Scope du projet - sujet V2

Groupe E :

- Rania Fekih
- Anis Khalili
- Benjamin Vouillon
- Robin Lambert

Bimestre 2

A. Description des nouveaux besoins:

1. Besoin fonctionnels :

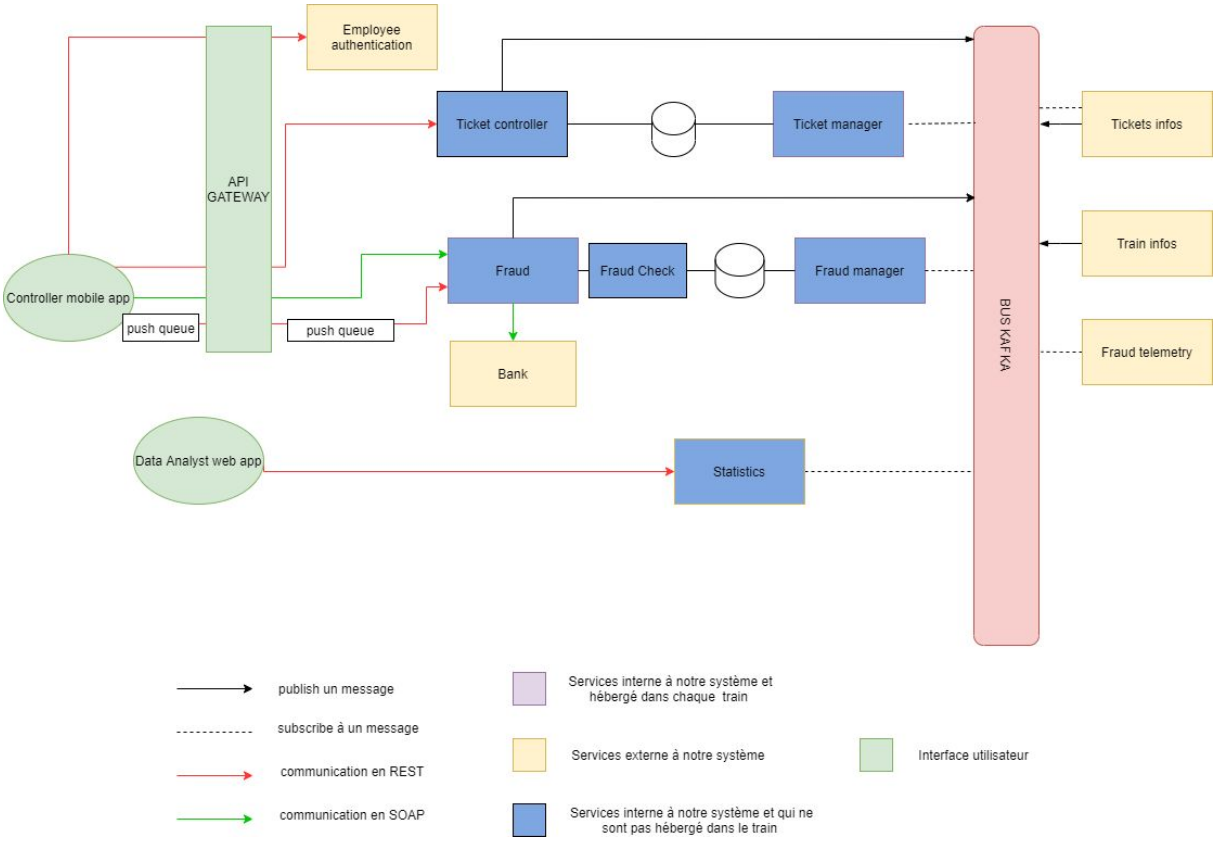
- Ajouter une interface administrateur pour consulter les statistiques et visualiser l'état général du travail.
- Développer davantage la partie statistiques en ajoutant des statistiques plus avancées.
- Prendre en considération les tickets achetés après le démarrage du train, étant donné que notre application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

1. Services externes -> contrainte

Dans cette nouvelle partie du projet une contrainte nouvelle apparaît, lors de changement de changement les trains peuvent séparer leurs wagons, fusionner des wagons, voir fusionner des trains:

2 trains sont accrochés pour n'en former plus que 1

Il se peut donc qu'il y ait plusieurs serveurs dans un train et que les wagons d'un même train aient des directions différentes. Comment faire alors pour synchroniser les informations informations entre les tickets testés avec les serveurs (ceux du train et ceux de là distances) ?

Dans un premier temps nous avons décidé de supprimer les serveurs du train. pour résoudre ce problème. Nous l'avons remplacé par une message Queue dans le téléphone qui attend la connexion au réseau et une gateway côté serveur pour gérer la charge des appels, et récupérer les tickets achetés entre temps (publish / subscribe).
Le problème est que la liste des tickets n'est pas à jour côté application du contrôleur, si il n'a pas de réseau et qu'un ticket a été acheté. Dans ce cas, le contrôleur peut déclarer une fraude, et le service **FraudCheck** pourra annuler la fraude si le billet a effectivement été acheté quand le contrôleur n'avait plus de réseau.

2. API Gateway

En hébergeant les services en dehors du train on se retrouve dans un nouveau problème c'est que les app des contrôleurs doivent connaître précisément tous les services pour pouvoir appeler ceux qui s'occupent du trajet demandé. Cela augmente potentiellement la latence réseau en obligeant la multiplication des aller-retours entre l'application cliente et le backend. Pour assurer une communication fluide entre les clients et notre backend + une latence raisonnable avec les services internes nous avons décidé de mettre en place un API Gateway qui sera notre unique point d'entrée pour le système.

Vu que le nombre de trajets qu'on a varie d'une saison à une autre, on compte implémenter un load balancer avec qui va nous fournir la scalabilité automatique des services tout dépend des besoins de notre système.

3. Push queues / Cache

Notre nouvelle contrainte architecturale ne doit pas rendre notre système inutilisable pour les contrôleurs en cas d'absence de réseau. L'objectif est qu'ils puissent continuer à contrôler des tickets indépendamment de l'endroit dans lequel le train se trouve.
Pour continuer à répondre à ce besoin, nous avons choisi d'utiliser un cache côté application mobile du contrôleur, qui garde en mémoire les tickets d'un trajet. Le cache est actualisé dès qu'un nouveau ticket est acheté pour le trajet en question et dès qu'il y a du réseau, pour ne pas perdre la cohérence des données trop longtemps. Nous priorisons la disponibilité de notre système et veillons à garder une cohérence éventuelle.

Enfin, nous souhaitons utiliser des push queues pour permettre au contrôleur de déclarer une fraude indépendamment du réseau. Les push queues permettent de stocker des messages dans une file d'attente, qui seront envoyés dès que le destinataire est disponible (ici l'API Gateway). Notre application est ainsi résistante aux pannes de réseau.

Évidemment, cela nous oblige à faire certains compromis : un cache important côté front, l'impossibilité pour le passager de payer l'amende d'une fraude si le réseau n'est pas disponible, les tickets achetés récemment ne sont pas pris en compte si l'application du contrôleur n'a pas trouvé de réseau, etc.

4. Ticket manager / fraud manager ET Bus kafka

Étant donné que nous avons choisi d'implémenter une architecture micro-service, nous avons remarqué que le service **Ticket Controller** 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 intermédiaire (**Ticket manager**) 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 Responsibility Segregation.

Pour la même raison, on a séparé le service Fraud en 2 micro-services : **Fraud** et **Fraud manager.**

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** récupère tous les tickets qui sont achetés avant le départ du train et ne prend pas en considération les nouveaux tickets.

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 **ticketManager** 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). Dans ce cas la on garantie la cohérence de données entre la BD et les données utilisées par le le service **Ticket Contrôle.**

Kafka nous garantit aussi 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).

5. Contrat fort

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.

On a une perte de cohérence des données de ticket du point de vue de l'application du contrôleur étant donné que les services ne sont plus dans le train. En revanche nous avons un service FraudCheck qui permet de s'assurer qu'on ne contrôle pas plusieurs fois le même utilisateur et qu'un passager qui a acheté son ticket récemment alors que la liste de ticket n'est pas à jour côté application du contrôleur ne sera pas sanctionné, même si le contrôleur crée une fraude.

Bimestre 1

Les utilisateurs

Avant de définir le scope de ce projet, il convient d'identifier les utilisateurs. Voici ceux qui vont utiliser notre solution :

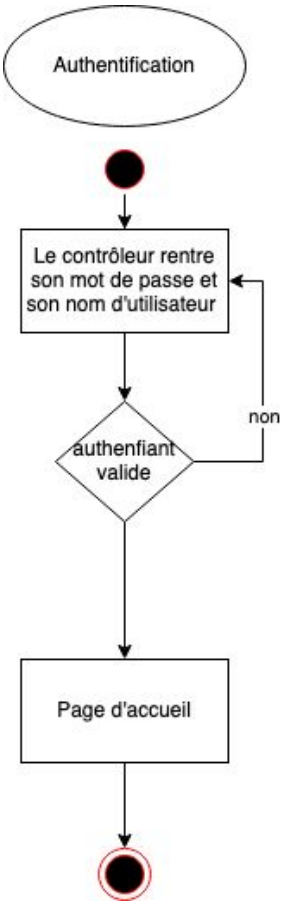
- **Le contrôleur** : Celui qui contrôle les billets dans le train et applique les sanctions aux fraudeurs.
- **L'analyste** : Celui qui consulte et analyse les données collectées par les contrôleurs à travers un dashboard.

Description du projet :

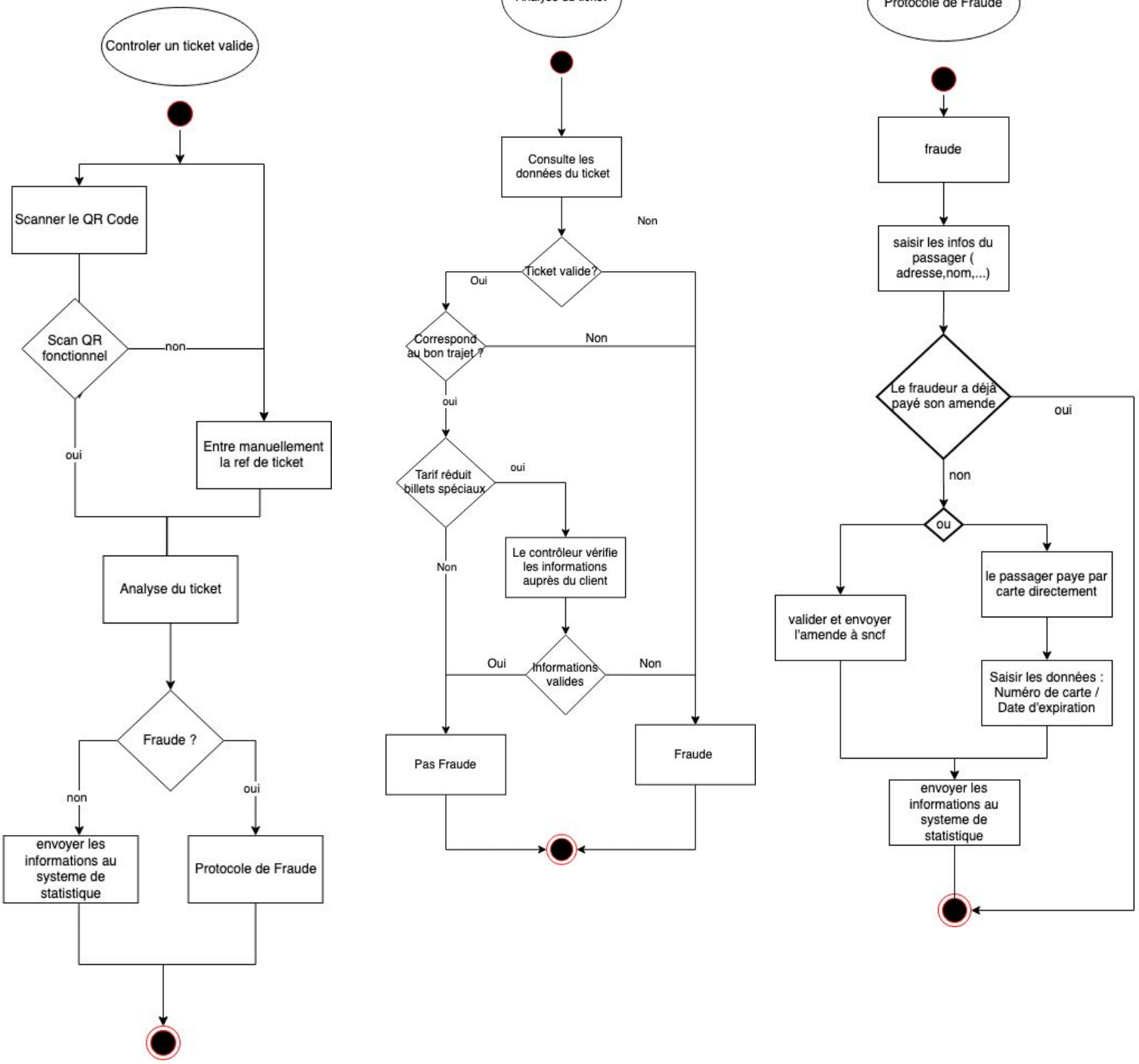
Dans un premier temps le projet du sujet V2 est de réaliser une solution permettant de contrôler le billet d'un passager dans un train (Ce billet est en fait un QR Code et un numéro qui a été remis au client à l'achat du billet.) et dans le cas où le client ne possède pas de billet valide alors il faut grâce au logiciel remonter une fraude au système. Pour cela voilà les différentes fonctionnalités de ce logiciel :

- Il faut pouvoir contrôler les billets en scannant le QR code, mais si le QRCode n'arrive pas à être lu le contrôleur doit pouvoir rentrer le numéro client à la main.
- Le logiciel permet d'ouvrir un dossier de fraude dans le cas où l'utilisateur le client n'a pas de billet ou son billet n'est pas valide.
- Le logiciel doit renvoyer les données de billet en cas de tarif réduit pour que le contrôleur s'assure que la réduction appliquée sur le ticket est bien applicable.
- Ce logiciel doit prendre en charge le paiement par carte bancaire et les paiements en espèces. Dans le cas où le client ne peut ou ne veut pas payer son amende directement, il doit y avoir un système de demande de paiement qui s'enclenche.
- Les informations de validité des billets et le nombre de fraudes doivent êtres stockées sur un serveur pour qu'il puisse les analyser.
- Le fait de vérifier le ticket se fait depuis un module qui appelle des services externes. Il appelle un service du projet v1 "booking" qui renvoie les informations qui correspondent au ticket (tarif réduit ou non, nom du passager, trajet, ...) et un service qui donne des informations sur le train (position en direct, trajet effectué, ...). Si le billet est un faux, ou bien que le trajet n'est pas le bon, le contrôleur est informé et peut lancer le protocole de fraude.

- Le protocole de fraude est géré par un service à part qui se connecte à un module externe de paiement et à un autre qui permet de tarifer les fraudes (dépend de l'infraction : tarif de réduction non applicable, billet pour une portion du trajet seulement, pas le bon nom de voyageur, pas de billet, etc.)



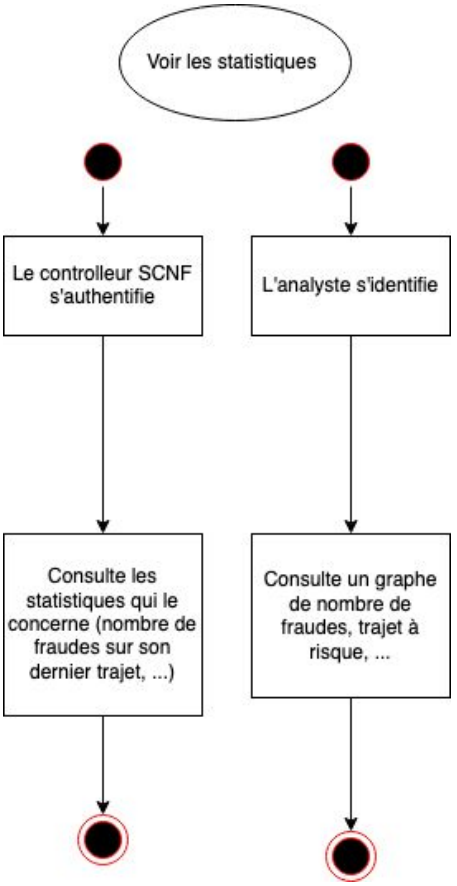
Scénario d'authentification du contrôleur



Scénarios de contrôle du ticket, de l'analyse du ticket et de l'application de la sanction pour les fraudeurs.

Dans un second temps, une autre solution doit-être mise à disposition des analystes de la compagnie de train et de permettre aux contrôleurs de consulter leurs statistiques de travail. Voilà les fonctions à réaliser:

- Le contrôleur a la possibilité de consulter différentes statistiques réalisées à partir de ses propres contrôles :
  - le nombre de voyageurs contrôlés
  - le nombre d'amendes rédigées
  - les nombres de trajets effectués
- L'analyste a accès à une api / dashboard pour voir toutes les statistiques des trajets :
  - le nombre des trains contrôlés
  - le nombre de fraudes par train
  - les trajets et les horaires où il y a un nombre important de fraudes



Scénarios liés aux statistiques, pour le contrôleur et l'analyste.

Ordre de réalisation

L'objectif pour la POC de début Novembre est de se focaliser sur le scénario le plus important de l'application : le contrôle de billets par le contrôleur. Il pourra également signaler une fraude (gestion des différents cas de fraude décrits dans les scénarios). Le module de statistiques pourra être utilisé par l'analyste : en revanche, il n'affiche que les informations qui sont stockées, sans calcul ni analyse de données.

Un scénario qui n'est pas prioritaire est la possibilité pour le contrôleur de voir les statistiques qui le concernent.

Diagramme de composants

L'avantage en terme d'architecture de séparer la gestion des fraudes du contrôle du ticket est de découpler ces deux actions métiers : le contrôleur peut sanctionner quelqu'un qui n'a pas son billet, et si l'un des deux services ne répond pas, ce n'est pas bloquant pour l'application (pas de single point of failure). Le module de statistique peut être partagé entre les deux applications, mais ne renverra pas les mêmes informations.

- *Authentification employé* : 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.
- *Informations du ticket* : renvoie le trajet, le nom du passager, le tarif réduit, la référence du train, ...
- *Informations du train :* renvoie le trajet effectué
- *Contrôle du ticket :* vérifie les données du ticket pour renvoyer à la fin si le ticket est valide ou non.
- *Statistiques :* stocke et affiche les informations statistiques aux analystes
- *Fraude :* génère et affiche les réclamations de fraudes, renvoie les tarifs et communique avec le module paiement
- *Fraude Telemetry :* Stocke la liste des frauds signalés par les contrôleur pour finaliser les différents procédures à faire après (envoyer les amendes par courrier aux gens qui ont décidé de payer plus tard, consulter le montant total payé aux contrôleurs ..)
- *Banque :* Un service externe de banque

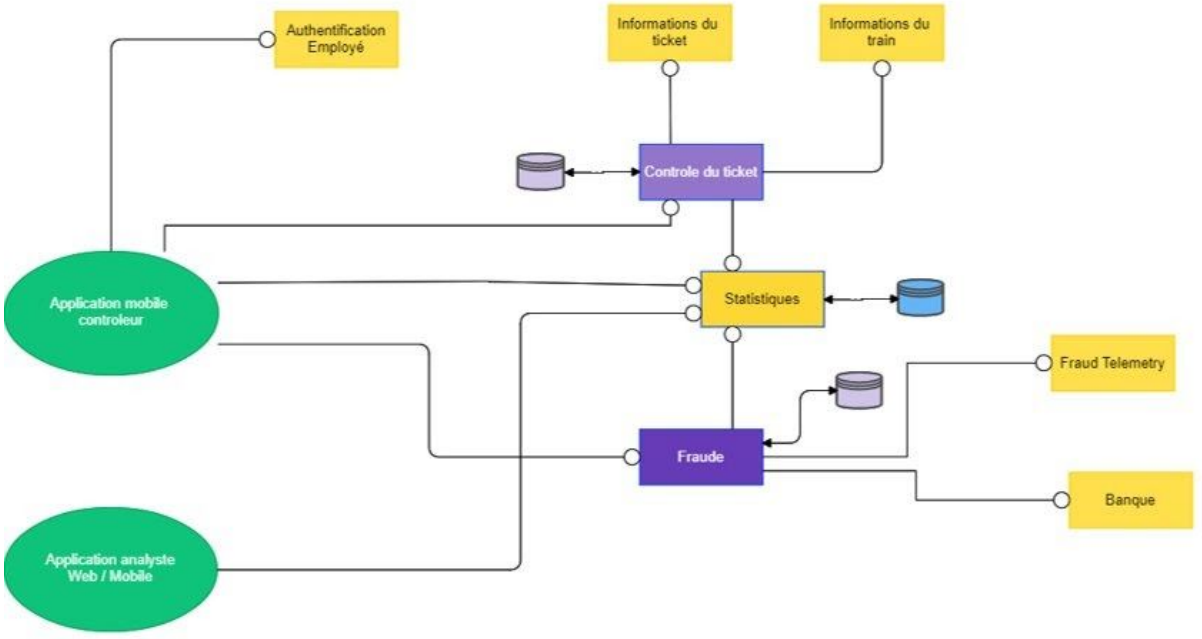
Au départ du chaque trajet **Contrôle ticket** demande les informations des tickets qui correspondent à ce trajet du composant **informations ticket** et le tableau de trajet à partir du composant **informations train**
L'utilisateur de l'**app mobile** s'authentifie en communiquant avec **Authentification employé**. L'utilisateur scanne le QR code, l'**app mobile** va envoyer les informations récupérées au **control ticket**.

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

Le **control ticket** va retourner à l'**app mobile** 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 l'**application mobile** va communiquer avec le composant **fraude** pour remplir les données nécessaires et retourne à la fin le montant à payer.
Le composant Fraude va renvoyer à la **banque** le montant récupéré et les données du paiement pour finaliser la procédure.

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

Le composant **Fraude** 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

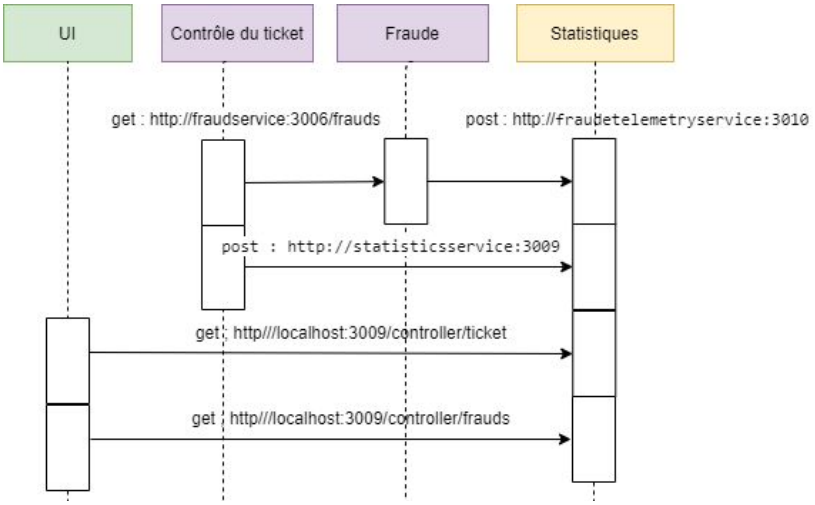


Sur ce schéma d'architecture, **les composants en violet** sont les composants qui vont être **physiquement présents dans les trains** (1 par train). Cela nous permet de nous assurer que ces services sont joignables partout dans le train depuis l'application mobile pour les contrôleurs (par réseau wifi interne avec des répéteurs pour chaque wagon, par exemple). **Le service de statistique 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 si il n'est pas disponible instantanément depuis l'application mobile.

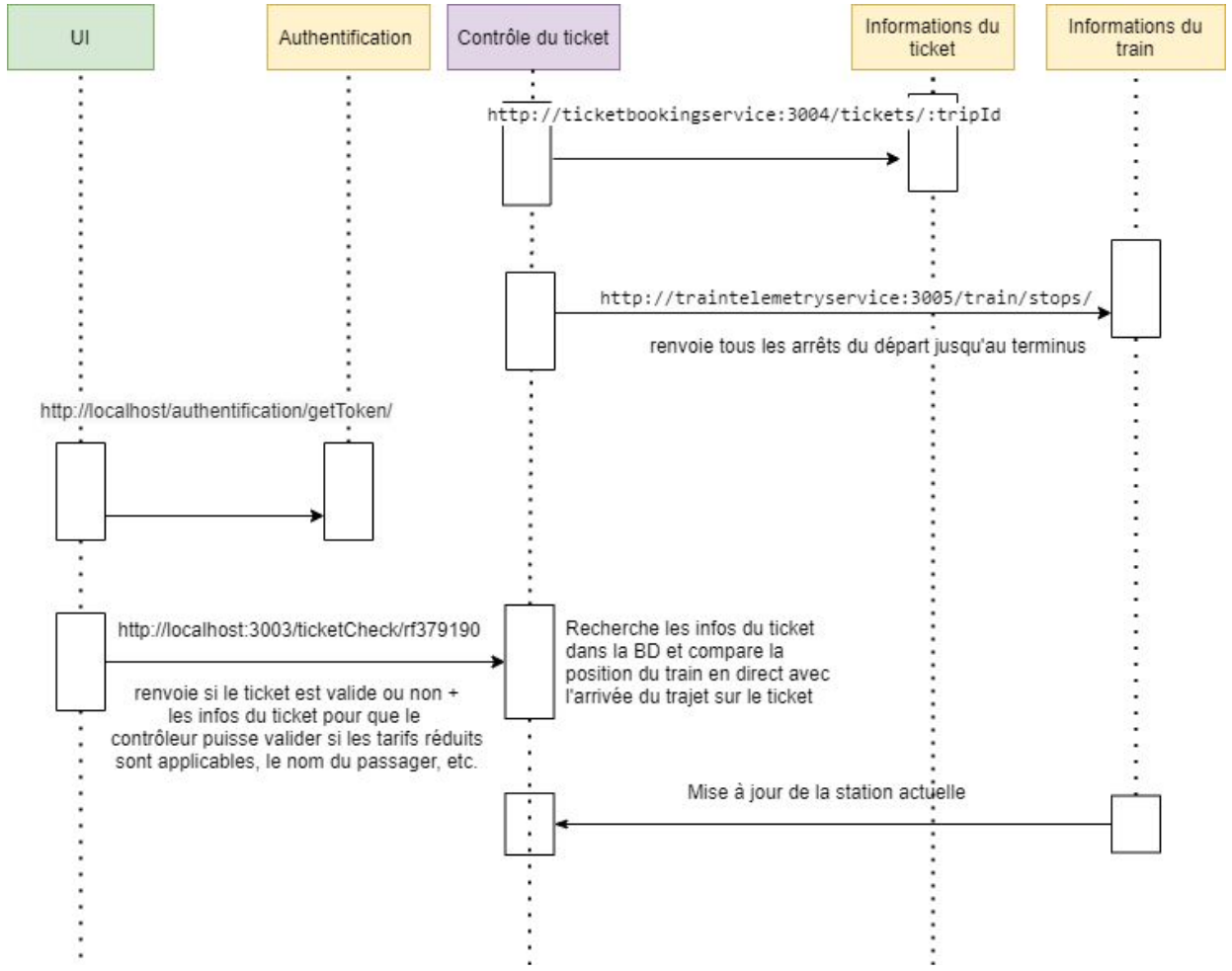
Les composants en jaune sont les **services externes,** qui ne font pas partie du scope de notre projet et que nous allons implémenter pour simuler l'intéraction avec d'autres services indspensables.

Diagrammes de séquence :

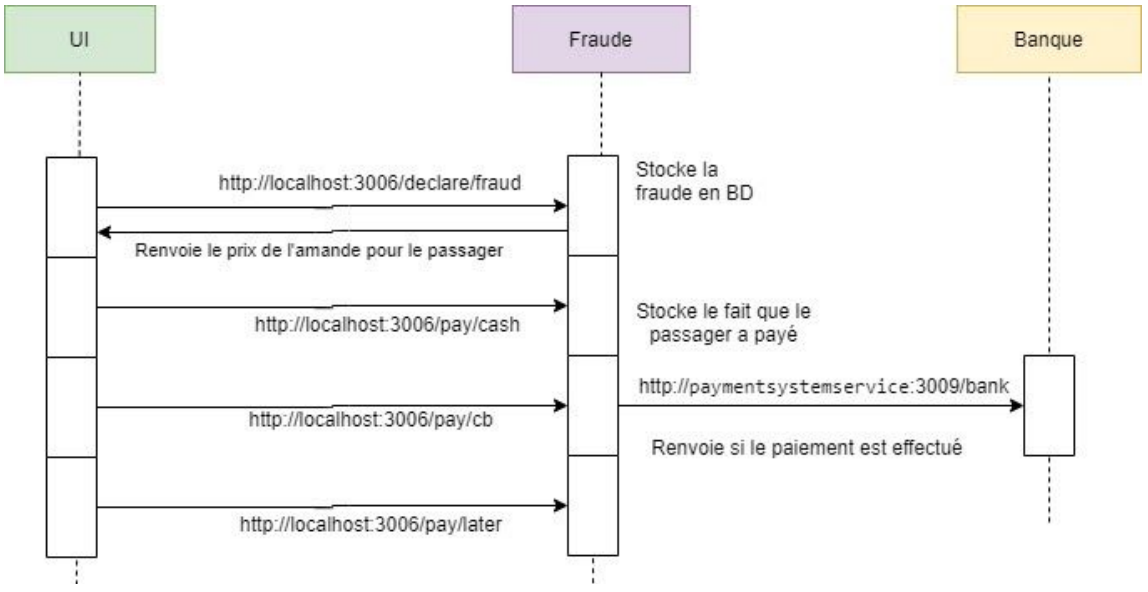
★ **Statistiques**



★ **Contrôler un ticket :**



★ **Signaler Fraud :**



Choix de technologies :

- **Front - end :** Android
  Après une recherche approfondie 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 différents. Même le développement d'applications mobiles Android peut se produire sous Windows, Mac OS ou Linux.

- **Back-end :** NodeJs
  Parmi les différents frameworks qui sont proposés ,on a bien choisi NodeJs .
  voici 3 bonnes raisons d'utiliser NodeJS dans notre projet :
  - Notre architecture est basée sur plusieurs services séparés qui communique entre eux avec des appels REST . Dans ce cas la, 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.

  - 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.
    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 .

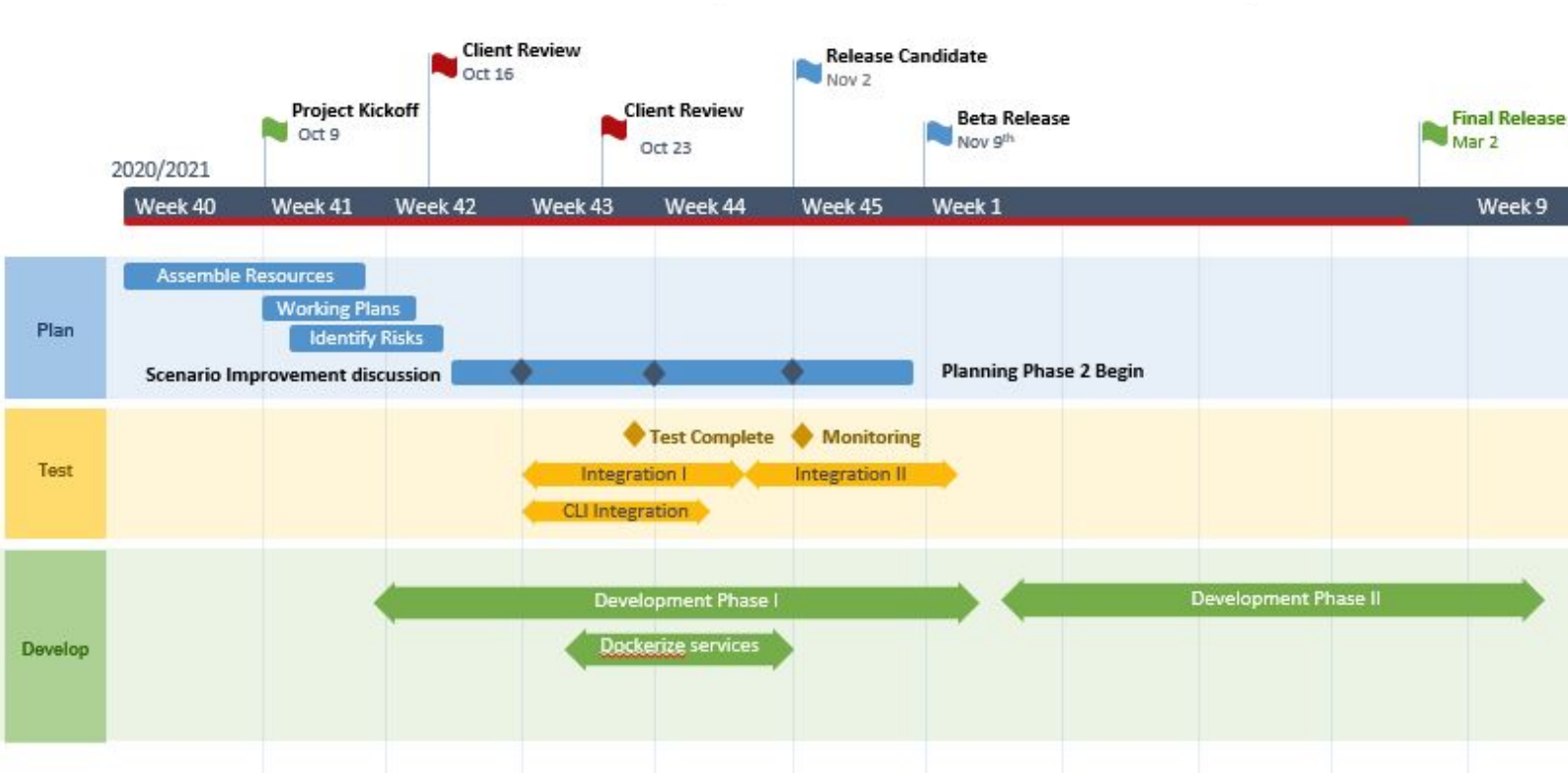
- **BD :** MongoDB
  L'architecture évolutive horizontale de MongoDB peut prendre en charge d'énormes volumes de données et de trafic et c'est ça ce qu'on a besoin dans notre application . On aura un grand flux de tickets à contrôler dans SNCF il y a 10 000 contrôleurs et chacun va contrôler minimum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

Risks :

- Un nombre énorme de requêtes qui peut entraîner un surcharge de serveur
- Un réseau instable qui peut perturber la récupération des données.
- Le composant paiement est un composant sensible aux attaques sécurité

Roadmap

Voici un schéma qui présente ce que l'on a prévu de faire pour ces prochaines semaines :



Auto-évaluation :

- ***Rania Fekih :***
  Ce projet m'a appris comment justifier mes choix en prenant toujours en compte les besoins du client ainsi que les éventuels risques qu'on pourrait avoir. J'ai aussi appris que l'architecture élaborée au début n'est jamais notre architecture finale, elle va évoluer au fur et à mesure de notre processus de développement grâce à un travail collectif. Nos réunions chaque semaine ont été un bon moyen pour voir les choses autrement et pour pouvoir prendre à la fin la meilleure décision.
  Malgré nos grands efforts pour tout finir à temps nous étions un peu en retard.Donc ce qu'on va changer la prochaine fois c'est de consacrer la dernière semaine pour la présentation même si notre produit n'est pas à 100% fini et consacrer plus de temps à l'intégration et les tests.
- ***Anis Khalili :***
  Personnellement, je trouve qu'on a bien respecté le Roadmap malgré quelques retards de liaison de la partie android (frontend) avec le backend .
  Le fait de fixer un jour permettant de résoudre les bugs et la discussion à propos de l'évolution du projet, ainsi que la répartition des tâches restantes pour la semaine d'aprés m'a aidé énormément à connaître ce que les autres membres ont effectués et de plus m'a permis d'avoir une vision globale sur le projet du backend frontend et la partie devops.
  Le point faible à signaler,est qu'on aurait dû éviter la pression de dernière minute pour faire les tests pour quelque scénario final.

- **Benjamin Vouillon:**

Cette première partie de projet a été riche en enseignement. Car malgré une Roadmap chargée nous avons fourni pour la démonstration un POC convaincant et réalisé en backend des traitements nécessaires aux fonctionnalités restantes. Et notre architecture nous permet de tirer partie de la production du POC pour construire les fonctionnalités restantes du projet.

Nous sommes conscients que les tests fonctionnels manquent à notre projet dû au manque de temps liés au problème d'ajout de la fonctionnalité de scan de QR. C'est pour cela que nous allons mettre en place en premiers lieux les tests qui nous font défaut dans la suite du développement de l'application.

Cependant le surcoût de conception et les soucis de développement liés à la mise en place du projet se trouvent désormais transformés en gain de productivité grâce à l'architecture plus mature de notre solution. Donc nous pourrons améliorer la qualité de notre production plus sereinement.

Nous avons aussi montré notre capacité à nous adapter face aux imprévus, pour sortir le meilleur malgré tout. Par exemple, à cause des problèmes liés à l'API de scan de ticket qui fait crasher l'application nous nous sommes réparties les tâches pour que malgré le temps dépensé dans cette mise en place, la démo se fasse de manière la plus complète possible et cela en enlevant les tests d'interfaces que nous faisons déjà à la mains

Nous avons aussi fait montre d'une grande capacité à travailler en équipe, chacun de nous a un background technique qu'il n'a pas hésité à transmettre aux autres pour que chacun comprenne ce que les autres font et à tout moment pouvoir aller remplacer ou aider dans tous les domaines. Cette cohésion d'équipe est vraiment un de nos points forts.

Pour conclure, je dirais que notre projet a subi un coup au début pour la mise en place de certaines fonctionnalités. Mais dans l'ensemble cette expérience est vraiment très positive. Nous sommes une équipe et allons avancer comme tels pour continuer à réaliser ce projet.

- **Robin Lambert:**
  La première phase de ce projet nous a permis de nous pencher sur des questions d'architecture essentielles pour concevoir un système complexe. Nous avons sû prioriser nos objectifs et choisir les fonctionnalités à implémenter pour ce working skeleton.
  En revanche et malgré nos efforts collectifs, nous avons rencontré des difficultés de timing qui nous ont suivi tout au long de la partie développement. Nous avons heureusement pu terminer l'application à temps, au détriment de tests côté frontend (avec Express par exemple) qu'il aurait été judicieux de faire plus tôt, quitte à laisser tomber certains points moins importants pour la démonstration.
  De façon générale, l'organisation est bonne et nous pensons avoir toutes les clés en main pour poursuivre ce projet sur la bonne voie. Les nombreuses discussions autour de l'architecture et des retours qui nous ont été faits en début de projet nous ont permis de travailler sur une base saine, que l'on pense extensible à coût raisonnable.

implication de chaque membre :

Rania Fekih	Benjamin Vouillon	Anis Khalili	Robin Lambert
100	100	100	100