Projet
Foosus géoconscient

Contrat d’architecture
de Conception
et
de Développement

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	14/07/2022	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
µservice(s)	Microservice(s)
AMQP	Advanced Message Queuing Protocol (trad. *protocole avancé de file d'attente de messages)*
DDD	Domain-Driven Design (trad. *conception piloté par le domaine*)
HATEOAS	Hypermedia As The Engine of Application State (trad. *hypermédia en tant que moteur de l'état d'application*)
IC	Intégration Continue
MSA	MicroServices Architecture (trad. *architecture de microservices*)
OSI	Open Systems Interconnection (trad. *interconnexion des systèmes ouverts*)
SGBDR	Système de Gestion de Base de Données Relationnelle
SSL	Secure Socket Layer (trad. *couche de sockets sécurisées*)
TDD	Test Driven Development (trad. *développement piloté par les tests*)
URI	Uniform Resource Identifier (trad. *identificateur de ressource uniforme*)
US	User’s Story
TLS	Transprt Layer Security (trad. *sécurité de la couche de transport*)
URL	Uniform Resource Locator (trad. *localisateur de ressource uniforme)*

Table des matières

I. Présentation et contexte

Dans le cadre de l’évolution de la plateforme applicative FOOSUS, une nouvelle architecture basée sur le MSA viendra progressivement suppléer la plateforme existante actuellement.

Ainsi, le but de ce document est d’identifier de manière formelle tous les acteurs qui participeront, directement ou indirectement, au cycle de vie de la nouvelle plateforme.

Les parties prenantes identifiées auront un rôle à jouer en ce qui concerne le développement, la gestion et la gouvernance de l’application FOOSUS.

Ce travail d’identification et de répartition des rôles permettra de favoriser la compréhension partagée des processus de création, de gestion, de maintenance et d’exécution de cette nouvelle architecture selon des principes de distribution de l’information ; une fois ces processus mis en place, cela renforcera significativement l'alignement entre les cultures d'entreprise et les technologies de l'information.

En outre, cette nouvelle architecture adoptera un principe d’hébergement différent et novateur puisqu’elle préconisera l'adoption d’un style architectural basé sur un environnement cloud.

Enfin, toutes les parties prenantes devront appréhender les tenants et aboutissants de cette nouvelle architecture afin d’opter, de façon pertinente, à des orientation métiers cohérentes.

II. Nature de l’accord

Ce document représentera un accord formel entre toutes les parties prenantes internes au projet de migration.

Cet accord permettra, autant aux parties métier qu’aux parties techniques, une bonne compréhension d’ensemble, en favorisant une symbiose permettant à tous les processus d’être menés correctement.

Il s'agit moins d'un contrôle manifeste et d'un strict respect des règles que d'une orientation et d'une utilisation efficace et équitable des ressources pour assurer, dans la durée, les objectifs stratégiques de FOOSUS.

En ce qui concerne la politique de FOOSUS, celle-ci respectera, autant que faire se peut, les préceptes suivants :

- se concentrer sur les droits : tous les utilisateurs adhérents à la plateforme ont des droits quant à leurs attentes respectives, par exemple, un client a le Droit de s’attendre à la Qualité du produit alimentaire qu’il achète, un fournisseur a le Droit d’être rémunéré suite à l’achat d’un de ses produits… ;

- assurer l’équité de rôle et de traitement des utilisateurs : peu importe la nature de l’utilisateur de la plateforme FOOSUS, celui devra être considéré avec égard et probité. Afin d’assurer sa place de leader sur le marché alimentaire, FOOSUS s’engage à faire montre d’intégrité, d’honnêteté et d’éthique envers tous les utilisateurs de sa plateforme. Les conflits seront réglés en privilégiant la communication en faveur des clients et/ou des fournisseurs référencés ;

- Assumer et afficher les décisions : toutes les décisions relatives à l’utilisation de la plateforme FOOSUS seront communiquées à toutes les parties prenantes intéressées. Ces décisions seront réfléchies et résolues après débat entre tous les acteurs impliqués directement et indirectement. Ceci contribuera à l’image de marque de FOOSUS en favorisant la transparence et l’acceptation des responsabilités.

Les préceptes énoncés ci-dessus assureront :

- une bonne orientation stratégique de l'organisation de FOOSUS ;
- un contrôle efficace de la gestion par le conseil d’administration ;
- une responsabilité du conseil d'administration envers toutes les parties prenantes de FOOSUS.

Ainsi, le conseil d’administration de FOOSUS aura comme responsabilités principales de :

- réviser et guider la stratégie de l'entreprise ;
- fixer et suivre la réalisation des objectifs de performance de la plateforme.

L’application des préconisations de ce document permettra d’améliorer :

- la **discipline** : toutes les parties concernées s'engageront à respecter les procédures, les processus et les structures d'autorité établis par FOOSUS;
- la **transparence** : toutes les actions mises en œuvre et leur aide à la décision seront disponibles pour inspection par FOOSUS et les fournisseurs référencés ;
- l’**indépendance** : tous les processus, prises de décision et mécanismes utilisés seront établis de manière à minimiser ou à éviter les conflits d'intérêts potentiels, cela vaut autant pour FOOSUS vis à vis de client et/ou de fournisseur qu’entre client et fournisseur eux-mêmes ;
- la **responsabilisation** : les groupes identifiables au sein de l'organisation - par exemple, les conseils de gouvernance qui prennent des mesures ou prennent des décisions - sont autorisés et responsables de leurs actions ;
- la **responsabilité** : chaque partie contractante, telle que les clients et les fournisseurs, est tenue d'agir de manière responsable envers l'organisation et ses parties prenantes ;
- la **justice** : toutes les décisions prises, les processus utilisés et leur mise en œuvre ne seront pas autorisés à créer un avantage injuste pour une partie en particulier.

III. Portée

Les exigences de qualité inhérentes à des travaux de développement en équipe ou en collaboration au sein même de FOOSUS ou en collaboration avec d'autres entreprises, la diversification des projets ainsi que le renouvellement des participants conduisent FOOSUS à utiliser des outils de gestion de configuration, tels que *subversion* ou *git*, et de développement dirigé par les tests (*Test Driven Development*).

Même si aujourd'hui, l'utilisation d'outils de gestion de configuration ainsi que la rédaction de tests est encouragé pour déboucher à un haut niveau de qualité logicielle, le constat est sans appel : ces pratiques restent à la marge au sein de FOOSUS.

En effet, de nombreuses projets de développement de fonctionnalités, débutés il y a plusieurs mois, contiennent d'importantes parties de code non testées.

Il serait donc nécessaire de passer un temps extrêmement important pour écrire les tests contrôlant toutes ces lignes de code, ce qui n'est objectivement pas concevable.

Ainsi, l'objectif de ce document s'inscrit dans la volonté de FOOSUS de mettre à jour, voire mettre en œuvre des pratiques de développement informatique, afin d'améliorer la qualité des développement des futures fonctionnalités et d'intégrer les fonctionnalités historiques au sein d'une architecture standardisée.

C'est dans ce contexte que l'utilisation de l'intégration continue entre en jeu, afin d'avoir un regard constant sur l'état des développements, ainsi que sur la qualité du code.

En effet, le but est de détecter au plus vite l'ajout d'éventuels problèmes ou erreurs au code existant, ainsi que de contrôler la qualité du code. Cela passe par la détection d'erreurs, la vérification des résultats des tests, la vérification de leur couverture, ou encore la détection de code dupliqué.

La mission principale consiste en la mise en place d'un système d'intégration continue, équipé d'outils de contrôle de qualité (couverture par les tests, contrôle de la redondance de code, etc) au sein de l'entreprise.

Le nouveau système proposera des solutions permettant de travailler efficacement dans divers langages, tels que le C, le C++, le Python, le Java ou encore le CUDA, pour n'en citer que quelques-uns.

Des prototypes d'applications basiques dans ces différents langages, avec des systèmes de construction divers, ainsi que la rédaction de tests unitaires, permettront ainsi de tester et valider la mise en place de la solution.

Un autre objectif de ce document consistera à donner des pistes relatives à la mise en place d'environnements de tests pour les codes précédemment déployés à intégrer en l'état sur ce système d'intégration continue.

Des solutions de type machine virtuelle ou conteneur (type *Docker* ou *Jenkins*) seront étudiées et la pertinence de leur couplage au système d'intégration continue sera également à prendre en compte.

Ainsi ce document devra orienter les futures études réalisées à propos de divers outils permettant la mise en place d'un système d'intégration continue, ainsi que sur les solutions autour des environnements de tests, dans le but d'avoir un aperçu sur les qualités et défauts de chacun et de vérifier s'ils correspondent ou non aux besoins.

IV. Description de l’architecture, principes stratégiques et conditions requises

La mise en place d’une nouvelle architecture basée sur les microservices mérite quelques précisions relatifs aux principes stratégiques de conception de cette MSA, aux stratégies de sa mise en œuvre, et aux limites de mise en œuvre de ces microservices.

Les microservices sont un style architectural permettant de créer des applications cloud résilientes, hautement évolutives, déployables indépendamment et capables d'évoluer rapidement.

Cependant, pour être plus qu'un terme technique, les microservices nécessitent une approche différente de la conception et de la création d'applications de style monolithique.

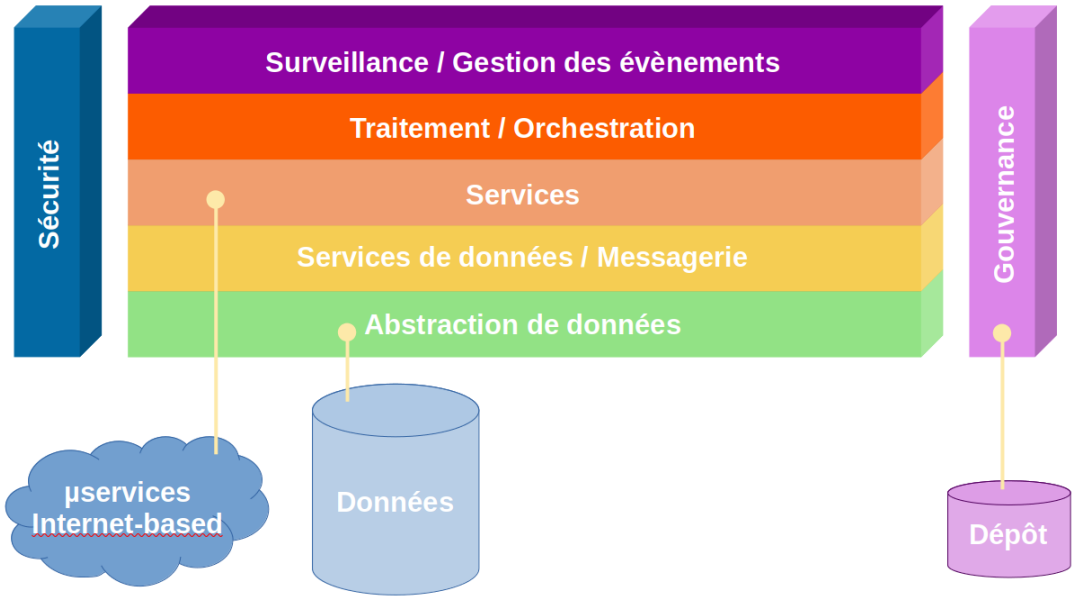
Aussi, afin de les présenter convenablement, il convient de décomposer leur nature en thème que nous aborderons dans les paragraphes qui suivent :

- Options de calcul pour les µservices ;
- Communication inter-µservices ;
- Conception d'API pour les µservices ;
- Passerelles API pour les µservices ;
- Considérations relatives aux données ;
- Modèles de conception pour les µservices.

IV.A. Généralités

Avant de rentrer dans des considérations spécifiques à la nature d’un microservice, il est important de définir une MSA de façon générale : une MSA consiste en une collection de petits services autonomes. Chaque service est autonome et doit implémenter une capacité métier unique dans un contexte délimité ; un contexte délimité sera une division naturelle au sein de FOOSUS et fournira une frontière explicite à l'intérieur de laquelle un modèle de domaine existe.

Une MSA peut alors être représentée de façon générique comme suit :



Maintenant qu’une définition générale d’une MSA est actée, définissons les µservices à partir d’axiomes dont il faudra tenir compte durant toute la phase de conception d’une MSA :

- les µservices sont petits, indépendants et faiblement couplés ;
- une seule petite équipe de développeurs peut écrire et maintenir un µservice ;
- chaque µservice est une base de code distincte qui peut être gérée par une petite équipe de développement ;
- les µservices peuvent être déployés indépendamment ;
- une équipe peut mettre à jour un µservice existant sans reconstruire et redéployer l'intégralité de l'application ;
- les µservices sont responsables de la persistance de leurs propres données ou de leur état externe (*nota : ici le terme ‘ou’ est considéré comme un OU logique exclusif*). Cette approche diffère du modèle traditionnel, où une couche de données distincte gère la persistance des données ;
- les µservices communiquent entre eux à l'aide d'API bien définies. Les détails de mise en œuvre interne de chaque µservice sont cachés aux autres µservices ;
- les µservices assument la programmation polyglotte ; les µservices n'ont pas besoin de partager la même pile technologique, les mêmes bibliothèques ou les mêmes frameworks.

Outre les µservices eux-mêmes, d'autres composants apparaissent dans une architecture de microservices typique :

- Traitement/Orchestration : ce composant est chargé de placer les services sur les nœuds, d'identifier les pannes, de rééquilibrer les services entre les nœuds, etc. Généralement, ce composant est une technologie prête à l'emploi telle que *Docker Swarm, Apache Mesos, Azk* ou encore *Kubernetes*, plutôt qu’une solution personnalisé ;
- Passerelle API : la passerelle API est le point d'entrée pour les clients. En ce qui concerne FOOSUS, la passerelle API utilisée sera une API REST. Ainsi, au lieu d'appeler directement les services, les clients appellent la passerelle API, qui transfère l'appel aux services appropriés sur le *backend*. L’utilisation d’une passerelle API présentent plusieurs avantages :
  - elle dissocie les clients des services. Les µservices peuvent être versionnés ou être modifié par *refactoring* sans qu'il soit nécessaire de mettre à jour tous les clients ;
  - les µservices peuvent utiliser des protocoles de messagerie qui ne sont pas compatibles avec le Web, comme AMQP ;
  - la passerelle API peut exécuter d'autres fonctions transversales telles que l'authentification, la journalisation, la terminaison SSL et l'équilibrage de charge ;
  - l’utilisation de politiques prêtes à l'emploi, comme pour la limitation, la mise en cache, la transformation, la validation...

IV.A.1. Bénéfices

La MSA et ses constituants étant définis, il convient maintenant de présenter les avantages pour FOOSUS quant à leur mise en œuvre et leur utilisation :

- **l'Agilité** : étant donné que les µservices sont déployés indépendamment, il est plus facile de gérer les correctifs de bogues et les versions de fonctionnalités. Ainsi, les équipes de développement de FOOSUS pourront mettre à jour un service sans redéployer l'ensemble de l'application et/ou annuler une mise à jour en cas de problème. Dans de nombreuses applications traditionnelles, si une erreur est trouvée dans une partie de l'application, elle peut bloquer l'ensemble du processus de publication. De nouvelles fonctionnalités peuvent être bloquées en attendant qu'un correctif d'erreur soit intégré, testé et publié ;
- **les petites équipes concentrées** : un µservice doit être suffisamment petit pour qu'une seule équipe technique puisse le créer, le tester et le déployer. En complément du point précédent, les petites équipes favorisent une plus grande agilité. Les grandes équipes ont tendance à être moins productives, car la communication est plus lente, les frais de gestion augmentent et l'agilité diminue ;
- **une petite base de code** : FOOSUS est une jeune start-up créée il y a trois ans. Aussi, depuis sa création elle a expérimenté, créé et fait émerger de nouvelles solutions. Toute cette expérience constitue une base de connaissance à conserver, à enrichir et à émuler. En outre, bien que l'architecture actuelle utilise beaucoup de technologies diverses et variées, celle-ci peut être considérée comme une architecture monolithique. Dans une application monolithique, les dépendances de code ont tendance à s'emmêler avec le temps. L'ajout d'une nouvelle fonctionnalité nécessite de toucher au code dans de nombreux endroits. En ne partageant pas de code ou de magasins de données, une MSA minimise les dépendances, ce qui facilite l'ajout de nouvelles fonctionnalités ;
- **le mélange de technologies** : les équipes de développement de FOOSUS doivent pouvoir continuer à choisir la technologie qui correspond le mieux à leur service, en utilisant une combinaison de piles technologiques, le cas échéant, et les µservices permettent cela ;
- **l'isolement d'anomalie** : si un µservice individuel devient indisponible, il ne perturbera pas l'ensemble de l'application, tant que les autres µservices en amont sont conçus pour gérer correctement les pannes (par exemple, en mettant en œuvre une coupure de circuit) ;
- **l'évolutivité** : les µservices peuvent être mis à l'échelle indépendamment, ce qui vous permet de faire évoluer les sous-systèmes qui nécessitent plus de ressources, sans faire évoluer l'ensemble de l'application. Les équipes de développement de FOOSUS pourront alors regrouper une plus grande densité de services sur un seul hôte, ce qui permettra une utilisation plus efficace des ressources ;
- **l'isolement des données** : il est beaucoup plus facile d'effectuer des mises à jour de schéma, car un seul µservice est affecté. Dans une application monolithique, les mises à jour de schéma peuvent devenir très difficiles, car différentes parties de l'application peuvent toutes toucher les mêmes données, ce qui rend toute modification du schéma risquée.

IV.A.2. Défis

Malgré tous les avantages des µservices présentés ci-dessous, il convient néanmoins d'annoncer que ceux-ci ne seront pas gratuits pour FOOSUS. Les équipes de développement devront tout de même considérer quelques défis technologiques avant de se lancer dans une MSA :

- **la complexité** : une application de µservices comporte plus de pièces mobiles que l'application monolithique équivalente. Chaque µservice est plus simple, mais l'ensemble du système dans son ensemble est plus complexe ;
- **le développement et tests** : l'écriture d'un µservice qui s'appuie sur d'autres µservices dépendants nécessite une approche différente de l'écriture d'une application traditionnelle monolithique ou en couches. Les outils existants ne sont pas toujours conçus pour fonctionner avec des dépendances de service/µservice. L'utilisation du *refactoring*, au-delà des limites de service, peut être complexe. Il est également complexe de tester les dépendances de µservice, en particulier lorsque l'application évolue rapidement ;
- **le manque de gouvernance** : l'approche décentralisée de la création de µservices présente des avantages, mais elle peut également entraîner des problèmes. Face à la diversité des technologies utilisées jusqu'alors par les équipes de FOOSUS, l'application devient difficile à maintenir. Il sera alors souhaitable de mettre en place des normes à l'échelle du projet, sans trop restreindre la flexibilité des équipes. Cela s'applique particulièrement aux fonctionnalités transversales telles que la journalisation ;
- **la congestion et latence du réseau** : l'utilisation de nombreux petits µservices granulaires peut entraîner davantage de communication inter-services. De plus, si la chaîne de dépendances de service devient trop longue (le service A appelle B, qui appelle C...), la latence supplémentaire peut devenir un problème. Les équipes de développement devront alors prendre grand soin de la conception des API REST. Il sera nécessaire d'appliquer la politique LEAN chère à FOOSUS : pas d'API trop bavardes, utilisation de formats de sérialisation et recherche de modèles de communication asynchrones, tel que le nivellement de charge basé sur la file d'attente ;
- **l'intégrité des données** : chaque µservice doit être responsable de sa propre persistance des données. Par conséquent, la cohérence des données peut devenir un véritable défi. Les équipes de développement devront alors adopter une politique de cohérence quant à la persistance, et s'y tenir pour l'ensemble des µservices développés ;
- **La gestion** : pour réussir avec les µservices, il faut une culture **DevOps** mature. La journalisation corrélée entre les µservices peut être difficile. En règle générale, la journalisation doit corréler plusieurs appels de service pour une seule opération utilisateur ;
- **le versioning** : les mises à jour d'un µservice ne doivent pas interrompre les autres µservices qui en dépendent. Plusieurs µservices peuvent être mis à jour à tout moment, donc sans une conception soignée, des problèmes de compatibilité ascendante ou descendante peuvent apparaître ;
- **l'ensemble de compétences** : les µservices sont des systèmes hautement distribués, FOOSUS devra soigneusement évaluer si les équipe possède les compétences et l'expérience nécessaires pour développer une MSA. Dans la négative, des formations techniques devront être envisagées.

IV.B. Options de calcul pour les µservices

Le terme ‘*calcul*’ du titre de ce paragraphe fait référence au modèle d'hébergement des ressources informatiques sur lesquelles va s'exécuter l’application FOOSUS.

Pour une MSA, deux types d’approches seront particulièrement adaptées :

un orchestrateur de services qui gère les services s'exécutant sur des nœuds dédiés (machine virtuelle VM) ;

une architecture sans serveur utilisant des fonctions en tant que service (SaaS).

Bien que ce ne soient pas les seules options, ce sont toutes deux des approches éprouvées qui seront adéquates à la migration des services historiques de l’application monolithique de FOOSUS. De plus, en ce qui concerne cette migration, différentes types d’hébergement technologiques seront à choisir :

- Orchestrateurs de services ;
- Conteneurs ;
- Sans serveur.

IV.B.1. Orchestrateurs de services

Un orchestrateur gère les tâches liées au déploiement et à la gestion d'un ensemble de services. Ces tâches incluent :

- le placement de µservices sur des nœuds ;
- la surveillance de la santé des µservices ;
- le redémarrage de µservices défectueux ;
- l'équilibrage de charge du trafic réseau entre les instances de µservice ;
- la découverte dynamique de nouveaux µservices ;
- la mise à l'échelle du nombre d'instances d'un µservice ;
- l'application de mises à jour de configuration.

De tels système d’orchestration nécessiteront l’utilisation de COTS, tels que *Service Fabric, DC/OS, Docker Swarm*…

IV.B.2. Conteneurs

Il est nécessaire de bien faire la distinction entre Conteneurs µservices ; ce sont deux objets très différents. En effet, il n’est pas nécessaire de posséder de conteneurs pour créer des µservices…

Néanmoins, les conteneurs présentent certains avantages particulièrement pertinents pour la mise en œuvre de µservices, tels que :

- la portabilité : une image de conteneur est un package autonome qui s'exécute sans qu'il soit nécessaire d'installer de bibliothèques ou d'autres dépendances. Cela les rend faciles à déployer. Les conteneurs peuvent être démarrés et arrêtés rapidement, ce qui permettra aux équipes de développement de FOOSUS de lancer de nouvelles instances pour gérer plus de charge ou pour récupérer des données après des pannes de nœud ;
- la densité : les conteneurs sont légers par rapport à l'exécution d'une machine virtuelle, car ils partagent les ressources du système d'exploitation. Cela permet de regrouper plusieurs conteneurs sur un seul nœud, ce qui est particulièrement utile lorsque l'application se compose de nombreux petits services ;
- l’isolement des ressources : il est tout à fait concevable de limiter la quantité de mémoire et de CPU disponible pour un conteneur, ce qui peut aider à garantir qu'un processus incontrôlable n'épuise pas les ressources de l'hôte.

IV.B.3. Sans serveur

Dév : sans serveur ???
Archi : ouais tkt, ça fonctionne en tant que service^^

Avec une architecture sans serveur, les équipes de FOOSUS ne pourront gérer ni les machines virtuelles ni l'infrastructure du réseau virtuel.

Au lieu de cela, les équipes de FOOSUS déploieront le code et le service d'hébergement gérera le placement de ce code sur une machine virtuelle et son exécution.

Cette approche tendra à favoriser les petites fonctions granulaires qui seront coordonnées à l'aide de déclencheurs basés sur des événements.

Par exemple, un message placé dans une file d'attente pourra déclencher une fonction qui *lira* le contenu du message à partir de la file d'attente puis traitera le message.

Certains COTS, tel que *Docker Swarm*, proposent est des services de calcul *sans serveur* qui prennent en charge divers déclencheurs de fonction, notamment les requêtes HTTP, les files d'attente *Service Bus* et les événements *Event Hubs*. Ces COTS mettront à disposition des équipes de FOOSUS des listes complètes de concepts de déclencheurs, de liaisons inter-µservices et de service de routage d’évènements.

IV.C. Communication inter-µservices

La communication entre les µservices doit être efficace et robuste. Avec de nombreux µservices granulaires interagissant pour mener à bien une seule activité, cela peut vite devenir un défi pour les équipes de développement de FOOSUS.

Il faudra alors procéder, au cas par cas, à un compromis entre la **messagerie asynchrone** et les **API synchrones**.

Un fois ce choix effectué, les équipes de développement devront se concentrer sur la conception d'une communication inter-µservices répondant aux critères suivants :

- la résiliente ;
- l'équilibrage de charge ;
- le traçage distribué ;
- les gestion des versions des µservices ;
- le cryptage SSL et l'authentification mutuelle TLS.

IV.C.1. Résilience

Il peut exister des dizaines, voire des centaines d'instances d'un seul et même µservice donné.

De plus, une instance peut échouer pour un certain nombre de raisons et il peut y avoir une défaillance au niveau du nœud, telle qu'une défaillance matérielle ou un redémarrage de la machine virtuelle. Une instance peut planter ou être submergée de requêtes et incapable de traiter de nouvelles requêtes.

Chacun de ces événements peut entraîner l'échec d'un appel réseau. Il existe deux modèles de conception peuvant aider à rendre les appels réseau de service à service plus résilients :

- la réitération : un appel réseau peut échouer à cause d'un défaut transitoire qui disparaît tout seul. Plutôt que d'échouer purement et simplement, l'appelant doit généralement réessayer l'opération un certain nombre de fois ou jusqu'à ce qu'un délai d'attente configuré se soit écoulé. Cependant, si une opération n'est pas idempotente, les nouvelles tentatives peuvent entraîner des effets secondaires imprévus. L'appel d'origine peut réussir, mais l'appelant n'obtient jamais de réponse. Si l'appelant réessaye, l'opération peut être invoquée deux fois. En règle générale, même avec l'utilisation de méthodes POST ou PATCH, l'appelant n'est pas sûr du résultat de nouvel essai, car ces méthodes ne sont pas garanties comme idempotentes ;

- le disjoncteur : un trop grand nombre de requêtes ayant échoué peut provoquer un goulot d'étranglement, car les requêtes en attente s'accumulent dans la file d'attente. Ces demandes bloquées peuvent contenir des ressources système critiques telles que la mémoire, les threads, les connexions à la base de données, etc., ce qui peut entraîner des échecs en cascade. Le modèle *disjoncteur* peut empêcher un service de tenter à plusieurs reprises une opération susceptible d'échouer.

IV.C.2. L'équilibrage de charge

Lorsque le µservice "A" appelle le µservice "B", la demande doit atteindre une instance en cours d'exécution du µservice "B". Le type de ressource du µservice doit alors fournir une adresse IP stable pour un groupe de nœuds logiques.

Le trafic réseau vers l'adresse IP du service est alors transmis à un nouveau nœud logique au moyen de règles de type *iptable*.

Par défaut, un nœud logique est choisi de façon. Un maillage de l'ensemble des µservices peut alors fournir des algorithmes d'équilibrage de charge plus intelligents basés sur la latence observée ou d'autres mesures.

IV.C.3. Traçage distribué

Une seule transaction peut couvrir plusieurs µservices. Ainsi, le transit entre plusieurs µservices peut grandement compliquer la surveillance des performances globales et de la santé du système.

Même si chaque µservice génère des journaux et des métriques, sans aucun moyen de les lier, ils sont d'une utilité limitée.

Pour palier à cette problématique il existe des collections de télémétrie pour les applications complexes pouvant être composées jusqu'à plusieurs millions de µservices existants simultanément.

IV.C.4. Gestion des versions des µservices

Lorsqu'une équipe de FOOSUS déploiera une nouvelle version d'un µservice, elle devra éviter de casser tout autre µservice ou client externe qui en dépend.

En outre, cette me équipe souhaitera peut-être exécuter plusieurs versions d'un même µservice côte à côte et acheminer les demandes vers une version particulière. Cette possibilité sera traitée dans le paragraphe suivant relatif à la conception d'API pour les µservices.

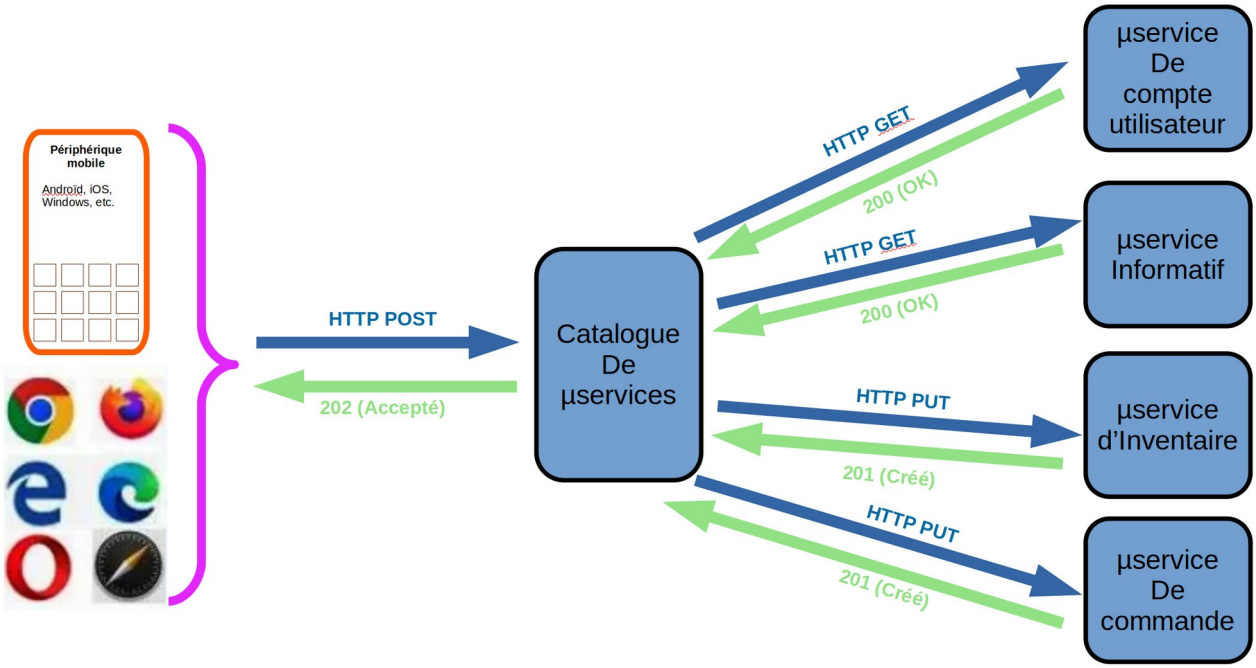
IV.C.5. Cryptage *S*SL et authentification mutuelle TLS

Pour des raisons de sécurité, FOOSUS devra chiffrer le trafic entre les µservices avec SSL et utiliser l'authentification mutuelle TLS pour authentifier les appelants.

IV.D. Conception d'API pour les µservices

Pour FOOSUS, la conception d'API est importante dans une MSA car tous les échanges de données entre les µservices se font via des messages ou des appels d'API. Ces dernières devront être efficaces pour éviter de créer des E/S bavardes, et donc en dehors des considérations *LEAN* dont FOOSUS est constitutée.

Étant donné que les µservices sont conçus par des équipes de développement travaillant de manière indépendante, les API devront avoir une sémantique et des schémas de version bien définis, afin que les mises à jour n'interfèrent pas ou n'interrompent pas les autres µservices, tel que présenté ci-dessous :



A partir de maintenant, il va être important de distinguer deux types d'API :

- les API publiques appelées par les applications clientes : cette API devra être compatible avec les applications clientes, généralement des applications de navigateur ou des applications mobiles natives. La plupart du temps, cela signifie que l'API publique utilisera **REST sur HTTP ;**
- les API *backend* utilisées pour la communication inter-µservices : ici, il faudra tenir compte tenir compte des performances du réseau. Selon la granularité des µservices, la communication inter-µservices pourrait entraîner un trafic réseau important. Les µservices peuvent alors rapidement devenir liés aux E/S. Pour cette raison, des considérations telles que la vitesse de sérialisation et la taille de la charge utile deviennent plus importantes. Certaines alternatives populaires à l'utilisation de REST sur HTTP incluent *gRPC*, *Apache Avro* et *Apache Thrift*. Ces protocoles prennent en charge la sérialisation binaire et sont généralement plus efficaces que HTTP.

En ce qui concerne FOOSUS, les API seront toutes développées selon la norme RESTful.

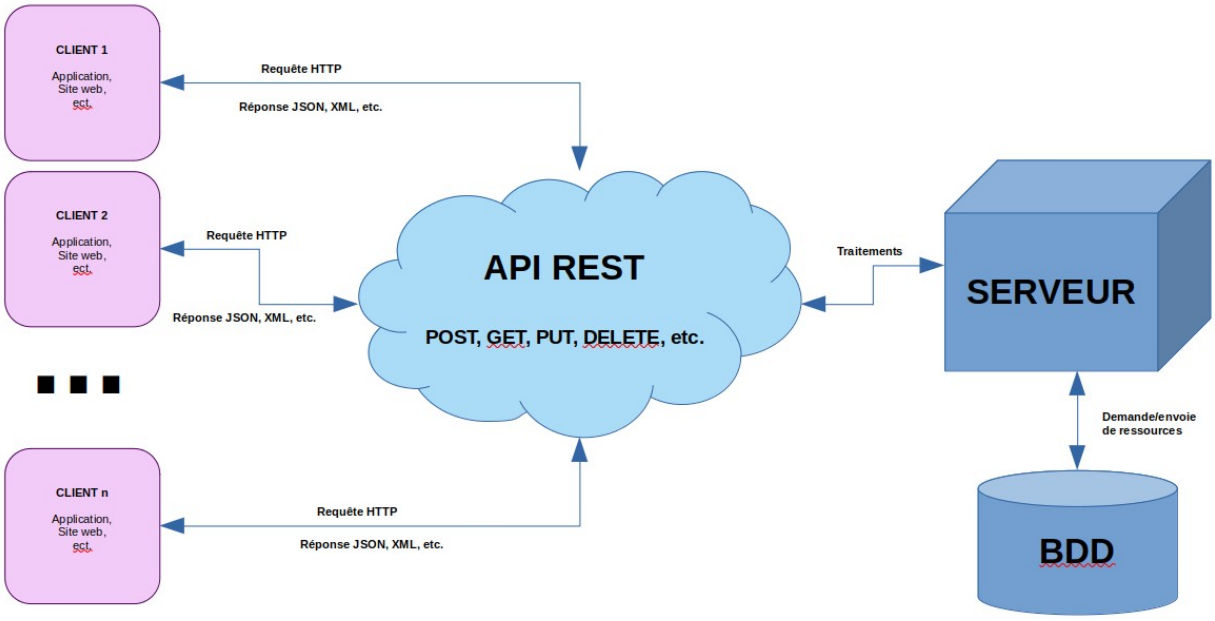
IV.D.1. Conception d'API RESTful

Lors de la conception d'API REST (ou RESTful), il sera nécessaire que les équipes de développement gardent certaines considérations à l'esprit :

- éviter les API qui divulguent des détails d'implémentation internes ou qui reflètent simplement un schéma de base de données interne. L'API doit modéliser le domaine. Il s'agit d'un contrat entre plusieurs µservices et, idéalement, cela ne devrait changer que lorsque de nouvelles fonctionnalités sont ajoutées, et non lors de processus de *refactoring* du code ou de normalisation de table de base de données ;
- utiliser différents types de clients, tels que les applications mobiles et les navigateurs Web de bureau, peuvent nécessiter des tailles de charge utile ou des modèles d'interaction différents. Il faudra que les développeurs de FOOSUS envisagent d'utiliser le modèle *Backends for Frontends*, tel qu'ils l'appliquaient historiquement lors de l'utilisation du système *StoreFront*. Cela permettra de créer des *backends* distincts pour chaque client qui exposent une interface optimale pour ce client ;
- pour les opérations avec des effets secondaires, il sera nécessaire de les rendre idempotentes et de les implémenter en tant que méthodes PUT. Cela permettra des tentatives sûres et améliorera grandement la résilience ;
- les méthodes HTTP peuvent avoir une sémantique asynchrone, où la méthode renvoie une réponse immédiatement, mais le service exécute l'opération de manière asynchrone. Dans ce cas, la méthode doit renvoyer un code de réponse *HTTP 202*, qui indiquera que la demande a été acceptée pour traitement, mais que le traitement n'est pas encore terminé.

Le lecteur est invité à prendre connaissance de la RFC 7231 (https://tools.ietf.org/html/rfc7231) pour avoir un détail exhaustif quant à la constitution du protocole HTTP.

De plus, les exigences architecturales de FOOSUS issues de l'API REST sont décrites au sein du document de Spécification des Conditions requises pour l'Architecture, *§Issues de l'API REST*.



IV.D.2. Mappage de REST aux langages Objets

Les modèles tels que l'entité, l'agrégat et l'objet de valeur sont conçus pour imposer certaines contraintes aux objets de votre modèle de domaine.

Lors d'approches conceptuelles comme le *DDD*, les modèles sont structurés à l'aide de concepts de langage orienté objet (*OO*) tels que les *constructeurs* ou les *getters* et *setters* de propriétés. Par exemple, les objets de valeur sont supposés être immuables.

Dans un langage de programmation *OO*, il sera donc nécessaire d'affecter les valeurs dans le constructeur et de rendre leurs propriétés associées en lecture seule.

Dans une MSA, les µservices ne partagent pas la même base de code et ne partagent pas les magasins de données. Au lieu de cela, ils communiquent via des API.

Prenons le cas où le Catalogue de µservices de FOOSUS demande des informations au µservice d'*Inventaire*. Ce dernier a son modèle interne d'*Inventaire,* exprimé par du code. Mais le *Catalogue* ne le voit pas. Au lieu de cela, il récupère une représentation de l'entité *Inventaire* - peut-être un objet JSON dans une réponse HTTP.

Ainsi, les objets seront définis selon des agrégats de plusieurs µservices. Ces agrégats seront à considérer comme des frontières de cohérence.

Les opérations sur les agrégats ne doivent jamais laisser un agrégat dans un état incohérent. Par conséquent, Les développeurs de FOOSUS ne devront jamais créer d'API qui permettent à un client de manipuler l'état interne d'un agrégat (composition de plusieurs µservices).

Il restera alors à privilégiez plutôt les API grossières qui exposent les agrégats en tant que ressources dont les entités ont des identités uniques.

Dans REST, les ressources ont des identifiants uniques sous la forme d'URL. Créer des URL de ressource correspond à identifier le domaine d'une entité.

Le mappage de l'URL à l'identité du domaine peut être opaque pour le client. Les entités enfants d'un agrégat peuvent être atteintes en naviguant à partir de l'entité racine. En suivant les principes HATEOAS, les entités enfants peuvent être atteintes via des liens dans la représentation de l'entité parent.

Étant donné que les objets de valeur sont immuables, les mises à jour seront effectuées en remplaçant l'intégralité de l'objet de valeur.

Dans REST, implémenter des mises à jour via des requêtes PUT ou PATCH est un approche correcte.

Un référentiel permet alors aux clients d'interroger, d'ajouter ou de supprimer des objets dans une collection, en faisant abstraction des détails du magasin de données sous-jacent. Dans REST, une collection peut être une ressource distincte, avec des méthodes pour interroger la collection ou ajouter de nouvelles entités à la collection.

IV.D.3. Versioning d’API

Il a été acté précédemment qu’une API est un contrat entre un service et des clients ou des consommateurs de ce service.

Si une API change, il y a un risque de casser les clients qui dépendent de l'API, qu'il s'agisse de clients externes ou d'autres microservices. Par conséquent, il est judicieux de minimiser le nombre de modifications d'API à effectuer.

Souvent, les modifications apportées à l'implémentation sous-jacente ne nécessitent aucune modification de l'API. De manière réaliste, cependant, à un moment donné, il sera nécessaire d’ajouter de nouvelles fonctionnalités ou de nouvelles capacités qui nécessitent de modifier une API existante.

Dans la mesure du possible, les développeurs devront rendre les modifications de l'API concernée rétrocompatibles. Par exemple, il faudra éviter de supprimer un champ d'un modèle, car cela peut casser les clients qui s'attendent à ce que le champ soit là.

L'ajout d'un champ ne rompt pas la compatibilité, car les clients doivent ignorer tous les champs qu'ils ne comprennent pas dans une réponse.

Cependant, le service doit gérer le cas où un ancien client omet le nouveau champ dans une requête.

Pour éviter ces problématiques, il faudra considérer la prise en charge de la gestion des versions dans les contrats d'API.

En introduisant une modification majeure de l'API, il faudra introduire une nouvelle version de l'API et continuer à prendre en charge la version précédente ; les clients pourront alors choisir la version d’API à appeler.

Il y a plusieurs façons de réaliser ceci. L'une d’entre elles consiste simplement à exposer les deux versions dans le même µservice.

Une autre option consiste à exécuter deux versions du service côte à côte et à acheminer les demandes vers l'une ou l'autre version, en fonction des règles de routage HTTP.

Néanmoins, la prise en charge de plusieurs versions a un coût, en termes de temps de développement, de tests et de frais généraux opérationnels. Par conséquent, il est bon de déprécier les anciennes versions le plus rapidement possible.

Pour les API internes, l'équipe de développement de l'API pourra travailler avec d'autres équipes pour les aider à migrer vers la nouvelle version. C'est à ce moment qu'un **processus de gouvernance** inter-équipes est utile.

Pour les API externes (publiques), il peut être plus difficile de déprécier une version d'API, en particulier si l'API est consommée par des tiers ou par des applications clientes natives. Il sera alors nécessaire de considérer le versioning d’API pour ces cas-là.

IV.D.4. Opérations idempotentes

Dans le cadre de l’application FOOSUS, un client, un fournisseur ou même un collaborateur pourra exécuter plusieurs fois la même requête ou le même traitement. A chaque série, il sera alors nécessaire que l’utilisateur reçoivent TOUJOURS la même réponse. C’est ce qu’on appelle les opérations idempotentes.

En d’autres termes, une opération idempotente appelée plusieurs fois devra systématiquement fournir le même résultat sans produire d'effets secondaires supplémentaires après le premier appel.

L'idempotence peut être une stratégie de résilience utile, car elle permet à un service en amont d'appeler en toute sécurité une opération plusieurs fois.

La spécification HTTP stipule que les méthodes GET, PUT et DELETE doivent être idempotentes.

Les méthodes POST ne sont pas garanties d'être idempotentes. Si une méthode POST crée une nouvelle ressource, il n'y a généralement aucune garantie que cette opération soit idempotente.

Selon la RFC 7231, la spécification de l'idempotence est la suivante : « *une méthode de demande est considérée comme "idempotente" si l'effet prévu sur le serveur de plusieurs demandes identiques avec cette méthode est le même que l'effet d'une seule de ces demandes.* »

Il est alors important de bien comprendre la différence entre la sémantique PUT et POST lors de la création d'une nouvelle entité. Dans les deux cas, le client envoie une représentation d'une entité dans le corps de la requête.

Néanmoins, la signification de l'URI est différente : pour une méthode POST, l'URI représente une ressource parente de la nouvelle entité, telle qu'une collection. Par exemple, pour créer un nouveau transport sur FOOSUS, l'URI peut être */api/deliveries*. Le serveur crée alors l'entité et lui attribue un nouvel URI, tel que */api/deliveries/39660*. Cet URI est renvoyé dans l'en-tête Location de la réponse. Chaque fois que le client envoie une requête, le serveur crée une nouvelle entité avec un nouvel URI.

Pour une méthode PUT, l'URI identifie l'entité. S'il existe déjà une entité avec cet URI, le serveur remplace l'entité existante par la version dans la demande. Si aucune entité n'existe avec cet URI, le serveur en crée une. Par exemple, si le client envoie une requête PUT à *api/deliveries/39660*, et en supposant qu'il n'y ait pas de livraison avec cet URI, le serveur en créera un nouveau. Maintenant, si le client envoie à nouveau la même requête, le serveur remplacera l'entité existante.

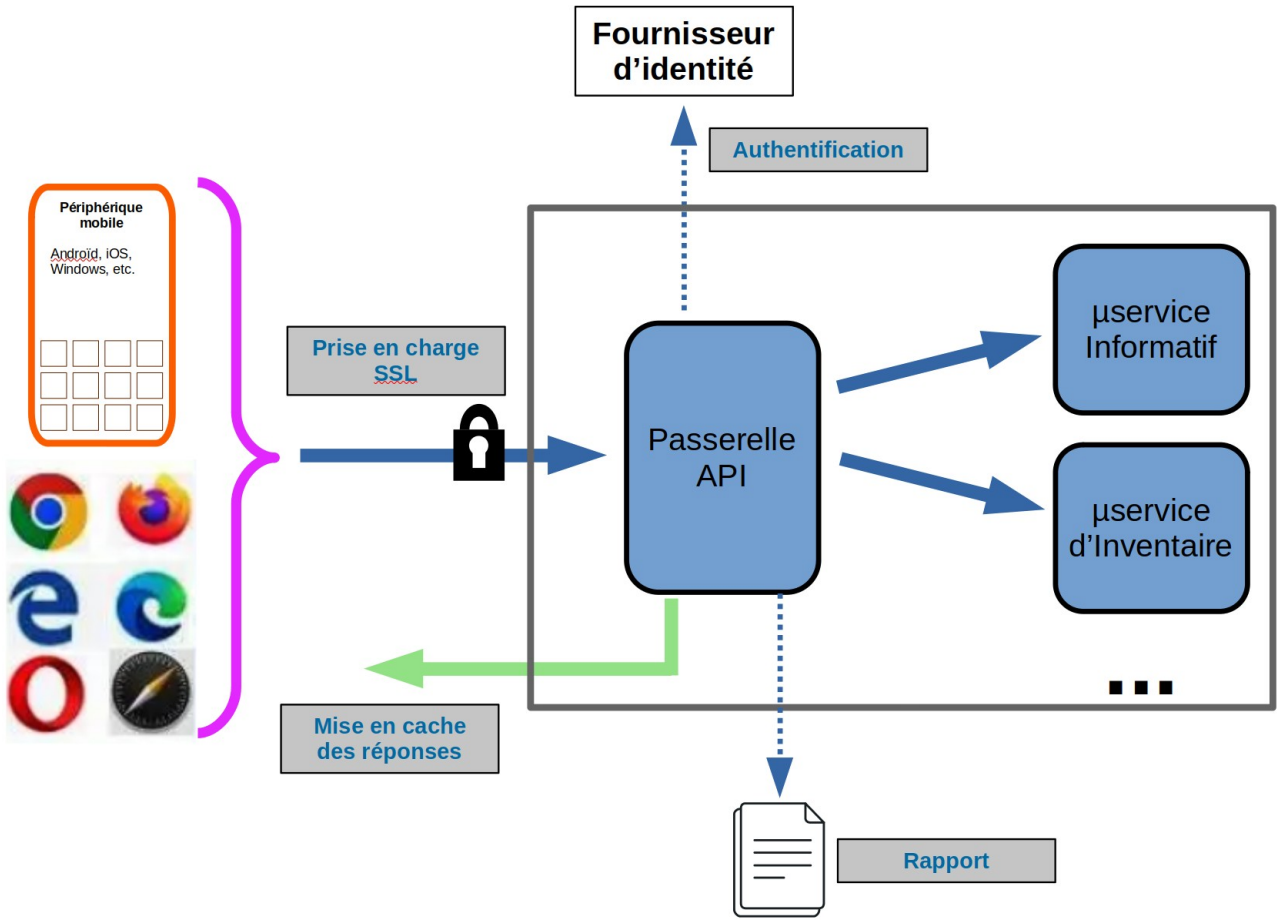
IV.E. Passerelles API pour les µservices

Dans une MSA, un client peut interagir avec plusieurs services frontaux.

En tenant compte de ce fait, les développeurs peuvent s'interroger sur plusieurs questions légitimes :

- comment un client sait-il quels terminaux appeler ?
- que se passe-t-il lorsque de nouveaux services sont introduits ou que des services existants sont en *refactoring* ?
- comment les services gèrent-ils la terminaison SSL, l'authentification et d'autres problèmes ?

Pour répondre à toutes ces questions, il n'y a qu'une seule et même réponse : l'utilisation d'une passerelle API, telle que schématisée ci-dessous :



Ainsi, comme il est possible de le constater au sein du schéma ci-dessus, une passerelle API se situe entre les clients et les µservices.

Il agit comme un proxy inverse, acheminant les demandes des clients vers les services.

Il peut également effectuer diverses tâches transversales telles que l'authentification, la terminaison SSL, la limitation du débit…

Si aucune passerelle n'est déployée, alors cela signifie que les clients devront envoyer toutes leurs requêtes directement aux services frontaux.

Cependant, il existe certains problèmes potentiels liés à l'exposition directe des services aux clients :

- *génération de code de µservice complexe* : le client (ici il est question d'un µservice expéditeur d'une requête et non pas d'un client au sens commercial) doit suivre plusieurs points de terminaison et gérer les pannes de manière résiliente ;
- *création d'un couplage entre µservices et le backend : p*our faire cela, le µservice a besoin de savoir comment les services individuels sont décomposés. Ce procédé rend plus difficile la maintenance et le *refactoring* des services ;
- *appels à plusieurs µservices pour une seule et même opération* : ce processus peut entraîner plusieurs allers-retours réseau entre le client et le serveur, ce qui ajoute une latence importante ;
- *gestion des problèmes d'authentification, SSL et limitation du débit client pour chaque µservice public* : les µservices doivent exposer un protocole convivial pour le client, tel que HTTP. Néanmoins, cela limite le choix des protocoles de communication. Les services avec des points de terminaison publics sont une surface d'attaque potentielle et doivent être renforcés.

Une passerelle aide à résoudre TOUS les problèmes cités ci-dessus en dissociant les clients des services.

Les passerelles peuvent exécuter un certain nombre de fonctions différentes regroupées dans les modèles de conception suivants :

- **le routage de passerelle** : une passerelle peut être utilisée comme proxy inverse pour acheminer les demandes vers un ou plusieurs services principaux, à l'aide du routage de couche 7 du modèle OSI. La passerelle fournit un point de terminaison unique pour les clients et aide à dissocier les clients des services ;
- **l'agrégation de passerelle** : une passerelle peut agréger plusieurs demandes individuelles en une seule demande. Ce modèle s'applique lorsqu'une seule opération nécessite des appels à plusieurs services principaux. Le client envoie une demande à la passerelle. La passerelle distribue les requêtes aux différents services *backend*, puis agrège les résultats et les renvoie au client. Cela aide à réduire le bavardage entre le client et le *backend,* en concordance avec le modèle *LEAN* de FOOSUS *;*
- **le déchargement** : une passerelle peut décharger les fonctionnalités de certains services individuels vers elle-même, en particulier les problèmes transversaux. Il est alors utile de regrouper ces fonctions en un seul endroit, plutôt que de confier à chaque service la responsabilité de les mettre en œuvre. Cela est particulièrement vrai pour les fonctionnalités qui nécessitent des compétences spécialisées pour être correctement mises en œuvre, telles que l'authentification et l'autorisation. En ce qui concerne FOOSUS, la passerelle pourrait décharger la résiliation SSL, les µservices d'authentification SSL, la gestion des listes d'adresses IP autorisées/bloquées, la limitation du débit client (*throttling*), la journalisation et la surveillance des µservices, la mise en cache des réponses Firewall d'applications Web, la compression de fichiers et la gestion du contenu statique.

IV.F. Considérations relatives aux données

Pour la contexte de FOOSUS, les différentes équipes de développement devront tenir compte de différentes considérations relatives à la gestion des données dans une MSA.
Étant donné que chaque µservice gère ses propres données, l'intégrité et la cohérence des données sont des défis à haute criticité.
Un principe de base des µservices est que chaque service gère ses propres données.
Ainsi, deux µservices ne doivent JAMAIS partager un magasin de données.
Au lieu de cela, chaque service est responsable de son propre magasin de données privé, auquel les autres µservices ne peuvent pas accéder directement.
La raison de cette règle est d'éviter un couplage involontaire entre les µservices, qui peut se produire si ceux-ci partagent les mêmes schémas de données sous-jacents.
S'il y a un changement dans le schéma de données, le changement doit être coordonné à travers chaque µservice qui s'appuie sur cette base de données, au cas par cas.
En isolant le magasin de données de chaque µservice, il est alors possible de limiter la portée du changement et préserver l'agilité de déploiements véritablement indépendants.
Une autre raison de ce cloisonnement de données est que chaque µservice peut avoir ses propres modèles de données, requêtes ou modèles de lecture/écriture. L'utilisation d'un magasin de données partagé limite la capacité de chaque équipe à optimiser le stockage des données pour son service particulier.
Cette approche conduit naturellement à la persistance polyglotte, c’est à dire l'utilisation de plusieurs technologies de stockage de données au sein d'une seule application.
Un µservice peut nécessiter les capacités de schéma à la lecture d'une base de données de documents. Un autre pourrait avoir besoin de l'intégrité référentielle fournie par un SGBDR. Chaque équipe est libre de faire le meilleur choix pour sa prestation. Pour en savoir plus sur le principe général de la persistance polyglotte, consultez Utiliser le meilleur magasin de données pour le travail.
Cette approche distribuée de la gestion des données relève néanmoins certaines difficultés, telle que la redondance dans les magasins de données, le même élément de données apparaissant à plusieurs endroits. Par exemple, les données peuvent être stockées dans le cadre d'une transaction, puis stockées ailleurs à des fins d'analyse, de création de rapports ou d'archivage. Les données dupliquées ou partitionnées peuvent entraîner des problèmes d'intégrité et de cohérence des données. Lorsque les relations de données s'étendent sur plusieurs %services, il n’est plus acceptable d’utiliser les techniques traditionnelles de gestion des données pour appliquer les relations ; la modélisation traditionnelle des données utilisant la règle « *un fait à un endroit* ». Chaque entité apparaît exactement une fois dans le schéma. D'autres entités peuvent contenir des références à celui-ci mais pas le dupliquer.
L'avantage évident de l'approche traditionnelle est que les mises à jour sont effectuées en un seul endroit, ce qui évite les problèmes de cohérence des données. Dans une MSA, il faudra tenir compte de la manière dont les mises à jour sont propagées entre les µservices et de la manière de gérer la cohérence éventuelle lorsque les données apparaissent à plusieurs endroits sans cohérence forte.

IV.G. Modèles de conception pour les µservices

L'objectif des µservices est d'augmenter la vitesse des versions d'applications, en décomposant l'application elle-même en µservices autonomes pouvant être déployés indépendamment. Une MSA pose néanmoins certains défis relatifs aux modèles de conception du développement de ces µservices.

Les modèles de conception présentés ci-dessous sont à adapter au cas par cas pour chaque µservice développé. De plus, cette liste n’est pas exhaustive et sera à compléter au fur et à mesure de l’avancée du projet de migration de FOOSUS :

- **Design Pattern *Ambassador*** : ce modèle de conception peut être utilisé pour décharger les tâches de connectivité client courantes telles que la surveillance, la journalisation, le routage et la sécurité (telles que TLS) d'une manière indépendante de la langue. Les services *Ambassador* sont souvent déployés en side-car (voir ci-dessous).
- **Design Pattern Couche anti-corruption** : ce modèle de conception implémente une façade entre les applications nouvelles et héritées, pour garantir que la conception d'une nouvelle application n'est pas limitée par des dépendances sur les systèmes hérités.
- **Design Pattern *Backends for Frontends :*** ce modèle de conception crée des services *backend* distincts pour différents types de clients, tels que les ordinateurs de bureau et les appareils mobiles. De cette façon, un service *backend* unique n'a pas besoin de gérer les exigences conflictuelles des différents types de clients. Ce modèle peut aider à garder chaque µservice simple, en séparant les préoccupations spécifiques au client.
- **Design Pattern *Bulkhead*** : ce modèle de conception isole les ressources critiques, telles que le pool de connexions, la mémoire et le processeur, pour chaque charge de travail ou service. En utilisant des cloisons, une seule charge de travail (ou service) ne peut pas consommer toutes les ressources, affamant les autres. Ce modèle augmente la résilience du système en empêchant les défaillances en cascade causées par un service.
- **Design Pattern d’agrégation de passerelle** : ce modèle de conception regroupe les requêtes adressées à plusieurs µservices individuels en une seule requête, réduisant ainsi le bavardage entre les consommateurs et les services (*concept LEAN cher à FOOSUS*).
- **Design Pattern de déchargement de passerelle** : ce modèle de conception permet à chaque µservice de décharger les fonctionnalités de service partagé, telles que l'utilisation de certificats SSL, vers une passerelle API.
- **Design Pattern de routage de passerelle** : ce modèle de conception achemine les demandes vers plusieurs µservices à l'aide d'un seul point de terminaison, de sorte que les consommateurs n'ont pas besoin de gérer plusieurs points de terminaison distincts.
- **Design Pattern *Sidecar*** : ce modèle de conception déploie les composants d'assistance d'une application en tant que conteneur ou processus distinct pour assurer l'isolation et l'encapsulation.
- **Design Pattern *Strangler Fig*** : prend en charge le processus de *refactoring* incrémentielle d'une application, en remplaçant progressivement des fonctionnalités spécifiques par de nouveaux services.

V. Processus et rôles de développement

V.A. Processus de développement

La première question à se poser sera relative aux différences existantes entre l'intégration classique et l'intégration continue.

En ce qui concerne l'intégration classique, celle-ci consiste à assembler, à la fin du projet, toutes les fonctionnalités développées. Ainsi, suite à la liaison de toutes ces parties logicielles, une longue phase de test et de correction d'anomalies est entamée. Il faut garder toutefois à l'esprit qu'il est extrêmement complexe et chronophage de corriger un code déjà réalisé plutôt que de tester celui-ci au fur et à mesure de sa rédaction. Cette étape entraîne donc des coûts supplémentaires et des délais pouvant impacter fortement la vie du projet.

L'intégration continue (IC) essaie de répondre aux difficultés énoncées ci-dessus. En cela, l'IC va segmenter les différentes étapes par l'utilisation de plusieurs outils spécialisés dans chacun de ces domaines interconnectés. Cet ensemble d'outils permettent ainsi de gérer finement l'avancée du projet, de configurer chaque étape de compilation et de générer des rapports ou des métadonnées à propos du projet. Le cycle itératif d'un élément de projet se découpe en 5 principales étapes:

- **Etape 1** : les différents développeurs travaillent de concert pour développer les fonctionnalités et modules qui leur sont attribués. Ainsi, chacun d'eux veille à tester son code à l'aide de tests unitaires et s'assurent de sa bonne intégration au sein du système. Une fois testé chaque développeur fusionne son code avec le code existant. Par la suite, les autres développeurs pourront accéder à cette modification afin de le relire, l'optimiser si nécessaire et le mettre à jour.
- **Etape 2** : l'utilisation d'un outil de gestion de version permet d'avoir une certaine visibilité sur l'avancée du projet. Ce suivi sera alors réalisé progressivement au fur et à mesure des fusions (commit) de modification réalisée par les développeurs.
- **Etape 3** : la compilation de chaque modification de code sera automatiquement associée à l'exécution automatique de tests fonctionnels préalablement écrits en fonction du *Backlog Product* existant.
- **Etape 4** : après la compilation réalisée et les tests exécutés, des informations quant au succès ou à l'échec de ces opérations sont remontées aux intervenants. Ces rapports pourront concerner la Qualité, la stabilité ou la découverte d'anomalie non répertoriée jusqu'à présent.
- **Etape 5** : l'analyse de ces rapports va permettre de reprendre le code incriminé et de l'améliorer pour qu'il réponde au besoin pour lequel il avait initialement été rédigé.

Grâce aux tests très réguliers du code rédigé, les anomalies sont détectées dès leur apparition. Il est ainsi plus aisé et plus immédiat de corriger ces anomalies en reprenant une petite partie de code - la seule partie ciblée par le test unitaire. Ainsi, le projet n'étant pas terminé, il y aura dès le moyen terme moins de parties de code à modifier, et donc un énorme gain de temps in fine.

L'application de l'ensemble de ces techniques nécessitera que :

- les tests unitaires soient écrits AVANT la rédaction du code source (*Test Driven Development*);
- un dépôt unique soit identifié et structuré pour permettre un accès au code source à chaque membre de l'équipe (la constitution de cette équipe sera définie dans un chapitre ultérieur);
- ce dépôt devra également permettre de mettre en place un système et/ou un outil de suivi de version (*versioning*) dans une optique de traçabilité, de sauvegarde et d'aide au maintien de la vision Produit;
- chaque développeur intègre ses modifications du code source au fur et à mesure du développement de chaque fonctionnalité;
- des tests d'intégration valident, si nécessaire, les imports de version dans une branche du logiciel de suivi de version différente que celle utilisée pour développer la modification du code en question.

Ainsi, les bénéfices, à court terme, de la mise en œuvre de ces "bonnes pratiques" seront :

- la vérification fréquente du code source et, donc, de sa bonne compilation;
- la réalisation systématique des tests unitaires et/ou fonctionnels;
- l'alerte immédiate en cas de régression fonctionnelle ou d'incompatibilité de code;
- la détection des problèmes d'intégration;
- la réparation et la maintenance continue, évitant les problèmes de dernière minute;
- au moins, toujours une version disponible pour un test, une distribution à fournir ou simplement une démonstration de l'incrément fourni;
- la possibilité d'obtenir périodiquement des remontées d'indicateurs cohérents, relatifs à la Qualité du code, la couverture des tests, la visibilité sur le travail qui a déjà été réalisé et de ce qui reste à faire…

V.B. Rôles de développement

L'ensemble des rôles définis dans les paragraphes qui suivent constituera une équipe de développement. Ainsi, dans plusieurs équipes nous rencontrerons donc forcément plusieurs fois le même rôle.

La méthode utilisée lors de la migration de FOOSUS sera SCRUM. Cette méthode permettra d'intégrer aisément le kanban utilisé historiquement par les équipes de développement et définira de nouvelles phases de développement très structurée.

Cette équipe SCR sera constituée de trois rôles essentiels :

- le Product Owner (PO) ;
- le SCRUM Master (SM) ;
- les développeurs (devs) .

V.B.1. Product Owner

Le Product Owner ou PO est la personne se rapprochant le plus du client final.

Il a la vision fonctionnelle du produit vers laquelle les développeurs doivent aboutir, et ne s'occupe que de celle-ci : il n'entre pas dans des considérations techniques, chasse gardée des développeurs.

La fonction principale du PO est ainsi de retranscrire les besoins fonctionnels du client au travers d'US (*User's Story*). Le PO n'est pas seul dans l'élaboration des US et est aidé par le *Scrum Master (*SM*)* et les développeurs en ce qui concerne le cadre et l'affinage des différentes US présentes au sein du *Product Backlog*.

Le PO est ainsi responsable envers les développeurs :

- de fournir une vision à court et moyen terme du produit, en projetant et en mettant à jour le *Release Plan* ;
- de travailler et affiner (en collaboration avec les développeurs) les US jusqu'à l'aboutissement à un DOR (*Definition Of Ready*) ;
- De tenir un rythme d'alimentation du *Product Backlog* et ainsi avoir suffisamment d'US à l'état *ready* dans le *Product Backlog* ;
- d'avoir une planification équilibrée ;
- d'être disponible pour trancher un choix ayant un impact fonctionnel.

Il définit en outre la forme et le contenu de la livraison.

V.B.2. Scrum Master

Le Scrum Master est avant tout un membre à part entière de l'équipe SCRUM.

Il est le garant du cadre méthodologique SCRUM. Ainsi, il n'a pas vocation à la diriger, mais à la guider dans l'application du cadre méthodologique Scrum.

Son rôle est d'**aider l'équipe à avancer** de manière autonome et en cherchant en permanence à s'améliorer. Pour y arriver, il sert d'interface entre l'équipe et le monde extérieur, la protégeant de tout élément susceptible de perturber son fonctionnement et sa concentration.

Au sein de l'équipe, il a pour mission de **former les membres aux pratiques agiles** et d'animer les différents « rituels » de Scrum : mêlées quotidiennes, planning pokers, rétrospectives…

Il peut être considéré commet un coach, et non comme un supérieur.

En effet, sa mission n'est pas de décider du rôle de chacun durant **les sprints** (les itérations de l'équipe) comme le ferait un chef de projet classique, car cette répartition revient à l'équipe elle-même.

Le rôle de Scrum Master est de s'assurer de l'implication de chaque membre et de les aider à franchir les différents obstacles qu'ils pourraient rencontrer.

Comme un coach, le Scrum Master n'a pas vocation à résoudre directement les problèmes, mais à **aider son équipe** dans la recherche et l'identification de solutions. Plus l'équipe gagnera en expérience et en autonomie, plus le périmètre et l'impact du Scrum Master se réduiront.

Pour réaliser ceci, le SM doit posséder des compétences humaines indispensables. Il doit s'assurer à la fois de l'implication de chaque membre de l'équipe, mais également de l'**auto-organisation** de celle-ci afin de respecter au mieux **le cadre méthodologique Scrum**.

Pour assurer cette fonction de "Coach", le Scrum Master doit nécessairement présenter de fortes "compétences douces" ou soft skills. La diplomatie, l'empathie, la pédagogie ou encore l'humilité seront ses meilleurs atouts.

De plus, outre le fait de s'assurer de l'**harmonie au sein de l'équipe**, il devra aussi s'assurer de la bonne compréhension entre le *Product Owner* et l'équipe de devs, et donc de la traduction des besoins techniques en besoins fonctionnels.

Pour conclure, un SCRUM Master peut être comparer à un maître de jeu. Il exposera aux autres joueurs le but commun et leur fonction respective. C'est également lui qui sera le trait d'union entre les règles dont il est le garant et les joueurs. Il sera amené à influencer les joueurs de manière plus ou moins subtile afin de les amener dans la bonne direction.

Cependant, les joueurs sont totalement libres de leurs actions et décident eux même de la manière dont ils arriveront à leur fin. Il devra **stimuler la créativité** des joueurs grâce à la liberté dont ils disposent.

V.B.3. Développeur Scrum

Le rôle des développeurs Scrum comprendra les fonctions suivantes :

- comprendre les exigences business, spécifiées par le Product Owner ;
- estimer les US dans le *backlog product* ou *backlog* de sprint ;
- développer le produit / service (livrables).

L'ensemble des développeurs Scrum est responsable de la livraison d'un élément fonctionnel (ou morceau) du produit à la fin de chaque sprint.

Chaque développeur Scrum sera également responsable de :

- livrer les travaux du sprint ;
- comprendre les besoins de l'entreprise ;
- assurer la transparence du développement ;
- l'auto-organisation de l'équipe entière.

Pour faire bref, l'esprit de l'équipe de développeurs Scrum est le collectif autonome.

VI. Mesure d’architecture cible

La mesure de la TBA sera réalisée dynamiquement durant tout le processus de développement même de FOOSUS. Pour réaliser ceci, les équipes de développement utiliseront les concepts **DevOps** réaliser le développement couplé à SCRUM en ce qui concerne la structure de l’équipe et l’organisation du projet.

VI.A. Définition de DevOps

DevOps (contraction de Développeur+Opérationnel) est un mouvement professionnel qui a vu le jour en 2008 et 2009 lors de ce que l’on nomme aujourd’hui les *DevOpsDays* (voir le site *devopsdays.org* pour de plus amples détails). Le terme « **DevOps** » fut employé pour la première fois par Patrick Debois en 2009 lors d’une conférence au *DevOpsDays*.

L’objectif de **DevOps** est de réconcilier les éléments de production (les Développeurs) avec les éléments d’opération (les Opérationnels).

Ce terme est un héritage découlant fonctionnellement du LEAN THINKING et techniquement d’AGILE.

Le LEAN THINKING est une approche fonctionnelle du projet d’un produit élimant tous les gaspillages (MUDA en japonais), et donc assurant une productivité tendant vers 100%. Cette politique est bien connue de FOOSUS puisqu’elle la met en pratique de puis sa création.

AGILE est une approche technique du projet d’un produit amenant de la flexibilité à sa réalisation, en terme d’organisation d’équipe et d’outillage.

Le point commun de ces 2 différentes approches est la **Qualité** ; point incontournable sur lequel les 2 approches font leur priorité.

Après avoir fusionné ces 2 approche projets, DevOps est né avec comme leitmotiv : « *plus vite, plus petit, plus souvent* ».

L’idée maîtresse ici est d’optimiser le flux, la productivité et, in fine, la Qualité.

Ainsi, il est possible de mettre en exergue six éléments essentiels pour DevOps :

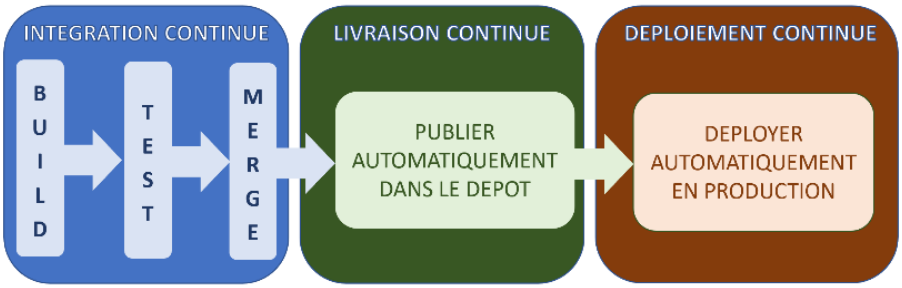
- la chaîne de Valeurs ;
- la suppression des goulets d’étranglement ;
- la suppression des gaspillages ;
- la diminution de la taille des lots jusqu’au « pièce à pièce » ;
- l’arrêt au premier défaut ;
- les unités autonomes de production ou équipes intégrées.

Ainsi DevOps peut avoir plusieurs objectifs différents en fonction de la nature du projet dont il est question, à savoir :

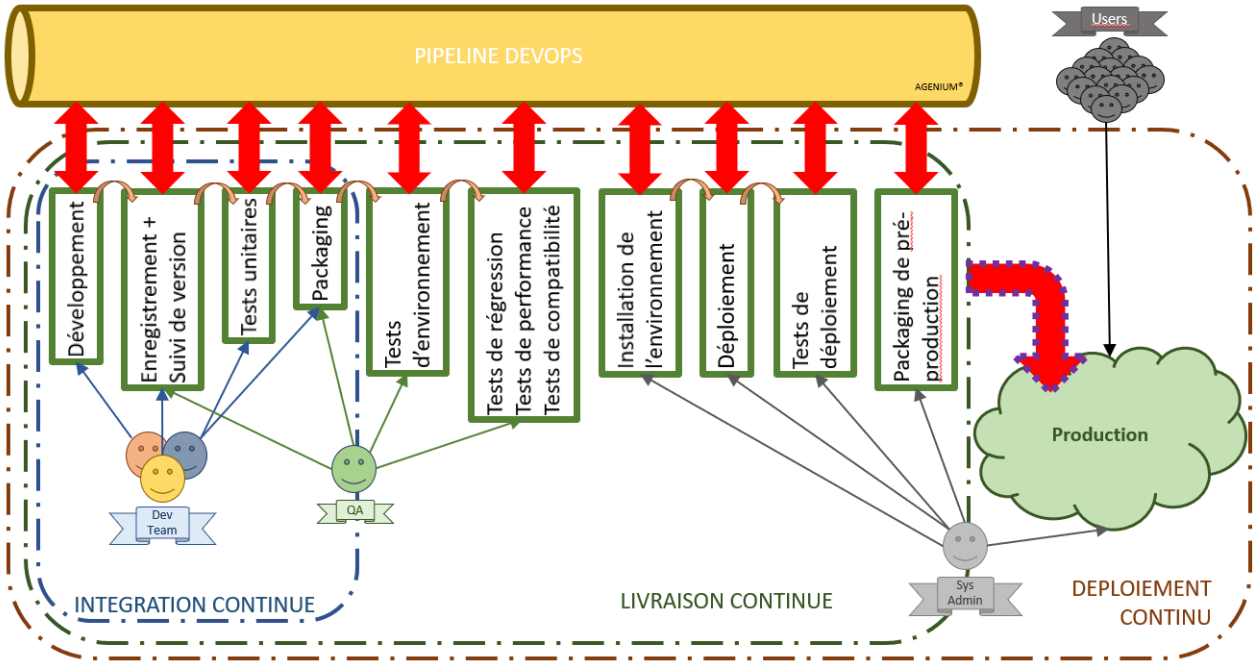
- L'Intégration Continue (IC) ;
- La Livraison Continue (LC) ;
- Le Déploiement Continue (DpC).

En outre, pour les trois notions ci-dessus, il est essentiel de garder à l'esprit un concept primordial qu'est l'**Amélioration Continue**, et qui ne sera pas traité au sein de ce document, mais qui est omniprésent dans chacune de ces notions.

En termes de séquence, et d'un point de vue purement fonctionnel, nous pourrions schématiser les trois points précédents selon le diagramme suivant :



En termes d'activités techniques, et afin d'obtenir une vision un peu plus détaillée de ces concepts, il serait intéressant de considérer le diagramme suivant :



Ainsi, le choix d'utilisation de tel ou tel composant sera fonction de la nature du projet dont il est question.

En se basant sur le diagramme d'activités techniques ci-dessus, les chapitres qui suivront seront architecturés selon les différents composants présentés.

VI.A.1. Le pipeline DevOps

Le premier élément à décrire dans un pipeline DevOps est le pipeline lui-même.

Un pipeline DevOps est un ensemble de pratiques pour lesquelles les équipes de développeurs (Devs) et les équipes d'opérationnels (Ops) concrétisent, testent, et déploient un produit logiciel plus vite et plus facilement que s'ils devaient le faire manuellement, car tout l'intérêt du pipeline réside dans l'automatisation de ces tâches : les développeurs peuvent alors consacrer plus de temps à développer.

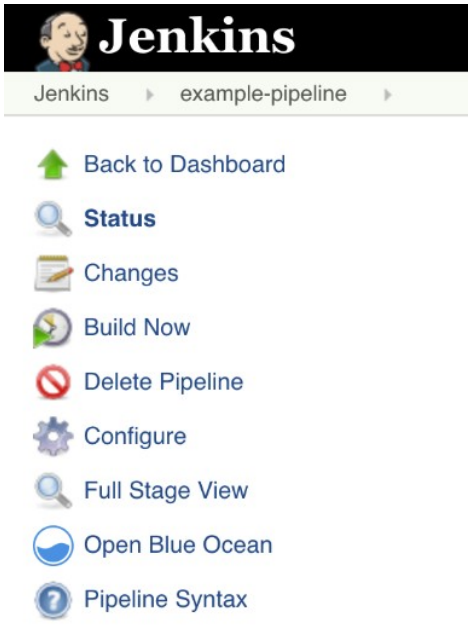
Une des premières caractéristiques d'un pipeline est de conserver le process de développement logiciel organisé et focalisé.

Ainsi, un pipeline DevOps va séquentiellement réaliser les étapes suivantes :

- Planifier ;
- Développer ;
- Compiler ;
- Tester ;
- Déployer ;
- Monitorer (action technique de surveillance d'un sujet).

En termes d'outillage, le choix proposé au sein de FOOSUS tournera autour de *Pipeline as Code with Jenkins*, plus communément appelé Pipeline.

En effet, *Jenkins Pipeline* est une suite de plugins supportant l'implémentation de l'intégration continue et de la livraison continue au sein d'un pipeline standard déclaré dans une instance standard de *Jenkins*. Le pipeline fournit ainsi un ensemble extensible d'outils pour la modélisation « Simple-To-Complex » d'une pipeline de livraison « as *cod*e » via le plugin Pipeline DSL (Domain Specific Language).



VI.A.2. Intégration Continue

VI.A.2.a. Phase d'analyse et de conception

La phase d'analyse est une étape essentielle dans la conception d'un logiciel et un prérequis au développement.

Néanmoins, il est vrai qu'elle n'apparaît pas dans le diagramme d'activités techniques en UML car elle est considérée comme une partie réalisée en amont du développement, voire pendant.

En ce qui concerne la méthode d'analyse, l'UML (*Unified Modeling Language*) sera à préconisé pour FOOSUS.

En effet, UML est particulièrement utilisé et adapté en Génie Logiciel et en conception orientée objet. Son principal intérêt est de modéliser GRAPHIQUEMENT les différents éléments constituant un système, selon trois types de diagramme :

- des diagrammes de structure ;
- des diagrammes de comportement ;
- des diagrammes d'interaction.

Le détail de chaque diagramme est présenté dans le tableau ci-dessous :

Catégorie	**Type de diagramme**	**utilité**
Diagramme de structure	Diagramme de classes	Représente les classes
	Diagramme d'objet	Affiche l'état d'un système à un moment donné
	Diagramme des composants	Affiche les dépendances et les composants de structure
	Diagramme de structure composite	Divise les modules ou les classes en leurs composantes et clarifie leurs relations.
	Diagramme de paquetage	Regroupe les classes en paquets, représente la hiérarchie et la structure des paquets
	Diagramme de déploiement	Affiche la distribution des composants aux nœuds de l'ordinateur
	Diagramme de profil	Illustre les relations d'usage au moyen de stéréotypes, de conditions aux limites, etc.
	Diagramme de cas d'utilisation	Représente divers usages
Diagrammes de comportement	Diagramme d'activité	Décrit le comportement de différents processus (parallèles) dans un système.
	Diagramme d'état-transition	Documente la façon dont un objet passe d'un état à un autre par le biais d'un événement.
Diagrammes d'interaction	Diagramme de séquence	Représente le moment des interactions entre les objets.
	Diagramme de communication	Affiche la répartition des rôles des objets au sein d'une interaction.
	Diagramme de temps	Démontre la limitation temporelle pour les événements qui mènent à un changement d'état.
	Diagramme d'aperçu d'interaction	Montre comment les séquences et les activités interagissent.

VI.A.3. Suivi de version

Un logiciel de gestion de version est composé d'une arborescence de fichiers permettant de conserver toutes les versions des fichiers, ainsi que leur historique de différence entre chaque mise à jour (*check out*).

Cette notion de suivi de version s'accompagne intrinsèquement par la notion de branche de version ou branche de développement.

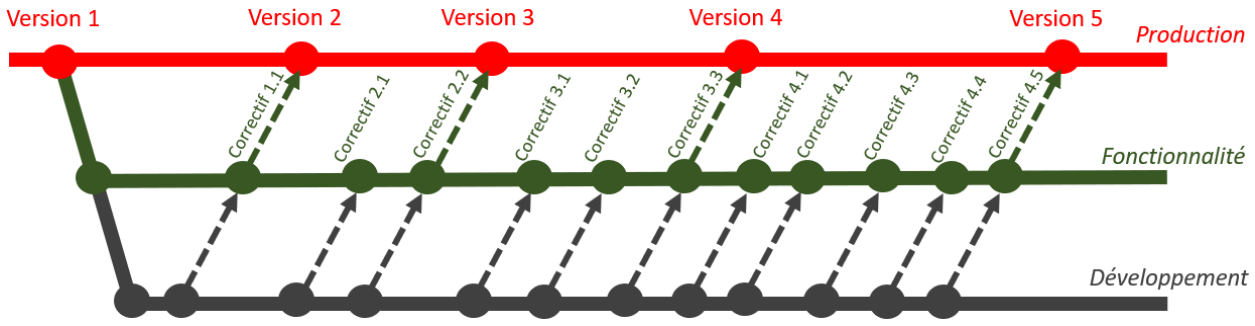
Avant d'aborder le type de branchement choisi au sein de FOOSUS parmi tous les types de branchement existant, il est nécessaire de décrire l'objectif de FOOSUS : avoir le minimum de branches possibles.

A ce stade, il est alors normal de se poser la question quant à la légitimité d'un tel axiome. Après tout, avoir beaucoup de branches différentes « devrait être » synonyme de flexibilité ou de persistance. Cependant, il n'en est pas du tout ainsi.

En effet, le fait d'avoir beaucoup de branches engendrent beaucoup de gaspillage de temps et de ressources, et c'est précisément cela que le DevOps s'efforce de minimiser, voire de supprimer.

Aussi, afin de réduire le champ de divergence entre branche et améliorer la Qualité ainsi que la productivité de reprise de code, les développeurs de FOOSUS s'efforceront à avoir le minimum de branche de version possible.

Aussi, la proposition ci-dessous se veut être une combinaison de l'axiome décrit précédemment (« avoir le minimum de branche possible ») tout en prenant en compte l'aspect opérationnel sur lequel l'appliquer :



VI.A.4. tests unitaires

Le test unitaire consiste à isoler une partie du code et à vérifier qu'il fonctionne parfaitement.

Il s'agit de petits tests qui valident l'attitude d'un objet et la logique du code.

Les tests unitaires sont effectués pendant la phase de développement des applications logicielles.

Ces tests sont effectués par les développeurs ; ils peuvent également être effectués par les responsables en assurance (QA *Quality Assurance*) afin de s'assurer particulièrement sur telle ou partie du code.

En termes de bonne pratique Qualité, FOOSUS favorisera fortement l'utilisation du TDD (*Test Driven Development*), voire du BDD (*Behavior Driven Developement*). Néanmoins, cette étude privilégiera le TDD et fera abstraction du BDD. En effet, ce dernier est une pratique AGILE orientée vers le fonctionnel, alors que ce document se veut être un ensemble de recommandations techniques.

L'utilisation du TDD est motivée par les raisons suivantes :

- le test unitaire révèle si la logique derrière le code est appropriée et si elle fonctionnera dans tous les cas ;
- le test unitaire améliore la lisibilité du code et aide les développeurs à comprendre le code de base, ce qui facilite grandement la mise en œuvre des modifications et cela, plus rapidement ;
- des tests unitaires bien conduits sont également de bons outils pour la documentation du projet ;
- en termes de performance, les tests sont effectués en un peu plus de quelques millisecondes, ce qui vous permet d'en réaliser des centaines en très peu de temps ;
- le test unitaire permet au développeur de remanier le code ultérieurement et de s'assurer que le module continue à fonctionner correctement. Des cas de test sont écrits à cet effet pour toutes les fonctions et méthodes afin que les erreurs puissent être rapidement identifiées et réparées, chaque fois que l'une d'elles est créée par l'introduction d'un changement dans le code ;
- la qualité finale du code s'améliorera parce qu'il s'agira en fin de compte d'un code propre et de haute qualité grâce à ces essais continus ;
- puisque le test unitaire divise le code en petits fragments, il est possible de tester différentes parties du projet sans avoir à attendre que d'autres parties soient terminées. Ici l'utilisation de Mock prend tout son sens et sa dimension.

En termes de framework d'utilisation, sur un projet à dominante technologique Java, il sera conseillé de travailler avec *Junit*. L'utilisation de *TestNG* - basé sur *JUnit* - est également possible.

VII. Phases définies des livrables

Il convient en premier lieu d'identifier une « phase » au sein du projet de migration de la plateforme FOOSUS.

Une phase correspondra à un découpage temporel ayant pour but de réaliser un certain nombre de tâches techniques annoncées et définies en amont du début de la phase.

En outre, comme il a été annoncé précédemment le projet sera organisé autour de deux méthodologies :

- gestion de projet et structuration de l'équipe : **SCRUM** ;
- développement technique : **DevOps**.

Le fait de coupler ces deux méthodologies a pour conséquence de phaser la livraison des différents livrables selon des périodes temporelles fixes et annoncées.

VII.A.Définition d'un Sprint

Un Sprint est une phase séquentielle de développement du produit.

Il s'agit de « **Sprint »**, car ce sont des itérations de relatives courtes durées pendant lesquelles l'équipe de développement réalise des tâches préalablement définies lors du *Sprint Planning*.

Ces cycles permettent de décomposer un processus de développement souvent très complexe afin de le rendre plus simple et plus facile à réadapter en fonction du résultat des évaluations intermédiaires.

Ainsi, un **Sprint** sera organisé en cinq cérémonies :

- le *Sprint Planning* ;
- le *Daily Meeting* ;
- la *Sprint Review* ;
- la *Sprint Retrospective* ;
- l'affinage du *Product Backlog* ;

Le résultat d'un Sprint est appelé incrément, et correspond aux US du *Product Backlog* réalisées en fin de Sprint.

Maintenant qu'une « phase » a été définie comme un Sprint, il convient de considérer le projet dans son intégralité et de l'appréhender comme un ensemble de Sprint.

Aussi, afin de bien gérer tous ces sprints, nous allons les organiser selon un *Release Plan*.

VII.B. Définition d'un Release Plan

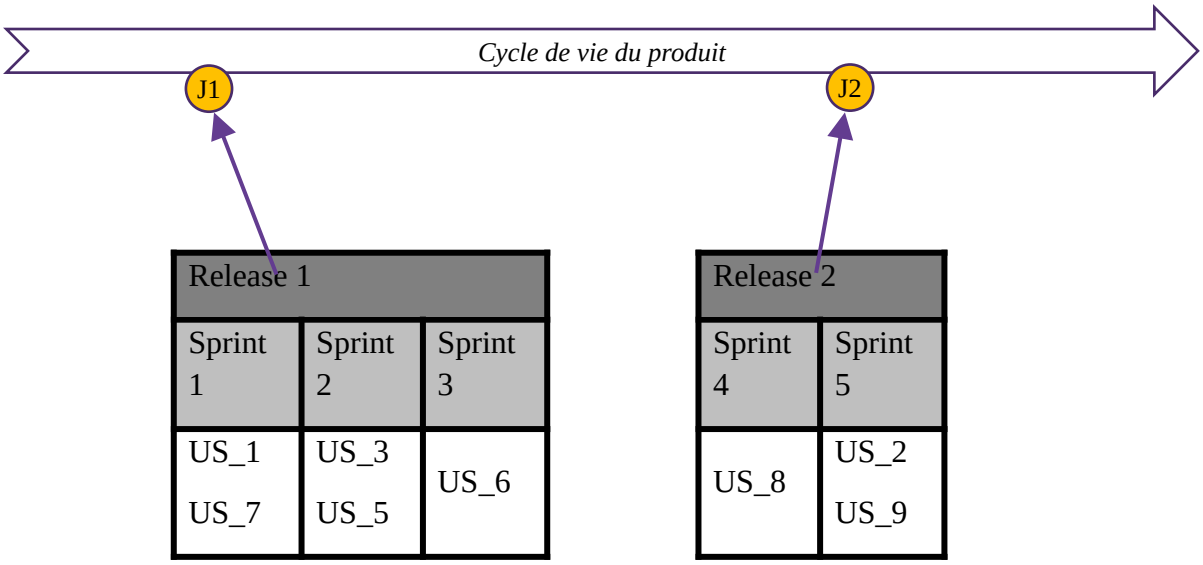
Le *Release Plan* définit et planifie les versions d'un produit, sachant qu'une version du produit est le résultat d'un ou plusieurs Sprint.

Pas toujours apprécié dans le monde de l'Agilité, le contexte projet impose parfois de devoir en réaliser un. C'est le cas de nombreux projets chez FOOSUS dans lesquels des jalons sont imposés par la migration elle-même afin de migrer sereinement dans les meilleurs délais.

La mise en place d'un *Release Plan* nécessite cependant certains pré-requis :

- les objectifs sont connus tant en termes de dates que de fonctionnalités à intégrer dans les différentes *releases* ;
- le *Product Backlog* doit être composé uniquement d'US à l'état « *Ready* », en incluant leur estimation ;
- la durée des Sprints est fixée dans l'optique de pouvoir estimer la capacité de l'équipe de développement à produire. Dans le contexte FOOSUS, la durée d'une sprint sera de trois semaines ;
- la stabilité de l'équipe doit être assurée (pas de perturbation hors projet).

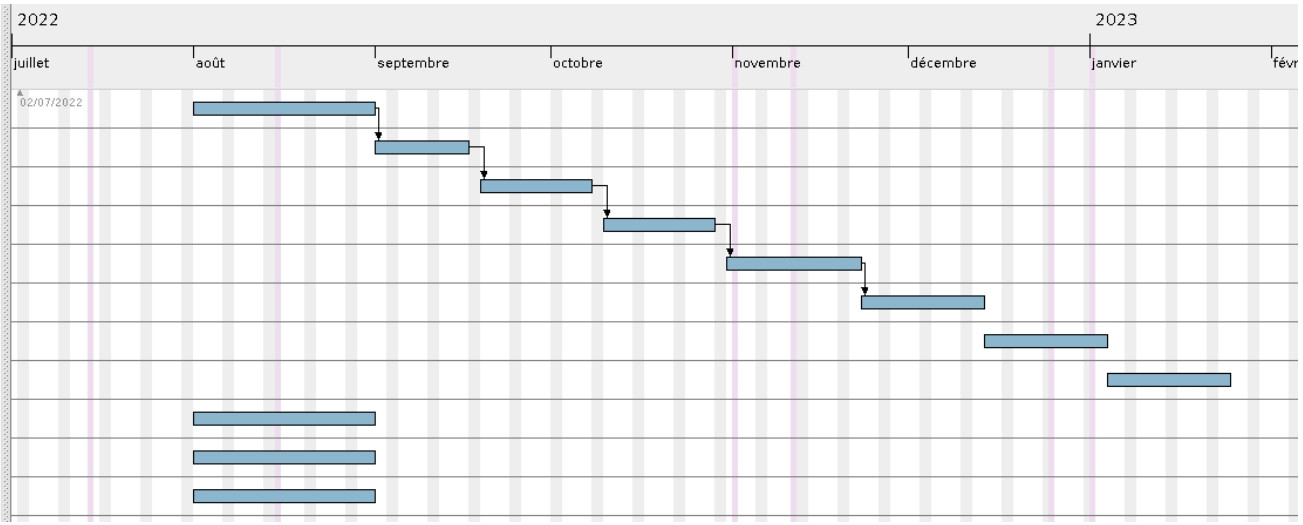
Un *Release Plan* peut être schématisé assez facilement de la manière suivante :



VIII. Plan de travail priorisé

Afin d'établir un plan de travail priorisé, adapté au projet de migration de FOOSUS, cette étude va se référer au plan de travail réalisé au sein document d'Enoncé des Travaux d'Architecture, *§Plan de travail*, représenté ci-après :

Nom	Date de début	Date de fin
Créer le catalogue de services	01/08/2022	31/08/2022
Intégrer la feature Informatif	01/09/2022	16/09/2022
Intégrer la feature Recherche	19/09/2022	07/10/2022
Intégrer la feature Inventaire	10/10/2022	28/10/2022
Intégrer la feature Facturation	31/10/2022	22/11/2022
Intégrer la feature Commande	23/11/2022	13/12/2022
Créer la feature de Géolocalisation	14/12/2022	03/01/2023
Créer la feature de calcul de la distance	04/01/2023	24/01/2023
Intégrer le système d'Identification	01/08/2022	31/08/2022
Intégrer le système d'Authentification	01/08/2022	31/08/2022
Créer le module de Crytptage SSL/TLS	01/08/2022	31/08/2022



Ainsi, il apparaît que les premières tâches de la figure ci-dessus représentent déjà les fonctions existantes au sein de la plateforme actuelle FOOSUS, à l'exception de la première relative à la création du catalogue de services.

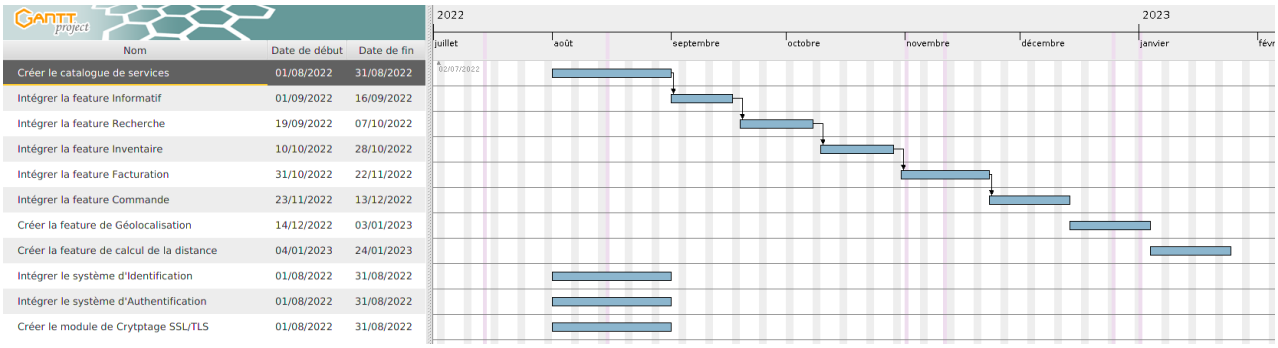
Aussi, la priorisation sera donc indiquée par ce Gantt, en commençant par créer le catalogue d'indexation des µservices, puis en intégrant les fonctionnalités existantes afin d'en standardiser la gestion.

Enfin, toutes les autres fonctionnalités à développer seront considérées au cas par cas par le *Product Owner* en terme de priorisation.

IX. Fenêtre temporelle

En ce qui concerne la migration de FOOSUS, la fenêtre temporelle correspondra à la marge pendant laquelle une tâche dans un réseau de projet peut être retardée sans entraîner de retard pour les autres tâches et la date d'achèvement du projet ; le flottement total est associé au chemin.

Pour le reste de ce paragraphe, il sera nécessaire de considérer la diagramme de Gantt ci-dessous relatif au document d'énoncé des travaux d'architecture :



Nom	Date de début	Date de fin
Créer le catalogue de services	01/08/2022	31/08/2022
Intégrer la feature Informatif	01/09/2022	16/09/2022
Intégrer la feature Recherche	19/09/2022	07/10/2022
Intégrer la feature Inventaire	10/10/2022	28/10/2022
Intégrer la feature Facturation	31/10/2022	22/11/2022
Intégrer la feature Commande	23/11/2022	13/12/2022
Créer la feature de Géolocalisation	14/12/2022	03/01/2023
Créer la feature de calcul de la distance	04/01/2023	24/01/2023
Intégrer le système d'Identification	01/08/2022	31/08/2022
Intégrer le système d'Authentification	01/08/2022	31/08/2022
Créer le module de Crytptage SSL/TLS	01/08/2022	31/08/2022

En considérant le diagramme de Gantt ci-dessus, il est possible d'identifier deux chemins non critiques, et donc deux valeurs flottantes :

- La création de la fonctionnalité de géolocalisation ;
- la création de la fonctionnalité relative au calcul de la distance.

Bien que ces deux fonctions se complètent, elles n'en demeurent pas moins différentes. Le fait de localiser un objet et le fait de calculer la distance entre cette objet et l'observateur, sont deux processus différents.

Ainsi, le flottant total représente la flexibilité du calendrier et peut également être mesuré en soustrayant les dates de début précoce des dates de début tardif de l'achèvement du chemin.

En outre, tel qu'exposé ci-dessus, les nouvelles fonctionnalités ajoutées à celles « *historiques* » seront toujours à considérer comme des fenêtres temporelle ne faisant pas partie du chemin critique, en considérant le système dans son ensemble. Néanmoins, en prenant comme référentiel une fonctionnalité particulière et non plus le système dans son ensemble, il sera alors possible de décomposer atomiquement cette fonctionnalité, et donc d'y identifier un chemin critique propre...

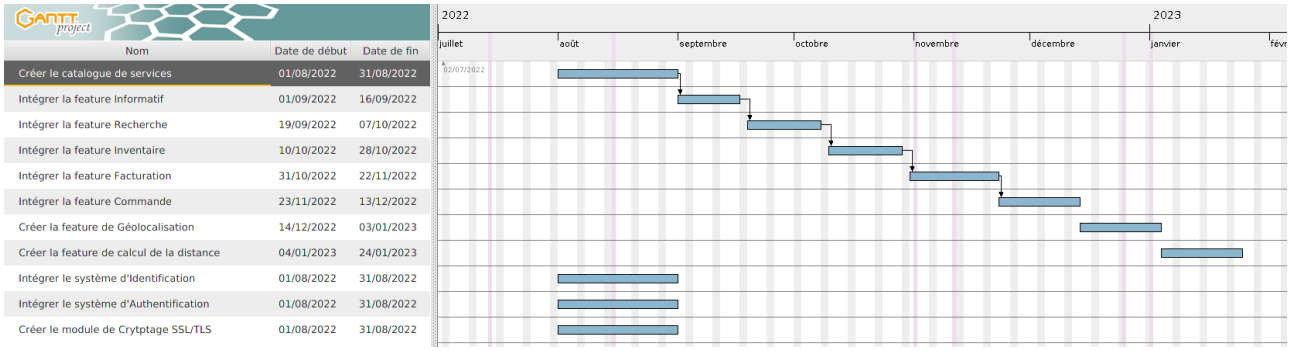
Toujours relativement au diagramme de Gantt précédent, il est alors possible d'envisager la fin de la migration de FOOSUS vers la MSA, pour décembre de cette année, en omettant la fenêtre temporelle.

X. Livraison de l’architecture et métriques commerciales

X.A. Livraison de l’architecture

Tel qu’annoncé dans le paragraphe §*Phases définies des livrables*, les livrables seront fournis à la fin de chaque Sprint.

Ayant convenu qu’un sprint aurait une durée de trois semaines, il est alors concevables d’attendre de l’équipe de développeurs un livrable pour chacune des phases définies dans le Gantt ci-dessous :



X.B. Métriques commerciales

En ce qui concerne les métriques commerciales, cette étude se basera sur le document de Contrat d’Architecture avec les utilisateurs métiers pour les définir. Ainsi, nous avons vu précédement les métriques commerciales suivantes :

- Planification (croissance de Chiffre d’Affaires, coûts, marge, trésorerie future) ;
- Prévisions annuelles ;
- Prévisions continues ;
- Analyse et extrapolation de l’évolution du Chiffre d’Affaires et de la marge.

Ces métriques mesureront la performance globale de la plateforme FOOSUS, en fonction des résultats qu’elle génère.

Les métriques clés de performance seront différents selon les objectifs. Il serait donc possible d'y rajouter :

- le nombre de ventes au sein de la plateforme ;
- le taux de réussite par fournisseur ;
- la valeur moyenne de chaque vente ;
- le temps moyen pour conclure une vente ;
- le revenu ou Chiffre d'Affaires généré sur une période définie ;
- la précision des prévisionnels de vente ;
- Le temps de réponse des fournisseurs pour garantir la disponibilité des produits vendus ;
- le temps passé à vendre, à rapprocher de l'efficacité de l'équipe commerciale ;
- le taux de réussite moyen de l'équipe commerciale ;
- les ventes réalisées ;
- les pourcentage d'atteinte des objectifs de vente ;
- les opportunités.
- Les KPI dépendront de la structure commerciale de FOOSUS.


Foosus