**D’une Architecture Monolithique Vers une Architecture micro-services**

[novembre 16, 2021](https://mohamedyoussfi.blogspot.com/2021/11/blogn-1-dunearchitecture-monolithique.html?m=1)

D’une Architecture Monolithique

Vers une Architecture micro-services

Par Pr. Mohamed YOUSSFI

**1. Rappel des composants principaux d’une Architecture Monolithique**

Une application monolithique est souvent une grande application développée en un seul Bloc. Une seule et unique application est développée pour un problème donné aussi grand qu’il soit. L’application est souvent déployée dans un serveur d’application qui peut être représenté par un Conteneur Web comme Tomcat et un Framework qui assure l’inversion de contrôle et l’injection des dépendances comme Spring IOC (Tomcat+SpringIOC). Tomcat joue le rôle de Web Container en gérant un pool de threads qui seront utilisés pour le traitement des l’ensemble de requêtes entrantes en se basant sur un composant Servlet qui constitue le contrôleur frontal de l’application. Dans le cas d’une architecture basée sur Spring MVC, ce contrôleur frontal est assuré par DispatcherServlet. Spring IOC assure l’inversion de contrôle et l’injection des dépendances dans l’application en permettant de séparer les aspects métiers des aspects techniques de l’application grâce à l’approche orientée aspect du Framework.  De ce fait, Spring IOC permet de minimiser à l’extrême le code du développeur en l’épargnant d’implémenter les aspects techniques de l’application comme la sécurité, l’accès aux données, la gestion des transactions, les caches, la journalisation etc.

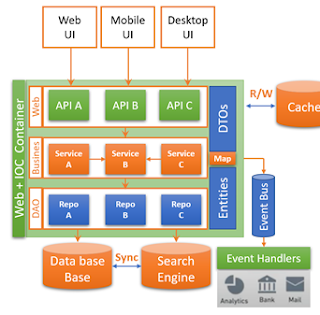
[](https://blogger.googleusercontent.com/img/a/AVvXsEhWk4xXuHUQjR7VWMn5zUfusDvaCjF0i8Aq4Bm5zfxROYSjPqhISNieYMDEzpmysZR1gsuS91y_5MeSqg_mji-ncfF0ph_G42X-rX52m_Qj3IYT3jbpFDDrVhhutgANkKaiJtrwPXInO9kHD6h6zWEKeJiFBATRMs6vud-2i8HbajiJLxE3V3S6oPt-ww=s342)

Figure 1 : Architecture Monolithique

Une application monolithique utilise souvent une seule grande base de données qui englobe toutes les données de l’application. L’application est structurée idéalement en plusieurs couches séparées dépendantes selon un couplage faible à travers des interfaces qui expose les traitements de chaque couche.

* **La couche DAO :**

La couche DAO Assure la gestion et la persistance des données. C’est une couche technique qui utilise toujours un Framework de persistance de données. Dans le cas d’une base de données relationnelle, l’API **JPA** (Java Persistence API) est souvent utilisée avec une implémentation comme **Hibernate** qui assure, le mapping objet relationnel (**ORM**). Cette couche nécessite de bien implanter le modèle de données de l’application en créant des entités persistantes JPA (**Entities**). Dans cette couche, il faut implémenter les opérations classiques de gestion des entités JPA de type CRUD. Pour faire plus simple, Spring apporte un module très précieux qui est **« Spring Data »** pour la gestion des données. Spring data est une couche d’abstraction générique qui facilite à l’extrême la persistance des données. En effet Spring Data a pris le soin d’implémenter toutes les opérations classiques « CRUD » de Gestion de entités de manière générique. Spring Data peut être utilisée aussi bien pour les SGBD relationnels et NoSQL comme MongoDb, Casandra, Elastic Search etc. Pour les bases de données relationnelles, Spring Data a défini l’implémentation générique « Spring Data JPA » qui permet d’assurer le mapping objet relationnel en utilisant l’API JPA et l’une de ses implémentations comme Hibernate. En créant de simples interfaces étendant l’interface générique JPARepository de Spring Data, le code de la persistance des données est réduit l’extrême en développant la couche DAO d’une manière déclarative (Uniquement des interfaces sans aucune classe). Ce qui permet de gagner beaucoup de temps nécessaire à la mise en place de cette couche.

* **La couche Service**

La couche service représente la couche qui implémente la logique métier d’une application. Toutes les spécifications fonctionnelles du problème sont implémentées dans cette couche. Cette couche est constituée d’interfaces et d’implémentations. Les opérations de cette couche sont souvent transactionnelles. Les transactions sont gérées facilement en utilisant « Spring Transactions » d’une manière déclarative à travers l’annotation @Transactional. Cette couche est liée aux interfaces de la couche DAO (Repositories) pour les opérations nécessitant l’accès aux bases de données de l’application. Pour éviter de faire propager les entités JPA qui sont lourdes du fait qu’elle soit surveillée par le Framework de persistance comme Hibernate, vers les couches supérieures de l’application comme la couche Web (UI), il est souvent recommandé de créer d’autres objet DTO (Data Transfer Object) pour le transfert des données issues des entités JPA vers la couche UI Web. Par exemple, si l’application gère des patients, nous aurons à créer :

o   L’entité JPA Patient qui contient tous les attributs persistants dans la base de données d’un patient avec les relations avec d’autres entités

o   Des Data Classes PatientRequestDTO qui contient uniquement les attributs d’un patient saisies et postés par la couche UI

o   Des Data classes PatientResponseDTO qui contient les attributs d’un patient qui seront retournée à la couche UI par le contrôleur de l’application

De ce fait, dans les bonnes pratiques, les opérations de la couche services reçoivent en entrée des paramètres de type RequestDTO et retourne des objets de type ResponseDTO. Ce qui impose de faire, dans les méthodes de la couche service, le mapping entre les objets des entités persistantes et les objet DTO.

Pour éviter de faire ces opérations manuellement, des Frameworks de mapping Objet Objet s’imposent. Parmi ces Framework, on peut Citer MapStruct et JMapper. Avec MapStruct, le développeur n’a qu’à déclarer des interfaces pour les mappers et MapStruct Processor se charge de générer à la volée l’implémentation de ces mappers grâce à un plugin Maven.

En plus des interfaces des Repositories pour l’accès aux bases de données et les Mappers pour le mapping Objet Objet entre les entités JPA et les DTO, la couche service a parfois besoin d’effectuer d’autres opération comme :

-          L’accès à d’autres services distants comme Les Web services SOAP ou Restful,

-          La publication des messages sur des Bus d’événement à travers des brokers comme KAFKA, RabbitMQ, ActiveMQ,

-          Démarrer des Handlers pour écouter des évènements du Bus d’évènement,

-          etc.

Pour tous ces services, Spring va encore se distinguer en offrant des solutions qui permet faciliter le développement de ces fonctionnalités d’une manière déclarative très simples. Par exemple pour interagir avec des API REST, on peut utiliser RestTemplate d’une manière programmatique ou encore OpenFeign d’une manière déclarative plus simple. Pour des opérations de Pub/Sub sur des Brokers, Spring Cloud Stream Functions qui permet de simplifier à l’extrême ce mécanisme pour laisser le code indépendant du Broker utilisé.

* **La couche Web :**

La couche Web représente la couche qui expose les fonctionnalités de l’application pour un client http. Pour le cas de Spring, la couche Web est basée sur Spring MVC qui déploie un contrôleur Frontal DispatcherServlet par lequel toutes les requêtes http passent avant d’être dispatchées vers les contrôleurs de cette couche Web.

Pour chaque requête cliente, une opération du contrôleur est invoquée par DispatcherSevlet en lui transmettant les données de la requête dans un objet de type RequestDTO conçu sur mesure à chaque type requête UI. L’opération du contrôleur fait souvent appel à la couche service pour appliquer la logique métier en lui transmettant en Input l’objet RequestDTO, après quoi le contrôleur reçoit en retour sous forme d’objet ResponseDTO. Il ne reste au contrôleur qu’à envoyer la partie UI les résultats les résultats au rendu UI, soit au format HTML ou encore au format JSON pour les modèles classiques ou au format binaire GRPC pour les modèles plus récents basés sur la technologie GRPC qui exploite les atouts du protocole Http2 apportant plus de performances au niveau de la communication ente le Browser et serveur Web. Il existe deux façons pour créer la partie UI de l’application Web : Le modèle Rendu coté serveur et le modèle rendu coté client :

·       Dans le modèle rendu coté serveur, le code HTML est généré coté serveur. Le Browser Web n’a pas besoin de faire des traitements à son niveau. Il envoie juste des requête http au contrôleur de la couche Web qui lui rend en réponse du code HTML. Dans ce cas-là, la couche Web a besoin d’utiliser un moteur de rendu HTML coté serveur comme Thymeleaf, FreeMaker, GroovyTemplates ou Mustache pour éviter le traditionnel JSP qui est très déconseillé. Pour implémenter ce modèle, la couche Web se doit de définir des contrôleurs et des vues basées sur le moteur de templates. Les données à afficher des vues sont préparées par le contrôleur dans l’objet Model de Spring MVC. Après avoir effectué le traitement, le contrôleur retourne à DispatcherServlet le nom de la vue et le modèle qui contient les données à afficher par la vue qui sera soumise au moteur de templates qui se charge de générer du code HTML qui sera envoyé dans la réponse HTTP.

·       Dans le modèle rendu coté client, le code HTML n’est pas généré par le serveur, mais plutôt par le client Web en faisant un appel à moteur de rendu HTML coté Browser. Ce qui nécessite coté frontend Web l’utilisation d’un Framework Java Script comme Angular, ReactJS ou VueJS. Dans ce cas de figure, la couche Web de l’application backend est représentée par des contrôleur sous forme de de Web API de type REST API dans le cas le plus courants, GrapheQL API ou encore GRPC API dans des cas particuliers.   Pour exposer des API RESTFUL, on peut utiliser des « RestControllers » au même titre que les contrôleurs classiques de Spring MVC. Il est également possible d’exposer des API Restful sans aucune implémentation de « RestContrôllers » en utilisant le module « Spring Data REST » qui expose facilement toutes les opérations des interfaces Repositories de la couche DAO juste en ajoutant de simples annotations (@RepositoryRestRessource, @RestResource, etc.) définies dans Spring data REST. Dans « Spring Data Rest », Spring a fourni astucieusement une implémentation générique d’un Web service RESTful (RestController générique) qui permet de gérer l’entité générique de l’interface Repository basée sur Spring Data. Ceci permet d’éviter décrire le code des API RESTFUL sous forme de « RestContrôllers ». Toutefois, Spring data REST est utiliser avec beaucoup de précautions uniquement dans le cas ou il s’agit des opérations CRUD classiques qui ne nécessitent pas des traitement métiers particuliers. Autrement, il est recommandé d’implémenter ses propres « RestControllers » d’une manière standard en permettant à chaque requête de déclencher des traitement métiers d’une manière plus professionnelle en profitant aussi des objets DTO pour mieux adapter les données aux besoins de la partie UI.

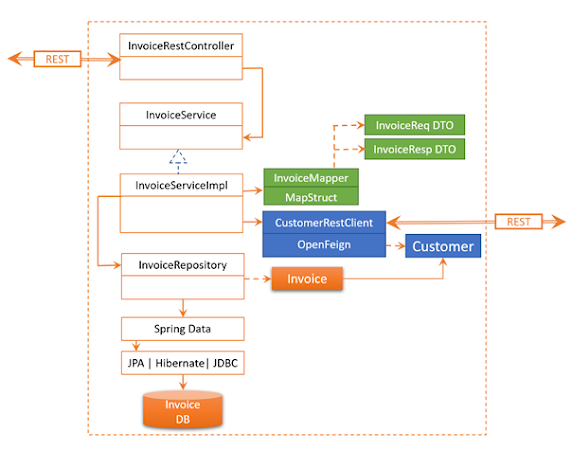
[](https://blogger.googleusercontent.com/img/a/AVvXsEhWzV-gjC6TSbTJg8Fix2vEn8fTcgoYMfOxIBHld-vbWzsTKIA9U6YOIcxDx6aqUmLFxOUw9K8Yz4TmqhG4ncBhP-qAQ1ZS-maQU-W61q6oiSEe2_AQ7HGMBD9EH4Imx0IM8pEu3gQMrr_K3Sxefk1UWInfxsFwiqiiVJL8omnraglvji9Q7LZ2kdbZnw=s604)

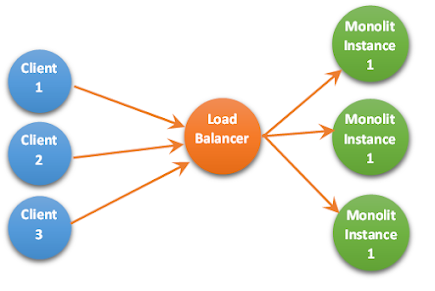
Figure 2 : Exemple d’une architecture modèle

**1.**       **2. Contraintes et problèmes des applications monolithiques**

Avec le temps, les applications monolithiques monolithiques peuvent présenter pas mal de contraintes et de problèmes à savoir :

* **Problèmes des performances :**

Une application monolithique implémente un nombre de fonctionnalités importantes. Dans une situation de montée en charge, la scalabilité horizontale s’impose. Ce qui signifie qu’il faudrait démarrer l’application en plusieurs instances avec la mise en place d’une Load balancer qui permet d’équilibrer la charge entre les différentes instances.

[](https://blogger.googleusercontent.com/img/a/AVvXsEiK0mDvSczyYZCWdPzO_dJkQT5_4hDtMMrpfASO12oB0o4ky1799ukve1JaQIeOmHqu337uiRKkZeOGOxAhsd9amLEwg0XDohHkhdAFj7ytnVr9hCpG_25CDKuzzwimvcOQl4XWpN5gScUe6g4Y_8Wc_BBasUtSx6HnLnMO1Wy6ZxT93Op9izCWGuVtww=s520)

Le problème c’est qu’une application monolithique est une grande application et le fait de la déployer en plusieurs instances signifie que toutes les fonctionnalités de l’application sont déployées en plusieurs instances (Scalabilité Horiontale). Or, En la réalité, quand il y a un problème de montée en charge, uniquement quelques fonctionnalités de l’application qui sont très sollicitées et font subir à l’application une grande charge. En réalité, seules ses fonctionnalités qui subissent la charge qui ont besoin d’être mises à l’échelle (Scalabilité) en les démarrant en plusieurs instances. Ce qui est impossible à faire pour une telle application monolithique. Ce qui fait que la scalabilité horizontale d’une application monolithique coute trop cher et les performances sont impactées car toutes les fonctionnalités sont condamnées à vivre dans le même conteneur et dans un même processus. En plus, quand on démarre l’application en plusieurs instances, il faudrait penser à partager les sessions d’authentification entre les différentes instances. Une opération qui ne se fait pas sans douleur avec l’obligation de mettre en plage d’un cache mémoire distribué en utilisant des solutions comme Hazelcast.

* **Toutes les fonctionnalités tournent dans un seul processus :**

Dans une application monolithique, toutes les fonctionnalités tournent dans un processus unique. Ce qui signifie, si une fonctionnalité bloque le processus pour donner suite à incident, c’est toutes les autres fonctionnalités qui seront affectée et restent indisponibles. Ce qui nécessite de redémarrer une nouvelle instance de toute l’application. Comme dans une grande application, la probabilité d’avoir un incident dans une fonctionnalité est logiquement proportionnelle au nombre de fonctionnalités présentes dans l’application, le facteur de disponibilité de l’application sera impacté négativement.

* **Difficiles à maintenir et à tester**

Généralement, une application monolithique est très difficile à maintenir. En effet, le fait d’avoir implémenté un grand nombre de fonctionnalité dans la même application rend difficile à identifier rapidement les composants qui doivent être maintenus. Et quand on met à jours une fonctionnalité particulière, des risques importants des effets de bords sont à considérer. Ce qui fait qu’il faudrait toujours tester les régressions. Comme le seul moyen de tester les régressions étant de rejouer tous les tests unitaires de l’application, la situation devient trop lourde et surtout que dans pas mal de cas, tous les tests unitaires ne sont pas implémentés. Ce qui fait qu’on ne saura jamais, si la modification effectuée ne va pas engendrer des effets de bords négatifs sur d’autres fonctionnalité. Les mauvaises surprises surgissent donc après le redéploiement de l’application en production et l’obligation d’ouvrir à nouveau le chantier pour retoucher le code source de l’application s’impose. Ceci qui augmente la durée de mise en production de chaque nouvelle version. Ceci peut vraiment devenir un cauchemar pour l’entreprise qui est liée par un contrat de maintenance avec son client. Ce qui nécessite de mobiliser beaucoup de ressources pour maintenir l’application et vite fait on a envie de découper cette application en plusieurs petites applications et l’intérêt d’une architecture micro-services commence à se ressentir.

* **Obligation d’utiliser une même technologie de développement**

Généralement, pour développer une application monolithique, l’entreprise est contrainte d’arrêter le choix sur le langage et la technologie à utiliser pour développer l’application monolithique. Le problème c’est que tous les langages et tous les Frameworks ont des points forts et des points faibles et que les développeurs de l’entreprise ont des préférences, des expériences et de l’expertise dans des technologiques différentes. Le fait d’opter pour une unique technologie signifie de former toutes les équipes du projet sur la même technologie et ignorer l’expertise qu’ils ont dans d’autres technologies. Un autre élément important c’est que quand on fait le découpage de l’application par domaine de fonctionnalités, on découvre que dans des services particuliers de l’application, il serait mieux et plus simples de les développer avec un autre langage et en utilisant d’autres Frameworks que ceux ayant été choisi pour développer toute l’application. Par exemple, si on opte pour la technologie Java/Spring, il se peut que dans une partie de l’application qu’on ait à développer des fonctionnalités de machines learning. Il serait peut-être judicieux de développer cette partie de l’application en utilisant le langage Python avec un Framewok comme TensorFlow par exemple. De temps plus que dans l’équipe on peut trouver facilement une ressource qui maitrise ce langage. Le problème c’est que dans une application monolithique, on sera obligé de développer cette fonctionnalité avec le même langage Java et qu’on sera obligé de chercher un équivalent de TensorFlow en langage Java. Heureusement que des solutions existent et que vous trouverez que DeepLearning4J ferait très bien l’affaire. Mais qu’il faudrait attendre la montée en compétences de l’équipe dans ce Framework. Ce qui risque de ralentir le processus de développement du projet et d’augmenter par conséquent augmenter le cout du projet.

* **Dépendances des équipes de développement :**

Dans un projet d’application monothéique, il est très difficile de rendre les équipes de développement intervenantes dans le projet complètement indépendant. Il est très courant que des réunions des différentes équipes s’imposent d’une manière fréquente pour synchroniser les différentes du développement à cause des éléments des couplages forts entre les différents modules et des éléments en commun aux différents modules de l’application monolithique.

* **Livraison en Bloc :**

Une application monolithique est grande application. Son développement prend beaucoup de temps. Ce qui fait que le client attend beaucoup de temps avant de voir la première version de son produit.

**3.     3. Architecture Micro-services**

L’architecture micro-services consiste à découper le périmètre fonctionnel d’un grand projet en plusieurs petites parties élémentaires. Pour chaque partie élémentaire, on développe une petite application indépendante dite micro-service.

[Une image contenant texte, capture d’écran, orange

Description générée automatiquement](https://blogger.googleusercontent.com/img/a/AVvXsEhpCQ09qEXAkYOZNX7QPdNDyptDVru61kJl37SL5a__1jF4f1_h-JT3dc5--Jw_ukC-6i1Mtj6vehKjIv_ig2dKZuyYxfi8bOi_YmMWVIAw-My_0CYkzOR8ySifUNH1VLC2vgR_QMuOHjzsMvqeOYSer9bsrOsvG5PyOErw5LiHH5nDvoYZT7ezWK9OMw=s1208)

Figure 3 : de l’architecture monolithique vers l’architecture micro-services

La figure 3 montre un exemple d’une architecture monolithique d’une application à 3 blocs fonctionnels A, B et C qui est découpée dans une architecture micro-services en 3 micro-services indépendants implémentant respectivement les blocs fonctionnels A, B et C.

* **Caractéristiques et avantages des micro-services :**
  + Chaque micro-service tourne dans un processus séparé avec son propre conteneur et ayant sa propre base de données. Ce qui fait que le blocage d’un micro-service ne va pas affecter les autres micro-services.
  + Chaque micro-services peut être développé, testé séparément des autres. Comme un micro-service est une petite application, Il est plus simple à développer et tester.
  + Chaque micro-service peut être développé avec un langage et une technologie différente. De ce fait on pourra bénéficier des atouts de tous les langages et toutes les technologies pour choisir la technologie la plus appropriée pour chaque Micro-servic
  + Les micro-services peuvent communiquer facilement d’une manière synchrone ou asynchrone. Pour le modèle de communication synchrone, les micro-services exposent toutes leurs fonctionnalités à travers des API (SOAP, REST, GRPC, GraphQL) qui permettent de les interconnecter faiblement. Les micro-services peuvent communiquer également d’une manière asynchrone à travers un bus d’événements basé sur des Brokers comme RabbitMQ, KAFKA et ActiveMQ. Ce mécanisme de communication asynchrone très important dans les architectures micro-services permet de mettre en place des mécanismes de synchronisation des micro-services.
  + Chaque micro-service peut être développé par une équipe complètement indépendante. Du fait que les micro-service peuvent communiquer via des mécanismes de couplage faible basés sur des API, les micro-services peuvent être développées en parallèle par des équipes indépendantes. Ceci permet d’augmenter la productivité dans le processus de développement du projet
  + Les micro-services sont des petites applications qui sont produites dans des cycles de développement très courts. Par conséquent, il est plus simple d’appliquer les méthodes agiles comme SCRUM dans le processus de développement des micro-services. Il est aussi plus simple d’appliquer le processus de développement Test Driven Développement (Développement Piloté par les Test). Ce qui permet de garantir la qualité logicielle
  + Les micro-services sont des cycles de développement couts. Ce qui permet de garantir une livraison en continue. On pourra livrer au client les fonctionnalités de l’application (Micro-services) au fur et à mesure de leurs développements.
* **Intégration des micro-services :**

Pour intégrer les micro-services dans le cadre d’une même application, il est nécessaire de mettre en place des services techniques fournis par le Framework permettant de développer les architectures micro-services comme Spring Cloud. Parmi ces services techniques, 3 s’imposent :

* ***Discovery Service :***

Représente un annuaire qui permet d’enregistrer la localisation de toutes les instances des micro-services de l’application. Au démarrage, chaque micro-service se connecte à ce micro-service pour enregistrer le nom du micro-service et l’URI du micro-service incluant l’adresse IP de la machine et le numéro de port. Un exemple de Discovery service fourni par Spring Cloud est Eureka Discovery, une implémentation de NetFlix.

* **Gateway Service :**

Pour interagir avec l’application, toutes les requêtes des applications frontend web, mobile et les applications externes doivent être envoyées au service Gateway. Ce dernier se charge d’acheminer cette requête vers le bon micro-service implémentant la fonctionnalité de la requête. Pour y arriver, Pour chaque requête, le service Gateway extrait le nom de du micro-service de la requête entrante et interroge le service Discovery pour récupérer l’URI du micros-service (IP, Port) sachant son nom. Ensuite, il dispatche la requête vers l’URI du micro-service. Pour mettre en place une Gateway, avec Spring Cloud, il existe deux modèles : un Modèle classique Multi Threads avec des entrées sorties bloquantes implémenté par ZUUL Proxy ; une implémentation de NetFlix et un autre modèle réactif plus performant, Single Thread avec des entrée sorties non bloquantes proposé par Spring Cloud à savoir Spring Cloud Gateway.

* **Config service :**

Ce service technique offre un mécanisme qui permet de centraliser la configuration de l’ensemble des micro-services dans un unique repository. En effet, ce service gère un fichier de configuration globale qui regroupe toutes les propriétés de configuration communes à tous les micro-services et des fichiers de configuration relatifs à chaque micro-service dans lequel chaque micro-service trouve ses particularités de configuration. Au démarrage, chaque micro-service commence par contacter ce service pour récupérer sa configuration avant de continuer le démarrage. Quand une mise à jour est apportée à la configuration, les micro-services reçoivent la nouvelle configuration à chaud. Ce qui permet de reconfigurer les micro-services à chaud en évitant par conséquent le redémarrage des services.

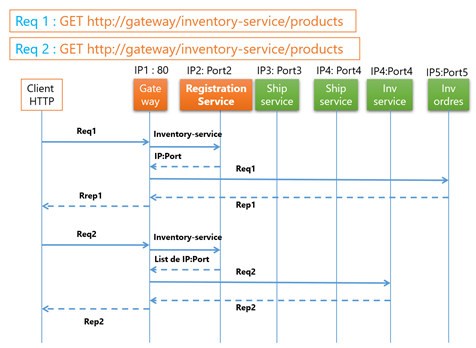
[](https://lh3.googleusercontent.com/-uFjhqh9ZQqw/YZPyRdKNSiI/AAAAAAAAYRM/0gYwezW47EI_CHfh_yYWjBdupzy4HdfoQCLcBGAsYHQ/image.png)

Figure 4 : Module d’interaction dans une architecture micro-services