NewBank : Merchant webApp

Equipe B

Yahiaoui Imène - Gazzeh Sourour - Ben aissa Nadim

Al Achkar Badr

Sommaire

I. ARCHITECTURE LOGICIELLE - EVOLUTION

Mise à jour de l'incrément

➔ Durant cette semaine nous avons :

- ◆ Clôturé notre adr-006 : Mise en place d'un TimeOut Pattern.
- ◆ Basculer l'export des métriques métier en format JSON compatible avec grafana pour faire un dashboard des métriques

Les modifications/ajouts dans ce rapport pour cet incrément durant cette phase d'évolution sont mises en rouge.

Introduction

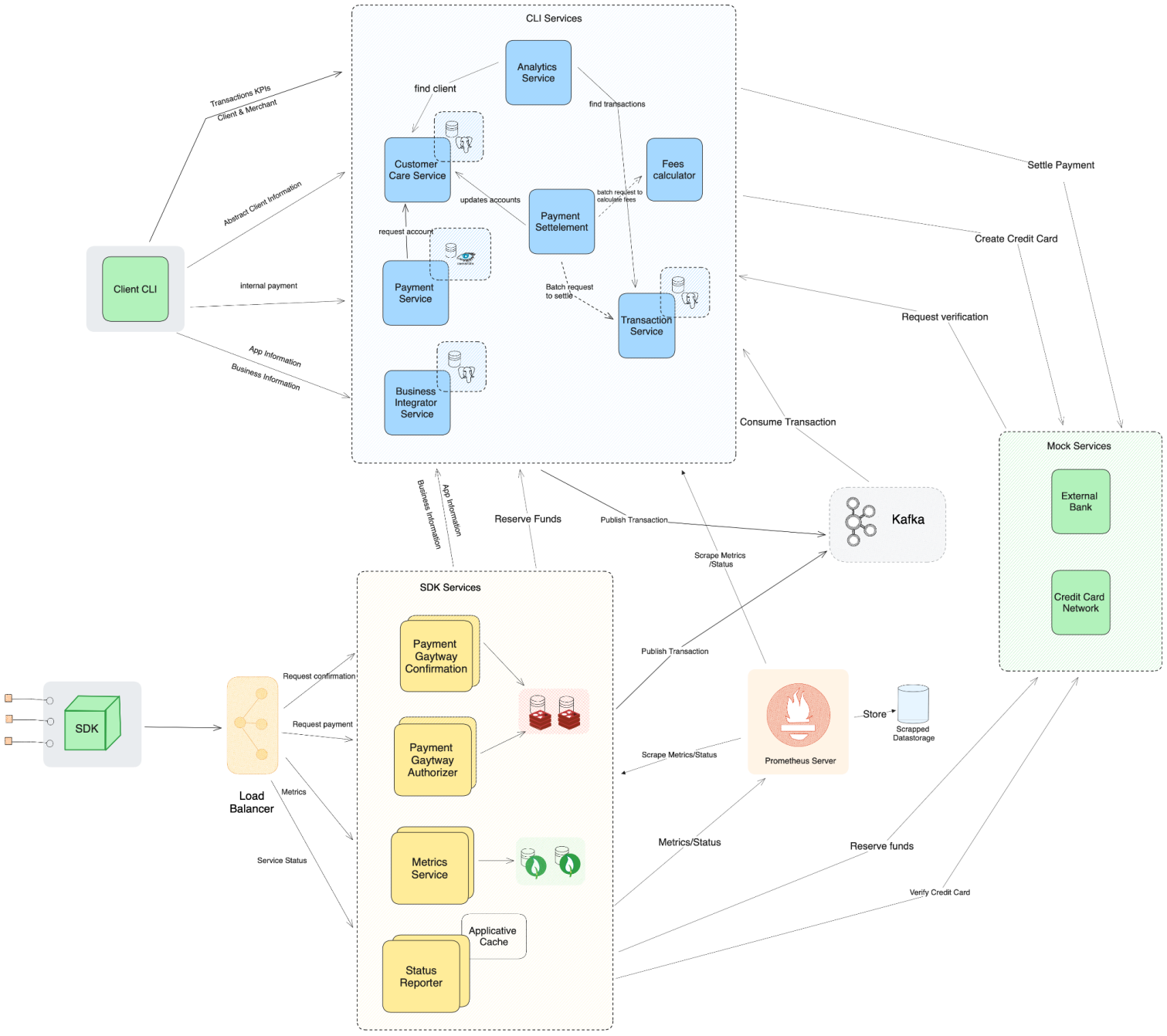
Dans cette seconde phase, notre système va subir des évolutions significatives. Notre objectif principal est de répondre aux nouveaux besoins des commerçants en introduisant des fonctionnalités de surveillance des statuts et de récupération des métriques métiers. Nous prévoyons également de renforcer la résilience du système et d'éliminer les points de défaillance, en particulier le Single Point of Failure (SPOF) représenté par le Payment Gateway.

Hypothèses

- Les métriques métier fournies aux commerçants incluent le nombre de transactions autorisées, confirmées et échouées.
- L'état des services retournés est soit UP, DOWN ou DEGRADED, pour avoir une vision globale et simple des statuts des services dans notre backend.

Architecture actuelle du système

[Version SVG]



Services et Composants ajoutés / modifiés dans cette phase d'évolution :

- **Payment Gateway Authorizer Service** :

Ce service a pour rôle principal d'assurer l'autorisation initiale des paiements des clients en appelant le Centre de Cartes Numériques (CCN).

- Composants et Interfaces Internes :

1. TransactionProcessor : Traite les transactions entrantes et délivre l'autorisation.
2. IRSA : Décrypte les informations de carte de crédit dans le contexte d'une demande de paiement, ainsi que la génération ou la récupération de clés publiques RSA pour une application spécifique.

- Interface externes :
    1. POST /api/gateway/authorize
        - Description : Autorise un paiement avec les détails fournis, en utilisant un jeton d'autorisation et des informations cryptées de carte.
    2. GET /api/gateway/applications/public-key
        - Description : Récupère la clé RSA publique d'une application en utilisant un jeton d'autorisation.

Payment Gateway Confirmation Service :
Ce service est responsable de confirmer la transaction, en communiquant avec le processeur de paiement dans le cas d'un paiement interne ou une banque externe si le paiement est externe.

- Composants et Interfaces Internes :
    1. TransactionConfirmation : Traite les transactions entrantes et confirme le paiement.
- Interface externes :
    1. POST /api/gateway-confirmation/{transactionId}
        - Description : Confirme un paiement avec l'identifiant de transaction fourni.

Metrics Service :
Ce service est chargé de fournir des métriques métier aux administrateurs, notamment en ce qui concerne les transactions, facilitant ainsi l'investigation des problèmes au sein de notre système.
Il permet aussi de faire une comparaison avec les données collectées par le SDK.

- Composants et Interfaces Internes :
    1. ITransactionFinder : Gère la recherche des transactions autorisées et confirmées.
- Interface externes :
    1. GET /api/metrics/transactions/authorized/merchantId
        - Description : Cherche les transactions autorisées d'un commerçant.
    2. GET /api/metrics/transactions/confirmed/metchantId
        - Description : Cherche les transactions confirmées d’un commerçant.

Status Reporter :

Pour communiquer les statuts de nos services aux clients, nous avons implémenté le service Status-Reporter. Ce service récupère les statuts d'un service Prometheus et les alertes levées d'un alerte Manager et renvoie les etats des services à l'SDK.
. Pour éviter de surcharger Prometheus, nous avons instauré un cache interne de 5 secondes en utilisant la stratégie de mise en cache "cache aside". Cela rend le service statefull.

- Interfaces Internes:
    1. IServiceStatusRetriever : Récupère les statuts des services.
- Interfaces externes:
    1. GET /api/status/healthcheck
        - Description : renvoie les statut des services du backend
    2. GET /api/status/availability
        - Description : vérifie si un service est disponible.

Newbank-Merchant SDK :

Le SDK Newbank-Merchant facilite l'intégration avec notre système de paiement, offrant aux développeurs une interface claire pour interagir avec les fonctionnalités de paiement, spécifiquement conçu pour être utilisé avec npm.

De plus, ce SDK a évolué pour offrir aux commerçants la possibilité de consulter des métriques relatives au nombre de transactions effectuées, autorisées et confirmées.

le SDK a été amélioré pour collecter activement des métriques liées aux transactions, telles que le nombre total de transactions, le nombre d'autorisations et de confirmations effectuées. Ces metrics sont récupérés du service Metrics de notre backend.

De plus, les tentatives de paiement seront désormais soumises à une stratégie de réessai exponentielle. On offre une manière pratique de configurer ce comportement de retry avec les paramètres suivants :

- retries : Le nombre maximal de tentatives de réessai. La valeur par défaut est de 3.
- factor : Le facteur exponentiel déterminant le délai entre les réessais. La valeur par défaut est de 2.
- minTimeout : Le temps minimum (en millisecondes) d'attente avant le premier réessai. La valeur par défaut est de 1000.
- maxTimeout : Le temps maximum (en millisecondes) entre deux tentatives de réessai. La valeur par défaut est de 3000.
- randomize : Un booléen indiquant s'il faut ou non randomiser les délais. La valeur par défaut est true.

Également,le SDK prend désormais en charge la spécification explicite d'un timeout, permettant aux clients d'éviter tout blocage indéfini au niveau du client.

Prérequis :
Téléchargez et installez Node.js et npm à partir d'ici.

Installation :
Utilisez npm pour installer notre SDK: **npm install @teamb/newbank-sdk**

Interfaces:

- authorizePayment(paymentInformation) : envoie au backend une demande d'autorisation de paiement et reçoit un Id de transaction si la demande est acceptée.
- confirmPayment(transactionId): envoie au backend une demande de confirmation du paiement de la transaction préalablement autorisée.
- pay(paymentInformation) : englobe les étapes d'envoi d'une demande d'autorisation de paiement au backend via la méthode "authorizePayment(paymentInformation)" et, en cas d'acceptation, elle confirme la transaction en utilisant la fonction "confirmPayment(transactionId)"
- getBackendStatus(): envoie une demande de récupération des statuts des services du backend.

Forces

- La division du service Payment Gateway en deux parties distinctes permet une spécialisation des responsabilités, favorise la modularité pour des évolutions indépendantes, et réduit les points de défaillance uniques (SPOF) en améliorant la résilience du système.

- La force d'une stratégie de retry avec backoff exponentielle réside dans sa capacité à espacer intelligemment les tentatives de retry après un échec, réduisant ainsi la charge sur le système de la banque. En augmentant progressivement les délais entre les réessais, cette approche atténue les effets des erreurs temporaires et prévient les surcharges potentielles.

```
payment request sent
Retry attempt 1 failed. Retrying...
Retry attempt 2 failed. Retrying...
Retry attempt 3 failed. Retrying...
All retry attempts failed. Last error: AxiosError: Request failed with status code 401
```

- Le fait de fournir l'API de statut des services au client n'a pas seulement répondu aux besoins métiers, mais nous a également permis de mettre en œuvre un

circuit-breaker côté client vu que nous avons ajouté le fait d'appeler implicitement le service d'état pour récupérer l'état des services avant de les interroger à chaque fois. Cela permet aux services de se remettre en route sans être bloqués par les appels lorsqu'ils redémarrent après une défaillance.

- La collecte des métriques métier se fait en temps réel grâce au CDC. Cela nous permet de suivre les événements métiers qui servent de métriques au fur et à mesure qu'ils se produisent, sans pour autant surcharger nos bases de données avec des appels de batch périodiques qui pourraient être mal programmés.

- Compte tenu de la nature en chaîne des appels effectués par notre système pour répondre aux besoins du sdk, le modèle de temporisation mis en œuvre permet d'interrompre les appels bloquants au cas où l'un des services de la chaîne s'avérerait être un service "sink" débordé et très lent à répondre, ce qui bloquerait le reste.

Faiblesses et points d'amélioration

- Notre approche actuelle pour l'envoi des métriques consiste à transmettre chaque nouvelle métrique à chaque étape de confirmation et d'autorisation du paiement. Cependant, cette méthode peut entraîner une surcharge du serveur de métriques, surtout avec un grand nombre d'utilisateurs. Une solution possible serait d'envoyer les métriques par lot (batch). Cependant, pour mettre en œuvre un envoi par lot, il est nécessaire de stocker les métriques de manière efficace avant l'envoi. Intégrer une base de données dans notre SDK ou mettre en place un système de cache sont des solutions que nous sommes en train de discuter et d'évaluer au cours des prochaines semaines.

- Notre SDK actuel transmet les erreurs provenant du système bancaire telles quelles au client. Une amélioration serait que le SDK gère chaque erreur en fournissant des informations détaillées sur l'erreur rencontrée. Cela permettrait aux développeurs utilisant notre SDK de mieux comprendre et réagir de manière appropriée aux problèmes qui surviennent, améliorant ainsi la robustesse des intégrations.

- Nous voulons également implémenter un mécanisme de backpressure directement dans le SDK. Cette fonctionnalité permettra de réguler de manière plus efficace le flux de transactions, et en cas de pointe de demandes, le SDK pourra réagir en temporisant ou en limitant la réception de nouvelles transactions, prévenant ainsi les échecs potentiels.

II. ARCHITECTURE LOGICIELLE - CONSTRUCTION

Projet

Le projet a pour objectif de concevoir un système bancaire sans numéraire qui supporte également la gestion des transactions en ligne pour des commerçants partenaires. Il inclut la conception d'un kit de développement logiciel (SDK) pour faciliter les paiements par carte de débit et de crédit sur les sites web des commerçants. De plus, il couvre la gestion des frais de transaction pour les transactions en ligne sans numéraire au sein de ce système bancaire.

Utilisateurs

1. Utilisateurs Principaux :
    - Commerçants : Ceux qui possèdent et exploitent des entreprises et qui utilisent l'application web pour gérer les transactions.
    - Clients : Individus souhaitant disposer d'un compte bancaire en ligne et d'une carte afin d'effectuer des transactions financières.
2. Utilisateurs Secondaires :
    - Administrateurs de la Banque : Responsables de la supervision du système bancaire en ligne sans numéraire.
    - Administrateurs Système : superviser la suppression de comptes et de cartes, de gérer les données sensibles et d'assurer la sécurité en cas de perte ou de vol.
    - Credit Card Network : ce réseau va nous solliciter afin d'autoriser une transaction chez nous (NewBank), notre système communiquera avec lui afin de lui demander de créer des cartes virtuelles.
    - Banques externes: Les banques externes nous sollicitent pour les transferts des fonds.

Cas d'Utilisation

- Enregistrement et Intégration des Commerçants :

Description : Un commerçant s'inscrit au service de gestion des transactions en ligne, fournissant les détails nécessaires à l'intégration au sdk.

- Gestion des Comptes Clients :

Description : Les clients peuvent créer et gérer leurs comptes, y compris les comptes d'épargne. Ils ont la capacité d'effectuer des opérations telles que l'ouverture de nouveaux comptes, la consultation des comptes existants.

- Consultation de Solde et Historique des Transactions Client :

Description : Les clients peuvent consulter le solde et l'historique des transactions de leur compte. Cette fonctionnalité permet d'obtenir des informations détaillées sur les mouvements associés au compte client.

- Virement entre Comptes :

Description : Les clients ont la possibilité de réaliser des virements entre leur compte bancaire et le compte associé à la carte de notre service.

- Traitement des Transactions avec Cartes de Débit/Crédit :

Description : Les commerçants initient des transactions en utilisant les détails des cartes de débit/crédit, qui sont traités de manière sécurisée par le système.

- Gestion des Frais de Transaction :

Description : Le système calcule et gère les frais de transaction associés à chaque transaction, assurant une facturation précise.

Hypothèses

- Les cartes virtuelles sont générées par les CCN, et par conséquent c'est le CCN qui se chargera de vérifier si un client possède une carte assignée à notre système NewBank pour nous appeler chez nous.
- La passerelle de paiement est chargée d'autoriser la transaction en effectuant les vérifications nécessaires sur l'appartenance du commerçant au système et en demandant l'autorisation externe le cas échéant pour les cartes de crédits des clients des marchands partenaires.
- Pour un paiement réussi, la passerelle de paiement doit effectuer deux appels : un premier appel au CCN pour confirmer l'éligibilité et la disponibilité des fonds sur la carte du client, et un second appel directement à la banque du client pour retenir les fonds.
- Les frais appliqués à une transaction varient et dépendent du type de carte, de notre marge en tant que passerelle et de la marge bancaire du commerçant.
- Les frais sont déduits durant le règlement, et le type de carte (débit ou crédit) est pris en considération par le service FeesCalculator qui fait cette déduction lors de l'application des frais.
- L'autorisation d'une transaction génère un jeton d'autorisation à usage unique de la banque cliente qui est récupéré par le CCN et conservé par la passerelle de paiement pour être utilisé pendant les étapes qui suivent.
- Si le commerçant est un client de notre banque, le service de règlement des paiements effectue lui-même la demande de règlement.
- Le règlement des transactions, qu'elles concernent notre passerelle ou notre banque interne, est effectué périodiquement et non pas toute de suite.

- Le SDK doit effectuer le cryptage des données de la carte, la vérification de la validité du numéro de carte saisit, et la demande de paiement avec le token de l'application.

Exigences et problèmes identifiées :

Exigences identifiées :

- Le système doit être disponible 24 heures sur 24 et 7 jours sur 7
- Il doit traiter potentiellement des milliers de transactions par seconde, provenant de commerçants partenaires.
- Une transaction "logique" confirmée ne doit jamais être perdue, dupliquée ou retrouvée dans un état corrompu.
- Les échanges financiers doivent être protégés de toute tierce partie.

Problèmes identifiés : évolutivité avec l'intégration des commerçants, sécurité des transactions, résilience en cas d'échec du paiement et disponibilité du système.

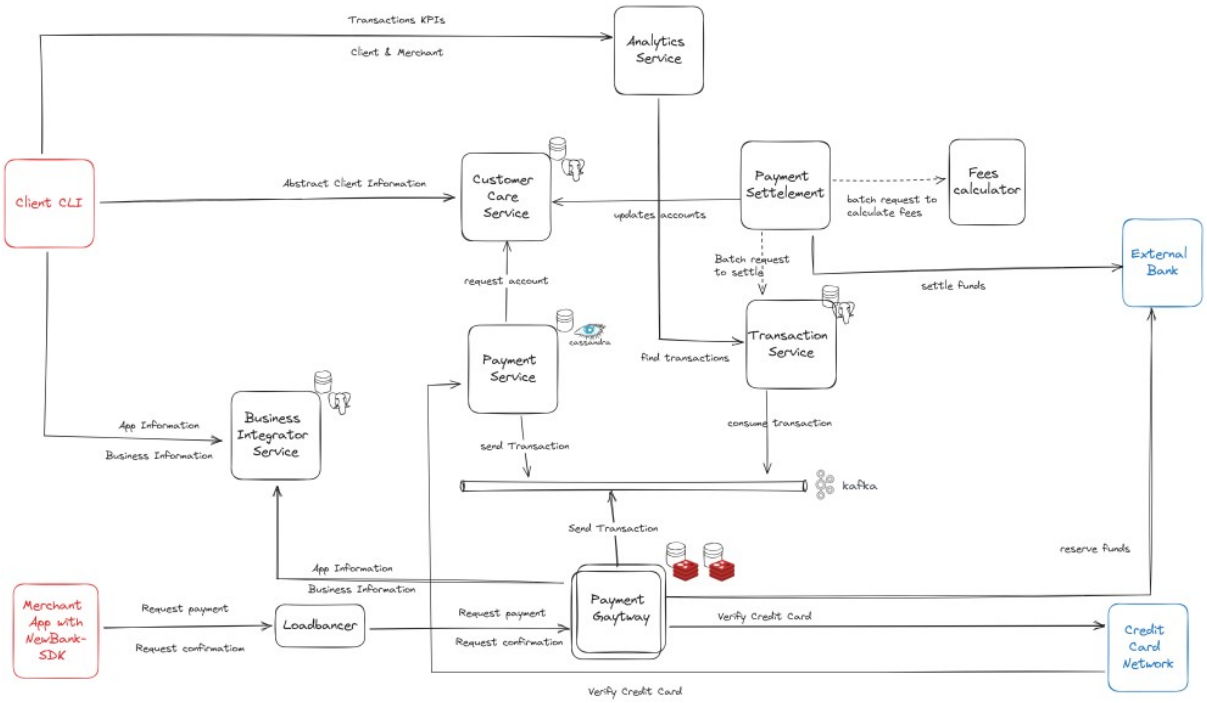
Choix technologiques :

- Le développement des services est fait en Java Spring Boot pour deux raisons : la familiarité de la stack technologique pour l'équipe et la robustesse de java (multithreading, strong typing).
- La technologie de la file d'attente décidée est Kafka vu sa capacité de traiter un débit très haut de messages, dans notre cas cela sera les transactions.
- Le reverse proxy du load balancer est implémenté avec Nginx. Nginx permet l'implémentation aisée de mécanismes tels que le “load balancing” et le "rate limiting".
- Pour les bases de données : Postgres a été choisi pour les données relationnelles, Redis pour les données temporaires "cachables" sous forme de clés-valeurs, et cassandra pour les données à haut débit d'écriture.

Architecture du système :

Vous trouverez ci-dessous notre diagramme d'architecture qui a été initialement dérivé du diagramme de composants conçu pour un monolithe et puis amélioré au fur et à mesure. Vous trouverez ci-après une explication détaillée des composants architecturaux, de leurs interfaces internes et de leurs interfaces externes.

Lien svg plus clair : [diagramme-architecture]



Nous avons choisi une architecture orientée services afin que chaque service de ce découpage représente une vue cohésive et claire du domaine du métier traité et ait une responsabilité claire qui reflète des fonctionnalités recherchées par la banque en ligne. Les fonctionnalités sont une réponse au parallèle que sont les besoins métiers. Notre découpage est axé sur le découpage du domaine lui-même.

Voici une description du rôle et des cas d'utilisation de chaque service ainsi que ses interfaces externes et internes :

- **CustomerCareService** :

Ce service gère la création de comptes, la logique d'épargne, les transferts, dépôts et débits, et attribue des cartes virtuelles à ses clients. Utilisant ***Postgres*** comme base de données, il manipule des données relationnelles claires, souvent sollicitées pour des opérations de lecture et d'écriture, et pourrait bénéficier de réplications pour des raisons de sécurité. En plus de ces opérations, il est chargé de toutes les interactions client, incluant la création de comptes utilisateurs et d'entreprises, la gestion des transactions financières directes, ainsi que la demande de cartes de crédit virtuelles pour les clients via un service externe : le réseau de cartes de crédit.

- Composants et Interfaces Internes :
    1. AccountFinder : Tout ce qui concerne la recherche du compte d'un client est géré par cette interface

2. AccountRegistration : En charge de l'intégration d'un client dans le système et de lui fournir son compte bancaire.
3. ChequeAccountHandler : il déplace les fonds en cas de dépôt, de transfert ou de débit.
4. SavingsAccountHandler : Différemment, il s'occupe de déplacer les fonds de l'épargne.
5. VirtualCardRequester : Chargé de demander et d'émettre une carte bancaire virtuelle à un client.
6. BusinessAccountHandler : permet de mettre à niveau le compte en business pour permettre d'augmenter la limite de paiement hebdomadaire.
7. AccountLimitHandler : permet de réinstaller la limite de paiement chaque semaine

- Interface externes :
    1. GET /api/costumer :
        - Description : Récupère tous les comptes client.
    2. GET /api/costumer/bankAccount :
        - Description : Récupère les donnée d'un compte
    3. GET /api/costumer/search
        - Description : Recherche un compte en fonction de différents critères.
        - Paramètres de Requête : iban, number, date, cvv
    4. GET /api/costumer/{id}
        - Description : Récupérer un compte par son identifiant.
    5. POST /api/costumer
        - Description : : Crée un nouveau compte.
    6. POST/api/costumer/{id}/virtualCard/debit
        - Description : Crée une carte virtuelle de débit pour un compte donné.
    7. POST /api/costumer/{id}/virtualCard/credit
        - Description : Crée une carte virtuelle de crédit pour un compte donné.
    8. PUT /api/costumer/{id}/funds
        - Description : Met à jour les fonds d'un compte.
    9. PUT /api/costumer/reservedfunds
        - Description : Met à jour les fonds réservés d'un compte.
    10. PUT /api/costumer/releasefunds
        - Description : Libère les fonds réservés d'un compte.
    11. PUT /api/costumer/{id}/movetosavings
        - Description : Transfère des fonds vers un compte d'épargne..
    12. POST /api/costumer/{id}/upgrade

        - Description : Met à niveau un compte vers un compte professionnel.
    13. PUT /api/costumer/{id}/deduceweeklylimit
        - Description : Dédue de la limite hebdomadaire d'un compte.
    14. POST /api/costumer/batchReleaseFunds
        - Description : Libère les fonds réservés pour plusieurs comptes.

- **PaymentProcessor service**:

Ce service est chargé d'autoriser la transaction réelle et la demande de paiement émanant d'une banque ou de notre réseau de cartes de crédit. Ce service effectue les contrôles de fraude nécessaires, et les contrôles de fonds, car il demande les informations sur le client au service clientèle avant d'approuver la transaction. Il envoie également l'événement à une queue afin de créer la transaction et de la stocker dans la base de données par le service dédié qui écoute sur la queue.
Le service payment–processor réalise une quantité significative de stockage des transactions provenant du gateway et de notre système interne, sans effectuer de mises à jour. Pour répondre à cette forte demande d'écriture, ***Cassandra*** est choisie en raison de sa capacité d'écriture robuste.

- Composants et Interfaces Internes :
    1. FundsHandler : application des contrôles nécessaires sur l'historique et le seuil des transactions, ainsi que vérification des fonds du compte.
    2. FraudDetector : Détecteur de fraude en cas, par exemple, de transactions multiples en même temps.
    3. TransactionProcessor : Vérifie l'existence du panier ou de l'IBAN et délivre une autorisation en cas d'approbation. Il envoie ensuite la transaction dans la queue interne.
- Interface externes :
    1. POST /api/payment/process
        - Description : :Traite un paiement en autorisant le paiement avec les détails fournis.
    2. POST /api/payment/checkCreditCard
        - Description : Vérifie une carte de crédit avec les informations fournies.
    3. GET /api/payment/transactions
        - Description : Récupère toutes les transactions.
    4. POST /api/payment/batchSaveTransactions
        - Description : Enregistre plusieurs transactions en lot.
    5. POST /api/payment/reserveFunds
        - Description : Réserve des fonds avec les détails fournis.
    6. POST /api/transfer/process

        - Description : Traite un transfert en autorisant le transfert avec les détails fournis.

- **Transactions service :**

Ce service s'occupera des insertions et des mises à jour de notre base de données de transactions. Il effectuera beaucoup d'écritures et il sera purement utilisé pour nous permettre d'effectuer des autorisations plus rapides dans d'autres services sans se soucier de la latence d'écriture directe et concurrente dans la base de données par ses services. Ce service va écouter les transactions sur un topic spécifique provenant du Payment processor Service et du PaymentGateway. Il va être sollicité pour récupérer ces transactions afin de les régler ou faire du reporting.

- Composants et Interfaces Internes :
    1. Persister : gère le stockage des transactions, qu'elles soient effectuées par l'intermédiaire du processeur de paiement ou de notre passerelle de paiement.
- Interface externes :
    1. POST /api/transactions
        - Description : Crée une nouvelle transaction.
    2. GET /api/transactions
        - Description : Récupère toutes les transactions.
    3. GET /api/transactions/toSettle
        - Description :Récupère les transactions en attente de règlement.
    4. PUT /api/transactions/settle
        - Description : Enregistre batch de transactions.
    5. GET /api/transactions/weekly
        - Description : Récupère les transactions hebdomadaires pour un IBAN donné.

- **PaymentSettlement service** :

Ce service s'occupera des mouvements de fonds et du règlement de toute transaction en cours qui a été approuvée à l'aide d'un cron job. Il recevra les demandes de transfert de fonds de nos comptes clients vers d'autres comptes bancaires, et inversement, il procédera aussi à la demande de règlement de fonds à partir d'autres comptes bancaires si ce sont nos clients qui doivent recevoir les fonds.

- Composants et Interfaces Internes :
    1. TransactionsDétenteur : Cette interface contient la logique de règlement des paiements : elle vérifie s'il s'agit d'un paiement entièrement interne ou externe et, si c'est le cas, rappelle l'organisme externe pour régler la transaction.
    2. FondsDéducateur : Cette interface est chargée de lancer l'application des mouvements de fonds liés aux comptes respectifs, ainsi que de

prendre la réduction qui a été calculée précédemment lors de l'appel de notre calculateur de frais.

- **Payment Gateway Service** :

Ce service a un rôle principal qui est le traitement des transactions effectuées par le SDK, l'appel au CCN pour l'autorisation de paiement du client et le déclenchement du règlement des fonds après l'autorisation. Ce service utilise ***Redis***, une base de données en mémoire de type clé-valeur, pour mettre en cache des autorisations extrêmement temporaires utilisées dans la confirmation des transactions entrantes. Il associe l'ID de transaction au jeton d'autorisation pour une confirmation rapide des transactions.

- Composants et Interfaces Internes :
    1. TransactionProcessor : Traite les transactions entrantes, délivre l'autorisation et demande une autorisation externe dans le cas où le client du commerçant utilisant notre sdk n'est pas un client de notre banque.
    2. IRSA : Décrypte les informations de carte de crédit dans le contexte d'une demande de paiement, ainsi que la génération ou la récupération de clés publiques RSA pour une application spécifique.
- Interface externes :
    1. POST /api/gateway/authorize
        - Description : Autorise un paiement avec les détails fournis, en utilisant un jeton d'autorisation et des informations cryptées de carte.
    2. GET /api/gateway/applications/public-key
        - Description : Récupère la clé RSA publique d'une application en utilisant un jeton d'autorisation.
    3. POST /api/gateway/confirmPayment/{transactionId}
        - Description : Confirme un paiement avec l'identifiant de transaction fourni.

- **Business Integrator Service** :

Ce service a pour rôle d'intégrer une entreprise et son application, de générer l'apiKey et de permettre l'utilisation de notre sdk en l'utilisant
Ce service utilise ***Postgres*** comme une base de données relationnelle pour l'association des commerçants et applications.

- Composants et Interfaces Internes :
    1. BusinessIntegrator : Intègre le commerçant dans notre système, à condition qu'il nous communique ses coordonnées bancaires.
    2. BusinessFinder : Gère la recherche des entreprises qui nous ont déjà rejoints.
    3. ApplicationIntegrator : gère l'intégration de l'application proprement dite en lui associant un apitoken, une clé publique et un commerçant.

4. ApplicationFinder : gère la recherche de l'application qui a été précédemment intégrée à notre passerell

- Interface externes :
    1. GET /api/integration/applications?name={name}
        - Description : Récupère une application en fonction de son nom.
    2. GET /api/integration/merchants/{id}
        - Description : Récupère un marchand en fonction de son identifiant.
    3. GET /api/integration/merchants?name={name}
        - Description : Récupère le compte bancaire d'un marchand en fonction de son nom.
    4. POST /api/integration/merchants
        - Description : Intègre un marchand avec les détails fournis.
    5. POST /api/integration/applications
        - Description : Intègre une application avec les détails fournis.
    6. POST /api/integration/applications/{id}/token
        - Description : Génère un jeton pour une application en fonction de son identifiant.
    7. GET /api/integration/applications/{id}/token
        - Description : Récupère le jeton d'une application en fonction de son identifiant.
    8. GET /api/validation?token={token}
        - Description : Valide un jeton d'application et renvoie les détails de l'application associée.

- **Analytics service** :

Ce service est responsable d'afficher les statistiques des activités financières des clients et des commerçants sur la plateforme, en fournissant des informations détaillées sur les transactions, les revenus, les dépenses et les soldes.

- Composants et Interfaces Internes :
    1. AnalyticsCustomer: est dédié aux transactions d'un client sur un mois donné. Il permet de fournir des détails sur les revenus et les dépenses journalières, ainsi que le solde mensuel, pendant un mois.
    2. AnalyticsMerchant :permet d'avoir des statistiques sur les revenus quotidiens des commerçants grâce à l'utilisation du SDK, ainsi que des frais quotidiens associés à leurs activités. De plus, il permet de suivre la variation quotidienne des profits des commerçants.
- Interface externes :
    1. /api/analytics/merchant?name={name}
        - Description : Récupère les analyses de bénéfices par jour pour un marchand donné.

    - Corps de Réponse : Liste d'objets MerchantAnalytics.
2. GET /api/analytics/customer?year={year}&month={month}
    - Description : Récupère les analyses clients pour un compte bancaire donné, une année et un mois donnés.

- **SDK :**

Ce composant représente le kit que nous fournissons et qui est utilisé par nos marchands partenaires. Il s'agit simplement d'une logique commerciale prête à être utilisée directement au lieu d'une interaction avec la passerelle via l'API. Il se charge de la récupération de la clé publique, le cryptage, la demande de paiement et la confirmation.

Protocole de paiement :

Lors de la simulation de paiements au sein de notre système, nous avons identifié un scénario courant de défaillance qui nécessitait notre attention : les échecs de paiement. Les échecs de paiement peuvent survenir pour diverses raisons, notamment lorsque le SDK (côté marchand) tombe en panne avant de recevoir une confirmation de 'succès', en cas de problèmes de réseau, d'interruptions de notre passerelle, et bien d'autres encore. En réponse à ces défis potentiels, nous avons reconnu la nécessité de rendre la procédure de paiement idempotente afin d'assurer un processus de paiement robuste et fiable.

Pour parvenir à l'idempotence, nous avons conçu un protocole de paiement en deux phases : l'autorisation et la confirmation. Les diagrammes de séquence présentés ci-dessous illustrent à la fois le scénario de paiement standard et un scénario de défaillance potentiel.

Lien vers la version SVG plus claire : SVG

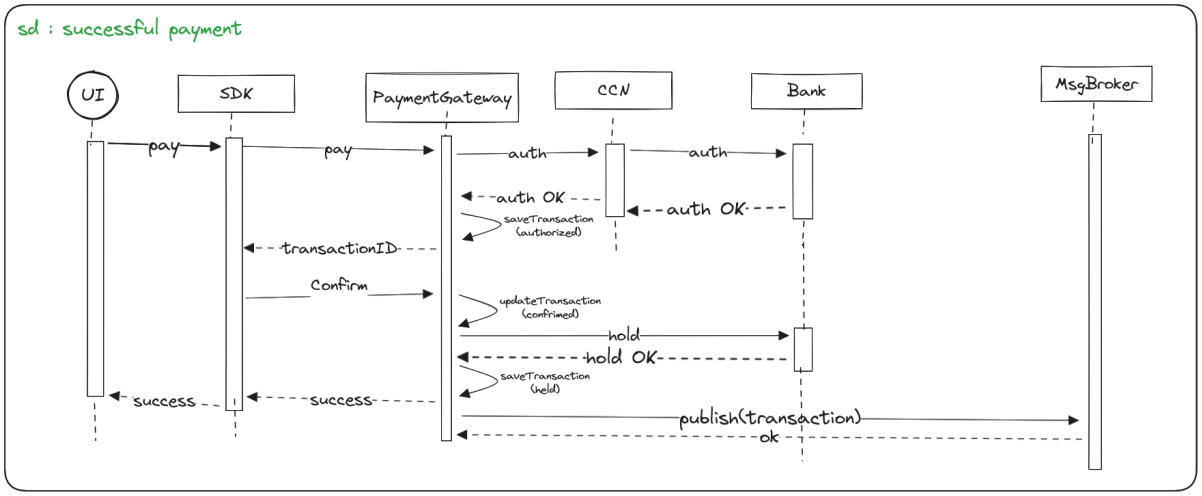


Figure 1 : Diagramme de séquence d'un paiement réussi

Lien vers la version SVG plus claire : SVG

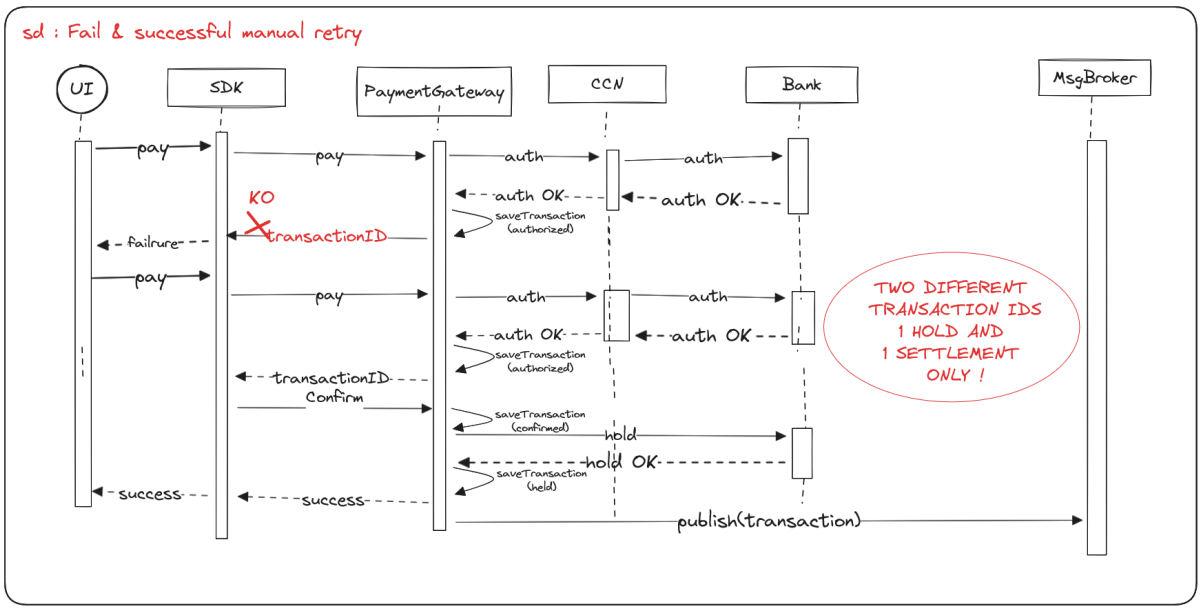


Figure 2 : Diagramme de séquence d'un paiement échoué et un retry réussi

Explications :

Dans la phase d'***Autorisation***, la passerelle traite la demande de paiement du client du marchand et effectue une vérification des fonds pour déterminer la solvabilité du client. Voici comment cela se déroule :

- Le client confirme le paiement et l'envoie via le SDK, qui en abstrait le processus.
- Le SDK initie le paiement en appelant la passerelle de paiement avec les informations de paiement encryptées.
- La passerelle de paiement effectue des vérifications de sécurité habituelles sur le jeton associé à la transaction.
- La passerelle de paiement demande l'autorisation de paiement au Réseau de Carte de Crédit (CCN).
- Le CCN contacte la banque émettrice de la carte de crédit et nous renvoie le jeton d'autorisation.
- Nous générons un identifiant de transaction, marquons la transaction comme 'APPROUVÉE' et la sauvegardons dans notre base de données avant de fournir à l'application SDK l'identifiant de transaction.

À la fin de la phase d'autorisation, aucun fonds ne sont retenus, et la transaction reste non confirmée.

Ensuite, dans la phase de ***'Confirmation'*** :

- Le SDK nous contacte à nouveau pour confirmer la transaction, en précisant l'identifiant de transaction.
- La passerelle de paiement effectue des vérifications de routine sur le jeton et d'autres détails.
- La passerelle de paiement met à jour le statut de la transaction en 'CONFIRMÉE'.
- La passerelle de paiement contacte directement la banque émettrice (la banque du client) pour bloquer les fonds à l'aide du jeton d'autorisation.
- Ensuite, la passerelle de paiement met à jour le statut de la transaction en 'RETENUE'.
- La passerelle de paiement répond au SDK avec une confirmation de succès.
- La passerelle de paiement place la transaction dans le message broker en vue d'une conservation et d'un règlement ultérieurs.

Ce protocole de paiement offre un certain degré d'idempotence en veillant à ce qu'en cas de défaillance, le client puisse réessayer et soit obtenir un nouvel identifiant de transaction ou bien résumer la procédure avec un identification de transaction déjà émis.

Si la défaillance survient après l'émission de l'identifiant de transaction, le marchand qui utilise notre SDK peut soit choisir de stocker et utiliser l'identifiant de transaction pour une nouvelle tentative, soit relancer le processus pour obtenir un nouvel identifiant de transaction. Notre SDK propose deux options pour mettre en œuvre ces procédures : l'utilisation directe de la méthode pour déclencher le protocole complet '.pay()' ou un contrôle à grains fins sur les phases avec les méthodes '.authorize()' et '.confirm()'.

En cas de défaillance survenant après la confirmation de la transaction, et si le marchand choisit de réessayer, notre passerelle peut vérifier le statut de la transaction.

Si le statut est 'RETENUE', la confirmation peut être effectuée directement. Si le statut est seulement 'CONFIRMÉE', le marchand peut réessayer avec le jeton d'autorisation. Cette hypothèse suppose que le jeton d'autorisation est un jeton à usage unique. Si la deuxième tentative de rétention échoue, cela signifie que la rétention précédente a réussi, et la transaction est considérée comme réussie et marquée 'retenue'."

Scénario MVP :

Lien vers la version SVG plus claire : SVG

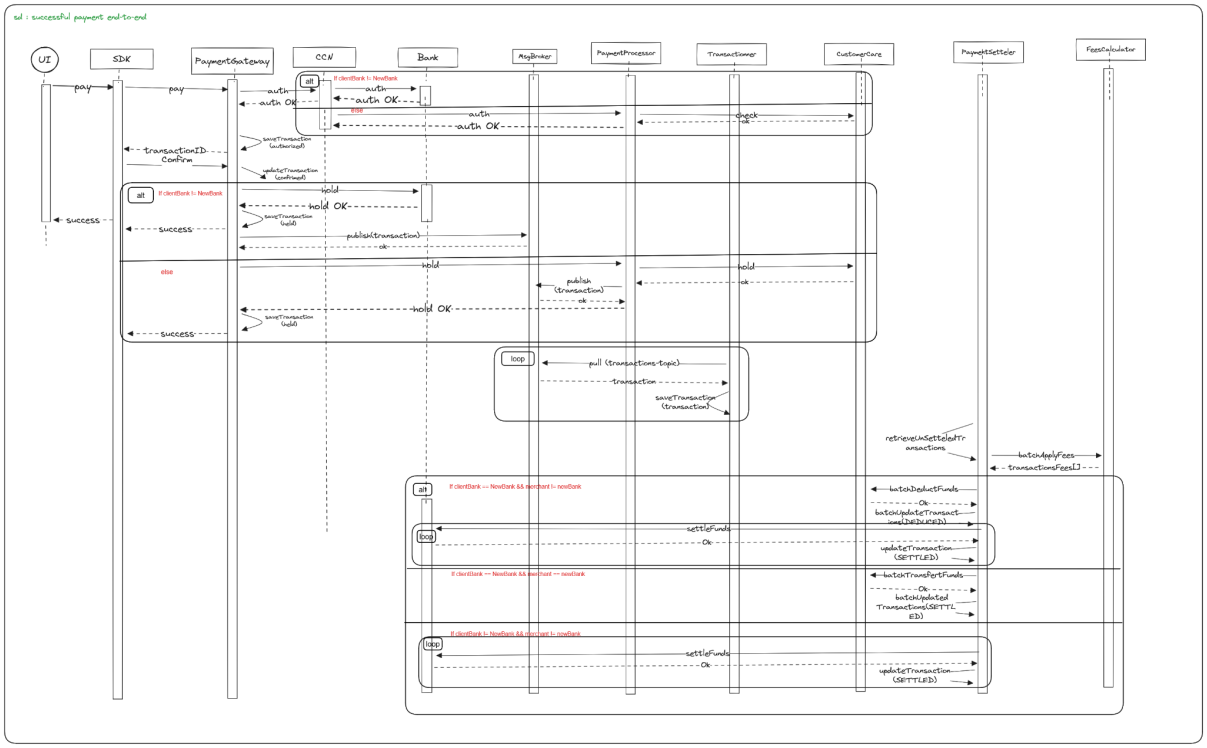


Figure 3 : Diagramme de séquence d'un paiement de bout-en-bout

Explication du scénario :

Pré-conditions : Ce scénario suppose que le commerçant s'est déjà joint à notre plateforme, a obtenu son jeton, installé le SDK et l'utilise pour permettre à ses clients de finaliser leurs achats et effectuer des paiements.

Le paiement se déroule en deux phases : la réussite de la transaction et le règlement du paiement. Ces deux phases se déroulent de manière séquentielle mais asynchrone.

La première phase :

Lorsque le client initie un paiement, le SDK envoie une requête à la passerelle de paiement, suivant le protocole défini pour le processus de paiement. Le Réseau de Carte de Crédit (CCN) détermine s'il doit contacter notre banque, Newbank, ou d'autres banques en fonction de la carte utilisée par le client.

En interne, lorsqu'il s'agit d'un client de notre banque, le processus de rétention des fonds est géré directement par notre processeur de paiement via la passerelle, activant le service client pour la rétention des fonds.

Une fois que le paiement est validé, une transaction est insérée dans notre file de messages, (une file Kafka). Un service de traitement des transactions surveille cette file, extrait les transactions réussies et les archive.

La deuxieme phase :

Les paiements sont réglés de façon asynchrone via une tâche planifiée. Le règlement est effectué par le régleur de paiement qui appelle le service qui stockent les transactions pour récupérer celles qui ne sont pas réglées.

Ensuite, le régleur de paiement sollicite le calculateur de frais pour appliquer les frais par lots. Une fois les frais récupérés, le règlement des fonds est effectué. Trois scénarios se présentent à ce stade : les deux parties impliquées possèdent un compte chez Newbank, facilitant un transfert direct des fonds et la marque des transactions comme réglées. Si seul le client est un client de notre banque, des mises à jour internes sont effectuées, suivies de demandes de règlement auprès des banques des commerçants.

Enfin, si ni le client ni le commerçant ne sont affiliés à notre banque, une requête est adressée à une banque fictive pour le règlement, puis nos transactions sont mises à jour en conséquence.

Prise de recul

Forces :

Notre architecture repose sur des principes solides de conception, notamment l'adoption du modèle CQRS (Command Query Responsibility Segregation), et divise distinctement les opérations d'écriture des services Payment Gateway et Payment Processor de celles de lecture du service Transaction. Cette séparation assure une meilleure cohérence du système, renforçant sa scalabilité et sa maintenabilité. En adoptant un traitement asynchrone des transactions, notre approche optimise les performances, réduit les goulots d'étranglement et offre une réactivité accrue, améliorant ainsi l'expérience utilisateur.

Dans notre système, le service Payment Gateway joue un rôle important dans le traitement et la validation des transactions issues de notre SDK. Afin de garantir une

disponibilité optimale et de gérer la charge, deux instances opérationnelles du service Payment Gateway sont mises en place, bénéficiant du soutien d'un load balancer. Également, sa base de données Redis, utilisée pour le stockage des transactions, est configurée en mode sharding, répartissant les données entre ces deux instances. Cette approche garantit une répartition équilibrée de la charge, ce qui améliore considérablement la réactivité globale du système, notamment avec la présence de deux instances opérationnelles du service Payment Gateway.

Notre composant proxy inverse continue de jouer un rôle essentiel dans notre système, agissant comme gardien limiteur de débit et équilibreur de charge, en se servant de la redondance active-active de notre passerelle de paiement afin d'améliorer la disponibilité. La combinaison de ces mesures vise à garantir une haute disponibilité et une robustesse dans notre système.

Pour l'extensibilité, nous avons utilisé REST pour que la fonctionnalité de notre passerelle soit facilement exposée par le biais d'un kit de développement logiciel. En effet, l'exposition d'interfaces externes simples, conformes aux normes REST et fournissant des points d'accès à la passerelle basés sur des cas d'utilisation simples, facilite l'extension et l'intégration de SDK à la passerelle.

En ce qui concerne la robustesse, notre approche a évolué. D'un point de vue fonctionnel, nous avons mis en place des mécanismes robustes de gestion des erreurs pour traiter à la fois les erreurs intentionnelles et non intentionnelles. Cependant, notre préoccupation architecturale principale porte désormais sur la garantie d'idempotence des paiements. Dans notre système de paiement, il est essentiel d'éviter la duplications des paiements clients pour une seule transaction. Ce défi est efficacement atténué grâce au protocole de paiement.

Notre SDK créé garantit la sécurité des transactions en utilisant un token JWT dans l'en-tête des requêtes, assurant l'authenticité des demandes et empêchant l'accès en cas de token invalide. De plus, il chiffre les informations de carte via l'algorithme RSA, assurant ainsi une transmission sécurisée des données sensibles. Facile à intégrer en tant que bibliothèque npm, il simplifie son utilisation pour les commerçants, bien que cela limite leurs options technologiques pour assurer une intégration optimale avec notre solution de paiement. Ainsi, notre SDK offre une sécurité renforcée pour les transactions, une intégration aisée et des mesures de cryptage robustes pour garantir la confidentialité des informations financières des clients.

Faiblesse :

Dans le processus du Payment Gateway, la phase de confirmation, actuellement gérée par ce service, pourrait être déléguée de manière asynchrone à un autre

service. Cette modification offrirait une meilleure flexibilité opérationnelle, en déchargeant le Gateway de certaines tâches et en permettant une meilleure gestion des échecs au niveau métier. Cependant, cette transition pourrait entraîner une complexité plus importante dans la gestion des échecs potentiels, exigeant une coordination plus étroite entre les services.

Également, la dépendance envers des systèmes externes tels que le service CCN et la banque externe peut introduire des problèmes de latence, engendrant des dépendances potentiellement critiques pour les opérations du système. Des règles strictes en matière de dépassement de temps peuvent s'avérer très utiles.

Du plus, l'utilisation d'un protocole d'encryptage asymétrique lourd avec une taille de 1024 bits peut contribuer au problème de latence.

Dans notre système, le traitement des transactions dans le service Payment Settlement se fait périodiquement par batch pour mettre à jour les bases de données. Cependant, l'exécution de ces batch peut entraîner une brève augmentation de la consommation de bande passante, impactant la vitesse de communication et surchargeant les services qu'il communique avec : CustomerCare, Transactions et FeesCalculator. Pour une efficacité optimale, il serait judicieux de définir une taille maximale pour les lots, limitant les perturbations potentielles sur le réseau et le traitement des transactions.

Bien que l'ajout de la base de données pour les transactions dans PaymentProcessor ait pour objectif principal de vérifier les limites de transactions, nous avons obtenu dans ce cas des redondances de stockage dans PaymentProcessor et TransactionService.

Le load balancer, essentiel pour répartir le trafic, risque de devenir un point de congestion en cas de ressources insuffisantes. Bien qu'il soit configuré avec un mécanisme de répartition et un dispositif de sécurité en cas de panne, son efficacité peut être compromise en cas de charge excessive due au manque de ressources. Cela pourrait causer des retards dans le traitement des requêtes, affectant ainsi les performances générales du système.

Dans le contexte des transactions via le SDK, des lectures répétées dans les bases de données pour vérifier les commerçants et leurs applications, ainsi que pour consulter les statistiques spécifiques à un commerçant, peuvent entraver les performances et la fluidité des opérations. Pour améliorer l'efficacité du système, l'adoption d'une architecture de réplication maître-esclave pour la base de données serait avantageuse. En déléguant les requêtes de lecture aux bases de données esclaves, cette configuration répartit la charge, accélérant ainsi le traitement des opérations.

Perspectives futures

En regardant vers l'avenir de notre projet, nous envisageons d'intégrer des loads balancers supplémentaires aux niveaux d'autres microservices pour optimiser la distribution de la charge sur ces microservices, en particulier ceux qui connaissent une demande plus soutenue.

Parallèlement, l'intégration d'une API Gateway constitue une étape stratégique, centralisant l'accès aux divers microservices tout en simplifiant la gestion des requêtes entrantes, et cachant les Endpoints qui ne sont pas censées être exposées.

Dans le but d'améliorer la résilience de notre système, nous prévoyons de mettre en place des mécanismes de réessai (retry mechanisms) pour les appels externes, assurant ainsi une gestion robuste des éventuelles défaillances temporaires. Des Circuit Breaker pourraient être utiles pour gérer le cas d'une communication interne vers les services internes défaillants. En outre, nous planifions d'implémenter la réplication comme mécanisme de sauvegarde (BACKUP), assurant la disponibilité des données critiques (transactions).

Sur le plan de la fonctionnalité, nous envisageons également d'ajouter le support de différents types de paiement tels qu'Apple Pay, Google Pay, et PayPal.

Organisation et auto-évaluation du travail

La répartition du travail au sein de l'équipe a été collaborative . Chacun d'entre nous a contribué activement au développement et à la conception de l'architecture du projet. Nous avons maintenu une collaboration continue tout au long du processus, travaillant ensemble sur les améliorations nécessaires. Pendant le projet, Imene a assumé la responsabilité des tests end to end pour garantir le bon fonctionnement de l'ensemble. De leur côté, Sourour et Nadim ont pris en charge les intégrations, notamment l'ajout de nouvelles bases de données, la mise en place du load balancer, et la création d'une pipeline avec GitHub Actions sur Git. Badr s'est concentré sur les tests de charge a l'aide prometheus grafana pour assurer une approche complète et équilibrée tout au long du projet.

Répartition du travail dans l'équipe sur un total de 400 points :

1. Nadim BEN AISSA : 100 points
2. Badr AL ACHKAR : 100 points
3. Sourour GAZZEH : 100 points
4. Imene YAHIAOUI : 100 points