NewBank : Merchant webApp

Equipe B

Yahiaoui Imène - Gazzeh Sourour
Ben Aissa Nadim - Al Achkar Badr

Sommaire

ARCHITECTURE LOGICIELLE - EVOLUTION

Projet

Le projet vise à développer un système bancaire sans numéraire, qui sert non seulement à gérer les transactions en ligne pour des commerçants partenaires, mais aussi à répondre à leurs besoins évolutifs. Il comprend la conception d'un kit de développement logiciel (SDK) pour intégrer facilement les paiements par carte de débit et de crédit sur les sites web des commerçants, tout en gérant les frais de transaction pour ces opérations en ligne. Ensuite, le système a été enrichi de fonctionnalités supplémentaires, telles que la surveillance des statuts et la récupération des métriques métiers, pour offrir une solution complète et adaptée aux défis du commerce électronique.

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.

- Récupérations de métriques métiers et surveillances des KPIs :

Description : Les commerçants sont en mesure de récupérer des métriques métiers du système et de suivre l'évolution de leurs indicateurs clés de performance au fil du temps.

- Surveillance de l'état des services :

Description : Les commerçants sont en capacité de connaître en temps réel la disponibilité de nos services.

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.
- Les métriques métier fournies aux commerçants sont déduites à partir des transactions effectuées (échouées et réussies).
- L'état des services qui est signalé est soit UP, DOWN, ou OVERLOADED. L'objectif est de fournir une vue d'ensemble des services commerciaux que le client peut vérifier en cas de comportement anormal et de rester informé des pannes.

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 utilisant le SDK.
- 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'échecs des paiements et des pannes système.

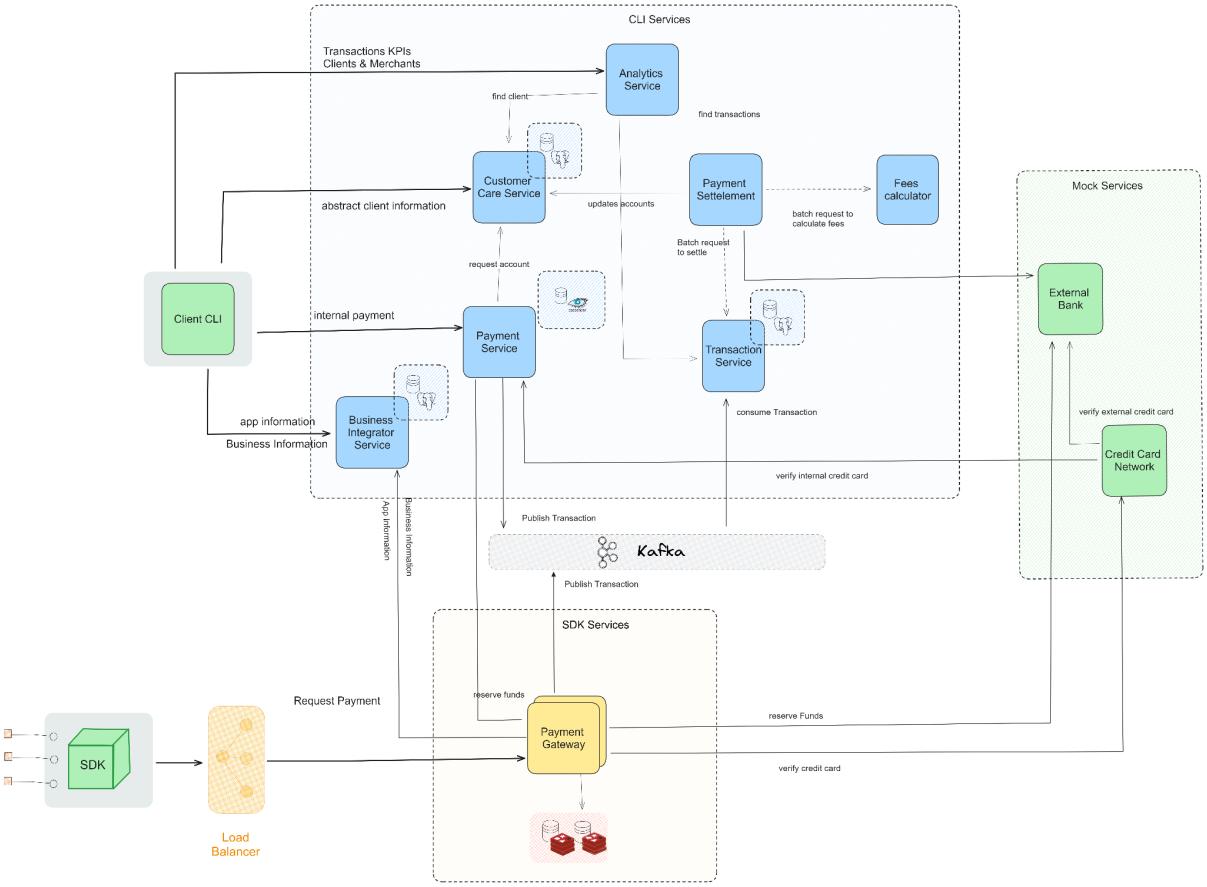
Choix technologiques :

- 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, Cassandra pour les données à haut débit d'écriture et finalement Mongo pour l'agrégation des données dans un schéma flexible.

Ancienne architecture du système :

Vous trouverez ci-dessous notre diagramme d'architecture initial. Vous trouverez ci-après une explication détaillée des composants architecturaux, de leurs interfaces internes et de leurs interfaces externes.

svg plus clair de l'ancienne architecture : ancien.svg



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** :

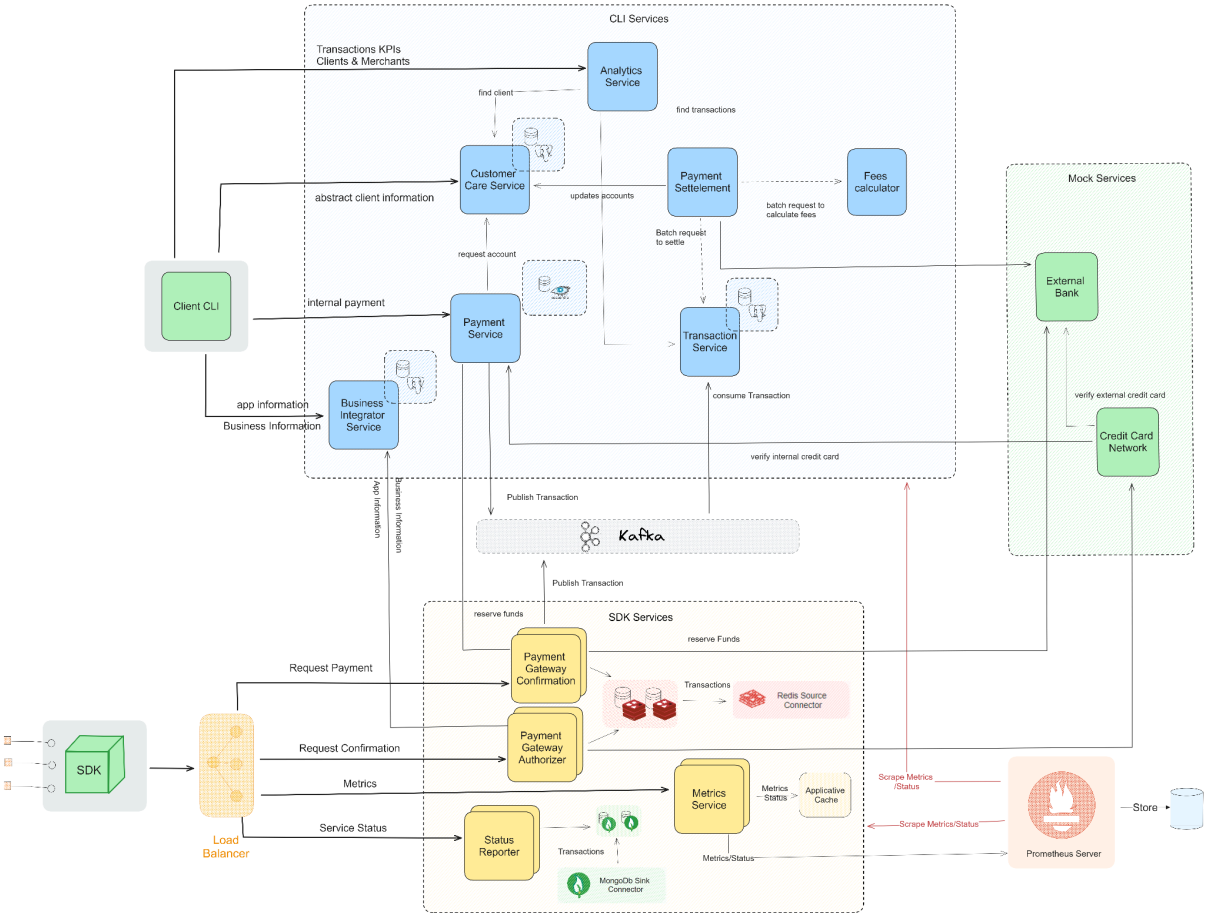
Ce service est responsable d'afficher les statistiques des activités financières des clients et des commerçants sur la plateforme, en fournissant des informations détaillées sur les transactions, les revenus, les dépenses et les soldes.

- 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.

Nouvelle architecture du système :

Vous ci-dessous notre nouvelle architecture après l'avoir fait évoluer pour s'adapter aux nouveaux besoins. Vous trouverez ci-après une explication détaillée des nouveaux composants, de leurs interfaces internes et de leurs interfaces externes.

Lien vers la version SVG plus claire : new-archi.svg



- **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 système et métier aux administrateurs, notamment en ce qui concerne les transactions, facilitant ainsi la visualisation des activités commerciales en extrayant et en analysant des données pertinentes.

- Interfaces Internes :
    1. IMetricsService: gère les requêtes de métriques en récupérant les métriques demandées et en traitant les données nécessaires associées
    2. RequestRegistery: enregistre les métriques des requêtes reçues par les services web des clients dans la base de données des métriques MongoDB.
- Interface externes :
    1. POST /api/metrics
        - Description : permet de recevoir des requêtes de récupération de métriques et de les traiter.
        - Body: une structure JSON contenant des paramètres spécifiques pour la récupération de métriques.
          Voici un exemple :

```
{
        "period": "L6H",
        "resolution": "5M",
        "metrics": [
        "transactionCount",
        "TransactionSuccessRate",
        "TransactionFailureRate",
        "totalAmountSpent"
        ],
        "filters": {
        "status": [
                "AUTHORISED",
                "CONFIRMED"
        ],
        "creditCardType": [
                "credit"
        ]
        }
}
```

Description des champs :

- **period** : Période pour laquelle les métriques doivent être récupérées (par exemple, "L6H" pour les dernières 6 heures).
- **resolution** : Résolution temporelle des métriques (par exemple, "5M" pour une résolution de 5 minutes).
- **metrics** : Liste des métriques spécifiques à récupérer (par exemple, "transactionCount", "TransactionSuccessRate", etc.).
- **filters** : Filtres facultatifs pour restreindre les données en fonction de certains critères (par exemple, le statut "AUTHORISED" ou "CONFIRMED", et le type de carte de crédit "credit").

2. Post /api/metrics/request
    - Description : Permet de reçevoir les demandes de création de métriques de requêtes provenant des applications des clients.
    - body:

```
{
        "time": "15-01-2024T1:14:50",
        "duration": 600,
        "status": "FAILED",
        "details": "informations_supplementaires"
}
```

- **time** : Date et heure de la demande.
- **duration**: Durée de la demande.
- **status** : Statut de la demande.
- **details** : Informations supplémentaires associées à la demande.

- **Status Reporter service :**

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.

Analyse comparative :

Scission de PaymentGateway

Le service original "Payment Gateway" gérait l'intégralité du protocole de paiement, c'est-à-dire à la fois l'autorisation et la confirmation du paiement. Il s'agissait donc d'un service de point d'entrée surchargé, ce qui représentait un SPOF (Single Point Of Failure) dans notre système.

Pour résoudre ce problème, nous avons décidé de le diviser en deux services, chacun gérant un aspect du protocole de paiement : "Payment Gateway Authorizer Service" et "Payment Gateway Confirmation Service". Cependant, nous continuons à accéder au même datastore Redis partagé. Le raisonnement derrière cela est qu'il n'y aura pas d'accès simultané à la même clé, par les deux services. Le protocole de paiement engendre naturellement un accès séquentiel.

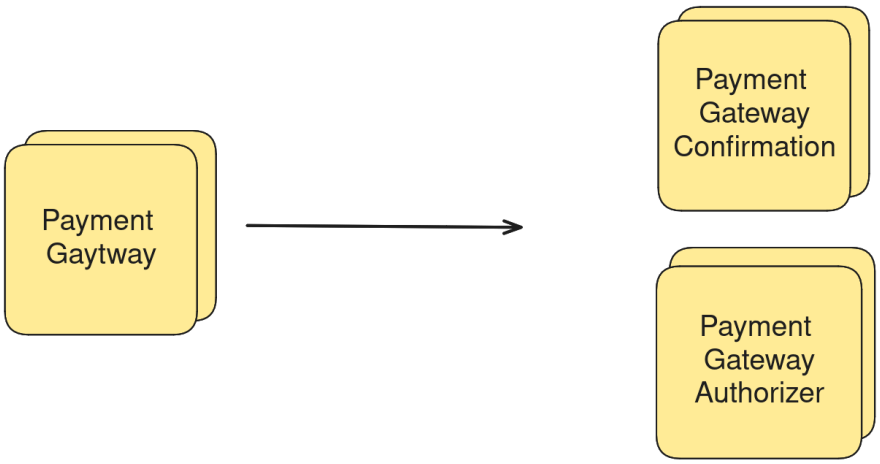


Figure 3 : Scission de PaymentGateway

Mise en Place d'un Moniteur de Services Internes Piloté par Prometheus

Le client a exprimé le besoin d'une nouvelle fonctionnalité : des informations sur le statut des services en temps réel. Notre système a donc été soumis à des changements et a évolué pour répondre à ce nouveau besoin.

Nos services, tels qu'ils sont, sont surveillés et scrappés périodiquement et constamment par un système de surveillance en cours d'exécution : Prometheus. Nous avons décidé de l'exploiter pour extraire des métriques sur l'état de nos systèmes, et de les exposer à travers un nouveau service : Status reporter. Le

principe de cette approche est de protéger notre instance de surveillance en cours d'exécution d'être exposée au monde extérieur, et d'agréger les informations nécessaires à donner à ce Status Reporter afin de prendre une décision informée sur l'état d'un certain service et de renvoyer le résultat ensuite à l'utilisateur final.

En outre, un cache applicatif a été mis en place pour mettre en place une stratégie de caching “read-through” afin de ne pas submerger le système de surveillance prometheus avec les requêtes en provenance de Status Reporter. Le cache est très rapidement invalidé et ne sert que de magasin très éphémère pour ne pas conserver des informations incohérentes.

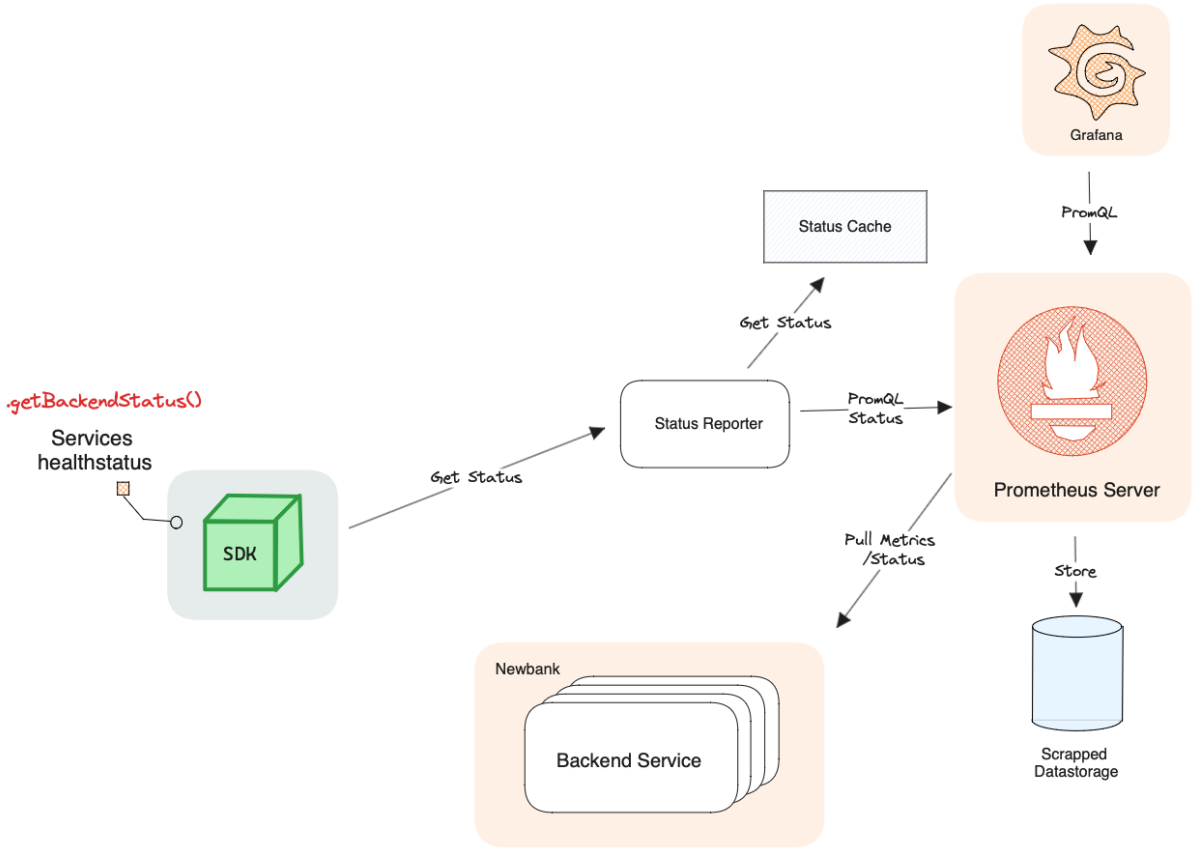


Figure 4 : Moniteur de Services Internes Piloté par Prometheus

Métriques : Introduction du pattern CDC pour l'agrégation des données

Similairement à la précédente évolution, un nouveau besoin a émergé : un tableau de bord des métriques métier.

Pour faire évoluer notre architecture et répondre à ce besoin particulier, nous avons choisi une approche whitebox dans laquelle non seulement nous extrayons les métriques de surface qui concernent les demandes métier, mais nous plongeons en

profondeur dans nos sources de données pour agréger les informations métier en temps réel et tenir le client informé à tout moment.

Les principales métriques métiers, qui concernent les transactions, sont stockées dans une base de données en mémoire au moment de l'émission de la transaction financière, d'abord pendant l'autorisation et la confirmation, puis transférées à une base de données relationnelle plus tard, avec un statut différent, au moment de la déduction des frais et du règlement.

L'extraction directe de ces métriques à partir d'une base de données en mémoire n'est pas efficace, car elle ne conserverait pas les données de tous les temps pour établir des rapports sur cette première phase. En outre, l'extraction de métriques à partir de notre base de données relationnelle stable exposerait notre source de vérité au risque d'un nombre alarmant de requêtes effectuées par un outil de visualisation afin de maintenir un tableau de bord à jour.

Pour remédier à ces lacunes, il a été décidé de créer un service de métriques dédié qui accède à une base de données de documents flexible NoSQL, et qui agrège toutes ces informations et renvoie les métriques nécessaires en temps réel.

Afin d'alimenter cette base de données au fur et à mesure que les autres sources de données sont alimentées, nous avons mis en œuvre un modèle architectural de CDC (change data capture), dans lequel nous accédons aux journaux d'opérations de nos bases de données, extrayons les événements qui arrivent et les publions dans notre magasin d'événements : la file d'attente Kafka. À l'autre extrémité de ce magasin d'événements, notre auditeur de changement consomme les événements et interagit avec notre magasin de documents pour le maintenir à jour.

Les indicateurs d'activité sont désormais accessibles grâce à un service “Metrics Service” s'appuyant sur une base de données riche.

De plus, nous avons créé un index dans la base de données mongodb sur le champ client (applicationId) puisque le filtrage est toujours effectué par ce champ afin de retourner les métriques spécifiques du marchand, et puisque ce n'est pas un identifiant unique mais l'identifiant de la transaction est utilisé à la place comme identifiant unique par défaut dans la base de données (c'est donc la colonne actuelle qui reçoit l'indexation par défaut lors de la création du document).

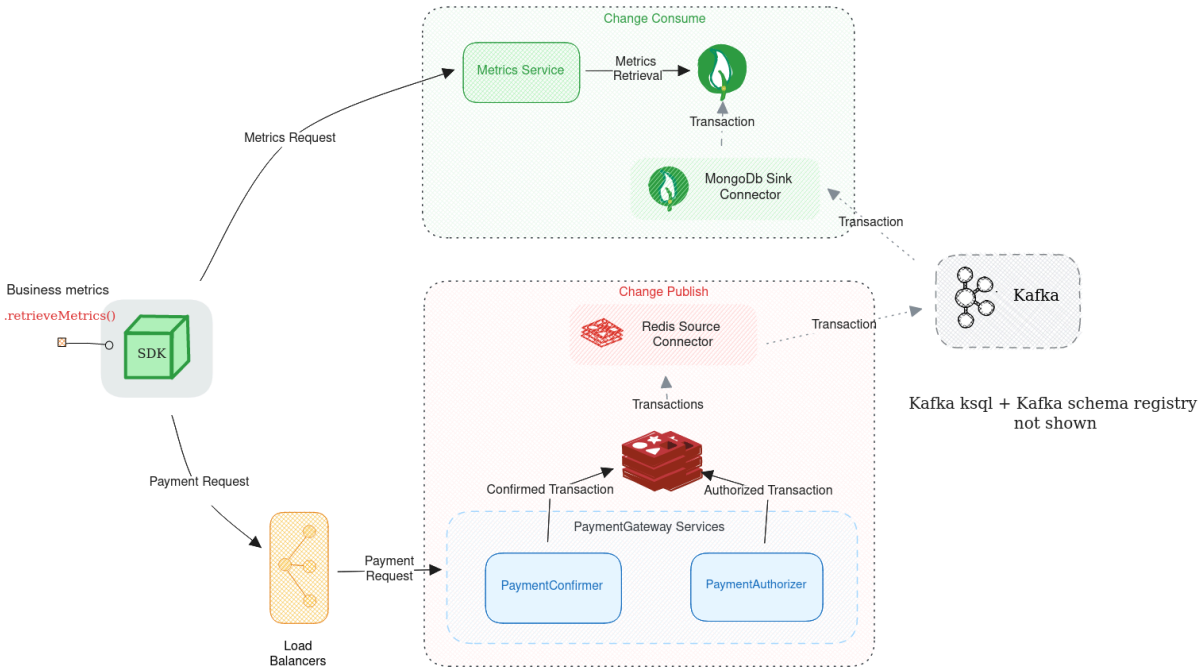


Figure 5 : CDC avec un connecteur source kafka sur redis

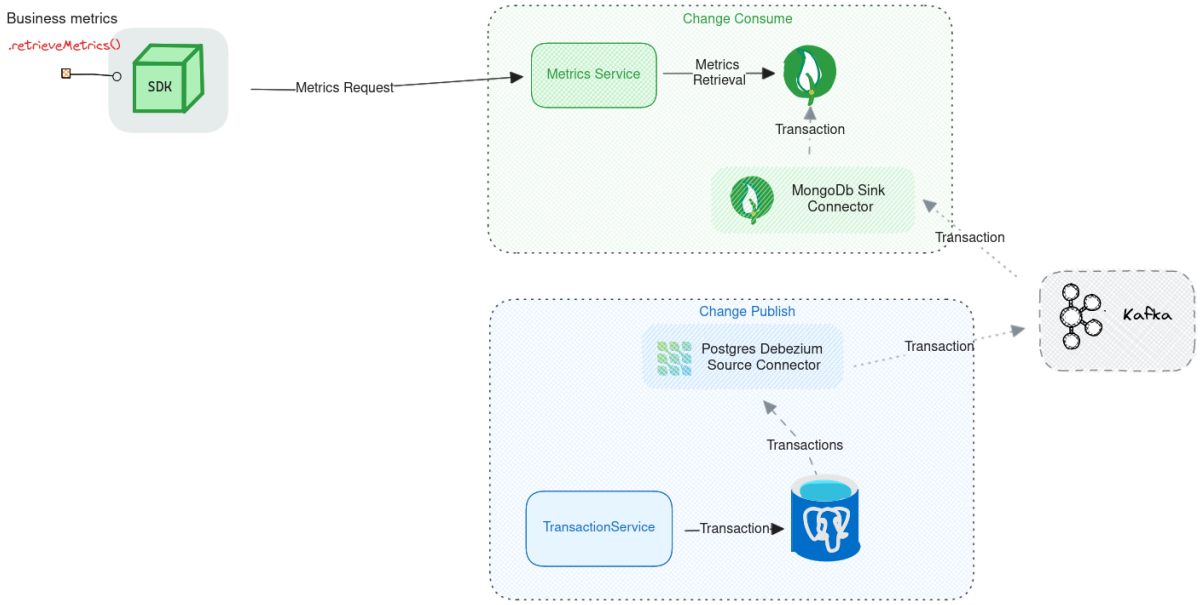


Figure 6 : CDC avec un connecteur source kafka sur postgres

Timeout Pattern : Mitigation des Appels Externes Chronophages

En adoptant une architecture SOA, nous étions conscients des problèmes qui pouvaient survenir en cas de défaillance du réseau ou d'autres problèmes de communication entre services. Certains services, en particulier le service Authorizer et le service Confirmation de la passerelle de paiement, récemment créés, communiquent fréquemment avec des composants internes et externes au système. Cela représente une grande menace pour les performances de notre système, car les appels en chaîne augmentent la latence et le risque de blocage. Pour remédier à ce problème, nous avons mis en place une politique de délai d'attente pour tous nos services afin de garantir que la latence ressentie par l'utilisateur final est acceptable.

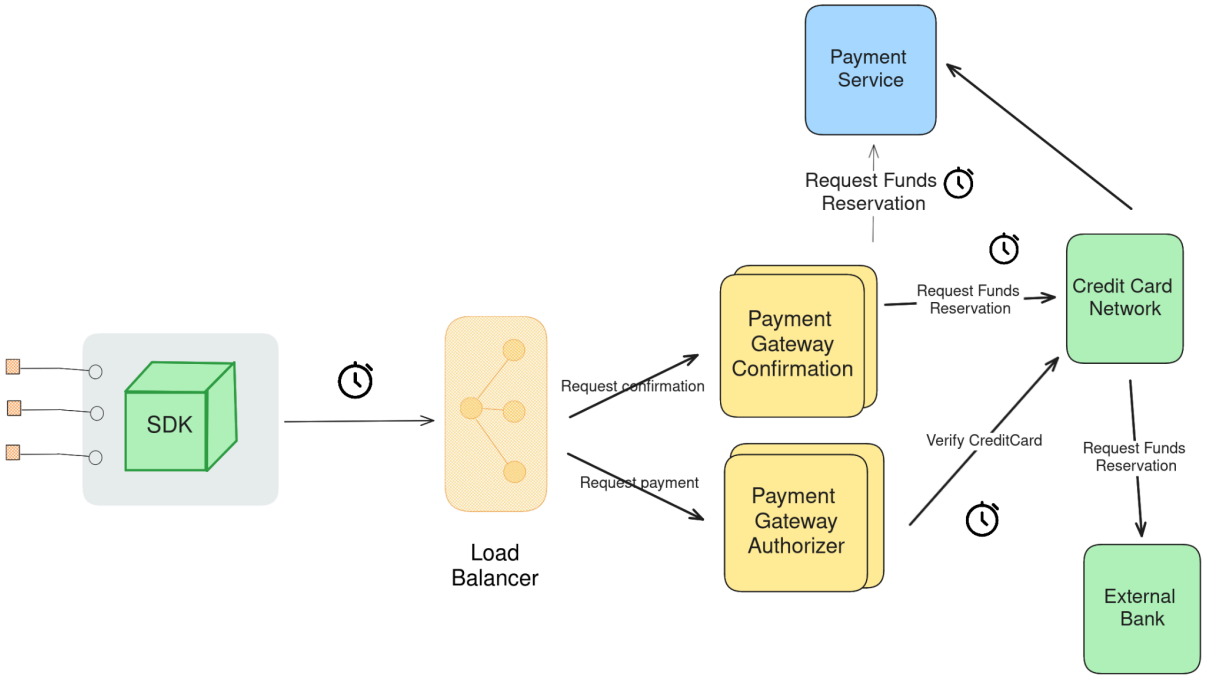


Figure 7 : Délais de Timeout au niveau des communication avec des services externes

SDK (KIT DE DÉVELOPPEMENT LOGICIEL) :

La plateforme Newbank Developer fournit un SDK que les développeurs peuvent utiliser pour intégrer les applications web de leurs commerçants avec les solutions commerciales de Newbank.

Si l'application web du commerçant peut envoyer une requête HTTP et recevoir une réponse, elle devrait être en mesure d'appeler n'importe quelle API externe de NewBank. Cependant, elle doit également effectuer d'autres tâches, telles que le choix du point de terminaison correct, le formatage des en-têtes de demande et des charges utiles, la construction des URL, l'envoi des demandes et l'analyse des corps de réponse, tout en respectant les protocoles mis en place pour profiter des services NewBank.

Le kit de développement logicielle fourni facilite l'intégration avec notre système de paiement, offrant aux développeurs une interface simple et claire pour interagir avec les fonctionnalités et les services offerts aux marchands partenaires.

La documentation technique du SDK avec l'onboarding du développeur est présente dans le README du SDK.

Choix technologiques

Le SDK est jusqu'à présent implémenté en typescript, afin de fournir un kit typesafe et accessible sur les applications web. Il nécessite une version minimale de node 16.17.0 en raison des opérateurs utilisés : optional chaining, nullish coalescing…

Fonctionnalités et interfaces

Paiement :

Le paiement est le service principal offert par Newbank aux commerçants partenaires. Il permet l'émission de transactions financières par les clients. Il est implémenté dans le SDK de manière à respecter notre protocole de paiement idempotent.

Protocole 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 mis en place 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.

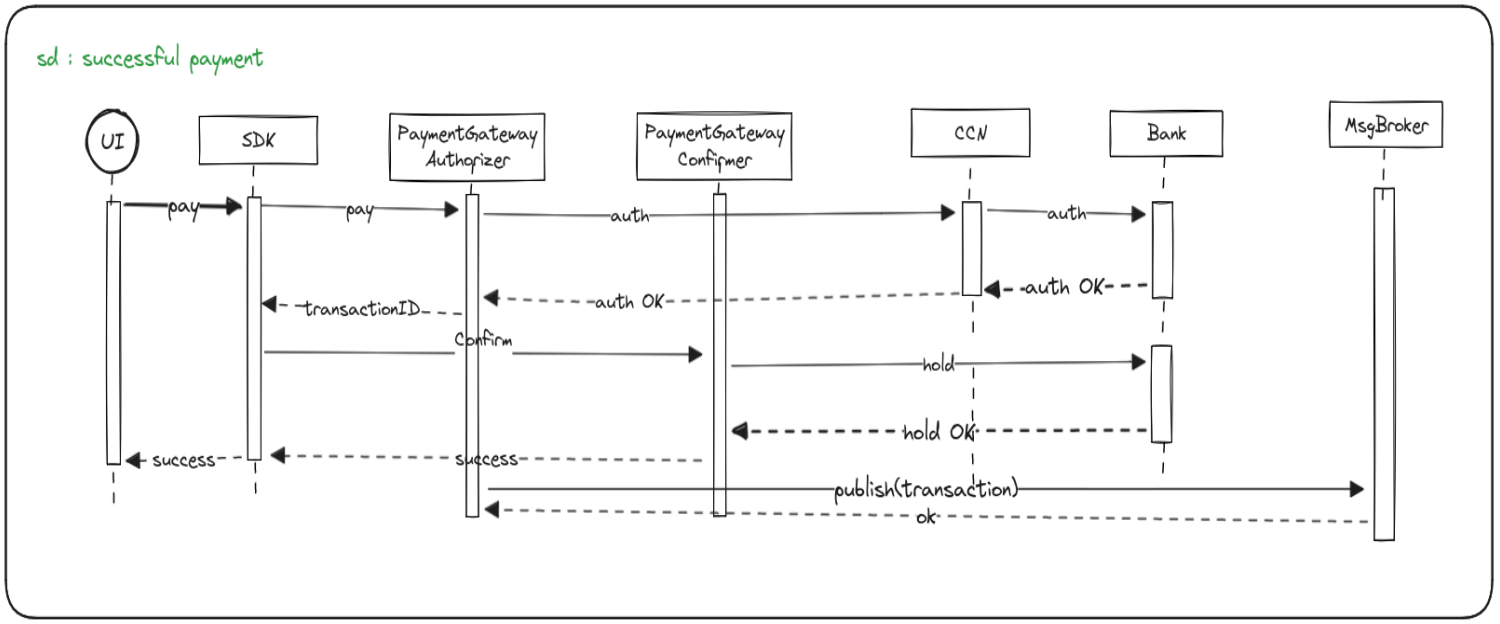


Figure 1 : Diagramme de séquence d'un paiement réussi

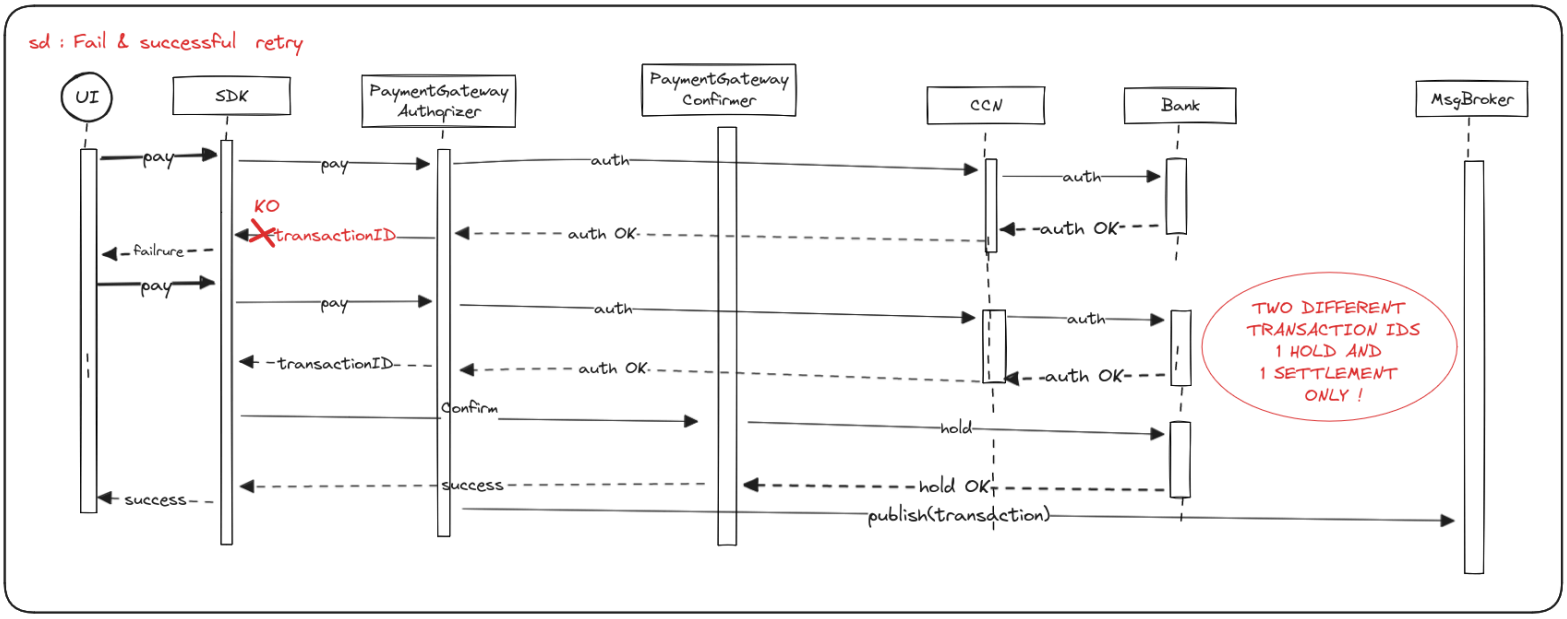


Figure 2 : Diagramme de séquence d'un paiement échoué et un retry réussi

Description :

Dans la phase d'***Autorisation***, le service d'authorisation 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 : récupération de clé de chiffrement, cryptage des données, envoie de la demande avec l'api token de l'application du commerçant.
- Le SDK initie le paiement en appelant le service d'autorisation de paiement avec les informations de paiement encryptées.
- Le service effectue des vérifications de sécurité habituelles sur le jeton associé à la transaction.
- Le service 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 auprés de service de confirmation des paiement, en précisant l'identifiant de transaction.
- Le service de confirmation effectue des vérifications de routine sur le jeton et d'autres détails.
- Le service de confirmation met à jour le statut de la transaction en 'CONFIRMÉE'.
- Le service de confirmation 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'.
- Le service de confirmation répond au SDK avec le succès de confirmation.
- Le service de confirmation 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'.

Interfaces de l’API :

- authorizePayment(paymentInformation) : Envoie une demande d'autorisation de paiement au Payment Gateway Authorizer Service et retourne un Id de la transaction si l’authorisation est accordée.

- confirmPayment(transactionId) : Envoie une demande de confirmation du paiement de la transaction préalablement autorisée au Payment Gateway Confirmation Service et retourne si la confirmation a abouti.
- pay(paymentInformation) : Regroupe les deux étapes d'envoi d'une demande d'autorisation et de confirmation de paiement, au cas où le marchand n'est pas contraint par la gestion des deux phases et des transactions ids dans son code métier.

Métriques :

Les métriques sont là pour fournir une visibilité sur les kpis de base de l'entreprise qui pourraient être extraits des métriques métiers renvoyées par l'api.

Notre SDK a évolué pour tenir compte de ce besoin et le présenter sous une forme plus conviviale. Ce service est présenté dans le kit sous une forme adaptée à la conception de tableaux de bord : il prend en charge la spécification de la période de récupération, la résolution du temps entre deux points de données et le filtrage des données renvoyées.

En outre, le SDK lui-même fournit certaines métriques métiers au système en mode Blackbox, en interrogeant implicitement le Metrics Service à chaque fois qu'un paiement est effectué pour l'informer de son échec ou de sa réussite. Ceci est utile pour collecter des métriques externes au système, détecter les divergences entre l'interne et l'externe et définir des alertes en cas de taux d'échec élevé. Ces métriques sont aussi accessibles via cet Api.

Interfaces de l'API :

- getMetrics(metricsConfiguration) : Demande au Metrics Service les métriques métiers à base de la configuration fournie par le client et renvoie le résultat sous un format json.

État des services :

L'état des services est une information cruciale à communiquer aux commerçants partenaires afin de les informer des pannes, de la maintenance ou d'autres interruptions de service importantes.

Au fur et à mesure que notre système évoluait pour répondre à ce besoin, nous l'avons également présenté dans notre SDK sous la forme d'une API facile à utiliser.

Interfaces de l’API :

- getBackendStatus(): Demande au statut reporter l’état des services et renvoie une liste d’objet json contenant cet état.

Mécanismes de résilience mises en places

Retry configurable

Étant donné que le paiement peut échouer et que nous nous sommes occupés des effets de bord de ce problème avec notre protocole de paiement, nous avons fourni un moyen d'effectuer de nouvelles tentatives sur ces paiements dans notre SDK.

L'objectif est de décourager les clients d'envelopper notre SDK dans une couche de relance et d'utiliser la nôtre qui met en œuvre un mécanisme de backoff exponentiel pour ne pas submerger les services.

Également, le SDK prend en charge aussi la spécification explicite d'un timeout, permettant aux clients d’éviter tout blocage indéfini au niveau de leur côté.

CircuitBreaker

Étant donné que des interruptions de service peuvent se produire et que le client peut être tenté de choisir une approche naïve consistant à réessayer indéfiniment, nous avons décidé de mettre en œuvre un circuit breaker dans notre SDK pour envelopper ses appels aux services.

L'approche consiste à exploiter notre Statut Reporter, et à l'interroger avant d'interroger tout autre service afin de connaître l'état du service que l'on souhaite appeler. Si le service est en état d'accepter des appels, il est interrogé, sinon l'appel est considéré échoué.

Le choix d'interroger le rapporteur d'état plutôt que le service directement a pour but de ne pas le submerger davantage d'appels de vérification d'état puisqu'un service d'état existe déjà.

Cette approche entraîne une prolifération d'appels externes à ce rapporteur en cas de perturbations du service. Cependant, étant donné qu'il ne s'agit pas d'un appel coûteux et que ce scénario n'est pas le flux normal et récurrent, nous avons accepté l’overhead qui en découle. L’avantage de ne pas submerger le service qui redémarre d’une panne est beaucoup plus pertinent.

Ce coupe-circuit est notre première ligne de défense pour rendre notre système résilient, soutenu par le rate limiter déjà fourni au niveau de nos load balancers pour interdire les tentatives malveillantes de déni de service.

Backpressure

Il existe cependant un scénario particulier, celui où le système subit une surcharge due à des périodes de pointe et non à des anomalies et perturbations liées à d'autres causes. Dans ce scénario, nous préférons exercer une pression sur l'appelant en lui renvoyant, dans notre SDK, un temps d'attente à respecter avant le prochain appel.

Ce temps d'attente est calculé à partir de l'état du service. Dans ce cas, notre couche coupe-circuit exploite l'état renvoyé par le rapporteur d'état, qui contient également le temps d'attente, pour sortir complètement de la boucle de réessai et remonter ce temps d'attente au client pour lui faire savoir que le service est occupé jusqu'à l'heure prévue.

Scénario MVP :

Lien vers la version SVG plus claire : mvp.svg

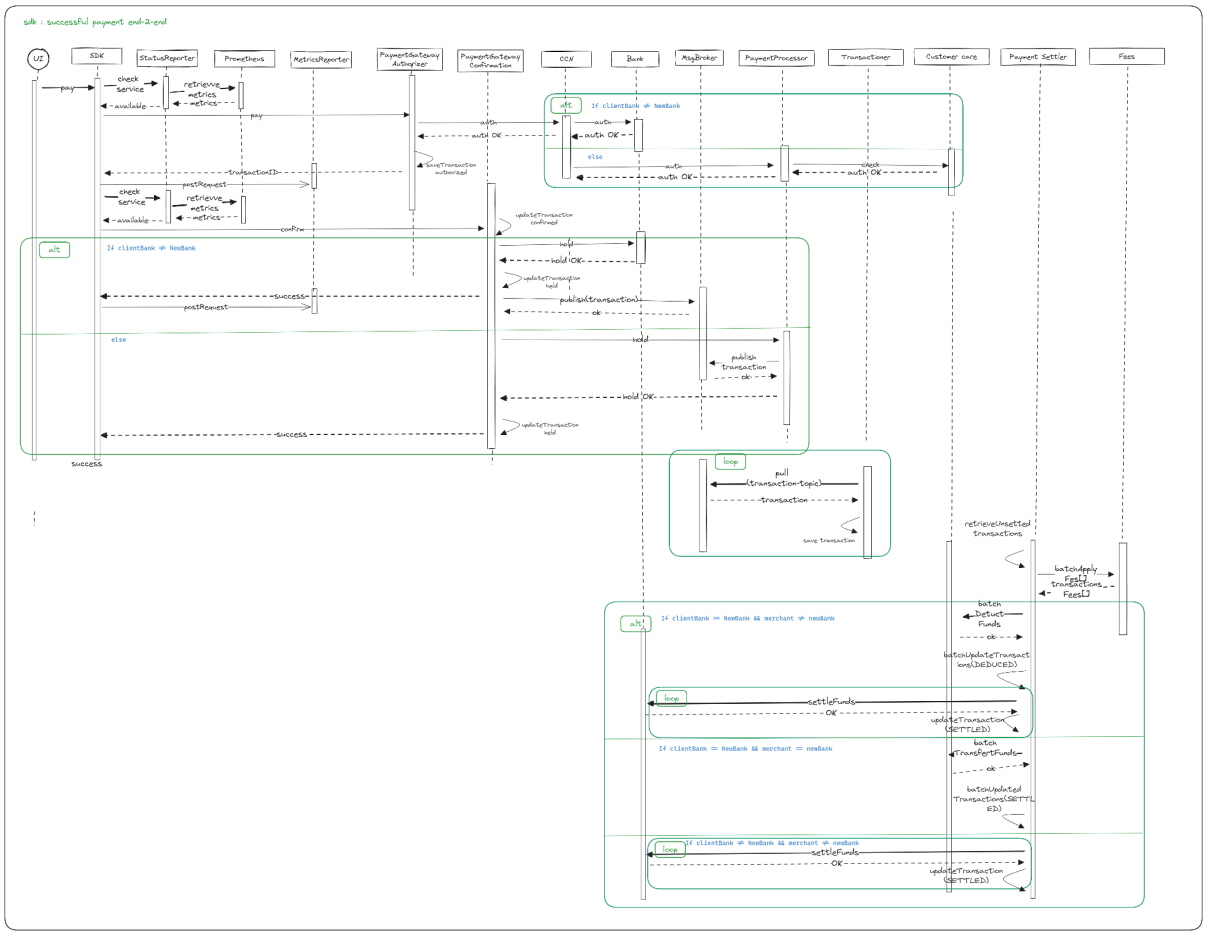


Figure 3 : Diagramme de séquence d'un paiement de bout-en-bout

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.

Étape de vérification des statuts des services :

Lorsque le client initie un paiement, le SDK envoie une requête au service des métriques pour vérifier la disponibilité du système, en récupérant les données à partir de Prometheus. En cas de défaillance du service, le SDK communique

immédiatement la réponse au client. En revanche, si le service est opérationnel, le processus de paiement est alors enclenché.

La première phase du paiement :

Le SDK envoie initialement une requête au Payment Gateway Authorizer, 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 afin d'autoriser ce paiement.

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 confirmé en seconde partie par le service Payment Gateway Confirmation, 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.

À chaque requête reçue par le SDK, qu'elle soit réussie ou non, celui-ci informe directement le Status Reporter pour enregistrer cette opération et son état, fournissant ainsi une information métrique utile pour le client.

La deuxième phase du paiement :

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 et faiblesse rapportées par la nouvelle architecture

Forces

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

- Les deux nouveaux services Status Report et Metrics Service ainsi que les services qui sont nés de la scission de PaymentGateway, Authorizer service et Confirmation service sont aussi mis derrière un load balancer en redondance active-active

- 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.

- 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 timeout 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.

- Nous avons implémenté 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 à travers le circuit breaker

établit va réagir en interrompant l'émission de nouvelles transactions, prévenant ainsi une surcharge additionnelle.

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 service de métriques, surtout avec un grand nombre de transactions. 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 et céder la responsabilité de transmettre ces métriques à nous au client utilisateur du SDK, responsabilité qu’on ne souhaite pas céder.

- Toujours sur le même point de surcharge, le Status Reporter et le Metrics Service étant interrogés trop souvent, ils constituent un nouveau point d'entrée critique, tout comme le Payment Gateway Authorizer Service et le Payment Gateway Confirmation Service. Il serait préférable de permettre une mise à l'échelle horizontale automatique pour ces composants critiques, ce qui impliquerait éventuellement de déployer notre système à l'aide d'une technologie d'orchestration à grande échelle.

- Avec la mise en œuvre du CDC basé sur les événements, nous avons fini par bombarder notre file d'attente Kafka avec beaucoup plus de messages, nos capacités de broker doivent s'adapter à la demande et peuvent atteindre une limite.

- La phase de confirmation est effectuée de manière synchrone, ce qui n'est pas la manière la plus efficace en termes de performances, car elle pourrait être effectuée de manière asynchrone, en publiant la confirmation directement dans la file d'attente kafka par le SDK. Mais cela signifierait également qu'il faudrait gérer les échecs de confirmation au niveau métier, par exemple en envoyant une notification aux clients.

Forces et faiblesse maintenue de l’ancienne architecture

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 Authorizer, Paymen Gateway Confirmer 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, les services Payment Gateway Authorizer et Payment Gateway Confirmation jouent 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 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 de chacun des services.

- Notre composant proxy inverse continue de jouer un rôle essentiel dans notre système, agissant comme gardien rate limiter et équilibreur de charge, en se servant de la redondance active-active de nos services de paiement : Payment Gateway Authorizer et Confirmer 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.

Faiblesses :

- L’utilisation d’un protocole d’encryptage asymétrique lourd avec une taille de 1024 bits contribue au problème de latence.

- 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.

Organisation et auto-évaluation du travail

La répartition du travail au sein de l'équipe a été collaborative . Chaque membre a participé activement à l'évolution du projet. Imene et Badr se sont principalement concentrés sur l’évolution du système NewBank, tandis que Nadim et Sourour ont travaillé sur l'amélioration du SDK. Une collaboration constante a été maintenue tout au long du processus, avec un travail conjoint sur les ajustements nécessaires.

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