Projet
Foosus géoconscient

Spécification des Conditions requises pour l’Architecture

Auteur(s) et contributeur(s)

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

Historique des modifications et des révisions

N° version	Date	Description et circonstance de la modification	Auteur
1.0	05/07/2022	Création du document	Gérald ATTARD

Validation

N° version	Nom & Qualité	Date & Signature	Commentaires & Réserves
1.0	Ash Callum CEO de Foosus		

Tableau des abréviations

Abr.	Sémantique
ABB	Architecture Building Block (trad. *bloc de construction d'architecture*)
ABF	Atomic Business Function (trad. *fonction métier atomique*)
ADM	Architecture Development Method (trad. *méthode de développement d'architecture*)
API	Application Program Interface (trad. *interface de programmation d'application*)
DDD	Domain-Driven Design (trad. *conception dirigée par le domaine*)
IoT	Internet of Things (trad. *Internet des objets*)
IT	Information Technology (trad. *technologie de l'information*)
ITSM	Information Technology Service Management (trad. *gestion des services de l'information*)
KPI	Key Performance Indicator (trad. *indicateur de performance clé*)
MSA	Microservices Architecture (trad. *architecture des microservices*)
OAS	Open API Specifications (trad. *spécifications d'API open source*)
SaaS	Software as a Service (trad. *logiciel en tant que service*)
SGVM	SOA Governance Vitality Method (trad. *Méthode de gouvernance Vitality*)
SLA	Service Level Agreement (trad. *accord de niveau de service*)
SLI	Service Level Objective (trad. *objectif de niveau de service*)
SLO	Service Level Indicator (trad. *indicateur de niveau de service*)
SOA	Service-Oriented Architecture (trad. *architecture orientée services*)
SRP	Single Responsability Principle (trad. *principe de responsabilité unique*)

Table des matières

I. Objectif du document

Le but de ce document est de fournir des conseils sur la façon d’appliquer l'architecture de microservices (MSA) préconisée pour développer, gérer et gouverner les microservices nécessaires à l’application FOOSUS.

En outre, cela favorisera la compréhension partagée du processus de création de MSA selon les principes de l'architecture distribuée, et renforcera significativement l'alignement entre les cultures d'entreprise et les technologies de l'information.

L'adoption ultérieure de MSA en tant que style architectural dans un environnement cloud entraînera l'utilisation de la méthode, du méta-modèle, des références et d'autres installations basées sur le référentiel TOGAF.

Il faudra tenir compte des nombreuses caractéristiques de MSA similaires à l'architecture orientée services (SOA) ; néanmoins, ce document est dédié à MSA uniquement.

De plus, afin d’utiliser nominalement les microservices, ceux-ci seront systématiquement introduits par API REST. Aussi, ce document fournira des préconisations relatives à ce type d‘interface.

II. Mesure du succès

La mesure de la réussite de ce projet sera effectué au travers de différents indicateurs en adéquation avec les visions des différentes parties prenantes concernées, à savoir :

Rôle	Indicateurs	périodicité
CEO, CFO	• Taux d'inscription des nouveaux utilisateurs ; • Nombre de commandes réalisées ; • Nombre de fournisseurs co-traitant ; • Nombre de produits proposés ; • Nombre de paiement reçus ; • Capacité publicitaire, exposition de la solution ; • Taux d'interruption de services ; • Taux d'incidents de production.	mensuelle
CIO, CPO, CFO	• Nombre de fonctionnalités demandées ; • Nombre de fonctionnalités répondues/réalisées ; • Nombre de technologies utilisées actuellement ; • Nombre de technologies prises en charge par MSA.	mensuelle
CMO, CIO, CPO, CFO	• Dette technique, nombre de User's Story émises ; • Dette technique, nombre de User's Story réalisées.	Par sprint
CMO, CPO, Directeur des Opérations	• Nombre de MSA en fonction ; • Nombre de total tests unitaires réalisés ; • Nombre de tests unitaires au *Vert* ; • Nombre de tests unitaires au *Rouge*.	Par sprint

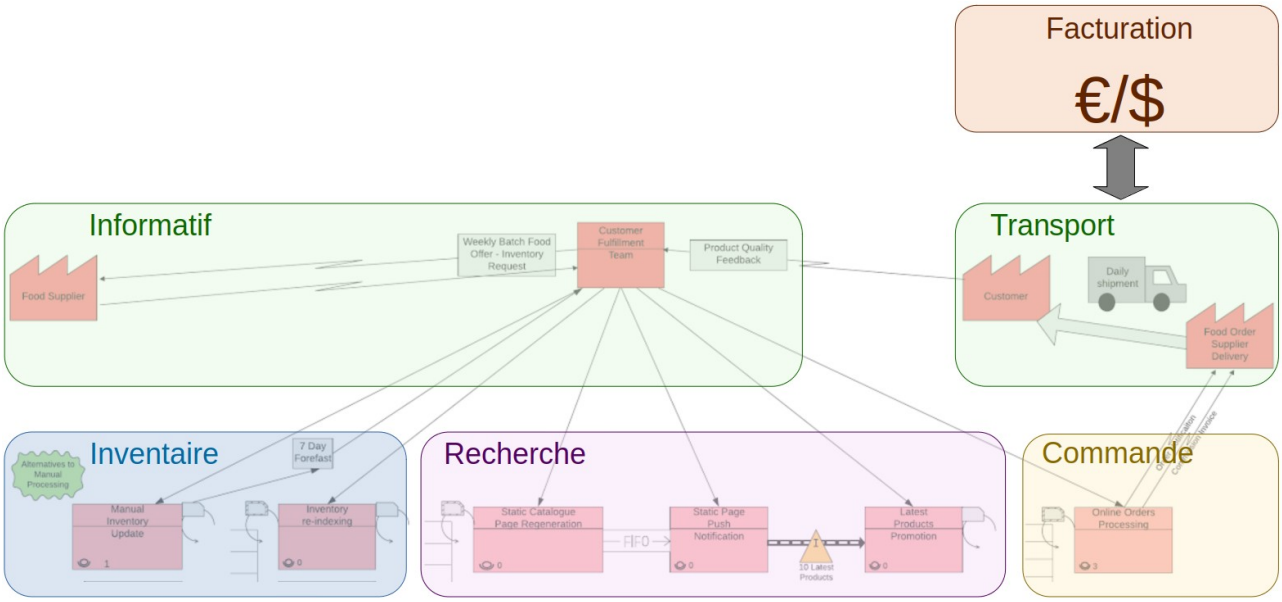
En plus des indicateurs spécifiques ci-dessus, d'autres indicateurs plus génériques seront à fournir à toutes les parties prenantes, à savoir :

- **délai de mise en œuvre (LT)** : temps total qu'il faut pour accomplir une tâche (LT=CT+QT) ;
- **temps de cycle (CT)** : durée nécessaire pour accomplir une tâche (temps à valeur ajoutée) ;
- **temps de queue (QT)** : temps sans valeur ajoutée entre les temps de cycle ;
- **temps de tâche** : rythme auquel un produit/service est produit/fourni ;
- **taux de défaut** : pourcentage d'une sortie qui ne répond pas aux spécifications ;
- **taux de change avec le temps (C/O)** : temps qu'il faut pour commencer à travailler sur l'unité suivante après avoir terminé l'unité actuelle.

III. Exigences architecturales

III.A. Issues de l'existant

A la base de l'application FOOSUS, six fonctions constituent le cœur de métier de l'application, telles que schématisées ci-dessous :



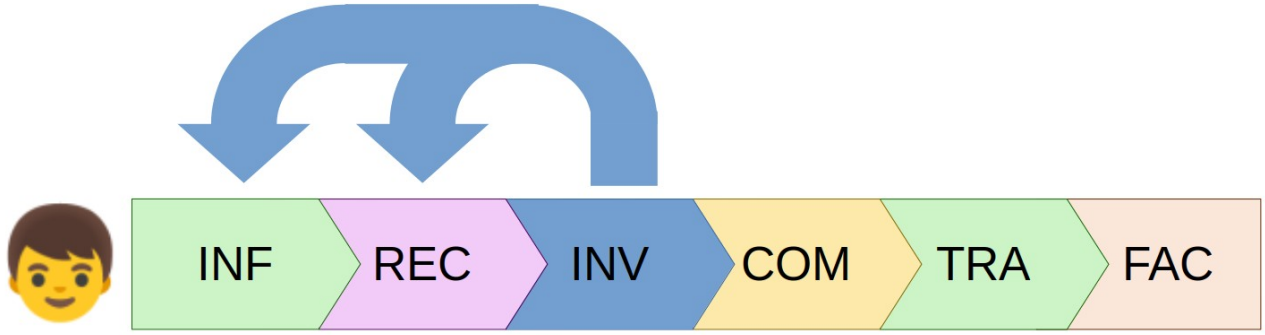
Ainsi, tel que schématisé ci-dessus, il est possible d'identifier six fonctions principales :

- **Inventaire** : ce module se charge de répertorier les offres et les quantités de produits disponibles chez un fournisseur référencé, ainsi que les besoins de client enregistré ;
- **Recherche** : ce module va croiser les données issues de la fonction ***Inventaire*** pour identifier les fournisseurs pouvant répondre aux besoins d'un client. Puis le système va ensuite comparer le positionnement géographique des fournisseurs identifiés avec celui du client, et proposer les fournisseurs qui sont les moins éloignés du client lui-même ;
- **Commande**: ce module va s'occuper d'informer le client de la réservation des produits choisis auprès des fournisseurs, tout en informant ces derniers que leur(s) produit(s) ont été vendus et donc réservés par un client. Il s'occupera également de fournir un état relatif au suivi de livraison des produits commandés ;
- **Transport** : dans la continuité du processus de ***Commande***, ce module prend en charge les modalités de livraison des produits commandés et donc réservés par le client ;
- **Facturation** : suite à la livraison des produits chez le client, le système en accord avec l'équipe Finance va émettre une facture et l'envoyer chez le client ou chez le fournisseur, en fonction du mode de livraison choisi par le client. Le système devra alors suivre l'avancée des transactions financières et informer toutes les parties de la finalisation du paiement ;

- **Informatif** : ce module a deux rôles principaux. D'un point de vue de FOOSUS, il va permettre à l'équipe commerciale de mettre à jour les produits disponibles chez les fournisseurs référencés. D'un point de vue clientèle, ce module va offrir la possibilité de faire toute sorte de retours d'expérience , aussi bien négatifs que positifs, à propos du vécu du client quant a l'utilisation de l'application FOOSUS, aux différents fournisseurs mandatés, au temps de livraison...

Ces fonctions principales devront être reprises au sein de la nouvelle architecture.

D'un point de vue métier, il est possible d'ordonnancer les fonctions décrites en suivant le cycle ci-dessous :



...avec comme légende :

- **INF** : le module ***Informatif*** ;
- **REC** : le module de ***Recherche*** ;
- **INV** : le module d'***Inventaire*** ;
- **COM** : le module de ***Commande*** ;
- **TRA** : le module de ***Transport*** ;
- **FAC** : le module de ***Facturation***.

III.B. Issues des MSA

Les microservices sont les principaux ABB d'une MSA et possèdent trois caractéristiques clés :

- **Indépendance des services** : chaque microservice est indépendant des autres microservices ; chaque microservice est développé, déployé et modifié indépendamment ;

- **Responsabilité unique** : chaque microservice est associé à une activité commerciale atomique (unique) dont il est responsable ;

- **Autonomie** : un microservice est une unité déployable autonome et indépendante englobant toutes les ressources informatiques externes (par exemple, les sources de données, les règles métier) nécessaires pour prendre en charge l'activité commerciale unique.

A ces trois caractéristiques, il est alors primordial d'y associer les trois principes suivants :

- **Déclaration** : un microservice est indépendant des autres microservices.

- **Raisonnement** : l'indépendance des services permet un développement et un déploiement rapides des services. Cela permet une grande évolutivité grâce à l'instanciation de services parallèles indépendants. Cette caractéristique fournit également de la résilience; un microservice est autorisé à échouer et ses responsabilités sont prises en charge par des instanciations parallèles (du même microservice), qui ne dépendent pas d'autres services. Lorsqu'un microservice échoue, il n'interrompt pas les autres services ; ce principe est la base de la robustesse et de la maintenabilité.

- **Conséquence** : l'indépendance de la conception et de l'exécution des services est requise. L'entreprise Foosus ayant besoin d'évolutivité et de résilience de la fonction commerciale, celles-ci doivent être considérées comme des considérations primordiales : MSA fournit un moyen d'atteindre ces caractéristiques.

Une MSA a pour objet de diviser une application en fonctionnalités encapsulées au sein de services autonomes. Chacun de ces services est géré et évolue indépendamment des autres services.

Les Microservices peuvent être mis à jour, étendus et déployés indépendamment les uns des autres et, par conséquent, beaucoup plus rapidement tout en limitant le risque d'une mise en péril de l'ensemble de l'applicatif.

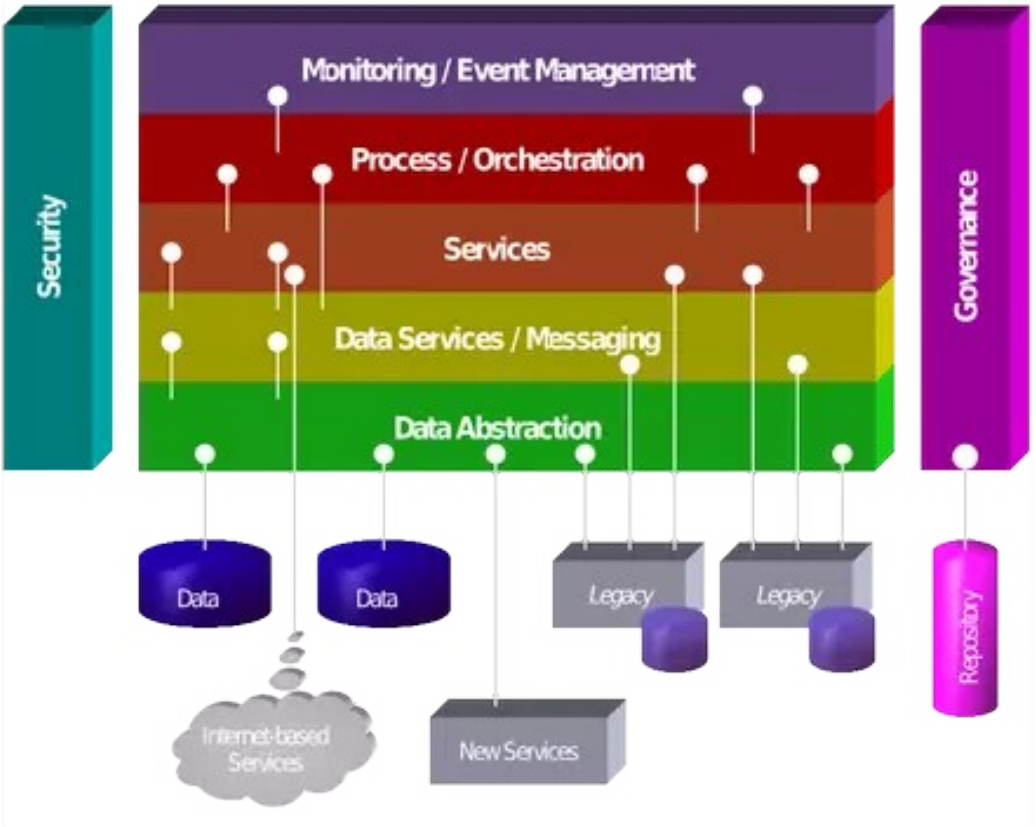
Les Microservices peuvent communiquer entre eux sans état, à l'aide d'interfaces de programmation d'application (API) indépendantes de tout langage.

Le choix technologique dans une MSA est donc totalement ouvert. Il dépend majoritairement des besoins, de la taille des équipes et de leurs compétences.

Ces architectures d'applications faiblement couplées qui reposent sur des Microservices avec des APIs et des pratiques DevOps constituent la base des applications cloud-native.

Le développement cloud-native est une manière d'accélérer la création des nouvelles applications, d'optimiser les applications existantes, d'harmoniser le développement et d'automatiser la gestion pour l'ensemble des clouds privés, publics et hybrides.

Le schéma ci-dessous représente une vision simplifiée de l'implémentation des Microservices :



Dans le schéma précédent, il est possible de dénombrer sept parties principales :

- le **Gestionnaire d'évènement (Monitoring)** : cette partie représente une activité de surveillance et de mesure des différents microservices en action. C'est une partie de permettant la supervision globale du système ;
- le **Planificateur de tâches (Process/Orchestration)** : cette partie permet de planifier l'exécution automatique et périodiques de microservices et ainsi de les ordonnancer en vue de rendre leur exécution cohérente et pertinente vis à vis du système global ;
- les **Services** : cette partie correspond aux activités métier réalisées par chaque microservice ;
- les **Services de données** : cette partie propose une technologie évoluée d'échange de données via un réseau et dont les données échangées sont regroupées dans un Espace Global de Données distribué dans le réseau pour éviter les problématiques liées aux goulots d'étranglements et aux pannes de gestionnaires de données ;
- l'**Abstraction de données** : cette partie est représentée par un modèle mathématique de types de données définissant le comportement d'un microservice en fonction de la sémantique d'une donnée d'un point de vue utilisateur ;

- la **Sécurité** : la sécurité globale d'un système utilisant des microservice est appliqué indépendamment à chacun d'entre eux afin de leur assurer la confidentialité des données qu'ils traitent, leur authenticité, leur intégrité et leur disponibilité. Chacune des opérations réalisées par un microservice est aussi tracée et imputée de façon unitaire ;

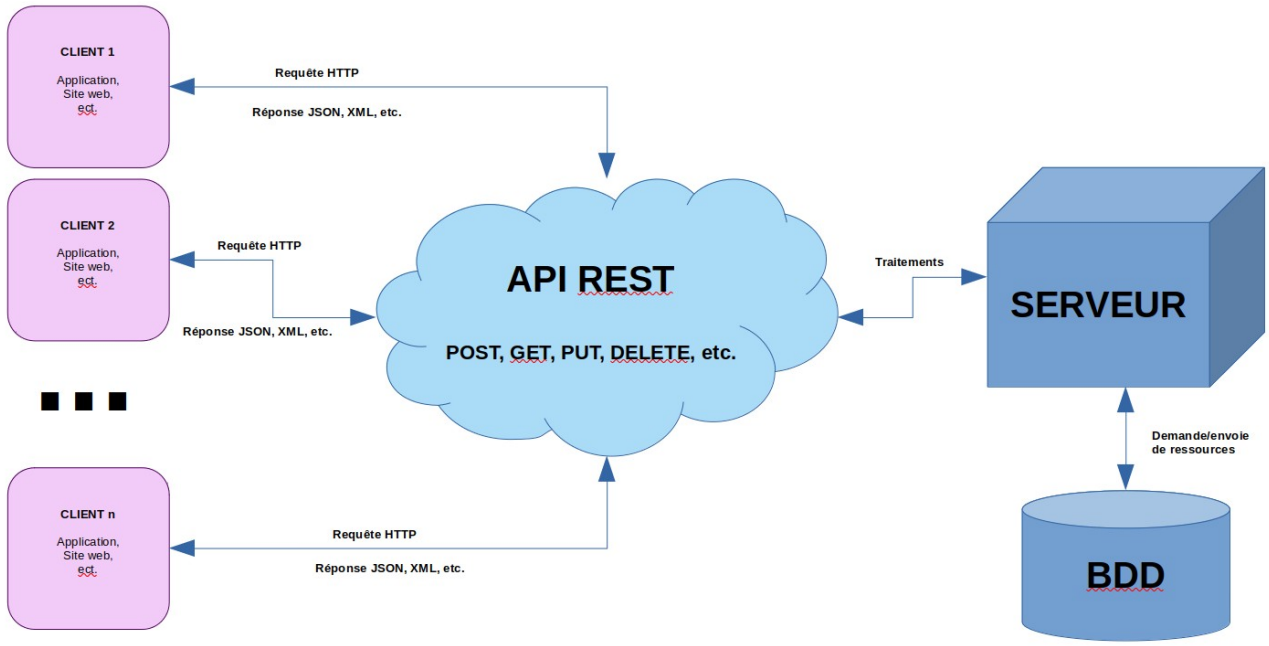
- la **Gouvernance** : cette partie définit le système d'information global utilisant l'ensemble des microservices. Cette démarche permet de définir la manière dont le système d'information, au travers des microservices, contribue à la création de valeur en précisant le rôle de chaque microservice utilisé.

III.C. Issues de l'API REST

Dans le contexte de cette étude, et dans la continuité de l'utilisation de MSA, 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* ou *RESTful* répond à six critères précis décrits dans les paragraphes qui suivent.

III.C.1. 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.

III.C.2. 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.

III.C.3. 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.

III.C.4. 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 auto-descriptives 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.

III.C.5. 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.

III.C.6. 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.

IV. Contrats de services

IV.A. Métiers

IV.A.1. Accords de niveau de service

IV.A.1.a. Fonction Inventaire

IV.A.1.a.i. Client

Du point de vue d'un client, la fonction ***Inventaire*** permet de conserver une liste des besoins réguliers/habituels d'un client, en indiquant la disponibilité de ceux-ci.

IV.A.1.a.ii. Equipe FOOSUS

Du point de vue de l'équipe commerciale de FOOSUS, ce module présente tous les produits disponibles chez tous les fournisseurs référencés.

De plus, il faudra distinguer deux types de date associée à chaque produit :

- la **date de fraîcheur** indiquera si le produit peut être considéré comme frais, c'est à dire si le produit est susceptible de conserver toutes ses caractéristiques nutritives et gustatives, comme s'il venait d'être ramassé/produit ;

- la **date de péremption** indiquera jusqu'à quelle date le produit peut être consommé ; cette date peut être ultérieure à la date de fraîcheur.

Ainsi, tous les produits proposés au sein de l'application FOOSUS respecteront drastiquement les dates de péremption de chacun d'entre eux. Ces dates seront systématiquement affichées au sein des fiches descriptives des produits. Après que cette date de péremption ait été dépassée, le produit ne devra plus être proposé au client par l'application.

En outre, une date de fraîcheur pourra être indiquée sur les fiches descriptives des produits disponibles. Cette date de fraîcheur sera une donnée purement informelle, non contractuelle et ne sera valable que jusqu'à la commande du produit, c'est à dire jusqu'à la réservation du produit chez le fournisseur. Au-delà de cette étape, cette information ne sera plus pérenne.

Néanmoins, des propositions de tarif dégressif pourront être émises, à la discrétion de l'équipe commerciale, de manière éthique et responsable, afin de limiter autant que faire se peut le gaspillage de denrées.

IV.A.1.b. Fonction Recherche

Cette fonction permettra à tous les utilisateurs de rechercher des produits ou des fournisseurs. Dans l'avenir cette fonction pourra également être étendues à la recherche de service(s) proposé(s) par les fournisseurs.

Ainsi, chaque recherche pourra être effectuée selon plusieurs critères, dont les principaux sont listés ci-après :

- par type de produit ;
- par disponibilité de produit ;
- par localisation géographique ;
- par fournisseur.

D'autres critères de recherche pourront être ajoutés à la discrétion de l'équipe FOOSUS.

IV.A.1.c. Fonction Commande

IV.A.1.c.i. Client

Du point de vue du client, le module de ***Commande*** réserve les produits auprès des fournisseurs.

Cette réservation a comme conséquence de décrémenter la quantité du type de produit identifié, du montant de la réservation dudit client. Ainsi, vis à vis de l'affichage des stocks restants, il sera alors affiché la quantité restante par la fonction ***Recherche***.

Par exemple, si un fournisseur met à disposition 5 tomates et qu'un client '*C*' lui en commande 2, la quantité affichée dorénavant par la fonction ***Recherche*** ne fera mention que de 3 tomates, puisque 2 tomates ont été réservées par le client '*C*'.

Ainsi, l'entrée du processus ***Commande*** est la sortie du processus ***Inventaire***, et la sortie du processus ***Commande*** est l'entrée du processus ***Transport***.

IV.A.1.c.ii. Fournisseur

Du point de vue du fournisseur, le module de ***Commande*** est avant un outil pour évaluer la quantité de produits réservés par les clients. Cette quantité, préalablement déclarée dans le module ***Informatif***, devra être fournie aux clients ayant exprimé le besoin.

IV.A.1.d. Fonction Transport

Suite à la commande d'un produit et donc sa réservation chez un fournisseur, le transport du produit dépendra de deux cas d'utilisation possibles, après la ***Commande*** :

- le client se déplace physiquement chez le fournisseur pour prendre en charge le produit : le client s'acquitte du paiement de la marchandise directement auprès du fournisseur. Une facture sera alors envoyée directement au fournisseur, indiquant les montants relatifs aux accords d'utilisation de la plateforme FOOSUS, au prorata de la commande facturée au client.
- Le client ne peut pas se déplacer et la livraison est prise en charge par un prestataire de transport local. Le choix de ce dernier peut soit être défini à la convenance du fournisseur, soir être choisi par FOOSUS si le fournisseur en émet le souhait. Il est également tout à fait envisageable que le fournisseur fasse lui-même les livraisons.

Dans tous les cas, peu importe le moyen de transport choisi, cette information DEVRA être affichée par la fiche récapitulative de la ***Commande***, au sein de l'application FOOSUS.

En outre, dans la continuité du processus de ***Commande*** correspondant à la réservation des produits chez le(s) fournisseur(s), ***la livraison ou la prise en charge des produits ne devra pas excédée deux jours*** pour garantir leur fraîcheur. Aussi, au cas où cette durée de livraison serait dépassée, si le client a choisi :

- **la livraison à domicile** : le transporteur et/ou le fournisseur devront remplacer les produits à livrer par des produits frais et effectuer une nouvelle livraison chez le client dans les meilleurs délais ;
- **le retrait des produits chez le fournisseur** : la commande sera annulée, et un montant au prorata de la valeur de la commande sera quand même facturée au client. Ce prorata sera à convenir au cas par cas entre FOOSUS et les fournisseurs de produits. L'application FOOSUS affichera le prorata négocié pour chaque fournisseur.

IV.A.1.e. Fonction Facturation

Lorsque le client aura pris en charge les produits identifiés au sein de la ***Commande***, une facture correspondant au montant des produits sera systématiquement émises et envoyées :

- *au client*, si ce dernier s'est fait livré à domicile ;
- *au fournisseur*, si le client s'est physiquement déplacé pour prendre livraison des produits réservés.

Ce module de facturation effectuera également le suivi des transactions et sera capable d'effectuer des rappels de paiement au bout d'une certaine durée d'impayé.

IV.A.1.f. Fonction Informatif

IV.A.1.f.i. Client

Du point de vue du client, le module informatif servira à mettre à jour les fiches descriptives des produits présents dans l'inventaire de celui-ci.

Ces produits auront été présélectionnés par le client lui-même et feront partie de l'***Inventaire*** de celui-ci en tant qu'articles déjà sélectionnés et/ou précédemment acquis par ***Commmande***.

Le module ***Informatif*** tiendra à jour les fiches descriptives de tous les produits disponibles par l'application FOOSUS, dont notamment celles des produits contenus dans l'***Inventaire*** du client.

Lorsque le client souhaitera effectuer une ***Commande***, celui-ci sélectionnera les produits désirés en les disposant dans un **panier virtuel**. Une fois la sélection de produit terminée, c'est ce panier virtuel qui sera transmis au module de ***Commande*** pour réserver les produits auprès du/des fournisseur(s).

Un article ayant été, au moins une fois, sélectionné pour faire partie du **panier virtuel** se retrouvera alors répertorié dans l'***Inventaire*** lors de tous les prochains accès du client à l'application FOOSUS.

Le client pourra également décider de retirer un article de son ***Inventaire*** de façon manuelle.

En outre, ce module servira également de panneau publicitaire en informant les clients des arrivages et des promotions saisonnières annoncées par l'équipe FOOSUS.

IV.A.1.f.ii. Equipe FOOSUS

Du point de vue de l'équipe FOOSUS, le module ***Informatif*** servira à annoncer les arrivages de produits et les annonces promotionnelles des produits.

Ce module tiendra également à jour les quantités et les disponibilités de tous les produits proposés par tous les fournisseurs référencés sur la plateforme FOOSUS.

IV.B. Applicatifs

Les **objectifs de niveau de service (SLO)** définissent des cibles pour certaines métriques, tandis que les **indicateurs de niveau de service (SLI)** sont le reflet continu de ces métriques ; les deux termes sont étroitement liés.

le **SLO** est un objectif ou une condition idéale qu'un service doit atteindre.

Relativement à l'axiome précédent, le **SLI** sera la valeur quantitative représentant le **SLO** ; cet indicateur se déplacera au fil du temps.

Ainsi, lorsque qu'une certaine valeur définie par un utilisateur est atteinte, une alerte se déclenche et oblige le responsable du **SLO** à intervenir.

Dans le cadre d'un écosystème de gestion des niveaux de service, les **SLO** et **SLI** ne peuvent exister l'un sans l'autre.

IV.B.1. Objectifs de niveau de services (SLO)

Les SLO nécessaires à ce projet devront être en adéquation avec les caractéristiques suivantes :

- **réalisme** : un SLO de 100 % est inaccessible sur le plan fonctionnel et représente un objectif théorique, pas un SLO utile. Les SLO doivent représenter un niveau de performance minimum acceptable et ne jamais être considérés comme impossibles ou inatteignables ;
- **pertinence** : les SLO doivent être des métriques présentant réellement une importance pour FOOSUS. Nous pourrions prendre pour exemple l'utilisation d'un processeur de serveur comme un SLO inutile. En effet, dans le dernier exemple, ce SLO ne représente aucune importance pour les utilisateurs et les activités de FOOSUS ;
- **capacité de mesure** : si une métrique ne peut pas être mesurée avec précision, elle ne sera pas utile en tant que SLO ;
- **capacité de contrôle** : un SLO qui fixe un seuil maximum au nombre de foudroiements sur le centre de données, par exemple, n'aurait aucune plus-value pour FOOSUS. Un SLO relatif au nombre d'enregistrements de clients mensuels serait pertinent pour assumer l'attractivité de l'application FOOSUS ;
- **intelligibilité** : certaines mesures n'ont pas une pertinence immédiate pour la gestion informatique. Par exemple, le taux de « *collision de paquets* » est couramment utilisé pour évaluer les performances du système. Cependant, cette mesure n'apporte pas réellement d'information pertinente aux utilisateurs et ne doit donc pas être utilisée comme SLO ;
- **rentabilité** : FOOSUS a-t-elle vraiment besoin d'une disponibilité de 99,9999 %, si 99,99 % est acceptable ? S'il faut investir dans plusieurs centres de données redondants, des protocoles de basculement et du personnel supplémentaire, le coût de l'investissement visant à garantir le SLO dépasse probablement de loin l'avantage apporté par ces 52 minutes de disponibilité supplémentaires annuelles ;
- **acceptation mutuelle** : les SLO doivent être acceptés par toutes les parties prenantes concernées. Si un SLO ne présente pas d'intérêt pour une partie prenante, soit le SLO n'est pas adéquat, soit la partie prenante doit être sortie de la portée de ce SLO.

En outre, tel que décrit dans le *§Mesure du succès*, plusieurs indicateurs seront à mettre en place et à remonter, de façon pertinente, en fonction des parties prenantes destinataires.

Plus spécifiquement, en ce qui concerne la mise en place de la MSA, les principaux SLO à atteindre seront les suivants :

- Augmentation de 10 % du nombre de clients enregistrés ;
- Passage à 4 référencements de fournisseurs mensuels ;
- Délai moyen de parution inférieur à 1 semaine ;
- Taux d'incidents de production inférieur ou égale à 1 par mois.

Une fois un **SLO** choisi, il convient de définir sa valeur. Cette opération doit être le fruit d'une négociation entre FOOSUS et les fournisseurs.

Il faut néanmoins garder à l'esprit que dans de nombreux cas, les **SLO** seront présentés comme des valeurs « à prendre ou à laisser ».

IV.B.2. Indicateurs de niveau de service (SLI)

IV.B.2.a. Métiers

Les indicateurs métiers sont relatifs à la Qualité, au nombre de clients, de fournisseurs, de transactions et d'interruption de services. Les principaux SLI relatifs à la plateforme FOOSUS sont détaillés ci-dessous :

- **Taux d'inscription des nouveaux utilisateurs ;**
- **Nombre de commandes réalisées ;**
- **Nombre de fournisseurs co-traitant ;**
- **Nombre de produits proposés ;**
- **Nombre de paiement reçus ;**
- **Capacité publicitaire, exposition de la solution ;**
- **Taux d'interruption de services ;**
- **Taux d'incidents de production.**

En ce qui concerne le dernier point énuméré ci-dessus, il est à noter que le taux d'erreur d'un **SLI** est la proportion d'activité qui tombe en dessous du seuil minimum du **SLO**. En d'autres termes, le taux d'erreur est l'inverse du **SLI** : 1 – SLI = taux d'erreur.

Le taux d'erreur, appelée *budget d'erreur*, est simplement une autre façon d'envisager la mesure des performances. Une disponibilité de 99% par mois peut être plus facile à interpréter (et plus prudente) lorsqu'elle est exprimée sous la forme d'un taux d'erreur de 1% de temps d'arrêt par mois, en particulier si le **SLO** est de l'ordre de 99,999% de disponibilité (soit un taux d'erreur de 0,001%). Cette notion peut également être exprimée sur une forme quantitative, par exemple, 7,2 heures d'indisponibilité par mois.

En plus de sa valeur pour les parties prenantes décisionnaires, le concept de *budget d'erreur* donne également à la *direction informatique* les outils dont elle a besoin pour prendre des décisions plus stratégiques concernant les temps d'arrêt et la latence. Par exemple, si la direction sait qu'un serveur doit absolument être mis hors ligne sans sauvegarde disponible, et qu'elle dispose d'un budget d'erreur de 7,2 heures par mois de temps d'arrêt autorisé, elle va s'efforcer d'inscrire l'intervention de maintenance dans cette fenêtre.

Outre ces SLI métiers, il est également concevable d'y ajouter deux autres indicateurs :

- **réactivité du centre de services** : la vitesse a laquelle les appels au centre d'assistance sont pris en charge ou résolus ;
- **niveau d'escalade** : la fréquence à laquelle les appels au service d'assistance sont transmis au niveau supérieur.

IV.B.2.b. Fonctionnels

Les indicateurs fonctionnels seront avant tout destinés au *Product Owner* du projet. Ce dernier ordonnancera alors le *Product Backlog* de façon à prioriser les *User's Stories* en fonction des priorités remontées par les différentes parties prenantes.

En ce qui concerne FOOSUS, la priorité sera donnée à la réduction de la **Dette Technique,** ainsi qu'à la **prise en charge des technologies existantes par la MSA**. En fonction de ces priorités, les SLI fonctionnels principaux, par priorité décroissante, seront les suivants :

- **Dette technique, nombre de User's Story émises ;**
- **Dette technique, nombre de User's Story réalisées ;**
- **Nombre de technologies utilisées actuellement ;**
- **Nombre de technologies prises en charge par MSA ;**
- **Nombre de fonctionnalités demandées ;**
- **Nombre de fonctionnalités répondues/réalisées.**

IV.B.2.c. Techniques

Les SLI techniques à la charge de FOOSUS seront examinés en temps réel pour superviser les conditions du système ; la fenêtre de temps réel des SLI sera à définir par le Responsable Ingénierie.

En outre, le stockage de ces **SLI,** nommés **SLI historiques**, sera analysé par le Responsable Ingénierie pour déterminer et/ou anticiper si et quand le système est susceptible de subir une interruption de service à l'avenir.

Il sera laissé à la discrétion du Responsable Ingénierie l'utilisation d'outils d'IA pouvant être particulièrement utiles pour cette analyse.

Comme pour tout **SLO**, l'augmentation de l'exigence de disponibilité se traduit par une augmentation des coûts. Ainsi, le Responsable Ingénierie devra être conscient qu'il est impossible d'assurer 100% de disponibilité. Dans bien des cas, d'ailleurs, il n'est ni approprié ni souhaitable de garantir une disponibilité de 100%. Pour de nombreux services, les temps d'arrêt planifiés sont intentionnels. Ils évitent d'exagérer la dépendance ou les attentes vis-à-vis de la disponibilité d'un seul service, et peuvent également contribuer à découvrir et à prévenir les utilisations inappropriées et les violations de la sécurité. Néanmoins, la redondance des microservices prises en compte par la MSA s'avérera particulièrement utile pour minimiser de façon dynamique ces interruptions programmés pour maintenance.

Ainsi, les SLI techniques principaux qui seront à considérer sont :

- **nombre de microservices en fonction** : quantité de groupe de microservices fonctionnels développés pour prendre en charge une technologie historique ;
- **nombre de total tests unitaires réalisés** ;
- **nombre de tests unitaires au *Vert*** : nombre de tests unitaires ayant été passés avec succès ;
- **nombre de tests unitaires au *Rouge***: nombre de tests unitaires ayant échoué ;
- **disponibilité du serveur** : proportions d'un serveur, un service web ou un microservice à être disponible et réactif ;
- **latence du serveur** : temps nécessaire à un serveur ou un microservice pour répondre à une requête ;
- **taux d'erreur** : fréquence d'une requête (à un serveur web, par exemple) entraînant une réponse d'erreur ;
- **débit/performance** : vitesse à laquelle les données sont livrées sur un canal donné ;
- **taux d'utilisation** : fréquence d'utilisation d'un certain microservice ou d'un groupement de microservice correspondant à une fonctionnalité (ce SLI est notamment utilisé par le MSA pour assurer l'équilibrage de charge/load-balancing) ;
- **fraîcheur des données** : proportions des données fournies aux utilisateurs correspondant bien à la version la plus récente.

V. Mise en œuvre

V.A. Directives

L'un des défis liés à l'utilisation de cette approche est de décider quand il est adapté de l'utiliser.

Lors du développement de la première version d'une application, il est difficilement concevable d’appréhender les problèmes que cette approche résout.

De plus, l'utilisation d'une architecture distribuée élaborée ralentira forcément le développement. Cela peut être un problème majeur pour les entreprises dont le plus grand défi est souvent de savoir comment faire évoluer rapidement le modèle commercial et l'application qui l'accompagne.

L'utilisation de fractionnements des fonctionnalités en microservices peut rendre beaucoup plus difficile les itérations de développement et présente des avantages indéniables en terme de robustesse, de fiabilité, de maintenabilité et d’extensibilité.

Néanmoins, lorsque le défi est de savoir comment évoluer et que le choix d’utilisation d’une architecture en microservice est acté, il est alors nécessaire d’utiliser la décomposition fonctionnelle. En effet, les dépendances enchevêtrées peuvent rendre difficile la décomposition de votre système d’information en un ensemble de microservices.

Ainsi, après le choix d’utilisation du MSA, il est indispensable de décider comment partitionner le système en microservices. Pour faire cela, il existe un certain nombre de stratégies pouvant aider à réaliser ce processus de décomposition :

- décomposer par capacité métier et définir les services correspondant aux capacités métier ;
- décomposer par sous-domaine de conception axée sur le domaine métier ;
- décomposer par verbe ou cas d'utilisation et définissez les services qui sont responsables d'actions particulières comme, par exemple, un service d'expédition responsable de l'expédition des commandes complètes ;
- décomposer par noms ou ressources en définissant un service responsable de toutes les opérations sur les entités/ressources d'un type donné, tel qu’un service de compte qui est responsable de la gestion des comptes d'utilisateurs.

Idéalement, chaque service ne se verra confier qu’un petit ensemble de responsabilités, on parle alors de la conception de classes en utilisant le principe de responsabilité unique (SRP).

Le SRP définit une responsabilité d'une classe comme une raison de changer et stipule qu'une classe ne devrait avoir qu'une seule raison de changer. Il est aisé d'appliquer également le SRP à un niveau plus élevé, tel que la conception de services.

Une autre analogie qui aide à la conception de services est la conception des utilitaires Unix, qui est un système d'exploitation fournissant un grand nombre d'utilitaires tels que grep, cat et find. Chaque utilitaire fait exactement une chose, souvent exceptionnellement bien, et est destiné à être combiné avec d'autres utilitaires à l'aide d'un script shell pour effectuer des tâches complexes.

Ainsi, une fois que l'utilisation de la MSA a été décidée, mise en place et est en cours d'utilisation, il restera encore à considérer deux domaines pour englober cette approche, concernant les bonnes pratiques relatives à :

- la maintenabilité d'une architecture de microservices ;
- l'extensibilité d'une telle architecture.

V.B. Spécifications

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 de FOOSUS;
- 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.

V.C. Framework Spring Boot

En tenant compte des éléments précédents, les frameworks, tels que **Spring Boot,** sont reconnus et fournissent une grande aide dans la construction de microservices.

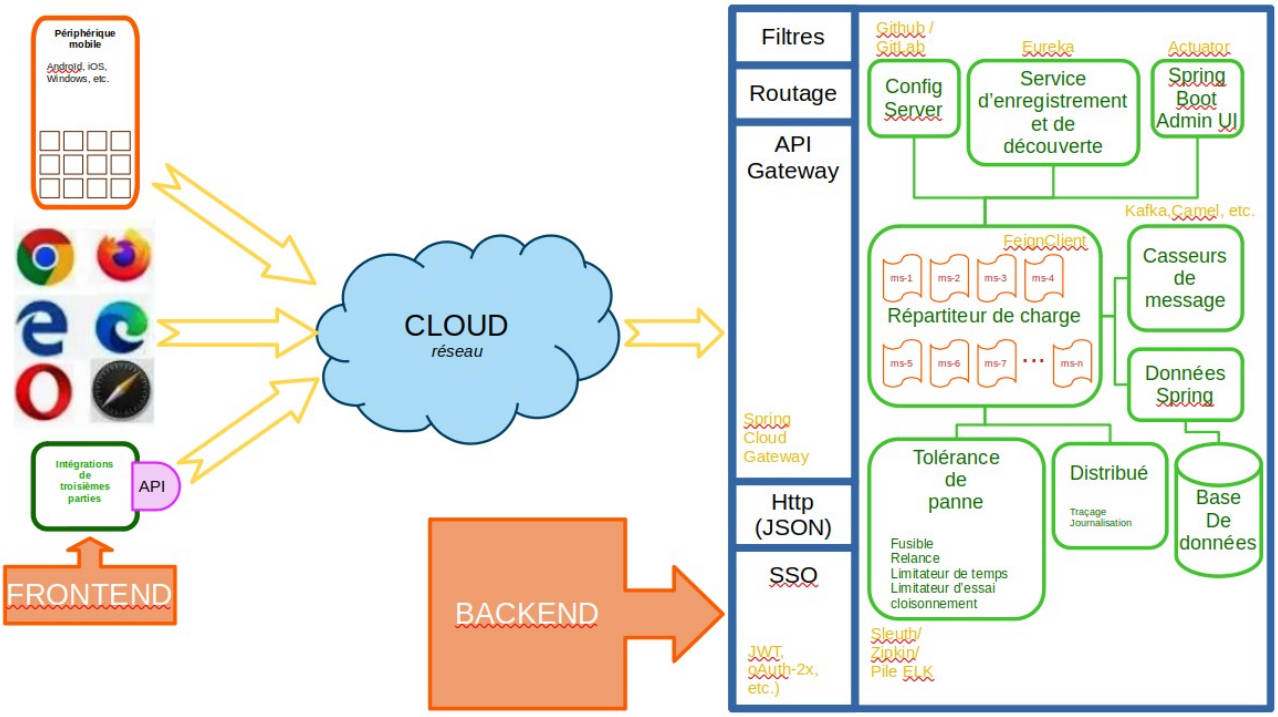
En effet, **Spring Boot** fournit des serveurs intégrés tels que *Tomcat*, *Jetty*, *Undertow*… les développeurs sont donc grandement accompagner pour créer très facilement des fichiers *.jar* avec un serveur Web intégré, et ainsi exécuté immédiatement leur création. Pas besoin de configuration particulière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



V.C.1. 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.

V.C.2. 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.

V.C.3. 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.*

V.C.4. 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*.

V.C.5. 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.*

V.C.6. 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.

V.C.6.a.i. 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*.

V.C.6.a.ii. 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.*

V.C.7. 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.

V.C.8. 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.

V.C.9. 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.*

V.C.10. 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.*

V.C.11. 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.

V.C.12. 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.

V.C.13. 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.

V.C.14. 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, FOOSUS 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.

V.C.15. 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* de FOOSUS, 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.*

V.C.16. 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.*

VI. Exigences

VI.A. D’interopérabilité

L’interopérabilité est importante au sein de l’application FOOSUS puisqu’elle permet **l’interconnexion de différents produits et services provenant de fournisseurs différents**. L’interopérabilité offre un éclectisme technique, sans tenir compte de l’utilisateur concerné, que ce soit un client, un fournisseur ou un membre de l’équipe FOOSUS.

Elle élargit l’éventail de choix en permettant de combiner une gamme plus large et plus polyvalente d’outils répondant aux besoins.

Elle permet également à FOOSUS de disposer de ressources limitées et de s’imposer sur le marché avec des produits innovants.

En effet, FOOSUS peut alors s’appuyer sur des outils existants pour proposer des modules complémentaires sans avoir besoin de créer des produits finaux très élaborés nécessitant une plus grande technicité.

En conséquence, l’interopérabilité est un facteur décisif d’innovation et d’enrichissement de l’écosystème numérique, s’inscrivant dans la politique d’entreprise de FOOSUS.

Malgré son statut de start-up, il faudra donc considérer, au sein de la nouvelle architecture, des normes établies pour que les clients, les fournisseurs et l’équipe FOOSUS puissent communiquer ensemble et échanger librement des informations et des services.

RAPPEL : chaque nouvel utilisateur renforce l’attractivité de la plateforme car il étend le nombre d’utilisateurs avec lesquels FOOSUS peut interagir.

De plus, en diversifiant le contenu de la plateforme, il permettra aux algorithmes d’affiner leurs sélections pour offrir un contenu plus pertinent à tous les clients comme dans le cas d’un moteur de recherche.

Enfin, une base de clients plus importante encouragera le développement de nouvelles applications comme sur les app-stores d’Android et d’Iphone.

Ce cercle vertueux pour les plateformes applicatives est appelé “effet de réseau”.

Ainsi, la domination du marché par FOOSUS se renforcera avec chaque nouveau client inscrit. Au fil du temps, la puissance de FOOSUS dissuadera également d’autres acteurs de créer des services concurrents, parce qu’ils savent qu’ils ne pourront jamais attirer assez de clients pour devenir attractifs et rentables.

VI.B. De gestion des services informatiques

La gestion des services informatiques (ITSM) est un ensemble de règles et de pratiques appliquées à la mise en œuvre, à la distribution et à la gestion des services IT destinés aux utilisateurs finaux, de manière à répondre à un cahier des charges définissant les besoins des utilisateurs finaux et aux objectifs de FOOSUS. Dans cette définition, le terme d'*utilisateurs finaux* peut inclure les clients, les fournisseurs ou les membres de l'équipe FOOSUS.

De plus, les *services IT* peuvent inclure toutes les ressources matérielles, logicielles ou informatiques fournies par FOOSUS. Les possibilités sont vastes : ordinateur portable professionnel, actif logiciel, application Web, application mobile, solution de stockage dans le cloud, serveur virtuel pour le développement, ou autres services.

Les principaux services informatiques dédiés à la mise en place de MSA et de l'API REST seront alors directement liés aux matériels et aux technologies utilisés, à savoir :

- **la gestion des incidents :** un incident désigne une panne ou une interruption non planifiée du service. La gestion des incidents définit le processus qui consiste à répondre à un incident dans le but de restaurer le service, avec un impact minimal pour les utilisateurs et FOOSUS ;

- **la gestion des problèmes :** ce processus consiste à identifier et résoudre la cause première d'un incident, les facteurs responsables de la cause première, et à déterminer la meilleure façon d'éliminer cette cause ;

- **la gestion des changements :** le changement est constant. La gestion du changement, également appelée '*intégration du changement*', est l'établissement de processus et de pratiques qui limitent au minimum les interruptions de service IT, les problèmes de conformité et les autres risques pouvant résulter des modifications apportées aux systèmes stratégiques ;

- **la gestion des actifs et des configurations :** ces processus d'autorisation, de surveillance et de documentation de la configuration des actifs logiciels et matériels (serveurs physiques et virtuels, systèmes d'exploitation, ordinateurs portables, appareils mobiles...) sont utilisés pour la distribution des services. Un des outils majeurs de gestion des actifs et des configurations est la *base de données de gestion de configuration (CMDB)*, qui sert de référentiel central pour tous les actifs IT et les relations entre ces actifs ;

- **la gestion des demandes de service :** cette fonctionnalité correspond aux processus de traitement des demandes de nouveaux services émanant de clients, de fournisseurs ou des membres de FOOSUS eux-mêmes ; l'équipe commerciale de FOOSUS peut d'ailleurs faire appel à cette fonctionnalité. Les demandes prises en compte peuvent être très variées : clients demandant de nouveaux produits, fournisseurs sollicitant un accès à un portail, équipe commerciale de FOOSUS demandant de nouveaux services applicatifs...FOOSUS aura tout intérêt à automatiser au maximum le flux de travaux des tickets via sa pratique du Kanban et le fonctionnement en "*libre-service*" de la gestion des demandes de service ;

- **le catalogue de services :** un menu ou un portail permettant aux utilisateurs de choisir eux-mêmes des services IT ; cette fonctionnalité sera apparentée au sein de l'application FOOSUS à une liste de choix faisant appel à différents *webapps*, eux-mêmes reliés directement aux microservices de gestion idoines ;

- **la gestion des connaissances :** cette pratique consiste à générer et partager des connaissances liées aux services IT de FOOSUS. Une base de connaissances en libre-service permet d'effectuer des recherches et des mises à jour. Elle est l'outil de base de cette pratique ;

- **La gestion des niveaux de service :** cette fonction consiste à s'entendre sur les niveaux de service requis ou souhaités pour différents groupes d'utilisateurs, puis à respecter ces niveaux, ou à "compenser" les utilisateurs lorsque les niveaux ne sont pas atteints. En règle générale, les niveaux de service convenus sont documentés dans un *accord sur les niveaux de service (SLA)*, qui fonctionne essentiellement comme un contrat entre l'IT et les utilisateurs ou l'entreprise. Les SLO sont décrits dans ce document au sein du *§Objectifs du niveau de service* ;

- **le service de support IT :** ce module est un sur-ensemble du service d'assistance standard. Il sert de point de contact unique (SPOC) pour le traitement et la gestion de tous les incidents, problèmes et demandes. C'est aussi un élément fondamental de l'ITSM, constituant la première étape pour tous les rapports d'incident, de problèmes et les demandes de service. Il s'agit également de l'endroit où les utilisateurs peuvent en suivre la progression de leur demande. Le service de support gère les licences logicielles, les fournisseurs de services et les contrats tiers liés à l'ITSM. Dans de nombreux cas de figure, le service de support exploite et gère les portails en libre-service et les bases de connaissances ITSM. FOOSUS pourra externaliser cette fonction à une entreprise tiers afin d'assumer sa politique d'Amélioration Continue de façon objective, pertinente et transparente.

VII. Contraintes

Les contraintes liées au gestion d'équipe devront prendre en considération l'esprit d'équipe développé précédemment.

Dans la continuité de l'esprit d'entreprise de FOOSUS, l'utilisation de méthode *Lean* devra être préconisée afin d'améliorer la responsabilisation individuelle et donc l'implication de groupe. Cette politique mettra en avant une politique de **Gouvernance** basée sur la bonne volonté des membres des équipes, ainsi que sur la reconnaissance individuelle.

En parallèle, l'utilisation de la MSA devra améliorée l'intégrité de l'architecture de FOOSUS. Cette amélioration sera mesurée et quantifiée par l'utilisation de SLA, SLO, SLI et/ou KPI.

La migration vers la MSA devra également modifiée la structure d'hébergement actuel en l'orientant vers des technologies *cloud* de type *SaaS*, sans exclure l'utilisation de solutions hybrides.

VIII. Hypothèses

Les hypothèses mentionnées ici prendront en considération les paragraphes précédents, dont notamment la prise en compte de l'infrastructure logicielle existante.

Cet existant devra être intégré à la nouvelle architecture. Ainsi, la MSA sera construite en fonction des technologies actuelles tout en assumant leur gestion. En effet, la réécriture des fonctionnalités programmées avec les technologies existantes n'auront pour effet que de retarder la pleine utilisation de la nouvelle architecture. Aussi, ces technologies existantes seront laissées en l'état et gérées à l'aide de l'utilisation d'API REST et de microservices. Ce principe aura pour conséquence d'adapter l'existant à la MSA pour obtenir, in fine, une architecture standardisée et cohérente, malgré la diversité des technologies utilisées jusqu'ici.

De plus, il est nécessaire de prendre en considération la connaissance des équipes de développement des technologies utilisées jusqu'ici. La MSA devra absorbée de manière progressive la multitude de technologies en gardant à l'esprit la politique de FOOSUS d'Amélioration Continue. S'écarter de cette politique serait synonyme de faux-raccourci et de contre-productivité, de par l'absence de Qualité avérée.

En outre, la coexistante des deux plateformes, basés sur l'architecture historique et sur la nouvelle à base de MSA, se conclura par une migration progressive de l'une vers l'autre. L'augmentation du nombre de clients sera absorbée par la nouvelle MSA. Cette montée en puissance sera également proportionnelle à l'évolution des fonctionnalités proposées.

De plus, les clients historiques se verront récompensés par la primeur d'utilisation de nouvelles fonctions développées ; la géolocalisation sera une de ces innovations.

Ces processus de mise en œuvre seront gérés à l'aide de méthode de gestion de type *Lean* afin d'assurer la continuité politique et stratégique de l'esprit d'entreprise de FOOSUS. Ainsi, les équipes FOOSUS conserveront toute leur autonomie et leur capacité d'innovation.

De plus, en utilisant des méthodes de développement issues du DevOps, les équipes de développement assureront un rythme de livraison durable et périodique.

ANNEXE

Liste des dépendances

Netflix Eurêka

```
<dépendance>
<!-- enregistrement des microservices par Eureka-->
<groupId>org.foosus.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>
```