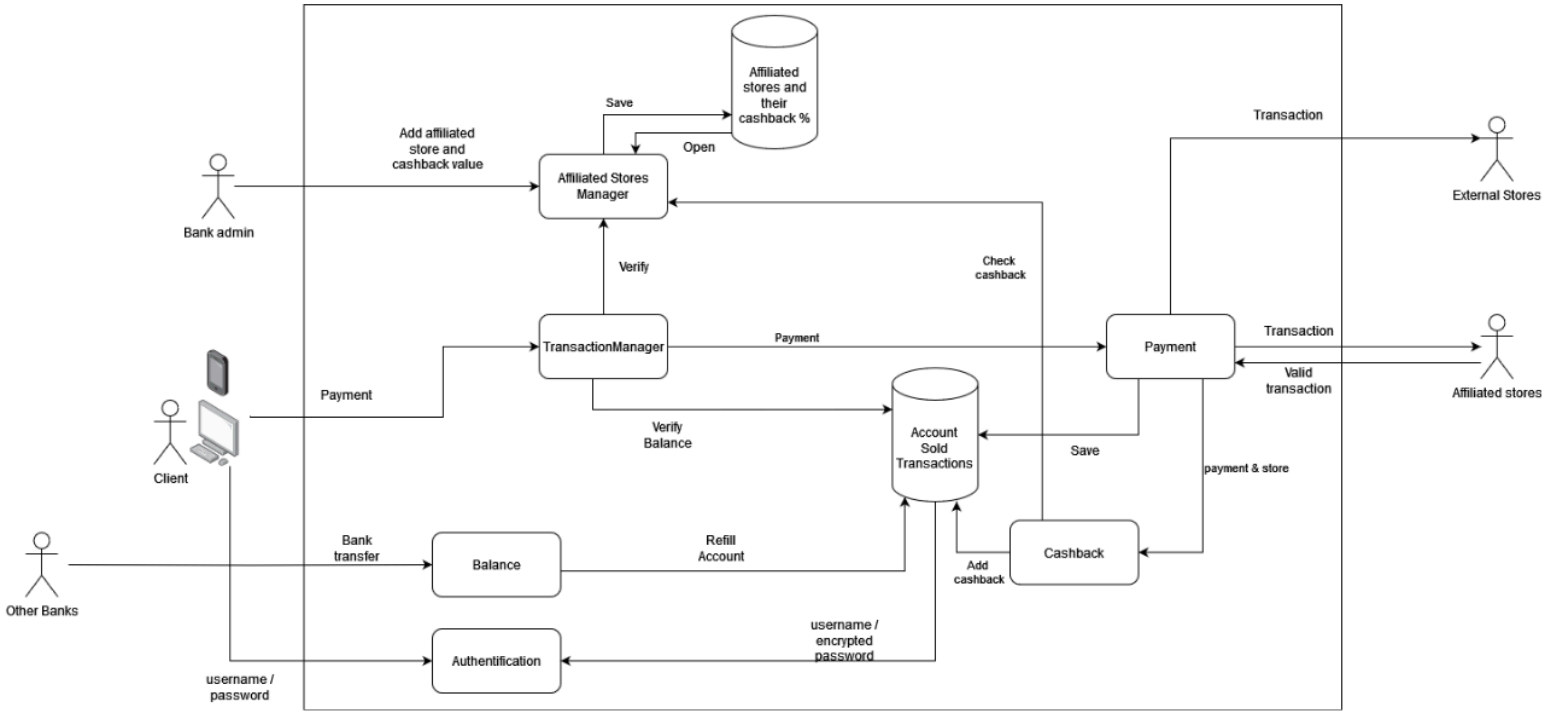


À cette date, notre application manquait cruellement de réalisme en ne faisant pratiquement aucun appel à des services externes. En effet, la transaction était directement transmise au magasin qui validait alors la transaction. Dans la réalité, le fonctionnement d'un paiement est bien plus complexe, en impliquant plusieurs acteurs externes pour effectuer le transfert des fonds.

L'application était donc fonctionnelle, mais certaines logiques n'étaient pas cohérentes. C'était le cas notamment avec le composant Transaction Manager qui avait la responsabilité d'ajouter le cashback dans le compte bancaire de l'utilisateur alors que théoriquement, il devrait s'agir d'une action effectuée depuis le service cashback.

Enfin, nous n'avions pas encore pensé à la fonctionnalité de vérification des transactions qui avaient du cashback mais qui ont été annulées ou remboursées par le magasin partenaire. Toute la partie avec les mocks simulant des API distante différentes afin de récupérer cette information n'était donc pas encore présente dans notre architecture.

4. 1 octobre 2023



Il s'agit du premier diagramme d'architecture élaboré dans le cadre de ce projet. Les briques les plus importantes de notre diagramme d'architecture actuel y sont déjà présentes telles que Affiliated Stores Manager, Transaction Manager ou bien Payment.

III. Annexes


```
{
  "generalStat": {
    "name": "Overall",
    "numberOfCashbackTransactions": 8,
    "amountSpent": 760,
    "cashBackReturned": 123
  },
  "statsByStore": [
    {
      "name": "Nike",
      "numberOfCashbackTransactions": 5,
      "amountSpent": 470,
      "cashBackReturned": 94
    },
    {
      "name": "Decathlon",
      "numberOfCashbackTransactions": 3,
      "amountSpent": 290,
      "cashBackReturned": 29
    }
  ]
}
```


Figure 1 : Statistique de cashback disponible pour le responsable des partenariats