Projet de mise à niveau
des outils de collaboration

Document de définition d’architecture

Auteur(s) et contributeur(s)

Nom & Coordonnées	Qualité & Rôle	Société
Gérald ATTARD	Architecte logiciel	Astra

Historique des modifications et des révisions

N° version	Date	Description et circonstance de la modification	Auteur
1.0	31/05/2022	Création du document	Gérald ATTARD

Validation

N° version	Nom & Qualité	Date & Signature	Commentaires & Réserves
1.0	Terry Strasberg CTO		

Tableau des abréviations

Abr.	Sémantique
ADM	Architecture Development method (trad. *méthode de développement d'architecture*)
AMOA	Assistance à Maîtrise d'OuvrAge
BBA	Baseline Business Architecture (trad. *architecture métier de référence*)
JVM	Java Virtual Machine (trad. *machine virtuelle Java*)
MOE	Maîtrise d'OEuvre
OLA	Operational Level Agreement (trad. *accord de niveau opérationnel*)
SLA	Service Level Agreement (trad. *accord de niveau de service*)
SRP	Single Responsability Principle (trad. *principe de responsabilité unique*)
TBA	Target Business Architecture (trad. *architecture métier cible*)
UML	Uniform Modeling Language (trad. *langage de modélisation uniforme*)

Table des matières

Situation actuelle

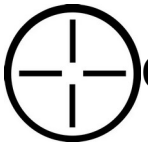
Leader dans le domaine de la recherche médicale depuis plusieurs années, l’IRA (Institut de Recherche Astra) s’est forgée une solide réputation basée sur la qualité de ses recherches médicales et la pertinence des résultats mis en évidence.

Ses activités ont toujours été accompagnées d’une forte propension à la coopération avec différents organismes, organisations et partenaires externes répartis à travers le monde.

Associée à ce contexte coopératif international, la croissance de l’IRA en terme d’effectif s’est vue accompagnée par une augmentation des frais de déplacement, impactant financièrement certains projets.

Ainsi, au travers du développement rapide des technologies de l’information, et en tenant compte des réglementation internationales en vigueur, autant politiques qu’industrielles, l’IRA se doit d’adapter ses outils collaboratifs afin de répondre à de nouveaux défis.

Ces challenges devront alors prendre en considération aussi bien l’infrastructure existante que les besoins actuels et prévisionnels, pour imaginer les outils collaboratifs de demain en accord avec les standards éthiques et moraux de l‘Institut.



Objectif du document

Le document de définition d'architecture est un livrable issu de l’ADM du TOGAF.

Ce document décrit l'architecture métier de base et cible d'un projet. Il contient les principaux artefacts architecturaux créés au cours d'un projet.

Les documents de définition d'architecture produits en phase B, C et D couvrent tous les domaines de l'architecture (entreprise, données, application et technologie) et examinent également tous les états pertinents de l'architecture (base, état(s) intermédiaire(s) et cible).

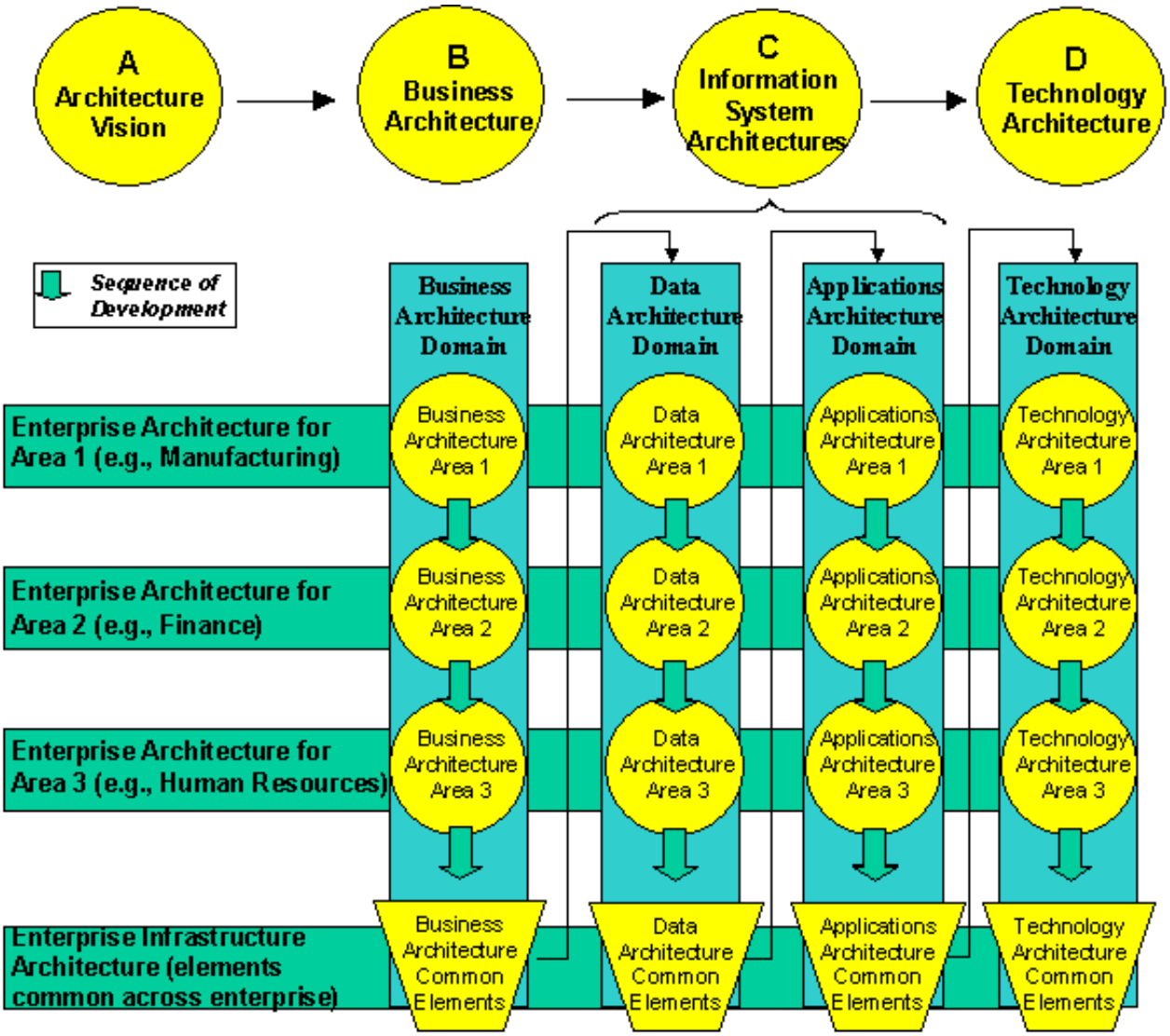
Les documents de définition d'architecture accompagnent la spécification des exigences d'architecture, avec un objectif complémentaire : les documents de définition d'architecture fournissent une vue qualitative de la solution et visent à communiquer l'intention des architectes.

La spécification des exigences d'architecture fournit une vue quantitative de la solution, en indiquant les critères mesurables qui doivent être respectés lors de la mise en œuvre de l'architecture.

Ainsi, ce document a pour ambition de décrire précisément l'architecture de microservices requise pour les nouvelles capacités du futur système d'information d'Astra, sous différents aspects, à savoir :

- la Business Architecture ;
- l'Application Architecture ;
- la Technical Architecture ;
- l'Infrastructure Architecture.

Ces différentes phases sont séquencées dans le schéma qui suit :



De plus, ce document se basera sur des ressources documentaires connexes telles que :

- l’étude exploratrice, dans laquelle il sera possible de prendre connaissance de la Baseline Business Architecture, ainsi que de la Target Business Architecture ;
- le framework d’architecture, décrivant des outils, des pratiques et des méthodes pouvant être mis en œuvre pour mener à bien l’implémentation de cette architecture.

En outre, ce document pourra également servir de référence pour réaliser les tâches administratives suivantes :

- décrire les décisions relatives aux blocs de construction de cette architecture ;
- Préparer les sections métier comprenant tout ou partie de :
  - une empreinte commerciale (une description de haut niveau des personnes et des emplacements impliqués dans les fonctions commerciales clés) ;
  - une description détaillée des fonctions de l'entreprise et de leurs besoins en informations ;
  - une empreinte de gestion (montrant l'étendue du contrôle et de la responsabilité) ;
  - normes, règles et directives indiquant les pratiques de travail, la législation, les mesures financières, etc ;
  - Une matrice de compétences et un ensemble de fiches de poste.

Méthodologie rédactionnelle

Pour réaliser les objectifs décrits supra, plusieurs étapes seront nécessaires :

- identifier les **responsabilités des parties prenantes** : décrire les responsabilités des parties prenantes pour l'architecture d'entreprise ;
- définir l’**architecture métier (Business Architecture)** en :
  - développant l’**architecture métier de référence (*BBA*)** en définissant l'architecture d'entreprise actuelle ;
  - développant l’**architecture métier cible (*TBA*)** en définissant l'architecture d'entreprise cible ;
  - Effectuer une **analyse des écarts** : analyser et décrire les écarts entre la *BBA* et la *TBA* ;
- réaliser une **analyse d’impact** : décrire les impacts associés à la migration vers cette nouvelle architecture métier.

Responsabilités des parties prenantes

Identification des parties prenantes

Dans cette section, nous nous efforcerons de penser à toutes les personnes qui seront affecter par ce projet, qui ont de l'influence ou du pouvoir sur lui, ou qui ont un intérêt à ce qu'il aboutisse ou non.

Ainsi, ce regroupement de rôle peut inclure des cadres supérieurs, des rôles d'organisation de projet, des rôles d'organisation cliente, des développeurs de système, des partenaires d'alliance, des fournisseurs, des opérations informatiques, des clients, etc.

Néanmoins, lors de l'identification des parties prenantes, il existera un risque de trop se concentrer sur la structure formelle d'une organisation comme base d'identification. Les groupes de parties prenantes informels peuvent être tout aussi puissants et influents que les groupes formels.

Les parties prenantes identifiées sont donc recensées dans le tableau qui suit :

Nom Prénom	Fonction
Strasberg Terry	CTO-Chief Technical Operation
Leonard Leslie	VPO-Vice Présidente des Opérations

Cette section devra être mise à jour tout au long du cycle de vie du projet lors de l’identification de nouvelles parties prenantes.

Positionnement des parties prenantes

Cette section permettra de développer une bonne compréhension des parties prenantes les plus importantes et d'enregistrer cette analyse pour référence.

Cette partie du document sera, également, à actualiser au cours du projet.

Fonction	Nom Prénom	Capacité à perturber le projet	Compréhension courante	Compréhension requise	Engagement actuel	Engagement requis	Support requis
CTO	Strasberg Terry	H	M	H	B	H	H
VPO	Leonard Leslie	M	H	H	B	H	H

Légende :

- H : *Haute*
- M : *Moyen*
- B : *Bas*

Gestion des parties prenantes

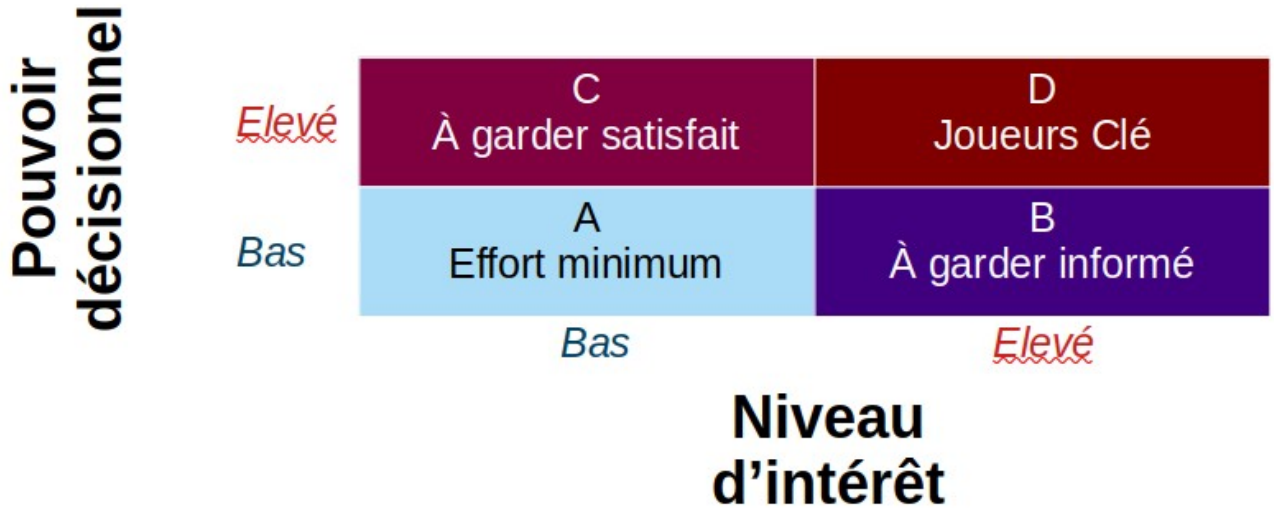
Les paragraphes précédents ont identifié une liste de personnes et d'organisations concernées par le projet d'architecture d'entreprise.

Certains d'entre eux peuvent avoir le pouvoir de bloquer ou d'avancer. Certains peuvent être intéressés par ce que fait l'initiative d'Architecture d'Entreprise, d'autres peuvent ne pas s'en soucier.

Cette étape permet aux équipes d'AMOA et de MOE de voir facilement quelles parties prenantes sont censées être des bloqueurs ou des critiques, et quelles parties prenantes sont susceptibles d'être des défenseurs et des partisans de l'initiative.

Ainsi, il est primordial de déterminer le pouvoir, l'influence et l'intérêt des parties prenantes, de manière à concentrer l'engagement de l'architecture d'entreprise sur les personnes clés. Ceux-ci peuvent être cartographiés sur une matrice pouvoir/intérêt, qui indique également la stratégie à adopter pour s'engager avec eux.

Ainsi, la matrice représente la catégorisation à réaliser pour chaque partie prenante, en fonction de son pouvoir décisionnel et de son intérêt pour le projet :



A partir de la matrice ci-dessus, nous pouvons l'appliquer à l'identification des parties prenantes réalisée précédemment :

Nom Prénom	Fonction	Niveau d'implication
Strasberg Terry	CTO	C – à garder satisfait
Leonard Leslie	VPO	C – à garder satisfait

Tout comme les précédents paragraphes précédents, celui-ci sera à maintenir à jour durant le cycle de vie du projet si de nouvelles parties prenantes étaient identifiées.

Engagement sur les livrables à fournir

Cette section a pour objectif d’identifier les catalogues, les matrices et les diagrammes que l'engagement d'architecture doit produire et valider avec chaque groupe de parties prenantes pour fournir un modèle d'architecture efficace.

Il est important d'accorder une attention particulière aux intérêts des parties prenantes en définissant des catalogues, des matrices et des diagrammes spécifiques qui sont pertinents pour un modèle d'architecture d'entreprise particulier.

Cela permettra de communiquer l'architecture afin qu’elle soit comprise par toutes les parties prenantes, tout en leur permettant de vérifier que l'initiative d'architecture d'entreprise répondra à leurs préoccupations.

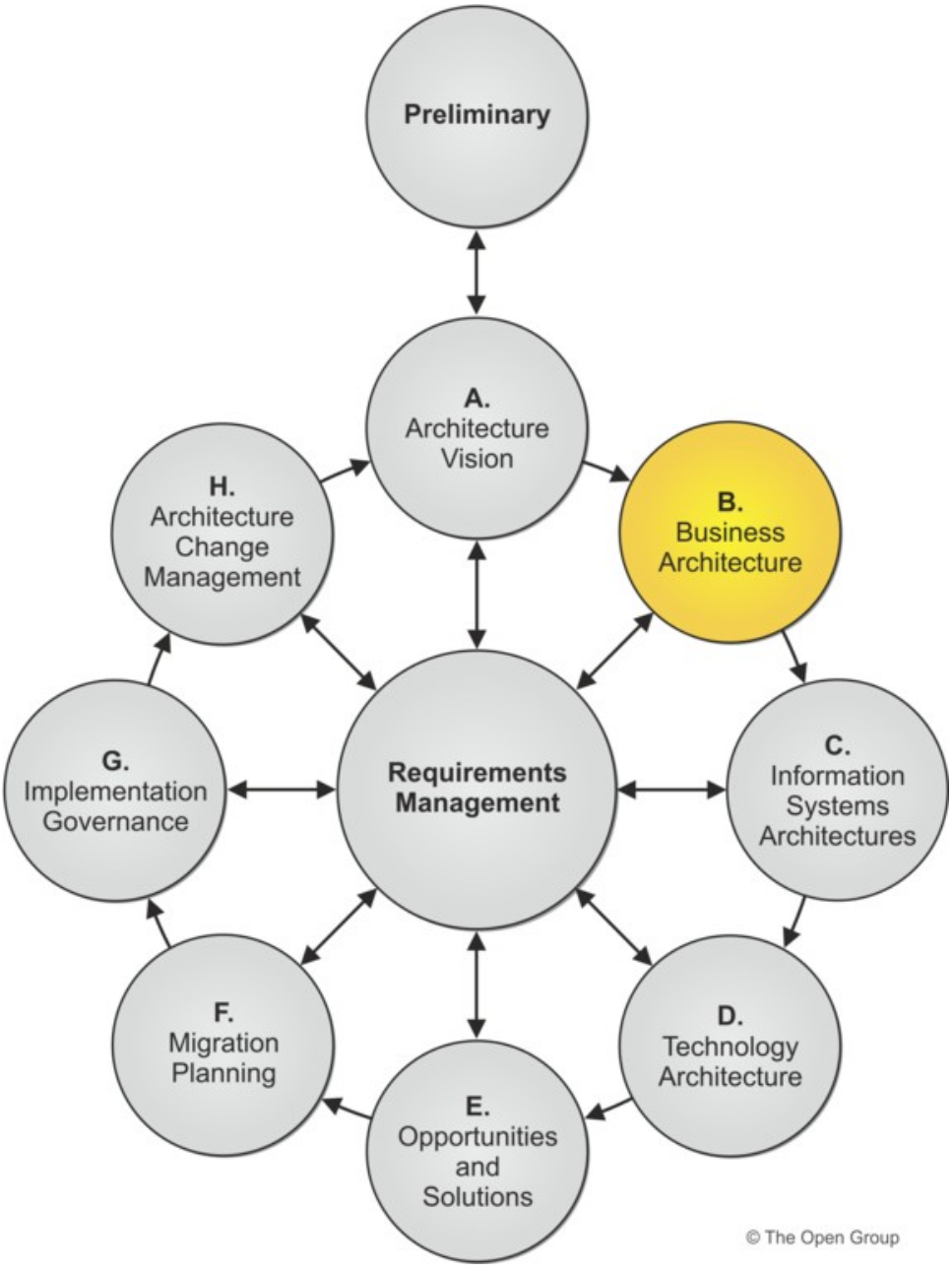
Nom Prénom	Fonction	Catalogue, Matrices et schémas à lui fournir
Strasberg Terry	CTO	• Diagramme de l'empreinte commerciale • Diagramme But/Objectif/Service commercial • Diagramme de décomposition de l'organisation • Catalogue des capacités métier • Matrice capacité/organisation • Carte des capacités de l'entreprise • Matrice Stratégie/Capacité • Matrice capacité/organisation • Diagramme du modèle d'affaires • Catalogue de flux de valeur • Catalogue des étapes de la chaîne de valeur • Matrice de flux de valeur/capacité • Carte de la chaîne de valeur
Leonard Leslie	VPO	• Catalogue des exigences • Diagramme du contexte du projet • Diagramme des avantages • Diagramme de l'empreinte commerciale • Diagramme de communication d'application • Carte de l'organisation • Catalogue des capacités métier • Matrice capacité/organisation • Carte des capacités de l'entreprise • Matrice Stratégie/Capacité • Matrice capacité/organisation • Diagramme du modèle d'affaires • Catalogue de flux de valeur • Catalogue des étapes de la chaîne de valeur • Matrice de flux de valeur/capacité • Carte de la chaîne de valeur

Architecture métier (Business Architecture)

L’architecture métier correspondant à la phase B du TOGAF correspond à deux objectifs :

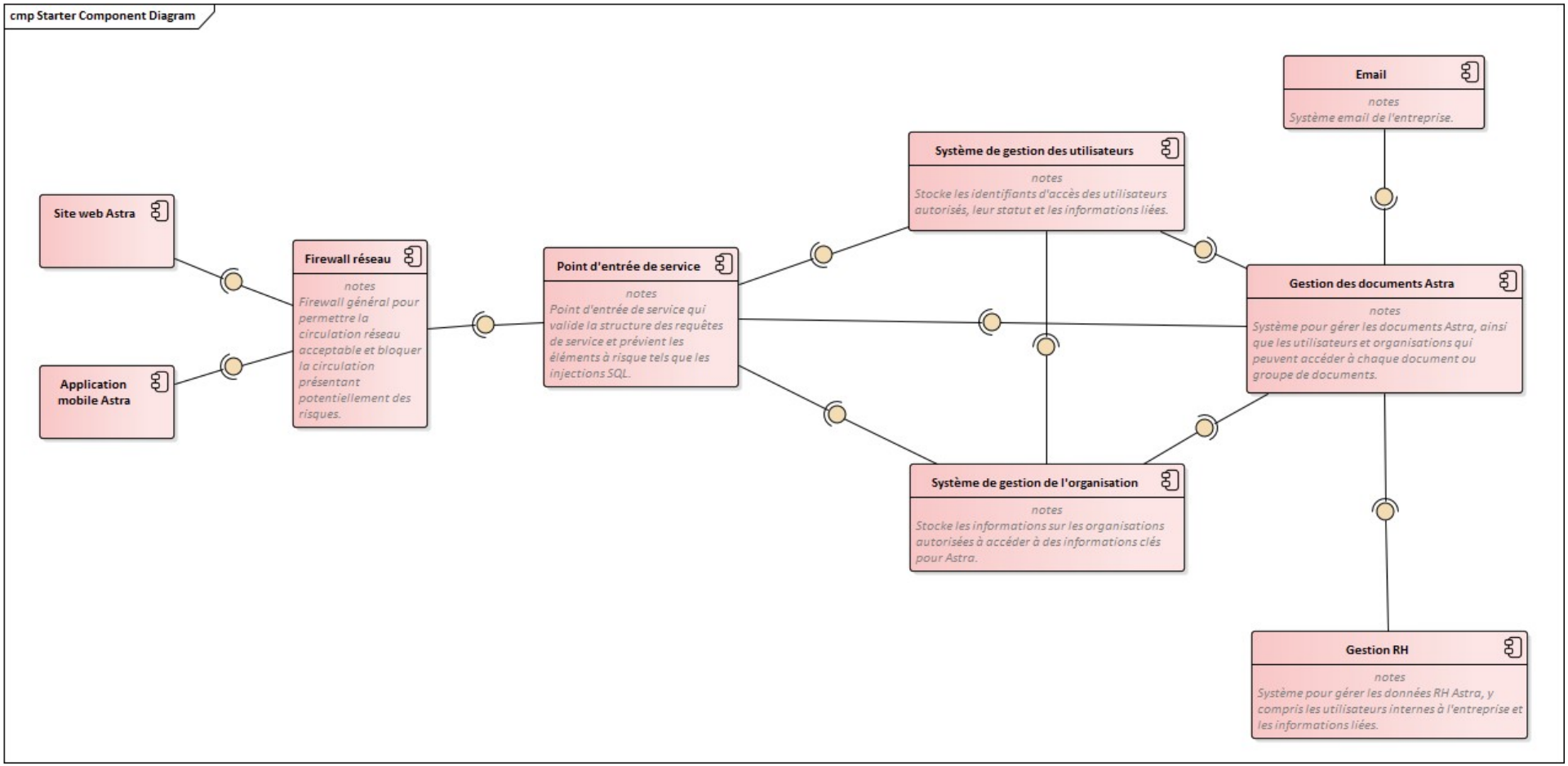
- développer l'architecture d'entreprise cible (*TBA*) qui décrit comment l'entreprise doit fonctionner pour atteindre les objectifs commerciaux et répondre aux moteurs stratégiques définis dans la vision de l'architecture, d'une manière qui répond à l'énoncé des travaux d'architecture et aux préoccupations des parties prenantes ;

- identifier les composants candidats de la feuille de route de l'architecture en fonction des écarts entre les architectures d'entreprise de référence (*BBA*) et cible (*TBA*).



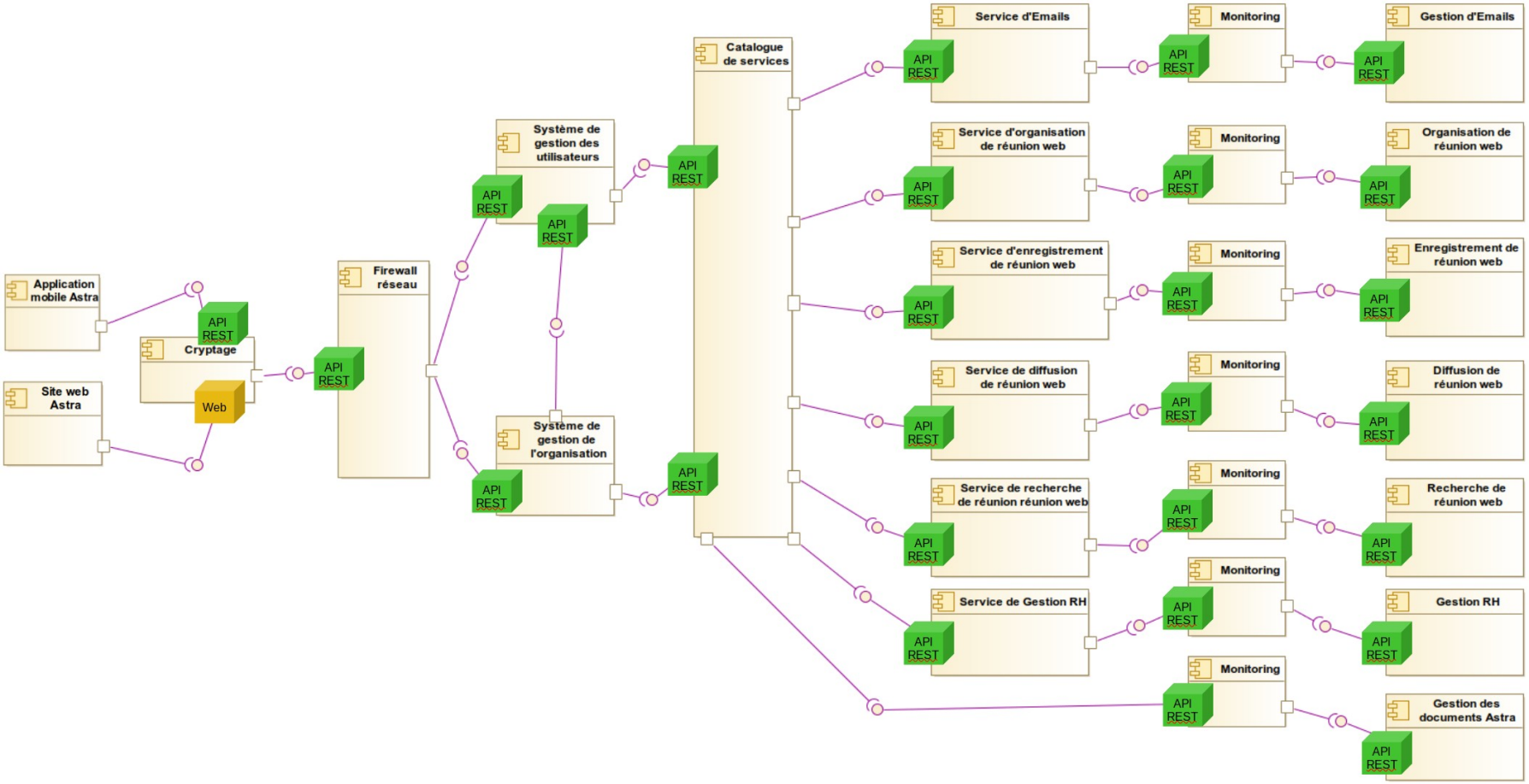
Architecture métier de référence (Baseline Business Architecture)

Tel qu'il a été rédigé au sein de l'étude exploratoire associée à ce projet, la *Baseline Business Architecture* se base sur le diagramme UML ci-dessous :



Architecture métier cible (Target Business Architecture)

Tel qu'il a été annoncé au sein de l'étude exploratoire associée à ce projet, la *Target Business Architecture* sera focalisera sur l'utilisation d'un modèle d'architecture des microservices. Ce dernier, appliqué aux besoins d'Astra, est défini selon le schéma de composants ci-dessous :



Définition des composants

Maintenant que la *BBA* et la *TBA* sont décrites, il convient de définir chaque composant les constituant. Aussi, chaque composant sera notifié des mentions *BBA* ou *TBA*, en fonction de leur appartenance à l'une ou l'autre architecture.

- **Application mobile Astra (*BBA*) :** application mobile pour les appareils Android et iOS permettant aux utilisateurs mobile d'accéder aux informations Astra autorisées par leur login et leur rôle d'utilisateur depuis un appareil mobile. L'application mobile permet un stockage limité de documents et d'autres fonctionnalités spécifiques à une application mobile au-delà de ce qui est permis par le site web ;

- **Catalogue de services (*TBA) :*** microservice recensant tous les microservices de Service (dans le sens Fonctionnalités offertes) et effectuant la répartition et l'orientation des responsabilités relativement aux requêtes d'entrée ;

- **Cryptage (*TBA) :*** microservice visant à rendre les données illisibles pour tout le monde sauf pour les personnes (ou les microservices) qui possèdent les clefs de déchiffrement. Ici, deux méthodes seront possibles et à utiliser en fonction des besoins, le chiffrement symétrique ou le chiffrement asymétrique ;

- **Diffusion de réunion web (*TBA*) :** microservice basé sur le streaming permettant la diffusion d'un flux audio et/ou vidéo en « direct » ou en différé ;

- **Email (*BBA*) :** service d'email typique pour recevoir et envoyer des emails, pour les utilisateurs internes à Astra. Gère les emails transactionnels envoyés par une API ;

- **Enregistrement de réunion web (*TBA*) :** microservice permettant l'enregistrement de session de diffusion de réunion selon le microservice de diffusion de réunion web ;

- **Firewall réseau (*BBA*) :** firewall général du réseau configuré pour protéger les systèmes Astra de la circulation réseau inattendue ou non planifiée. Fournit l'accès port 80 aux systèmes et services exposés alors que les systèmes internes peuvent utiliser différents ports HTTP pour la protection des données ;

- **Gestion des documents Astra (*BBA*) :** microservice gèrant les documents Astra avec des protections permettant uniquement aux utilisateurs internes et externes autorisés d'accéder à des documents spécifiques, sur la base du rôle utilisateur ou en tant qu'utilisateur ayant la permission d'accéder à des documents et dossiers spécifiques ;

- **Gestion d'Emails (*TBA*) :** microservice spécialisé dans la gestion des emails reçus basé sur des processus de dissociation des méta-données (en-tête de message) et des données (corps de message) reçues au sein de chaque email ;

- **Gestion RH (*BBA) :*** système pour gérer les utilisateurs, salariés et prestataires internes à Astra, inclut le rôle, le département et les permissions d'accès de l'utilisateur ;

- **Monitoring (*TBA*) :** microservice spécifique à chaque microservice auquel il est associé pour surveiller et définir une mesure d'activité idoine ;

- **Organisation de réunion Web (*TBA*) :** microservice permettant la planification d'une future réunion web ou initiant la réalisation d'une réunion web immédiate ;
- **Point d'entrée de service (*BBA*) :** dispositif vérifiant que les utilisateurs accèdent uniquement aux services auxquels ils ont accès ;
- **Recherche de réunion web (*TBA*) :** microservice permettant de trouver les informations relatives à une réunion web ayant déjà été effectuée ou ayant été planifiée ultérieurement ;
- **Service de diffusion de réunion web (*TBA*) :** microservice gérant les méta-données associées à la diffusion d'une réunion web spécifique ;**Service d'Emails (*TBA)* :** Service d'email typique pour recevoir et envoyer des emails, pour les utilisateurs internes à Astra. Gère les emails transactionnels envoyés par une API ;
- **Service d'enregistrement de réunion web (*TBA*) :** microservice gérant les méta-données associées à l'enregistrement d'une réunion web spécifique ;
- **Service d'organisation de réunion web *(TBA)* :** microservice gérant les méta-données associées à l'organisation d'une réunion web spécifique ;
- **Service de Gestion RH (*TBA*) :** microservice gérant les méta-données associées au système de gestion RH ;
- **Site Web Astra (*BBA*) :** affiche des informations publiques sur l'entreprise de même que des informations protégées par login basées sur l'organisation et le rôle de l'utilisateur. Le site web est construit comme une application web réactive permettant l'accès depuis une variété d'appareils et de tailles d'écran.
- **Système de gestion des utilisateurs (*BBA*) :** système pour gérant les utilisateurs ayant la permission d'accéder aux services et à d'autres systèmes internes, dont le rôle, l'authentification, et les capacités liées des utilisateurs ;
- **Système de gestion de l'organisation (*BBA*) :** système gérant les organisations ayant accès aux données et services Astra. Les utilisateurs doivent appartenir à une organisation autorisée. Certains services permettent à tout utilisateur d'une organisation d'accéder à des données et documents limités.

Interactions entre composant

L'identification précise des composants étant réalisée, il convient à présent d'analyser les interactions entre ces composants. Pour cela, cette étude se basera sur les diagrammes d'interaction UML. Le but de ces diagrammes est de capturer l'aspect dynamique d'une système, c'est à dire l'instantané du système en cours d'exécution à un moment particulier.

Ainsi, les diagrammes utilisés seront :
- le diagramme de collaboration pour décrire l'organisation structurelle des objets participant à l'interaction ;
- le diagramme de séquence pour capturer l'ordre des messages circulant d'un composant à un autre.

Ces deux types de diagramme sont utilisés pour comprendre :
- le flux de message ou séquence du flux de contrôle d'un objet à un autre ;
- l'organisation structurelle et visuelle d'un système.

Diagramme de collaboration

Le diagramme de collaboration voit son utilité dans la définition des éléments utiles pour obtenir un résultat en spécifiant les RÔLES et les INTERACTIONS des composants précédemment identifiés. Ainsi, le schéma ci-dessous représente ces rôles ainsi que les messages de retours propres à chacun :

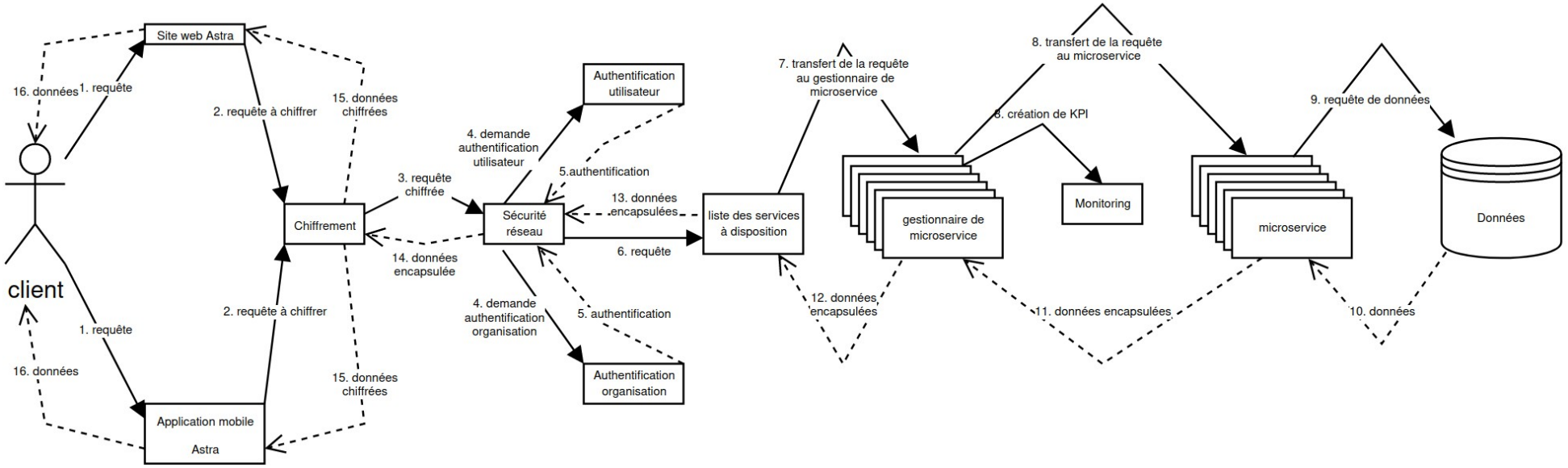
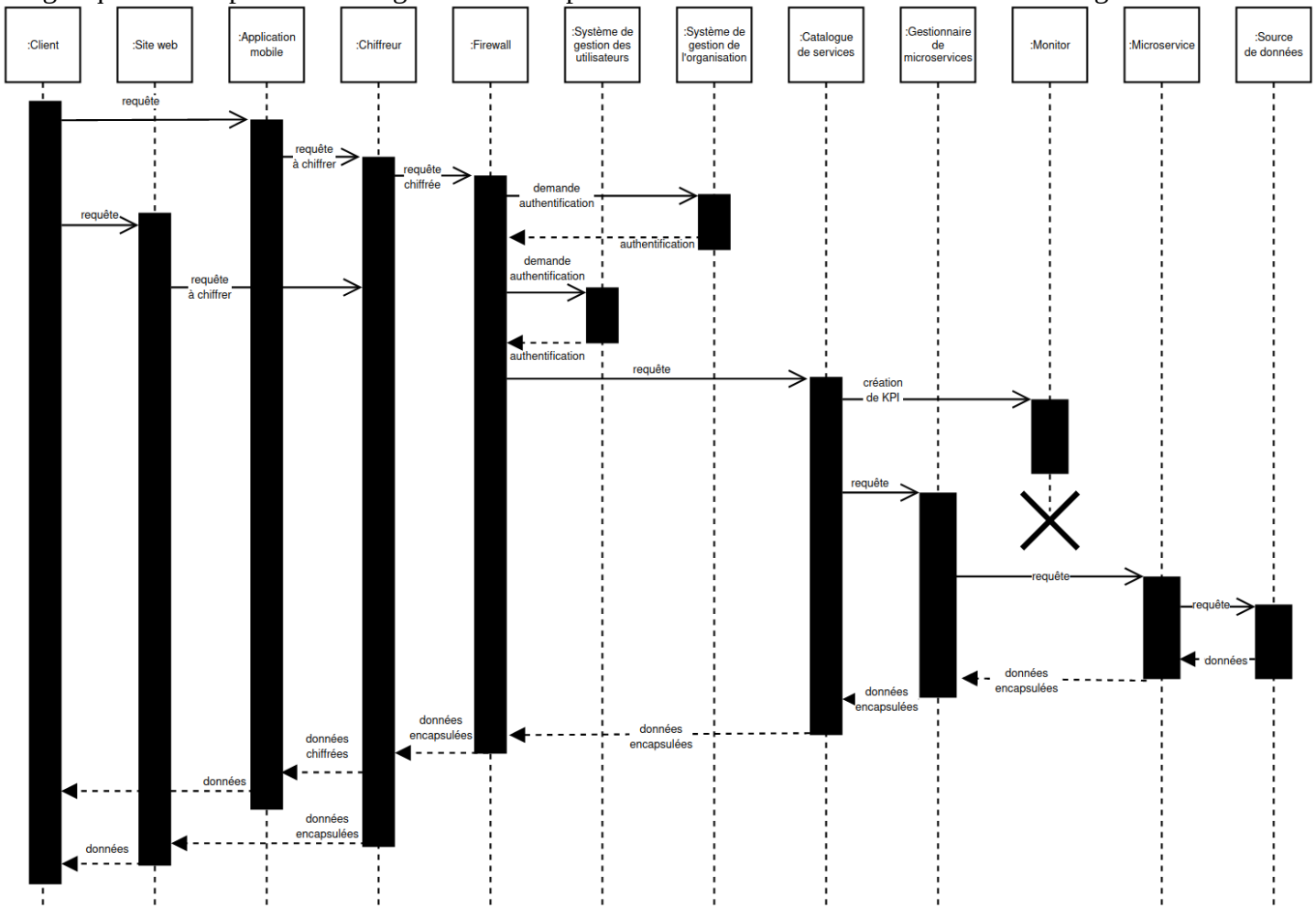


Diagramme de séquence

En se basant sur le diagramme de collaboration élaboré précédemment, il est alors possible d'en déduire un diagramme de séquence. Ce dernier se concentre sur les messages que les composants échangent entre eux pour exercer une fonction avant la fin de la ligne de vie.



Analyse des écarts

L'analyse des écarts a été réalisée dans le framework d'architecture.

Néanmoins ,le tableau ci-dessous recense les composants déjà existant à modifier (MOD), et ceux à créer (ADD) :

Composant	Emplacement	Opération
Application mobile Astra	BBA	MOD
Catalogue de services	TBA	MOD
Cryptage	TBA	ADD
Diffusion de réunion web	TBA	ADD
Email	BBA	MOD
Enregistrement de réunion web	TBA	ADD
Firewall réseau	BBA	MOD
Gestion des documents Astra	BBA	MOD
Gestion d'Emails	TBA	MOD
Gestion RH	BBA	MOD
Monitoring	TBA	ADD
Organisation de réunion Web	TBA	ADD
Point d'entrée de service	BBA	MOD
Recherche de réunion web	TBA	ADD
Service de diffusion de réunion web	TBA	ADD
Service d'Emails	TBA	MOD
Service d'enregistrement de réunion web	TBA	ADD
Service d'organisation de réunion web	TBA	ADD
Service de Gestion RH	TBA	MOD
Site Web Astra	BBA	MOD
Système de gestion des utilisateurs	BBA	MOD
Système de gestion de l'organisation	BBA	MOD

Analyse d’impact

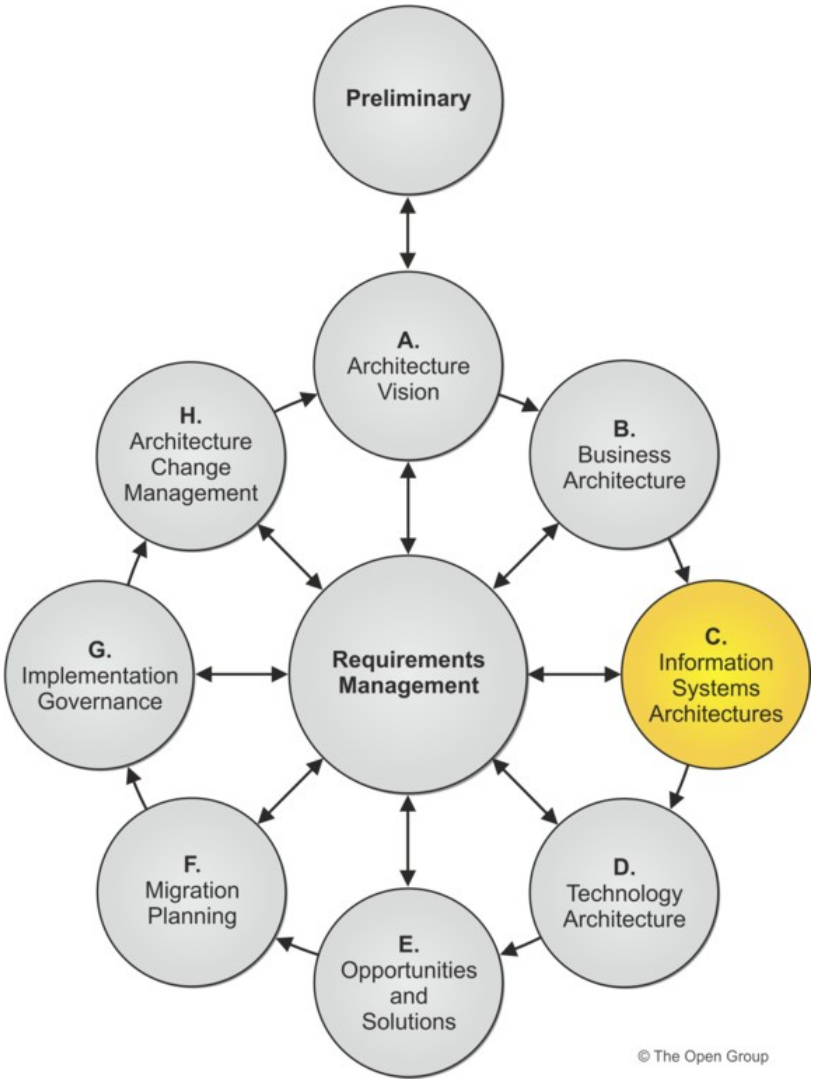
L’analyse d’impact sera structurée selon 3 différentes vues :

- Application Architecture ;
- Technical Architecture ;
- Infrastructure Architecture.

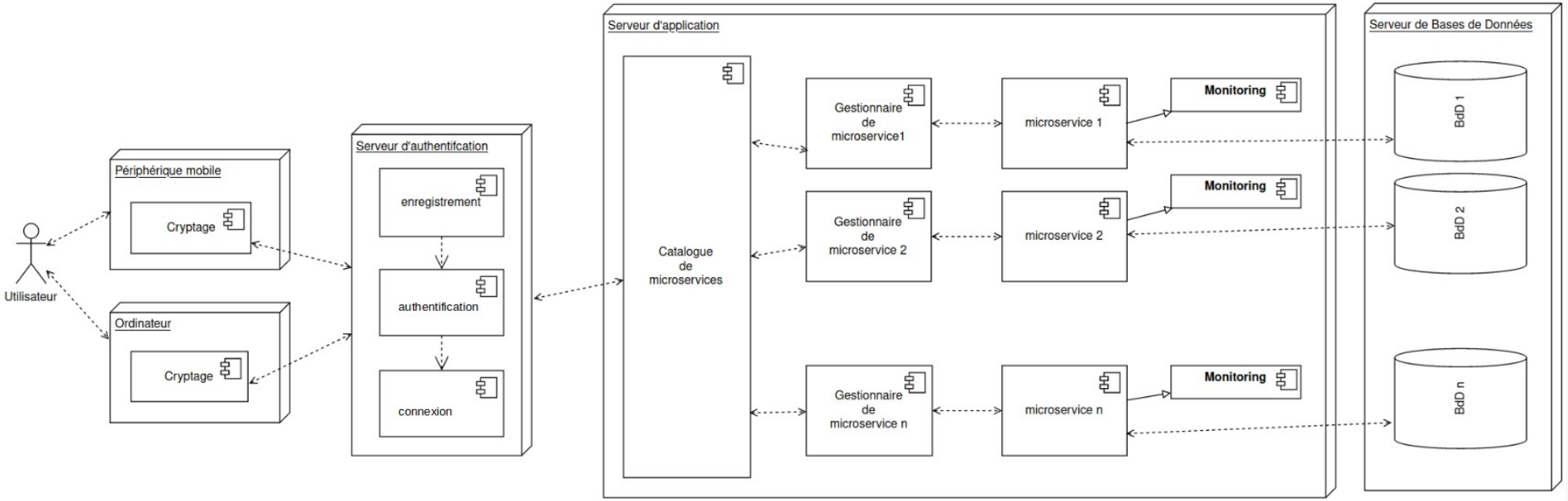
Application Architecture

Les objectifs de la partie architecture d'application de la phase C sont les suivants :

- développer l'architecture d'application cible qui permet l'architecture d'entreprise et la vision de l'architecture, d'une manière qui répond à l'énoncé des travaux d'architecture et aux préoccupations des parties prenantes ;
- identifier les composants candidats de la feuille de route de l'architecture en fonction des écarts entre les architectures d'application de base et cible.



Ainsi l'application se base sur les microservices selon le diagramme de déploiement ci-dessous :



Le schéma ci-dessous présente plusieurs blocs représentant les composants hébergeant une ou plusieurs fonctionnalités. Ces composants sont :

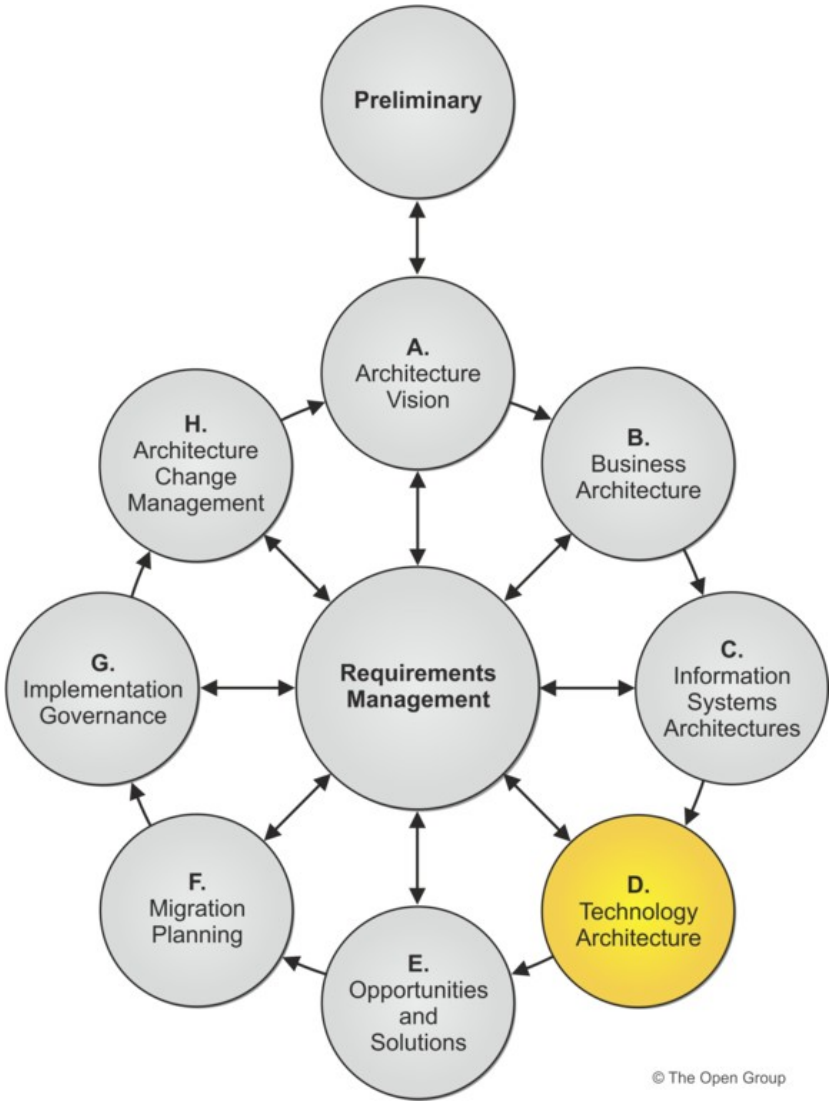
- *Périphérique mobile* : ce module peut représenter une tablette numérique ou un smartphone exécutant une application mobile (il est à noter que l'application mobile contiendra un module de cryptage) ;
- *Ordinateur* : ce module représente un ordinateur implémentant un navigateur internet interrogeant un serveur à travers des requêtes HTTP(S) communiquant via un protocole TCP/IP ;
- *Serveur d'authentification* : ce module représente le serveur chargé des enregistrements des comptes utilisateur, de l'authentification et de la connexion des utilisateurs ;

- *Serveur d'application* : module offrant un contexte d'exécution pour des composants applicatifs ;
- *Serveur de base de données* : également nommé Système de Gestion de Base de Données, ce module hébergera les base de données nécessaires au stockage des données.

Technical Architecture

Les objectifs de la phase D sont de :

- développer l'architecture technologique cible qui permet à la vision de l'architecture, aux activités cibles, aux données et aux blocs de construction d'applications d'être fournis par le biais de composants technologiques et de services technologiques, d'une manière qui répond à l'énoncé des travaux d'architecture et aux préoccupations des parties prenantes ;
- identifier les composants candidats de la feuille de route de l'architecture en fonction des écarts entre les architectures technologiques de base et cibles.

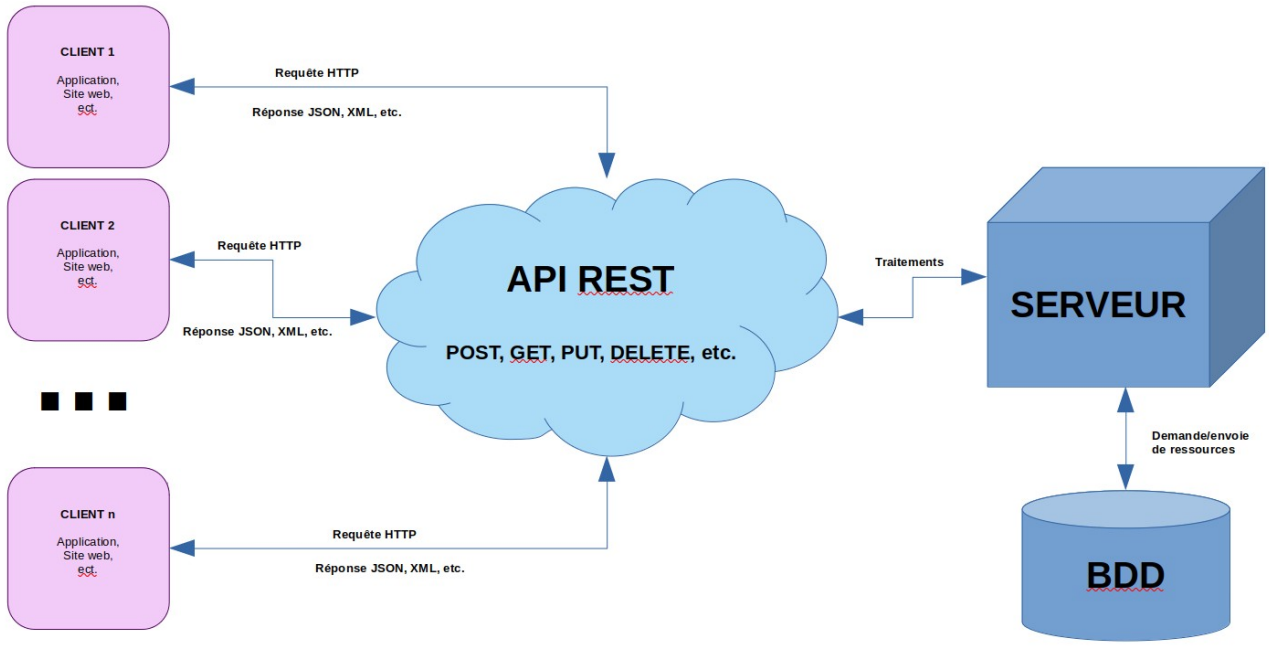


API REST

Dans le contexte de cette étude, le principal rôle d'une API REST est de servir d'intermédiaire entre le client et le serveur. En d'autres termes, c'est l'API REST qui réceptionnera les requêtes émises par le client, les transmet à l'entité demandée au serveur, prend les réponses données par ce dernier puis les retransmet au client.

En faisant appel à une URI, il est possible de récupérer quantité de données. Cela offre comme avantage le fait que les clients puissent augmenter la performance de leur application et de leurs sites ; les serveurs gagnant en lisibilité.

En outre, une API REST aura le formalisme général suivant, peu importe le microservice :



Enfin, une API REST répond à sept critères précis décrits dans les paragraphes qui suivent.

L'architecture client-serveur

Dans une API RESTful, le client et le serveur effectuent leurs traitements indépendamment. Cela veut dire que chaque protagoniste peut manipuler leurs programmes sans impacter les fonctionnalités de l'autre. Tant que la communication entre eux est maintenue, tout va bien. Cette communication s'effectue essentiellement à travers des requêtes HTTP(S).

C'est grâce à ce principe que plusieurs applications situées sur plusieurs plateformes différentes peuvent échanger avec le même API REST et recevoir chacune les réponses attendues.

Sans état

Lors d'un échange entre le client et le serveur, aucune information du client n'est conservée par le serveur, mis à part les informations d'authentification pour celles qui en ont besoin. Chaque requête est traitée indépendamment et le serveur ne sauvegarde pas l'état de la session de l'utilisateur. Seules les données nécessaires à la réalisation de la demande sont à fournir par le client.

Le fait qu'une API REST soit *stateless* permet au serveur de traiter plusieurs requêtes venant de plusieurs utilisateurs différents sans subir de saturation risque de le saturer.

Cacheable

Afin de limiter les appels au serveur et éviter de le surcharger, une API REST doit permettre de sauvegarder certaines informations en cache. Donc, le client peut réutiliser des ressources précédemment fournies par le serveur si cela s'avère utile.

Cependant, l'API REST doit spécifier la durée de mise en cache ainsi que la nature des informations pouvant accepter ce procédé pour maintenir la véracité de ces dernières.

Interface uniforme

Une API REST doit se soumettre à une architecture spécifique que le client et le serveur doivent absolument respecter. Cette uniformité permet de simplifier l'interaction entre les deux parties, améliore la visibilité de celle-ci et renforce également leur autonomie.

Il y a quatre contraintes d'interface qu'il faut respecter dans l'élaboration d'une API REST : l'identification des ressources, le traitement de celles-ci par des représentations, les interactions autodescriptives et l'utilisation de l'hypermédia comme moteur de l'état de l'application. D'ailleurs, nous allons détailler ces deux dernières contraintes sous peu.

Système en couches

Une API REST peut contenir plusieurs systèmes en couches, c'est-à-dire qu'il peut y avoir plusieurs serveurs qui travaillent ensemble pour élaborer la réponse finale demandée par le client. Cette architecture offre plusieurs avantages à savoir une sécurité importante des données échangées par les deux parties, car les interactions se limitent entre les couches qui se suivent, une grande flexibilité et une évolutivité des applications.

Code à la demande

Cette contrainte, qui est optionnelle, veut dire que l'API REST peut fournir du code exécutable aux clients au lieu des réponses conventionnelles. Ces codes peuvent être des scripts ou des applets. De ce fait, les clients peuvent étendre les fonctionnalités de leurs applications ou de leurs sites sans avoir à pré-implémenter certaines méthodes ou modules. Ce qui facilite grandement la vie des programmeurs.

Microservices en Java

Il y a de nombreux langages qui peuvent servir au développement de microservices, tels que Python, C++, Ruby, Golang…

Cette étude ne pourra présenter le développement des microservices dans chacun des langages existant, et il sera souhaitable de choisir celui-ci en fonction de :

- l'existant de l'infrastructure logicielle d'Astra ;
- la matrice des compétences réalisée en vue de constituer de l'équipe de développement.

Cependant, afin de donner un exemple concret relatif au choix du langage de programmation idoine pour le développement de microservices, ce document présentera le cas de l'utilisation de Java.

En effet, la tendance actuelle d'utilisation de Java avec les microservices ne cesse de corroborer ce choix. Java possède un grand nombre de ressources et de bibliothèques et répond à une exigence fonctionnelle de ce projet, à savoir la **portabilité**.

De plus, outre le fait qu'il soit aisé de trouver des développeurs en Java, de nombreux fournisseurs de cloud peuvent facilement faire évoluer les microservices basés sur Java.

Entre autres avantages, les annotations en Java sont très conviviales pour les développeurs et plus faciles à lire. Lors de l'écriture de microservices, les annotations Java facilitent grandement la vie des développeurs. Même s'ils sont proposés par un framework comme *Spring Boot*, cela devient plus simple à maintenir et à étendre. De plus, ces commentaires incluent beaucoup de valeurs dans la lisibilité, en particulier lorsqu'il s'agit de travailler sur des applications complexes.

L'utilisation de la JVM, plate-forme notable en Java, offre aux développeurs d'excellente opportunité quant à l'utilisation de ce langage.

Ainsi, la force de l'architecture Microservice basée sur Java permet aux développeurs d'analyser plus facilement avec d'autres langages ou frameworks également sans avoir de risque d'interprétation très élevé.

Une chose extraordinaire à propos de l'écosystème Java (JVM) est que nous écrivons notre code Java une fois, et nous pouvons l'exécuter fondamentalement sur n'importe quel autre système d'exploitation. Il y a cependant une condition à cette portabilité : il ne faut pas que le code soit compilé avec une version Java plus récente que les versions de votre JVM cible. Si cette condition n'est pas respectée, le développeur se verra confronté à une exception de type *java.lang.UnsupportedClassVersionError* lors de l'exécution.

Spring Boot

Outre les éléments précédents, les frameworks tels que Spring Boot sont reconnus et fournissent une grande aide dans la construction de microservices.

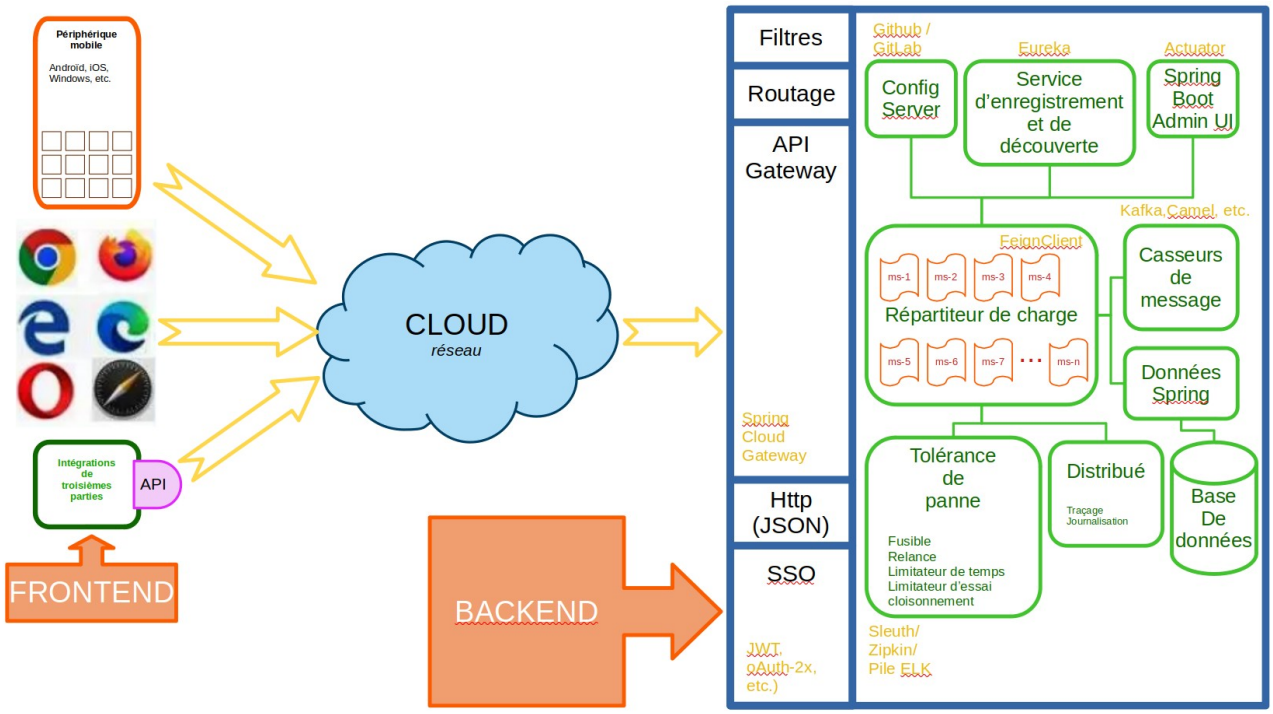
En effet, Spring Boot fournit des serveurs intégrés tels que *Tomcat*, *Jetty*, *Undertow*...les développeurs sont donc grandement accompagner pour créer très facilement des fichiers *.jar* avec un serveur Web intégré, et ainsi exécuté immédiatement leur création. Pas besoin de configuration particulière d'un serveur spécifique pour exécuter l'application.

L'objectif du développement de l'application importe peu, que ce soit pour la configuration, la sécurité, l'API REST, le traitement par lots, le mobile ou même le Big Data, il existe un projet Spring Boot pour faciliter et accélérer le développement.

Généralement, le terme *Spring Boot Microservices* est utilisé pour une application basée sur les microservices en utilisant Java.

Spring Boot n'est rien d'autre qu'un framework pour développer une application prête à l'emploi en Java prenant également en charge le développement de microservices en Java.

De plus, il existe une grande variété de 'démarreurs' dans Spring Boot, aidant à développer facilement une application basée sur des microservices. Aussi, ce document recense ci-dessous la partie théorique des "**Microservices en Java**", pour être en mesure de reconnaître les outils, frameworks, technologies utilisés pour développer un microservice dans son ensemble, tel que démontré dans le schéma ci-dessous :



Flexibilité

Spring Cloud aide les développeurs avec les configurations de service, la découverte de service, la coupure de circuit, l'équilibrage de charge, le traçage, la journalisation distribués et la surveillance.

Il peut également faire office de passerelle API si le besoin s’en fait sentir.

Enregistrement et découverte des services

Lorsque plusieurs services s'exécutent ensemble dans une application, ils doivent se détecter pour communiquer. C'est la première et la plus importante caractéristique des microservices. Afin de rendre cela possible, il devrait exister un support où tous les microservices s'enregistrent eux-mêmes, d’où la présence du module *Catalogue de services* sur le schéma de composant de la *TBA*.

Par la suite, lorsqu'un service souhaite communiquer, il peut se connecter à ce support et découvrir un autre service pour communiquer. Ce support n'est rien d'autre qu’un microservice d’enregistrement et de découverte.

De plus, ceci est similaire au mécanisme RMI de Java, où il est possible de travailler avec un registre central afin que les processus RMI puissent se trouver. Les microservices ont ici la même obligation.

Netflix Eurêka

Netflix Eureka permet de créer une sorte de serveur qui peut enregistrer les microservices et les découvrir lorsque cela est requis par un autre microservice. *Eureka Server* ou *Discovery server* sont des outils permettant cette fonctionnalité.

Chaque microservice s'enregistrera alors sur le serveur *Eureka* avec un identifiant de service et le serveur *Eureka* aura des informations (port, adresses IP, etc.) sur tous les microservices exécutés en tant qu'applications clientes.

De plus, afin de l'implémenter, il est possible de créer un microservice à l'aide du projet *Spring Boot* et d’inclure obligatoirement une dépendance annoté avec *@EnableEurekaServer* .

Ainsi, un microservice annoté avec *@EnableEurekaServer* fonctionnera comme un serveur *Eureka.*

La dépendance associée à ce microservice est décrite en *§Annexe.*

Intra-communication Microservices

Un microservice publié communique avec un autre microservice à l'aide d'*Eureka Server* lui-même.

Ainsi, l'utilisation d'un simple navigateur Internet permet d'obtenir les détails d'un microservice d'*Eureka Server*.

Un microservice annoté avec *@EnableEurekaClient* fonctionnera comme une application *Eureka Client*. Ces clients potentiels peuvent être :

- ***DiscoveryClient*** (client hérité) : il s'agit d'un client de base qui prend en charge la récupération des instances de service à partir du serveur *Eureka* en fonction de l'ID de service en tant que type de liste. Le développeur doit alors choisir manuellement une instance avec un facteur de charge inférieur (pas de concept d'équilibrage automatique pris en charge) dans la liste des instances de service ;

- ***LoadBalancerClient*** (nouveau client) : ce microservice ne récupérera qu'une seule instance de service *d'Eureka* en fonction de l'ID du microservice qui a moins le facteur de charge. *LoadBalancerClient* est une interface dont l'implémentation est fournie par '*Spring Cloud Netflix-Ribbon*' ;

- ***FeignClient/Open Feign*** (Abstract client) : appelée aussi *Abstract Client* ou *Declarative Rest Client, FeignClient* est une interface dont l'exécution génére une classe à l'aide de *Dynamic Proxy Pattern*. Ce concept offre deux annotations : *@EnableFeignClients* à la classe de démarrage et définit l'interface pour un consommateur avec l'annotation *@FeignClient(name="ServiceId"*).

DiscoveryClient et *LoadBalancerClient* communiquent avec *Eureka* pour obtenir les détails de l'instance de service, mais ils ne prennent pas en charge les appels HTTP.

Feign Client agit comme un client combiné : il obtient l'instance de service d'*Eureka Server* et prend en charge l'appel HTTP. En utilisant un des 3 clients ci-dessus, il est alors possible d'obtenir les détails d'instance du microservice d'*Eureka Server* en utilisant l'ID de service comme entrée. Une fois les données de l'instance de service obtenues, il faut utiliser le client HTTP : *RestTemplate* pour effectuer une requête HTTP pour l'application du fournisseur de services.

Il faut cependant prendre en compte que les numéros IP et PORT peuvent être modifiés en fonction du nombre d'instances de système et de déploiement. Il est donc possible lire ces détails à partir d'*Eureka Server* en utilisant *Service Instance*.

Équilibreur de charge

Bien que les versions précédentes de Spring Boot utilisaient le module "*ruban*" pour activer la fonction d'équilibrage de charge, il est désormais possible de l'activer cette fonction d'équilibrage de charge en utilisant "*Feign Client*" et "*Eureka*".

La dépendance associée à ce microservice est décrite en *§Annexe*.

Passerelle API

API Gateway est le point d'entrée et de sortie unique de tous les microservices de l'application.

Étant donné que chaque microservice a sa propre adresse IP et son propre port, et qu'il n'est pas possible de fournir plusieurs détails d'adresse IP et de port au client/utilisateur final, il est indispensable qu'il y ait un seul point d'entrée et de sortie.

C'est également un microservice qui appelle tous les autres microservices utilisant *Eureka*, et il doit également être enregistré auprès du serveur *Eureka* comme n'importe quel autre microservice.

Il génère alors une classe (proxy) basée sur l'ID de service fourni avec le chemin (URL) à l'aide d'un client d'équilibrage de charge.

Ensuite, il sélectionne une instance de service *d'Eureka* et effectue l'appel HTTP. Ceci est évidemment nécessaire car *Eureka Server* lui-même ne peut communiquer avec aucun microservice. En effet, ce dernier sert uniquement à enregistrer et à découvrir les microservices. Un microservice peut communiquer avec un autre à l'aide d'*Eureka Server* uniquement.

Eureka ne prend jamais en charge les appels HTTP vers un microservice.

Le module *API Gateway* aide alors à mettre en œuvre la sécurité, à appliquer des filtres, SSO (Single Sing On), routage, etc.

Serveur proxy Zuul

Afin d'implémenter *API Gateway*, il est possible d'utiliser *Zuul Proxy Server* qui gère toutes les requêtes et effectue le routage dynamique des microservices.

Le routage dynamique ne consiste qu'à choisir une instance de microservice et à effectuer un appel HTTP en fonction de la charge.

Ce serveur est parfois appelé *Zuul Server* ou *Edge Server*.

Pour l'activer et l'exécuter, il est alors nécessaire d'inclure l'annotation *@EnableZuulProxy* au sein de la classe d'application principale pour que l'application *Spring Boot* agisse comme un serveur *Zuul Proxy.*

La dépendance associée à ce microservice est décrite en *§Annexe.*

nota : les nouvelles versions de *Spring Boot* suggèrent d'utiliser *Spring Cloud Gateway* à la place de *Zuul*.

Passerelle Spring Cloud

Spring Cloud Gateway est un moyen simple mais efficace d'acheminer les données de chaque microservice vers les API de transport de données.

Cette passerelle offre également la mise en œuvre de diverses préoccupations transversales telles que la sécurité, la journalisation, la surveillance/les métriques, etc. Elle est construite sur *Spring Webflux* (une approche de programmation réactive).

Les fonctionnalités principales de *Spring Cloud Gateway* permettent de :

- faire correspondre les itinéraires sur n'importe quel attribut de requête ;
- définir les prédicats et les filtres ;
- intégrer les différents microservices à *Spring Cloud Discovery Client* (équilibrage de charge) ;
- réécrire les URI de chemin.

La dépendance associée à ce microservice est décrite en *§Annexe.*

Disjoncteur

Si la méthode réelle d'un microservice génère en permanence une exception, il faut arrêter d'exécuter cette méthode et rediriger chaque demande vers une méthode de secours. Ce concept est ainsi appelé *Disjoncteur*.

Dans cette situation, il faut configurer une méthode factice qui s'exécutera et renverra une réponse au client telle que "*Le service ne fonctionne pas*", ou "*Impossible de traiter la demande pour le moment*", ou encore "*essayez après un certain temps*"… etc.

Il est donc nécessaire d'appeler une telle méthode fictive *Fallback*.

Il existe alors deux types de circuit :

- le circuit ouvert qui consiste à faire passer la demande du client directement à la méthode de secours ;
- le circuit fermé qui redirige la demande du client directement à la méthode de service réelle.

Spring Cloud Hystrix

Afin de mettre en œuvre le mécanisme de disjoncteur, vu dans le paragraphe précédent, il faut nous utiliser *Spring cloud Hystrix*.

Pour cela, il faut inclure l'annotation *@EnableHystrix* à la classe d'application principale pour que l'application *Spring Boot* agisse comme un disjoncteur.

De plus, il faut également ajouter la proprité *@HystrixCommand(fallbackMethod = "DUMMY METHOD NAME")* au niveau de la méthode *RestController*.

La dépendance associée à ce microservice est décrite en *§Annexe.*

nota : le support actuel d'*Hystrix* n'est pas disponible car il est resté en phase de maintenance. L'outil le plus populaire est *Resilience4j* qu'il est alors possible d'utiliser pour tirer parti du mécanisme de disjoncteur pour la tolérance aux pannes.

Tolérance aux pannes et microservices

Afin de mettre en œuvre une tolérance aux pannes complète, y compris pour le disjoncteur, il est préconisé d'utiliser l'API *Resilience4j*.

Cette API comporte plusieurs modules distincts tels que *Rate Limiter*, *Time Limiter*, *Bulkhead*, *Circuit Breaker*, *Retry*, etc.

Il existe plusieurs annotations distinctes pour chaque fonctionnalité comme *@RateLimiter*, *@TimeLimiter*, *@Bulkhead*, *@CircuitBreaker*, *@Retry* respectivement.

Cependant, il est également possible d'implémenter ces fonctionnalités par programme en utilisant le *design pattern Decorator*.

La dépendance pour implémenter *Resilience4j* est décrite en *§Annexe.*

Spring Cloud Config Server

En bref, ce module se nomme également serveur de configuration.

Dans le cas où l'association des mêmes propriétés "*clé = valeur*" se retrouve au sein de chaque microservice, il est alors possible de toutes les définir dans un seul et même fichier de propriétés commun en dehors de tous les projets de microservices.

Pour ce faire, il faut créer un microservice qui aura les droits de modification sur ce fichier de propriétés commun ; ce microservice s'appellera *Config Server*.

Ce fichier de propriétés commun sera alors associé à chaque microservice utilisant *Config Server*.

Les propriétés *"clé=valeur*" courantes sont, par exemple, la connexion à la base de données, l'e-mail, la sécurité, etc.

Cependant, il est possible de le gérer de deux manières à partir de :

- un serveur de configuration externe en utilisant *GitHub, GitLab, Bitbucket*, etc ;
- un serveur de configuration natif en implémentant un serveur de configuration natif simplement en utilisant les lecteurs locaux de notre système, ce qui convient uniquement à l'environnement de développement.

De plus, dans le cas du dernier point ci-dessus, il faudra ajouter l'annotation *@EnableConfigServer* sur notre classe d'application principale pour que notre application *Spring Boot* agisse comme un serveur de configuration.

La dépendance associée à ce microservice est décrite en *§Annexe.*

Traçage et journalisation distribués

En temps réel, une application peut avoir plusieurs microservices. Ainsi, une demande peut impliquer plusieurs microservices jusqu'à l'achèvement de celle-ci.

Le traçage manuel de tous les microservices impliqués dans une requête devient alors une tâche longue et complexe. Ici, le traçage est le processus de recherche d'un chemin d'exécution ou d'un flux de plusieurs microservices impliqués dans le traitement d'une requête.

La journalisation ne sera pas un processus simple car chaque microservice aura son propre fichier journal. Par conséquent, la mise en œuvre d'un mécanisme de traçage et de journalisation distribués devient obligatoire.

Lorsqu'il n'est implémenté une fois, les développeurs n'ont pas besoin de vérifier encore et encore le code pour le flux d'une requête, qu'il s'agisse de test, de débogage ou de développement. Il fournit l'ordre d'exécution des microservices et les lignes de journal associées. Cependant, en cas de recherches multiples, il faut utiliser un module tel que *Sleuth*.

Un détective nommé Sleuth

Afin d'implémenter le module *Distributed Tracing & Logging*, *Spring Cloud API* propose deux composants cloud : *Sleuth* & *Zipkin*.

Sleuth fournit des identifiants uniques pour les flux de requêtes.

Le développeur utilise cet ID pour trouver le flux d'exécution d'une requête.

Il existe deux types d'identifiants :

- l'identifiant de trace est un identifiant unique pour un flux complet (de la demande à la réponse). À l'aide de cet ID, le développeur peut trouver les journaux de tous les microservices impliqués dans le flux ;
- l'identifiant d'étendue ou *Span Id* est un ID unique pour un flux de microservice. À l'aide de cet ID, le développeur peut trouver des messages de journal pour un microservice particulier.

Zipkin

Zipkin fonctionne dans un modèle client-serveur. Au sein de chaque microservice, il est obligatoire d’ajouter cette dépendance avec *Sleuth*.

Il contient *Sampler* qui permet de collecter les données du microservice à l'aide de *Sleuth* et de les fournir au serveur *Zipkin.*

Il ne doit y avoir qu'un seul serveur *Zipkin* centralisé qui collecte toutes les données du client *Zipkin* et les affiche sous forme d'interface utilisateur.

Le développeur doit alors faire une demande et accéder au serveur *Zipkin* pour trouver également l'ID de trace, l'ID d'étendue et le flux lui-même.

Ensuite, le développeur doit ouvrir les fichiers journaux pour voir les lignes de journal liées à l'ID de trace actuel.

Pile ELK

En ce qui concerne le monitoring, *ELK Stack* (comprenant *Elasticsearch, Logstash, Kibana*) est l'un des outils les plus populaires pour surveiller l’application via l'analyse des journaux.

L'analyse de logs est le processus consistant à donner du sens aux messages de logs générés par le système dans le but d'utiliser ces données pour améliorer ou résoudre des problèmes de performances au sein d'une application ou d'une infrastructure.

Dans une vue d'ensemble, Astra analysera les logs pour atténuer les risques de manière proactive et réactive, se conformer aux politiques de sécurité, aux audits et aux réglementations, et permettre également de mieux comprendre le comportement de ses utilisateurs utilisant les applications.

Tableau de bord d'administration

Spring Boot Admin est une application Web qui gère et surveille plusieurs applications de démarrage *Spring* (microservices dans notre cas) et affiche les résultats sous la forme d'un tableau de bord unique.

Généralement, les développeurs de microservices l'utilisent pour surveiller les services Web.

En ajoutant *Spring Actuator* aux applications *Spring Boot* d'Astra, il sera possible d'obtenir plusieurs points de terminaison pour surveiller et traiter les applications *Spring Boot*.

Chaque application *Spring Boot* agit en tant que client et s'enregistre auprès du serveur d'administration *Spring Boot*.

Les points de terminaison *Spring Boot Actuator* fournissent la magie derrière la scène.

Il faudra donc ajouter *Actuator* aux applications client *Spring Boot Admin* et attendre les résultats dans *Spring Boot Admin Server* sous la forme d'un tableau de bord.

De plus, il sera nécessaire d'ajouter l'annotation *@EnableAdminServer* à la classe d'application principale pour que notre application Spring Boot agisse comme un serveur d'administration *Spring Boot.*

La dépendance associée à ce microservice est décrite en *§Annexe.*

Spring Boot Actuator

Spring Boot Actuator fournit des points de terminaison pour la gestion et la surveillance des applications *Spring Boot*.

Tous les points de terminaison de l'actionneur sont sécurisés par défaut.

Les points de terminaison ne sont que les métadonnées pour accéder à un service Web tels que le chemin (*/emp/data*), la méthode http(*GET*), l'input(*String*), l'output(*JSON*) etc.

Afin d'activer *Spring Boot Actuator* dans une application, nous avons besoin d'ajouter la dépendance *Spring Boot Starter Actuator* dans le fichier *pim.xml.*

La dépendance associée à ce microservice est décrite en *§Annexe.*

Certains des paramètres d'actionneur populaires et importants sont indiqués ci-dessous. Il suffira de les entrer dans un navigateur Web pour surveiller le comportement des applications de chaque microservice :

- **env** : permet de connaître les variables d'environnement utilisées dans l'application ;
- **beans** : permet de visualiser les *beans Spring* et leurs types, portées et dépendances utilisés dans l'application ;
- **health** : permet de visualiser la santé de l'application en indiquant si chaque microservice a bien démarré ou pas. De plus, il fournit également des données telles que la mémoire pour l'espace disque, le statut PING, etc ;
- **info** : permet d'obtenir des informations sur le microservice actuel à d'autres clients, utilisateurs, ou développeurs ;
- **trace** : permet de visualiser la liste des traces de vos *endpoints Rest* ;
- **metrics** : permet de visualiser les métriques de l'application telles que la mémoire utilisée, la mémoire libre, les classes, les threads, la disponibilité du système, etc.

Infrastructure Architecture

Les objectifs de la phase G sont de :

- Assurer la conformité avec l'architecture cible par des projets de mise en œuvre ;
- Exécuter les fonctions de gouvernance de l'architecture appropriées pour la solution et toute demande de changement d'architecture axée sur la mise en œuvre.

L'essor du Big Data, du cloud computing et de la multiplicité des appareils distants a exercé une pression intense sur l'infrastructure.

Les problèmes sans cesse croissants d'intégration, de sécurité et de gestion de l'information sont autant de défis auxquels sont confrontées les organisations modernes.

L'architecture d'infrastructure est une approche structurée pour gérer ce nouveau paysage d'infrastructure.

Une architecture d'infrastructure réussie clarifie quelle fonctionnalité est requise, par qui et à quel niveau de qualité.

Cette analyse facilite la prise de décision éclairée dans le processus de conception, de construction et de gestion de l'infrastructure d'une organisation. Il permet d'utiliser rapidement les bons modèles, ce qui conduit à des résultats de qualité.

Une excellente architecture d'infrastructure garantit la confiance : dans les SLA, les OLA et dans les clients qui comptent sur des délais de livraison précis, une facturation précise et opportune et une prestation de services fiable...

Néanmoins, dans le cadre de ce projet, il sera nécessaire aux parties prenantes décisionnaires de fournir les informations d’infrastructure supplémentaires sur l’infrastructure actuelle d’Astra, afin de réaliser une telle étude.

ANNEXES

Liste des dépendances

Netflix Eurêka

```
<dépendance>
<!-- enregistrement des microservices par Eureka-->
<groupId>org.astra.cloud</groupId>
<artifactId>spring-cloud-starter-eureka-server</artifactId>
</dépendance>
```

Équilibreur de charge

```
<dépendance>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-openfeign</artifactId>
</dépendance>
```

Serveur proxy Zuul

```
<dépendance>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-zuul</artifactId>
</dépendance>
```

Passerelle Spring Cloud

```
<dépendance>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-passerelle</artifactId>
</dépendance>
```

Spring Cloud Hystrix

```
<dépendance>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-hystrix</artifactId>
</dépendance>
```

Resilience4j

```
<dépendance>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dépendance>

<dépendance>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artifactId>
</dépendance>

<dépendance>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-aop</artifactId>
</dépendance>
```

Spring Cloud Config Server

```
<dépendance>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-config-server</artifactId>
</dépendance>
```

Tableau de bord d’administration

```
<dépendance>
<groupId>de.codecentric</groupId>
<artifactId>spring-boot-admin-starter-server</artifactId>
</dépendance>
```

Spring Boot Actuator

```
<dépendance>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-admin-starter-actuator</artifactId>
</dépendance>
```


Astra Recherche