PROJET DE PLATEFORME COLLABORATIVE ET DE VISIOCONFÉRENCE

FRAMEWORK D’ARCHITECTURE

SOMMAIRE

ANALYSE DE L'ÉCART ENTRE LA BASELINE ET LA TARGET

L'architecture de base nécessite d'ajouter ou modifier des composants afin de couvrir l'intégralité des besoins fonctionnels et des exigences techniques requis pour ce projet de réalisation complète de l'architecture et du design du système.

De plus, certains anciens services doivent être remplacés par des services plus adaptés aux exigences techniques.

Nouveaux composants techniques

Passerelle d'API

Le point d'entrée de services est remplacé par une passerelle d'API. Une passerelle d'API est un outil de gestion des interfaces de programmation d'application (ou API) qui se positionne entre un client et une collection de services back-end.

Elle agit comme un proxy inversé qui accepte tous les appels des API, rassemble les différents services requis pour y répondre et renvoie le résultat souhaité.

L'objectif principal d'une passerelle API est de simplifier et de stabiliser les interfaces exposées aux clients (mobiles, navigateurs...). De plus, en raison de la position unique d'une passerelle API dans l'architecture, divers avantages complémentaires sont activés, tels que la surveillance, la journalisation, la sécurité, l'équilibrage de charge et la manipulation du trafic.

Service unique de gestion des accès et des identités (IAM)

Ce nouveau service d'IAM (Identity and Access Management) sera mis en place en remplacement des 3 systèmes actuellement utilisés, afin de :

- gérer les profils individuels et les groupes d'utilisateurs
- gérer les profils internes et externes
- gérer les droits d'accès aux différents services et ressources fournis par Astra

Il contiendra un profil invité permettant d'accéder aux ressources publiques uniquement depuis l'application mobile et le site internet d'Astra.

.

Il contiendra les anciennes données de l'architecture actuelle soient :

- pour les utilisateurs internes :
  - le rôle
  - le département
  - le statut de l'utilisateur (salarié/prestataire/stagiaire)
- pour les utilisateurs externes :
  - le rôle
  - le groupe auquel l'utilisateur appartient (pré-requis)

Afin de sécuriser les accès et les données, l'exploitation, pour les profils externes comme internes, d'une double authentification sera faite.

Nouveaux composants fonctionnels

Plateforme collaborative

Ce nouveau service permettra d'animer des réunions web interactives, sur invitation, avec streaming audio/vidéo, enregistrement du flux de streaming, partage d'écran et chat entre participants ou entre participant et animateur.

Cette plateforme permettra d'organiser des présentations web avec streaming audio/vidéo, sur invitation, avec chat entre participants ou entre participant et animateur. Le nombre de participants pourra aller de 10 à plus de 500.

Plateforme de streaming

Ce nouveau service permettra la visualisation de vidéo live ou enregistrées et ne permettra pas le téléchargement local à l'exception du cache nécessaire à la bonne exécution du service..

L'accès aux différentes ressources vidéos se fera selon les droits de chaque utilisateur y compris pour les utilisateurs ayant un accès public.

Ce nouveau service pourra être un composant de la plateforme collaborative.

Outil de chat

Ce nouveau service permettra les discussions instantanées entre membres d'une réunion/présentation ou entre l'animateur d'une réunion/présentation et les membres d'une réunion.

Ce nouveau service pourra être un composant de la plateforme collaborative.

Outil de planification

Ce nouveau service permettra d'organiser et de planifier les réunions et présentations en envoyant des mails d'invitation aux utilisateurs privés concernés.

Ce nouveau service pourra être un composant de la plateforme collaborative.

Modifications apportées aux services existants

Gestion des documents

La gestion des documents n'est plus le service central permettant d'accéder à tous les autres services.

Ceci permet de sécuriser les données contenues dans la gestion des documents car les services ouverts sur l'extérieur (plateforme collaborative et plateforme de streaming) ne passent plus par ce point.

Cela permet également de supprimer un illogisme fonctionnel qui fait que désormais la plateforme collaborative et la plateforme de streaming ne dépendent plus de la gestion des documents ainsi les services sont tous indépendants (avec un couplage le plus faible possible).

Point d'entrée de service

Comme expliqué plus haut, le point d'entrée de service est supprimé et remplacé par une passerelle d'API qui gérera les accès aux différents services entre eux ou en fonction des droits d'accès des utilisateurs en se basant sur le service d'identification.

Contrôles faits sur les outils front existants

Les deux portails front (site web et application mobile) sont conservés, a priori en l'état, car ils sont déjà dits réactifs. Ils permettent donc de s'adapter à tous types de supports ou de tailles d'écran.

Les applications utilisant un client dit léger (un navigateur web), elles ne sont que peu impactées par les OS et seront donc uniquement tributaire du choix du navigateur en termes de compatibilité.

Cependant, afin de s'assurer que le site web répond bien à ce besoin de compatibilité, des tests de compatibilité-navigateur (cross-browser testing en anglais) seront effectués et les éléments frontend non compatibles feront l'objet

d'une "normalisation" (la normalisation consiste à ré-ajuster les styles par défaut des navigateurs afin d'éviter les différences de styles et de positionnement).

BONNES PRATIQUES DE L'ARCHITECTURE

Architecture REST

REST signifie REpresentational State Transfer (ou transfert d'état de représentation, en français), et constitue un ensemble de normes architecturales qui structurent la façon de communiquer les données entre une application et le reste du monde, ou entre différents composants d'une même application. Nous utilisons l'adjectif RESTful pour décrire les API REST. Les API RESTful se basent sur le protocole HTTP pour transférer les informations.

Six contraintes architecturales définissent un système REST. Ces contraintes restreignent la façon dont le serveur peut traiter et répondre aux requêtes du client afin que, en agissant dans ces contraintes, le système gagne des propriétés non fonctionnelles désirables, telles que la performance, l'extensibilité, la simplicité, l'évolutivité, la visibilité, la portabilité et la fiabilité.

Les contraintes architecturales de REST confèrent aux systèmes qui les respectent les propriétés architecturales suivantes :

- performance dans les interactions des composants, qui peuvent être le facteur dominant dans la performance perçue par l'utilisateur et l'efficacité du réseau ;
- extensibilité permettant de supporter un grand nombre de composants et leurs interactions ;
- simplicité d'une interface uniforme ;
- évolutivité des composants pour répondre aux besoins (même lorsque l'application est en cours de fonctionnement) ;
- visibilité des communications entre les composants par des agents de service ;
- portabilité des composants en déplaçant le code avec les données ;
- fiabilité dans la résistance aux pannes du système en cas de panne des composants, des connecteurs ou des données.

Liste des contraintes architecturales de REST :

Client-serveur

Les responsabilités sont séparées entre le client et le serveur. Découpler l'interface utilisateur du stockage des données améliore la portabilité de l'interface utilisateur sur plusieurs plateformes. L'extensibilité du système se retrouve aussi améliorée par la simplification des composants serveurs

Sans état

La communication client–serveur s'effectue sans conservation de l'état de la session de communication sur le serveur entre deux requêtes successives. L'état de la session est conservé par le client et transmis à chaque nouvelle requête. Les requêtes du client contiennent donc toute l'information nécessaire pour que le serveur puisse y répondre. La visibilité des interactions entre les composants s'en retrouve améliorée puisque les requêtes sont complètes. La tolérance aux échecs est également plus grande. De plus, le fait de ne pas avoir à maintenir une connexion permanente entre le client et le serveur permet au serveur de répondre à d'autres requêtes venant d'autres clients sans saturer l'ensemble de ses ports de communication, ce qui améliore l'extensibilité du système.

Cependant une exception usuelle à ce mode sans état est la gestion de l'authentification du client, afin que celui-ci n'ait pas à renvoyer ces informations à chacune de ses requêtes.

Avec mise en cache

Les clients et les serveurs intermédiaires peuvent mettre en cache les réponses. Les réponses doivent donc, implicitement ou explicitement, se définir comme pouvant être mise en cache ou non, afin d'empêcher les clients de récupérer des données obsolètes ou inappropriées en réponse à des requêtes ultérieures. Une mise en cache bien gérée élimine partiellement voire totalement certaines interactions client–serveur, améliorant davantage l'extensibilité et la performance du système.

En couches

Un client ne peut habituellement pas dire s'il est connecté directement au serveur final ou à un serveur intermédiaire. Les serveurs intermédiaires peuvent améliorer l'extensibilité du système en mettant en place une répartition de charge et un cache partagé. Ils peuvent aussi renforcer les politiques de sécurité.

Avec code à la demande (facultative)

Les serveurs peuvent temporairement étendre ou modifier les fonctionnalités d'un client en lui transférant du code exécutable. Cela permet de simplifier les clients en réduisant le nombre de fonctionnalités qu'ils doivent mettre en œuvre par défaut et améliore l'extensibilité du système. En revanche, cela réduit aussi la visibilité de l'organisation des ressources. De ce fait, elle constitue une contrainte facultative dans une architecture REST.

Interface uniforme

La contrainte d'interface uniforme est fondamentale dans la conception d'un système REST. Elle simplifie et découple l'architecture, ce qui permet à chaque composant d'évoluer indépendamment. Les quatre contraintes de l'interface uniforme sont les suivantes :

- Identification des ressources dans les requêtes :
  Chaque ressource est identifiée dans les requêtes, par exemple par un URI dans le cas des systèmes REST basés sur le Web. Les ressources elles-mêmes sont conceptuellement distinctes des représentations qui sont retournées au client. Par exemple, le serveur peut envoyer des données de sa base de données en HTML, XML ou JSON, qui sont des représentations différentes de la représentation interne de la ressource.
- Manipulation des ressources par des représentations :
  Chaque représentation d'une ressource fournit suffisamment d'informations au client pour modifier ou supprimer la ressource.

- Messages auto-descriptifs :
  Chaque message contient assez d'informations pour savoir comment l'interpréter. Par exemple, l'interpréteur à invoquer peut être décrit par un type de médias.
- Hypermédia comme moteur d'état de l'application (HATEOAS) :
  Après avoir accédé à un URI initial de l'application — de manière analogue aux humains accédant à la page d'accueil d'un site web —, le client doit être en mesure d'utiliser dynamiquement les hyperliens fournis par le serveur pour découvrir toutes les autres actions possibles et les ressources dont il a besoin pour poursuivre la navigation. Il n'est pas nécessaire pour le client de coder en dur cette information concernant la structure ou la dynamique de l'application.

Hébergement cloud hybride

Il est recommandé d'utiliser un cloud hybride alliant cloud public et cloud privé

Un cloud privé est une infrastructure qui est totalement dédiée à l'entreprise. Ce sont un ou plusieurs serveurs privatifs qui géreront l'ensemble des données et l'ensemble des utilisateurs de l'organisation. L'administration peut être gérée en interne ou externalisée. Un Cloud privé peut être hébergé en interne ou en Datacenter, il est très proche d'une infrastructure « On-Premise ».

Un cloud public est une infrastructure mutualisée où données et applications sont hébergées sur une multitude de serveurs. Les utilisateurs peuvent y accéder de n'importe où. La maintenance de l'infrastructure est assurée par le fournisseur.

Le Cloud hybride adopte donc une infrastructure mixte qui s'appuie sur le matériel propre à l'entreprise ou sur son cloud privé et à la fois sur des ressources « On-demand » via des fournisseurs de cloud.

Les avantages du cloud hybride sont :

- La sécurité : Les données les plus sensibles peuvent être séparées des autres données en les hébergeant respectivement sur un cloud privé et un cloud public.
- La flexibilité et l'élasticité : Autrement appelé scalabilité, ceci a pour avantage de permettre à notre architecture de s'adapter automatiquement à un changement d'ordre de grandeur de la demande (montée en charge), en particulier sa capacité à maintenir ses fonctionnalités et ses performances en cas de forte demande. Pour cela, on peut utiliser la conteneurisation qui permet d'héberger un microservice sur une instance et de multiplier le nombre d'instances de ce microservice en fonction de la demande.
- Les coûts : S'appuyer en partie sur le cloud public permet de limiter le coût de l'infrastructure locale avec à la clé moins de machines, moins de maintenance et ainsi, de passer, pour certaines applications, de l'achat de licence à l'abonnement ou au paiement à l'usage.
- L'interopérabilité et l'indépendance : PaaS (Platform as a service ou Plateforme en tant que service) et conteneurs permet de s'affranchir de la dépendance à une infrastructure matérielle. Cela permet, par exemple, de déplacer facilement des applications entre différents fournisseurs d'infrastructure.

L'hébergement cloud s'appuie également sur les principes de la conteneurisation qui a pour avantages :

- Flexibilité : Toutes les applications peuvent être transformées en des conteneurs
- Légèreté : Contrairement à la virtualisation classique, les conteneurs exploitent et partagent le système d'exploitation de l'hôte, ce qui les rend très efficaces en terme d'utilisation des ressources du système
- Portabilité : Il est possible de créer, déployer et démarrer des conteneurs sur un serveur local ou distant (voire même sur un PC local)
- Adaptation : L'installation et la désinstallation de conteneurs ne dépend pas des autres conteneurs installés. Ce qui permet de mettre à jour ou remplacer un conteneur sans modifier les autres
- Scalabilité : Dupliquer un container est extrêmement simple, ce qui permet de réaliser de la scalabilité horizontale aisément

Sécurité des accès

Afin de sécuriser les accès, il est recommandé d'utiliser une solution unique d'identification qui exploitera une double authentification.

A cette fin, les 3 services d'identification actuellement utilisés seront fusionnés pour n'en faire qu'un et seront pilotés par le service RH qui assurera la création des utilisateurs et la gestion des droits d'accès.

Il est également conseillé d'utiliser une matrice de rôles en créant des modèles d'habilitation, reposant sur la définition de rôles (RBAC : Role-Based Access Control). Ceci permet d'attribuer les autorisations à des rôles "professionnels" plutôt qu'à des personnes.

De plus, un accès public sera possible depuis les portails web et mobiles via un processus de création de compte utilisateur ayant un accès limité aux applications et données (grâce à l'utilisation d'une matrice de rôles).

Stockage des données

Il est recommandé que le stockage des données s'appuie sur le cloud hybride afin de sécuriser l'accès aux données les plus sensibles.

En matière de sécurité, les données les plus sensibles pourraient être stockées sur un cloud privé (soit sur un des nos serveurs Linux) tandis que les données destinées au public pourraient être hébergées sur un cloud public.

Cependant, il convient de considérer l'ensemble des données gérées par Astra comme sensibles et de ne faire qu'un stockage local.

De plus, afin de garantir la haute disponibilité des données, il est conseillé de mettre en place une stratégie de reprise après sinistre (Disaster Recovery) en utilisant des outils permettant la sauvegarde automatisée et régulière (ou redondance volontaire) des données. Cette redondance doit également s'appuyer sur le cloud hybride en diversifiant les "lieux" de stockage (physiques et virtuels).

CYCLE DE VIE DE L'IMPLÉMENTATION DE L'ARCHITECTURE

Les différents phases de l'implémentation de l'architecture sont présentées dans le diagramme de transition applicatif (*1) et décrites ci-dessous :

Etape 1 - Implémentation de la sécurité
- Création du nouveau service de gestion des identités (IAM)
- Migrer les données d'accès dans une nouvelle base de données locale
- Tester le service IAM
- Conteneuriser le service IAM
- Supprimer les anciens services de gestion des accès

Etape 2 - Implémentation de la passerelle d'API
- Mettre en place la passerelle d'API dans un conteneur

Etape 3 - Conteneurisation des services conservés
- Conteneuriser la gestion des documents
- Tester le service IAM en contexte gestion des documents
- Conteneuriser le service mail
- Tester le service IAM en contexte service mail

Etape 4 - Implémentation de la plateforme collaborative/streaming/chat (CSC)
- Créer la plateforme CSC
- Créer la base de données locale de la plateforme CSC
- Tester la plateforme CSC
- Conteneuriser la plateforme CSC
- Tester le service IAM en contexte plateforme collaborative

Etape 5 - Implémentation de l'outil de planification
- Créer l'outil de planification
- Créer la base de données locale de l'outil de planification
- Tester l'outil de planification
- Conteneuriser l'outil de planification
- Tester le service IAM en contexte outil de planification

Etape 6 - Implémentation des nouvelles applications front

- Vérification de la compatibilité-navigateur du site web
- Modification du site web le cas échéant
- Vérification de la compatibilité-navigateur de l'application mobile
- Modification de l'application mobile le cas échéant
- Ajout des nouvelles fonctionnalités dans les applications front
    - Plateforme collaborative
    - Service de streaming
    - Outil de planification
    - Outil de chat
- Développement d'une fonctionnalité de recherche de vidéos
- Test End-to-End

(*1) Voir annexe Diagramme de transition applicatif

CYCLE DE VIE POUR L'IMPLÉMENTATION DE L'EXTENSIBILITÉ

L'architecture mise en place est naturellement extensible notamment grâce à :

- La passerelle d'API
- La conteneurisation

Exploitation de la passerelle pour l'extensibilité

Lors de la création d'un nouveau microservice, celui-ci sera implémenté dans la passerelle d'API via son API. Le microservice sera alors connu de la passerelle et pour être donc faire l'objet de requêtes par les applications clientes.

En utilisant le principe d'interface uniforme de l'architecture REST (notamment celui d'hypermédia), notre API pourra être exploitée facilement depuis les pages d'accueil des applications clientes sans codage supplémentaire

Si toutefois, l'ajout d'un microservice nécessitait de modifier le code des applications clientes, il pourrait être très simple et rapide, tel l'ajout d'un bouton ou d'une entrée de menu pour accéder, par exemple, à un outil de messagerie instantanée interne (en complément du service déjà disponible dans la plateforme collaborative par exemple).

Exploitation de la conteneurisation pour l'extensibilité

La mise en place d'un nouveau conteneur pour héberger un nouveau microservice se fait très rapidement grâce aux avantages de la conteneurisation (portabilité, légèreté, flexibilité).

Le remplacement d'un microservice par un autre peut se faire facilement (de la même manière que l'ajout d'un nouveau microservice) en permettant le déploiement progressif (par zones géographiques ou par groupes d'utilisateurs par exemple) de ce nouveau microservice tout en conservant l'ancien microservice en production jusqu'à l'arrêt définitif de celui-ci une fois le déploiement terminé (comme le montre le diagramme de cycle de vie des microservices (*1)).

(*1) Voir annexe Diagramme du cycle de vie des microservices

BONNES PRATIQUES POUR LA SCALABILITÉ ET LA MAINTENABILITÉ

Concevoir nos microservices en vue de leur évolution

Il est recommandé d'avoir un couplage le plus faible possible entre les microservices afin de permettre leur modification de manière indépendante des autres microservices. Cela assurera ainsi leur autonomie de fonctionnement, augmentera leur tolérance aux pannes, les rendra plus facilement scalables, facilitera leur déploiement, …

Il faut également assurer la cohésion entre les microservices en rendant les échanges entre eux le plus cohérent possible. On peut pour cela utiliser des interfaces claires avec des types précis et en évitant de définir des fonctions qui ont plusieurs objectifs (il est préférable de limiter une fonction à un seul cas d'utilisation).

On doit éviter les choix technologiques trop "exotiques" dans les communications entre microservices, tels que l'utilisation de middlewares qui sont généralement coûteux en licence et peuvent avoir une empreinte forte dans l'implémentation des microservices.

De plus, Il faut penser les interfaces pour limiter les "breaking changes" lors des évolutions des microservices car cela nécessite la modification des microservices qui y font appel et peuvent compliquer les déploiements.

Enfin, il faut proscrire l'exposition des détails de l'implémentation interne d'un microservices car cela peut donner des indices sur son fonctionnement. D'autres microservices peuvent ainsi en tirer partie et avoir une implémentation dépendante de ce fonctionnement qui rend le couplage plus important entre les microservices.

Utiliser le DDD pour la conception des microservices

L'intérêt de l'approche du "Domain-Driven-Design" (ou contexte borné) est de proposer une solution pour séparer une application en microservices. Un contexte borné peut correspondre à plusieurs microservices ayant en commun un contexte fonctionnel. Chaque contexte borné répond à un besoin fonctionnel qui possède un langage spécifique, c'est "l'ubiquitous language". Ce langage permet d'avoir une logique spécifique au contexte borné qui ne déborde pas de ce contexte.

Les frontières du contexte borné sont franchies seulement avec des interfaces qui sont exposées en dehors du contexte borné de façon à volontairement limiter les échanges entre contexte borné à ces interfaces. Cette limitation permet de contrôler et de maîtriser les interfaces et donc les échanges.

Garder les microservices aussi petits et simples que possible

La taille d'un microservice n'est pas forcément mesurable en nombre de lignes de code. Il est plus opportun de se baser sur une couverture fonctionnelle définie et limitée.

Par exemple, dans notre cas, le service de streaming fournira les flux audio/vidéo pour la diffusion de contenu en direct ou en replay. Le service de réunion fournira aussi des flux audio/vidéo mais ne contiendra pas de code gérant ces flux audio/vidéo. C'est le service de streaming qui sera alors mis à contribution.

A l'opposé, il n'est pas recommandé de pousser le concept à l'extrême (on peut alors parler de nanoservice). Si une fonctionnalité peut être couverte par une librairie déjà existante, il n'est pas utile de développer un microservice pour cela.

Par exemple, pour la gestion des rendez-vous de réunion, il n'est pas nécessaire de développer un microservice permettant de générer des rendez-vous au format iCalendar car des librairies existent pour tous types de langages de programmation.

Limiter la responsabilité des équipes

Tout comme les microservices, les équipes (de développement) doivent avoir des responsabilités limitées (fonctionnellement parlant). Leur champ d'intervention n'est pas forcément limité à un seul microservice. L'objectif est surtout de ne pas rendre toutes les équipes responsables de tous les microservices.

Dans notre cas, il n'est pas recommandé qu'une seule équipe ait la responsabilité des microservices de gestion des documents et de streaming car ceux-ci sont fonctionnellement assez éloignés. Mais une seule équipe pourrait être responsable à la fois de la plateforme collaborative et de l'outil de chat (le chat étant une fonctionnalité dite "collaborative").

Avoir une approche technologique agnostique

Le marché évoluant rapidement, les communications entre microservices ne doivent pas être dictées par une stack technologique. Cela pourra aider dans le futur à faire évoluer plus facilement nos microservices en utilisant la technologie la plus adaptée à un microservice en particulier en conservant les autres microservices sur leur stack technologique d'origine.

Par exemple, la plateforme de streaming utilisera des technologies spécifiques permettant de fluidifier les flux audio/vidéos et ainsi améliorer l'expérience utilisateur alors que l'outil de planification a besoin d'outils beaucoup moins performants car moins consommateur de données.

Penser scalabilité

La scalabilité est la capacité de notre architecture à s'adapter à un changement d'ordre de grandeur de la demande (montée en charge), en particulier sa capacité à maintenir ses fonctionnalités et ses performances en cas de forte demande.

On parle ici de scalabilité horizontale. Notre architecture doit être capable d'accepter une montée en charge en s'adaptant au besoin. Pour cela, on peut utiliser la

conteneurisation qui permet d'héberger un microservice sur une instance et de multiplier le nombre d'instances de ce microservice en fonction de la demande.

Par exemple, dans le cas de notre service de streaming, la mise à disposition d'une nouvelle vidéo de cours et l'envoi d'un message à une liste d'utilisateurs potentiellement intéressés par celle-ci pourrait faire croitre temporairement le nombre de requêtes au service de streaming et provoquer ainsi une montée en charge de l'utilisation de ce service qui pourrait augmenter le temps de latence. En multipliant le nombre d'instances du service, on permet à celui-ci d'absorber la montée en charge et de conserver une latence acceptable.

Adapter le stockage des données

Afin de conserver un couplage lâche, il est recommandé que chaque microservice gère ses propres données. Cela implique qu'un microservice ne peut pas accéder directement aux données d'un autre. Le succès de cette organisation des données dépend d'une définition efficace des contextes délimités de chaque microservice.

Cette approche permet d'adapter le type de stockage au type de données et d'être facilement scalable. Cela permet en outre de sécuriser les données en ne permettant l'accès aux données qu'au travers d'API.

Dans notre contexte, les données les plus sensibles pourront ainsi être mieux protégées en étant séparées des données plus accessibles.

Une autre bonne pratique, propre à notre contexte, est de ne pas stocker les données dans un cloud mais sur nos propres serveurs dont nous aurons la maîtrise en terme de sécurité.

ANNEXES

Diagramme de transition applicatif

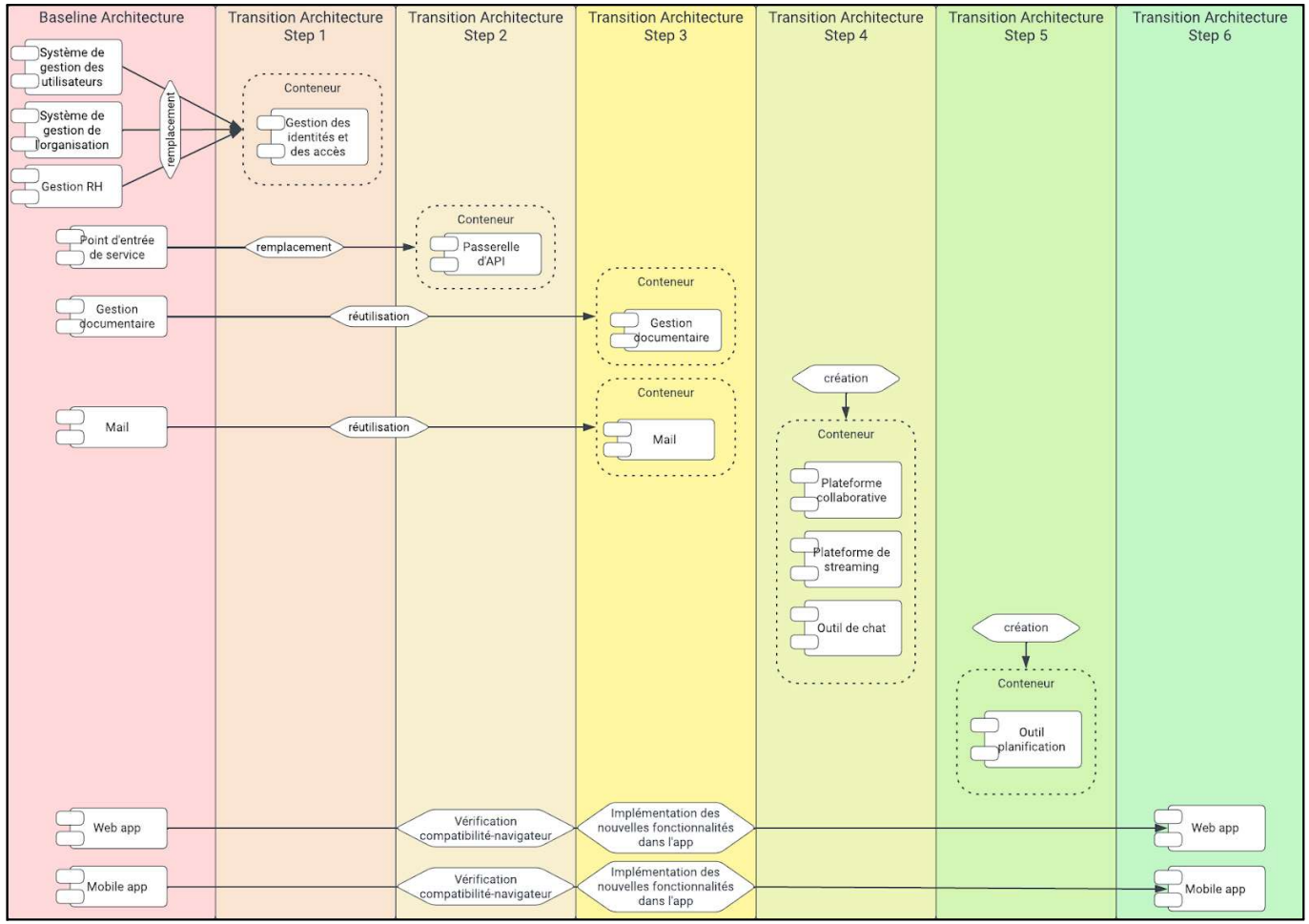

Diagramme du cycle de vie des microservices