POLYTECH
NICE-SOPHIA

Université Nice Sophia Antipolis

Réflexion et planning du projet V2

Ticket control

Groupe E

Robin Lambert, Rania Fekih, Anis Khalili, Benjamin Vouillon

POLYTECH NICE-SOPHIA

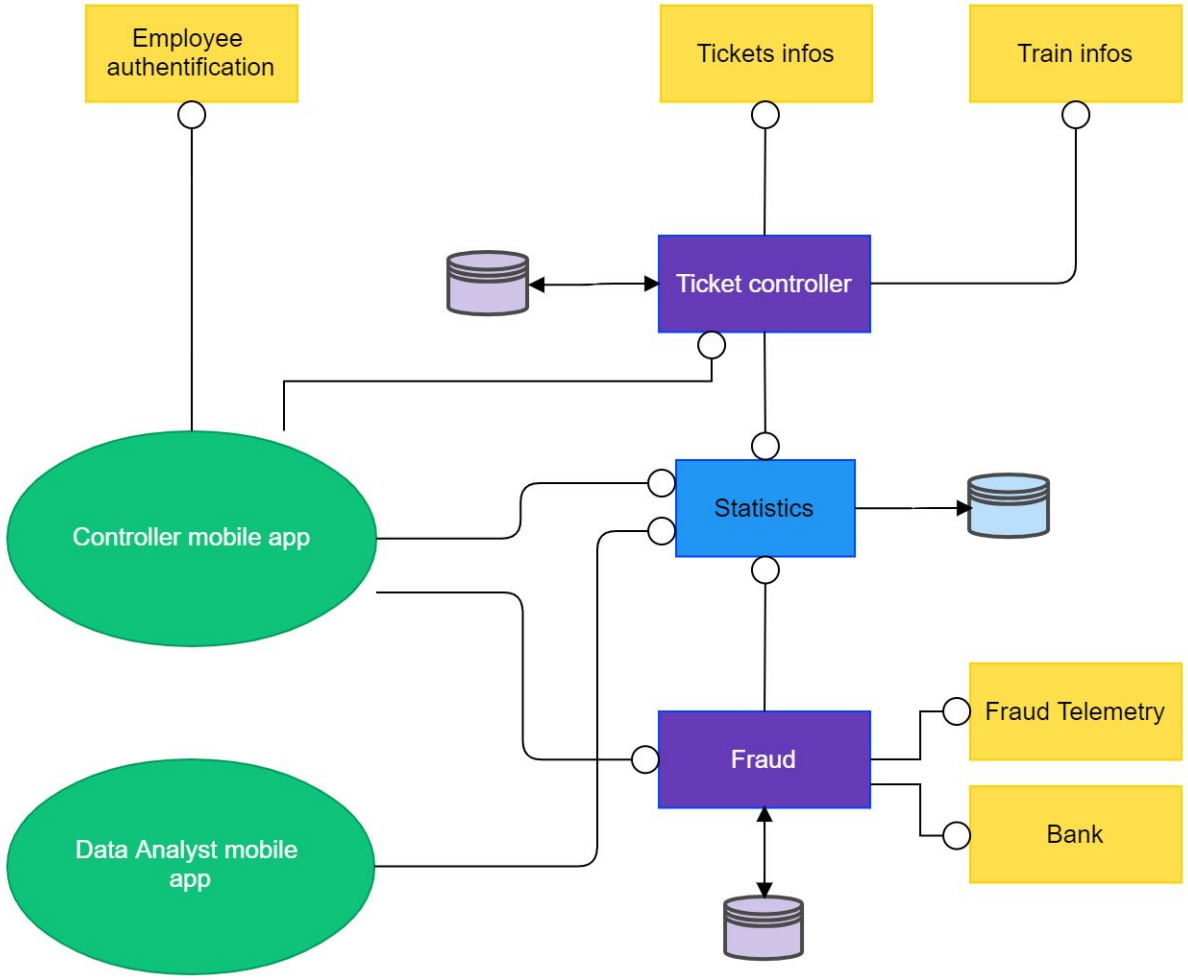
Université Nice Sophia Antipolis

SOMMAIRE

I. Etat Actuel de l'application :

A. Diagramme d'architecture

Notre architecture actuelle est une architecture hybride entre une architecture micro-service et une architecture orientée service. Nous allons présenter dans cette partie l'état actuel de notre application sans parti pris:



Sur ce schéma d'architecture, **les composants en violet** sont les composants qui vont être **physiquement présents dans les trains** (1 par train). Cela nous permet de nous assurer que ces services sont joignables partout dans le train depuis l'application mobile pour les contrôleurs (par réseau wifi interne avec des répéteurs pour chaque wagon, par exemple). **Statistics est interne à notre système** mais n'est **pas embarqué dans le train** : il est commun à tous les trains, et ce n'est pas critique s'il n'est pas disponible instantanément depuis l'application mobile.

Les composants en jaune sont les **services externes,** qui ne font pas partie du scope de notre projet et que nous allons implémenter pour simuler l'intéraction avec d'autres services indspensables.

B. Présentation des services :

- *Employee Authentification* : il permet au contrôleur de se connecter à son profil pour lui donner accès à l'application et aux statistiques de contrôle qui le concerne, et permet au module de statistique d'identifier qui effectue le contrôle.
- *Ticket infos* : renvoie le trajet, le nom du passager, le tarif réduit, la référence du train, ...
- *Train Infos :* renvoie le trajet effectué
- *Ticket Controller :* Récupère les tickets du trajet en cours et vérifie les données du ticket pour renvoyer à la fin si le ticket est valide ou non.
- *Statistics :* stocke et affiche les informations statistiques aux analystes
- *Fraud :* génère et affiche les réclamations de fraudes, renvoie les tarifs et communique avec le module paiement
- *Fraud Telemetry :* Stocke la liste des fraudes signalés par les contrôleurs pour finaliser les différentes procédures à faire après comme envoyer les amendes par courrier aux gens qui ont décidé de payer plus tard, consulter le montant total payé aux contrôleurs ...
- *Bank :* Un service externe de paiement, permettant de réaliser les opérations bancaires en ligne.

C. Communication entre services :

Au départ du chaque trajet **ticket controller** demande les informations la listes des tickets de ce trajet au composant **ticket infos** et le tableau de trajet à partir du composant **train infos.**

L'utilisateur de l'**app mobile** s'authentifie utilisant le service **employee authentification**.

L'utilisateur scanne le QR code, l'**app mobile** va envoyer les informations récupérées au **ticket controller**.

Le composant **ticket controller** va comparer les données de tickets avec les données stockées dans sa base .

Le **ticket controller** va retourner à la **mobile app** si le ticket est valide, si c'est un tarif réduit à vérifier ou bien c'est si c'est une fraude (faux ticket, pas le bon train, trajet payé trop court si la personne est restée dans le train après la station de destination de son ticket).

En cas de fraude la **mobile app** va communiquer avec le composant **fraud** pour remplir les données nécessaires et retourne à la fin le montant à payer.

Le composant **fraud** va renvoyer au **bank** le montant récupéré et les données du paiement pour finaliser la procédure.

À la fin de chaque trajet les informations des tickets contrôlés et les fraudes signalées seront envoyées vers **le composant statistics et fraud telemetry** pour exécuter les différents traitements des analystes.

Le composant **fraud** va également envoyer ses informations vers **fraud telemetry** pour pouvoir envoyer les courriers des amendes pour les personnes qui ont choisi de payer plus tard.

II. Les forces

A. La Résilience & Disaster Recovery

Les conditions d'utilisations de notre applications sont assez complexes, une partie du projet se déroule au sein de l'infrastructure de serveurs fixes existant (stockage des données long termes, traitement des données de fraude,...) et au sein d'un environnement mobile qui est le train (scan des tickets, création d'une fraude). Ce train traverse des zones non couvertes par le réseau internet mobile, mais il faut que malgré cela l'application fonctionne, que l'on sache si le billet est valide et stocker les tickets de fraude. Il faut donc une résilience certaine.

Dans un premier temps, nous avons rendu le système résilient en implémentant un serveur dans le train qui vient répliquer la liste des billets valides et des fraudes. Le système est donc disponible en permanence pour le contrôleur, qu'il y ait du réseau ou non.

Dans un second temps nous avons enlevé tous les risques de Single Point of Failure (SPOF). Pour ce faire, le backend est morcelé en services métiers indépendants.

B. Le Contrat de Service

Pour assurer une efficacité du système importante nous avons fait en sorte que chaque service expose ses ressources. Donc notre choix s'est porté sur REST qui est parfaitement adapté à notre besoin, car il est orienté ressources.

C. Scalabilité

Dans notre explication scope de projet nous avons parlé de 300 tickets testées en moyenne par contrôleur chaque jour pour 10 000 contrôleurs, soit plusieurs millions de requêtes par jour. De plus, le système n'est pas sollicité de manière homogène les 24h de la journée, Il faut donc pouvoir adapter la quantité de puissance de nos serveurs en fonction du flux de traitement.

Nous avons une architecture hybride avec des services orientés métiers disposant de leurs propres bases de données. De plus, nous utilisons des conteneurs pour nos services ce qui rend l'architecture facilement scalable à l'avenir.

D. Choix de Technologie

1. Le Front-end

Après une recherche, nous avons trouvé que la plupart des applications dédiées à ce genre d'utilisation sont implémentées avec Android vu que sur le marché, il existe une large gamme de périphériques matériels alimentés par le système d'exploitation Android, y compris de nombreux téléphones et tablettes. Même le développement d'applications mobiles Android peut se produire sous Windows, Mac OS ou Linux.

De plus, Android est disponible sur différents appareils, où plusieurs entreprises sont en concurrence sur le marché des téléphones Android, ce qui permettra à l'entreprise de renouveler son parc de machines pour moins chères, avec des appareils mieux adaptés à son besoin et d'adapter ses produits à ses contrôleurs. Par exemple, un contrôleur avec un handicap visuel disposera d'un téléphone avec un écran plus grand.

2. Le Back-end

Parmi les différents frameworks qui sont proposés, on a bien choisi NodeJs .
voici 2 bonnes raisons d'utiliser NodeJS dans notre projet :

a) Plusieurs services

Notre architecture est basée sur plusieurs services séparés qui communiquent entre eux avec des appels REST . Dans ce cas là, on doit avoir un framework puissant côté temps de réponse pour les requêtes REST. Donc , on a trouvé NodeJs est un bon choix côté rapidité des requêtes REST.

b) Qualité de service

Comme notre application est conçue pour être utilisée par plusieurs contrôleurs au même temps sur différente ligne de train, on doit donc garantir que les serveurs (des services) restent fonctionnels.

c) Conclusion

L'utilisation de Node.js en tant que serveur web permet de traiter un gros volume de requêtes simultanément de manière efficace et asynchrone permettant d'éviter les attentes .

3. La Base de données

L'architecture évolutive horizontale de MongoDB peut prendre en charge d'énormes volumes de données et de trafic. C'est exactement ce dont on a besoin dans notre application . On aura un grand flux de tickets à contrôler dans SNCF il y a 10 000 contrôleurs et chacun va contrôler minimum 300 tickets par jour donc là on parle d'un énorme flux de données.

III. Faiblesses

Toutes les architectures ont des points critiques ou des faiblesses : il est important de les identifier pour les faire évoluer, ou de trouver un compromis adapté aux besoins.

A. Couplage

Malgré le fait de répartir les différents besoins métier de notre application sur des services séparés, il reste des services qui sont fortement couplés entre eux.
Pour le stockage des statistiques , le service **statistics** utilise une seule base de données pour stocker les statistiques des tickets et les statistiques des fraudes. L'analyse de ces données est aussi gérée par le même service.
Dans le cas ou l'un des services **fraud** ou **ticket controller** ne renvoie pas les données à la fin du trajet, l'analyse des statistiques va être incohérente car il se base sur les 2 types de données (une en état de mise à jour et l'autre non!).
Le service **statistique** dépend donc fortement des deux services **fraud** et **ticket controller** pour assurer son bon fonctionnement.
=> couplage fort

B. Latence

Le service **ticket controller** est lié fortement avec le service externe **train infos** qui lui donne les informations sur les arrêts et les positions de train. Cela crée une dépendance entre ces deux services. On peut aussi avoir un problème de temps de réponse étant donné qu'il n'y a qu'un seul service **train infos** partagé par tous les trains.

Donc il peut y avoir une surcharge de serveur. Une solution serait de faire un système de cache qui stocke les données entre 2 arrêts pour éviter d'appeler **train infos** trop régulièrement si on veut avoir des données cohérentes. Actuellement on contourne le problème en n'appelant **train infos** qu'au début de chaque trajet, ce qui pose un problème de cohérence. On peut aussi réduire les échanges entre ces deux services, en ne faisant qu'une seule requête entre 2 arrêts pour tous les billets qui vont être contrôlés. Mais on ne fait que déplacer le problème de cohérence, et la charge sur **train infos** reste importante.

C. Fiabilité

À ce stade, notre architecture nous limite à un certain nombre de billets de train et ne prend pas en considération les billets qui sont pris lorsque le train est en marche .
Comme nous l'avons vu ci-dessus, le service **ticket controller** stocke tous les billets qui sont déjà achetés par les voyageurs avant la sortie du train. Ensuite, ce service ne dialogue plus avec **ticket controller.**
Les tickets achetés après le départ du train ne sont donc pas pris en compte. Alors notre choix de minimiser l'appel à un service externe (pour éviter les risques de coupure brusque de réseau qui est très fréquente dans les trains) nous a posé ici un **problème de fiabilité** car l'application ne peut pas vérifier un certain nombre de nouveaux tickets.

D. Résilience

Dans notre architecture le seul point d'entrée à notre application que ce soit pour le contrôleur ou le responsable des statistiques c'est le service **employee authentification** .

Donc même si ce n'est pas un service important dans le déroulement des différents scénarios de l'application, il reste un point de faiblesse bien marqué car si le service d'authentification ou la base de données reliée à ce service tombe on risque d'avoir un blocage des utilisation de l'application. Donc on doit garantir que ce service fonctionne parfaitement.

E. Contrat de service

Dans notre architecture on a utilisé les contrats faibles **(communication REST entre tous services)**. En revanche, nous avons des services qui demandent plus de sécurité et qui déclenchent une action (pas d'appel de ressource CRUD).
Par exemple, entre le service de **bank** et de **fraud**, on fait une requête Rest (GET) où on passe les données des cartes bancaires des utilisateurs. Cette requête doit être sécurisée avec un modèle de données plus strict: on doit utiliser un contrat fort RPC.
Donc on doit vérifier les choix des contrats de services qui peuvent être un point de **faiblesse côté sécurité** pour notre application.

IV. Planning du 2ème bimestre

A. Description des nouveaux besoins:

Comme précisé dans la 2ème partie nous avons pas mal de points à améliorer dans l'application. Dans les deux premières parties nous avons plutôt présenté la version technique des choses, mais on ne peut pas négliger qu'il y a pas mal de choses à améliorer aussi par rapport aux fonctionnalités. Donc on va séparer les points à faire dans le 2ème bimestre en deux catégories besoins fonctionnels et non fonctionnels.

1. Besoin fonctionnels :

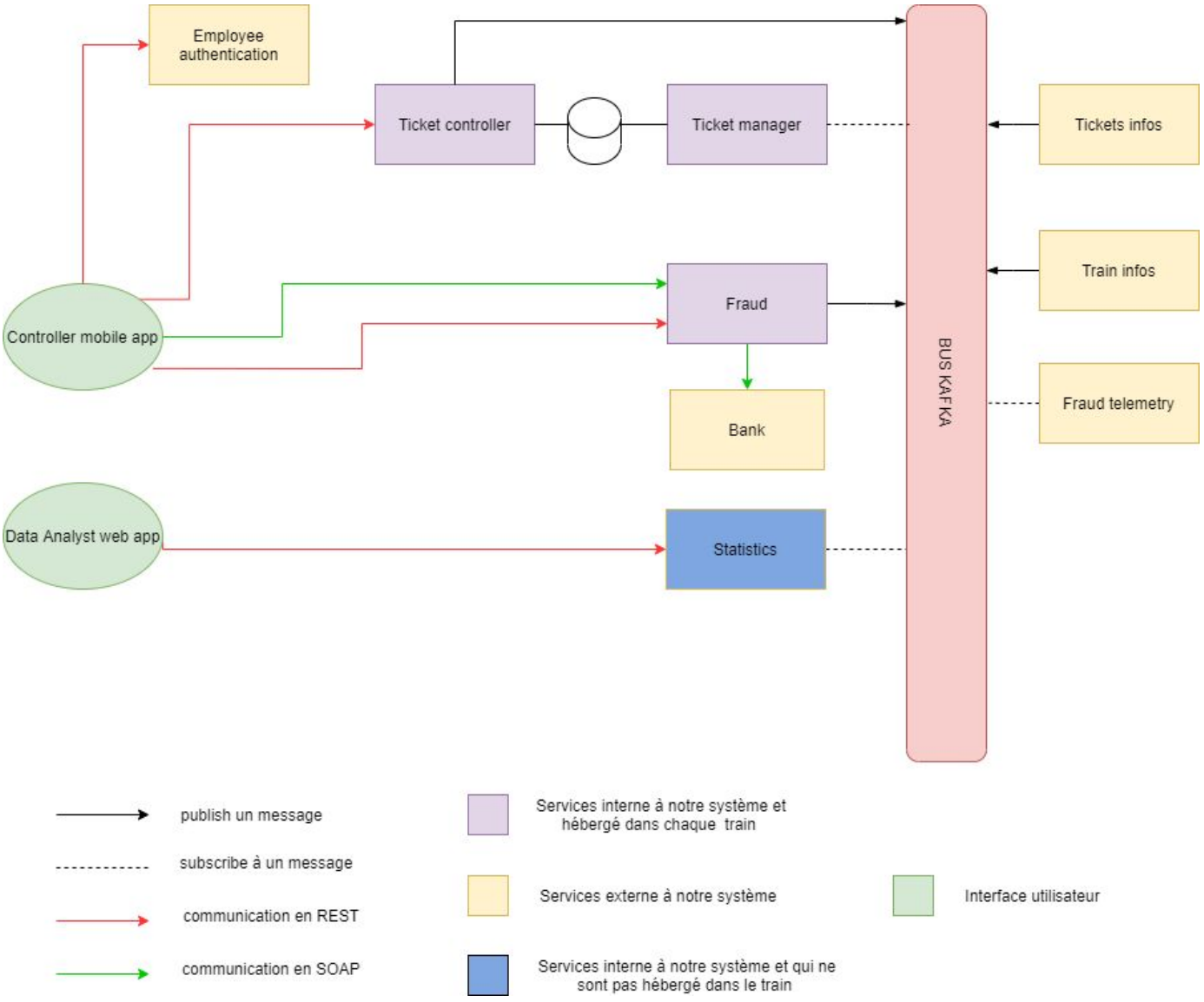
- Ajouter une interface administrateur pour consulter les statistiques et visualiser l'état général du travail.
- Développer davantage la partie statistiques en ajoutant des statistiques plus avancées.
- Prendre en considération les tickets achetés après le démarrage du train. Étant donné que notre application ne demande les tickets achetés qu'au moment du démarrage du train.

2. Besoins non fonctionnels :

- Scalabilité
- Resilience
- Tolérance à la panne et haute disponibilité

Cette partie présente brièvement nos objectifs. La justification des choix et la façon de réalisation seront expliqués dans les prochaines parties.

B. Diagramme d'architecture



Dans un premier temps, nous séparons le service **Ticket controller** en deux : **Ticket controller** et **Ticket manager. Ticket controller** gère la partie métier de contrôle du ticket, tandis que **Ticket manager** s'occupe de récupérer les tickets et de mettre à jour la base de données. Ces deux micro-services partagent la même base pour des raisons de simplicité, comme ils sont tous les deux physiquement présents dans le train. L'objectif de cette séparation est de séparer la logique métier, et de permettre au contrôleur de continuer d'utiliser l'application même si la récupération des tickets par le **Ticket Manager** ne fonctionne plus.

Dans notre nouvelle architecture, nous souhaitons utiliser un bus Kafka entre les micro-services de notre système. Certains écrivent dans le bus d'événements (**Ticket Controller**, **Fraud**, **Ticket infos**, **Train infos**) et d'autres consomment les événements (**Fraud telemetry**, **Statistics**, **Ticket Manager**). Le **Ticket Manager** récupère donc les tickets en temps réel, dès qu'un passager achète un ticket.

Nous souhaitons également utiliser un contrat fort entre l'application mobile et le service de fraude, et entre le service de fraude et le service externe de la banque.

C. Justification des choix

Étant donné que nous avons choisi d'implémenter une architecture micro-service, nous avons remarqué que le service contrôle ticket avait plusieurs fonctionnalités. Donc nous avons décidé de le diviser en deux micro-services: un qui est responsable de la récupération de flux de ticket et de l'enregistrement des tickets dans la BD interne au train et un autre qui est responsable du contrôle des tickets (récupération des infos de ticket, comparer les données et décider si le ticket est valide ou non). Ce pattern est appelé CQRS, pour Command Query Responsability Segregation.
Comme précisé précédemment nous avons un couplage fort dans notre système qui se présente dans plusieurs scénarios.
L'un des problèmes majeurs de notre système à l'heure actuelle est qu'il ne permet pas aux utilisateurs ayant acheté un ticket après le départ du train d'être contrôlé en règle. En effet dans l'architecture précédente, le service **Ticket Controller** qui se trouvait physiquement dans tous les trains ne récupérait les tickets d'un trajet qu'au départ de ce trajet. Nous pourrions lui demander de faire cette requête plus régulièrement, mais on perdrait en efficacité et en performances : faire cette requête pour tous les trains au moins à chaque arrêt ajouterait un nombre considérable de requêtes parfois inutiles si personne n'achète de tickets après le départ du train.
L'un des moyens de régler ce problème est d'utiliser un bus d'évènements (Kafka) qui permet à un service ou micro-service de consommer les événements d'un canal (channel).
Le micro-service de **ticket manager** va pouvoir s'abonner au canal du trajet en question, et à chaque fois qu'un ticket est acheté, il va pouvoir consommer l'évènement (système publish / subscribe). Nous n'avons donc plus un couplage fort entre le système interne au train et le système externe et nous aurons en plus une cohérence de données entre **la BD du système interne** au train et la BD de **ticket infos**, ce qui n'était pas assurée par notre architecture précédente.

En effet, si l'on avait utilisé une route REST comme un POST depuis le services de **ticket infos** vers le service de **ticket controller** interne au train, le service de **ticket infos** serait complètement dépendant des pannes de réseau (assez fréquentes pour un train qui se déplace partout en France, parfois à grande vitesse...).
Kafka nous garantit donc le Fault Tolerance et le High availability. En effet, il peut rejouer les événements qui sont stockés dans une file, pour pallier des problèmes de pertes de réseaux, de messages non consumés, etc. (Fault tolerance). Cela permet également d'avoir quasiment en temps réel la liste des tickets valides pour un trajet, et le service **ticket manager** n'a pas à attendre que le train ait de nouveau du réseau pour l'informer de l'achat d'un nouveau ticket (High availability).

Ce principe fonctionne pour tous les autres services qui étaient en couplage fort dans l'ancienne architecture. Nous avons donc un système plus scalable avec un couplage faible.

Comme vu plus haut, cette architecture permet de relier entre eux les micro-services en diminuant le couplage. Cela permet aussi de s'assurer de l'aspect asynchrone et idempotent des opérations, et l'utilisation d'évènements force notre système à parler métier plus que « code » ou « ressource ». Cela fait partie de bonnes pratiques à utiliser pour tirer

parti au mieux d'une architecture microservice (Event Driven Design).

Nous avons également choisi d'utiliser un contrat fort pour le service de **fraud** et le service externe de la banque : **bank**. Cela permet une couche supplémentaire de vérification du modèle de données envoyé, indispensable pour effectuer des actions aussi importantes que le paiement. L'interface est donc orientée "action" plus que ressource, ce qui est aussi plus adapté sémantiquement.

D. Risques envisagés :

Comme vous le disiez tout le temps, l'architecture micro-service n'est pas une baguette magique. Nous sommes conscients que cela peut nous poser des problèmes d'intégration ou même de monitoring vu que le tracking des erreurs peut être plus compliqué. C'est pour cela que nous envisageons de mettre plus d'efforts dans la partie devops pour implémenter plus de tests et automatiser les pipelines qui peuvent nous aider à éviter ces problèmes.

V. Si j'avais su …

A. Backend

Si nous avions su que nous allions potentiellement implémenter un bus d'évènement, nous l'aurions fait dès le début. Le bus d'événement est relié à plusieurs services dans notre application, et donc l'intégration de kafka à ce stade peut prendre un peu de temps : il faut changer tous les services + implémenter les tests nécessaires et configurer les pipelines d'intégration et de déploiement. En revanche, le passage à l'échelle pourra être beaucoup plus facile.

B. Front-End

Si nous avions su pour le front-end que la feature de Scan de Ticket prendrait autant de temps et serait si complexe à mettre en place, nous n'aurions pas cherché à l'implémenter et nous nous serions concentrés sur la mise en place de tests UI/UX et de tests d'intégration. D'autant plus que cette fonctionnalité n'est absolument pas essentielle à notre MVP.