Rapport final Uberoo
Service Oriented Architecture

CANAVA Thomas, GARDAIRE Loïc, JUNAC Jérémy, MELKONIAN François

Sommaire

Justification des technologies utilisées

Node Js

Pour ce projet, nous avons utilisé *Node Js*. Nous avons fait le choix de cette technologie pour la facilité avec laquelle nous pouvions créer des micro services. En effet, quelques lignes de code suffisent pour créer un micro service minimum. En suivant le principe KISS (*Keep It Simple, Stupid*), nous voulions avoir cette facilité d'implémentation pour pouvoir se focaliser sur l'architecture de notre système et les interactions entre nos services.

De plus, nous voulions une technologie rapidement déployable pour profiter au maximum des avantages de scalabilité apportés par une architecture micro service. *Node Js* permet aussi de faire des changements sans phase de compilation excessive.

Base de données

Pour persister les données nécessaires au fonctionnement de nos services, nous nous sommes confrontés au choix d'une base ACID ou BASIC. Nos services, petits par construction, ont chacun un modèle objet qui se réduit à une hiérarchisation des données pour ne garder que les données nécessaires à son fonctionnement. D'autre part, après la définition de nos contextes bornés, nous nous sommes rendus compte que dans nos processus critique, aucun service ne faisait des requêtes nécessitant de parcourir une large quantité de données, nous n'utilisons pas l'avantage de la cohérence globale des base de données. De plus, de part la nature de notre architecture, nous allons avoir de la duplication de donnée entre service pour garder l'unicité et éviter le couplage entre ces services, et les bases BASIC présentent l'avantage d'être très performantes lorsqu'il y a beaucoup de données. De plus, nous avions vu en cours la complexité amenée par la gestion des transactions dans des systèmes distribués. Nous avons donc, dès le départ, modéliser notre système pour éviter au maximum ces transactions.

Nous nous sommes donc tournés vers une base BASIC. Lors de nos recherches comparatives sur la recherche de base de données NoSQL, nous avons trouvé que la plus adaptée était *Mongo*. D'une part, nous ne pouvions pas faire de supposition sur le type de base de données, d'autre part, *Mongo* nous semblait suffisamment documenté pour le bon déroulement du projet. Cette supposition s'est vérifiée au cours du développement.

Paradigmes de style d'architecture exposée aux clients

Le sujet est centré sur une gestion de restaurant et de menus. Il nous est donc nécessaire de gérer différentes ressources : les différents menus des restaurants, les commandes des clients, la liste des clients, etc. Le choix du style de service *Ressource* semble donc approprié pour résoudre notre problème. Il nous permet à la fois d'exposer des listes de ressources accessibles facilement via une adresse et à la fois de récupérer parmi elles un élément précis grâce à un identifiant. D'autre part, nous ne faisons que des opérations basiques sur ces ressources (*e.g.* création, mise à jour), ce qui est adapté à ce paradigme.

De plus, ce type de service laisse la possibilité au restaurant de se connecter à notre système et de pouvoir créer une interface de façon simplifiée. Avec une API très générale (utilisation de GET et POST), le restaurant peut facilement se connecter à notre back-end avec un minimum de suppositions sur celui-ci. Le format *Ressource* est donc le moins contraignant à la fois pour nous et pour l'utilisateur tout en remplissant les demandes de ce dernier.

Cependant, nous sommes conscients que nous n'avons pas mis l'énergie nécessaire pour appliquer de façon correcte le nommage des routes. Ainsi, lors de l'ajout de nouvelles fonctionnalités, nous les avons souvent modélisés comme étant des ressources à part entière, sans appliquer une topologie propre à REST. Prenons l'exemple des avis sur un plat : au lieu de créer une route tel que */restaurants/:restaurantId/meals/:mealId/feedbacks*, nous avons défini une route */feedbacks/:restaurantId*.

Cette erreur n'est pas un gros point noir de notre système et nous avons jugés que ce n'était pas prioritaire d'avoir des routes respectant scrupuleusement les principes de REST.

Rappel de notre architecture de départ

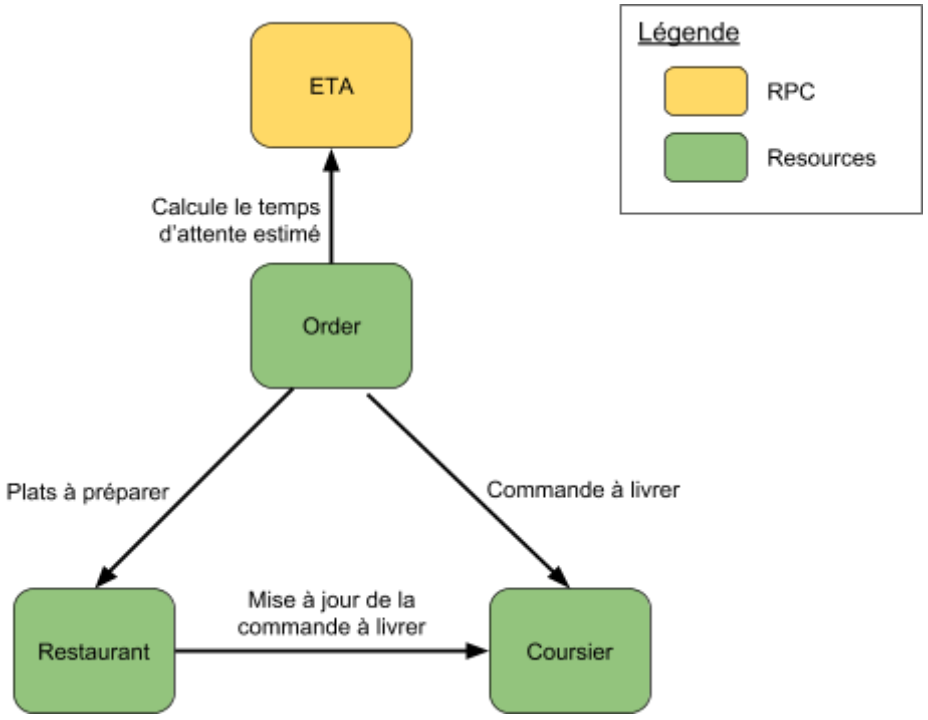


Figure n°1: Représentation initiale de nos services

Nous avons initialement fait le choix de découper notre application en 4 services.

Le service nommé *Order* était un point d'entrée de notre système pour le client. Il possédait toutes les fonctionnalités dont ce dernier a besoin pour recevoir son repas qui vont de lister les repas pouvant être commandés à passer sa commande à un restaurant. Ainsi, il contenait la liste des plats.

Le service nommé *Restaurant* s'occupait de gérer l'interface utilisée par les différents restaurants partenaires de notre application. C'est ici que ces derniers pouvaient consulter les commandes à préparer par exemple.

Le service nommé *Coursier*, quant à lui, centralisait l'état des livraisons. Il permettait aux différents acteurs de savoir dans quel état était une commande à tout moment. Par exemple, il permettait au livreur de savoir lorsque le repas a fini d'être préparé par le restaurant et est donc prêt à être livré au client.

Et enfin le service estimant le temps d'attente était, comme son nom l'indique, celui nous permettant de calculer l'ETA de chacune des commandes.

Évolution lors des changements de chaque semaine

Première version

Le passage de nos services qui n'étaient pas connectés à notre architecture utilisant *Kafka* nous a fait revoir notre architecture, notamment pour prendre en compte les feedbacks du premier rendu.

D'une part, avant l'interconnexion, le service *Order* avait la responsabilité de connaître le menu et de stocker les différents comptes (les comptes des coursiers et des clients). Nous avons alors séparé la liste des plats dans un service *Catalog*.

D'autre part, la première version du produit demandé par le client comportait de nouvelles fonctionnalités par rapport au MVP rendu initialement. Ces dernières nous ont demandé d'ajouter 2 nouveaux services. Le premier est un service de gestion du paiement permettant à Erin de payer directement avec une carte de crédit sur le site. Et le second est un service de compte de livreur permettant à Jamie le livreur d'être crédité lors de la fin d'une course.

Une autre problématique est survenue avec *Kafka*. Notre architecture était totalement synchrone lors de la précédente version. Or, *Kafka* est basé sur des évènements. Nous avons alors ajouté une passerelle entre les différents clients et nos services. Cette passerelle reçoit les requêtes des clients, valide la forme des messages puis envoie les messages correspondants dans notre queue Kafka. Il va alors mettre en attente le client jusqu'à recevoir la réponse de notre système. Cette séparation nous a permis de simplifier la stack technologique de nos services, et de faciliter la mise à l'échelle à granularité faible.

Ajout de la semaine 43

Le second ajout de fonctionnalités s'est fait en semaine 43 avec l'arrivée de Terry, le patron du restaurant voulant des statistiques sur son restaurant. De plus, le chef du restaurant souhaitait avoir du feedback sur ses plats et les clients voulaient pouvoir géolocaliser le livreur en temps réel.

Ces fonctionnalités nous ont poussé à ajouter un nouveau service nommé *Statistics* qui remplit la story de Terry. Il consomme différents messages au cours de la vie d'une commande : "*les plats sont prêts à être livrés*", "*le livreur a livré la commande*", … Il produit en sortie des statistiques par livreur pour un restaurant donné. Pour son utilisation, nous avons ajouté une route dans la *Gateway* du restaurant pour faire une demande de statistique et écouter la réponse. Un des avantages de cette solution est l'évolutivité : à l'avenir, si notre système évolue en ajoutant de nouveaux états, les statistiques ne seront pas impactées. Cependant, c'est aussi un inconvénient car l'ajout d'un nouveau type de coursier par exemple entraînerait une nécessité de normalisation des événements ou une évolution du service le rendant sensible aux nouveaux messages. Le second avantage est en terme de performance ; il est intéressant d'avoir un service indépendant responsable des statistiques car en cas de surcharge des services responsables des commandes, une demande de statistiques en surcroît ferait s'écrouler le système. Cependant, le service de statistiques possède l'intelligence de savoir lorsqu'une commande est terminée, intelligence déjà présente dans le service responsable des commandes. Nous avons donc une duplication de cette dernière.

Pour les feedbacks, le service nommé *Catalog* créé la semaine précédente, reçoit alors de nouvelles fonctionnalités. À présent, il remplit la story du chef de restaurant et permet d'organiser les plats suivant les restaurants et de stocker leur feedbacks respectifs. Il nous semblait logique d'ajouter les feedbacks dans le catalogue contenant déjà les repas de tous les restaurants. Nous avons ensuite ajouté deux routes : une sur la *Gateway* du restaurateur permettant de lister les feedbacks et une sur la *Gateway* du client permettant d'ajouter un feedback sur un plat.

Enfin, nous avons aussi ajouté le système de localisation en temps réel en permettant au coursier d'envoyer sa position à intervalle de temps fixe. Cette localisation est stockée en base de données et le client envoie une requête pour récupérer la dernière localisation stockée. Cette responsabilité a été ajoutée au service responsable du coursier, qui avait déjà l'information de l'attribution des commandes aux livreurs en fonction de leur position. L'ajout de cette fonctionnalité a donc eu un impact faible sur notre système. L'inconvénient majeur de notre solution, basée sur l'envoi à intervalle régulier de la position des coursiers, est le fait qu'elle peut rapidement amener une charge importante pour notre système. Nous avons donc souhaité limiter le nombre de microservices nécessaires pour cette fonctionnalité afin d'éviter une surcharge globale du système.

Ajout de la semaine 44

Les nouvelles demandes du client sont l'ajout de codes promotionnels pour les restaurants de façon à attirer plus de clients et l'interruption inopinée d'une livraison (accident) entraînant un remplacement de la commande.

La première a impacté le service responsable des restaurants. avec l'ajout d'un message permettant d'ajouter un nouveau code promotionnel. De plus, nous avons fait le choix d'ajouter un nouveau service nommé *Pricer* responsable d'appliquer les réductions sur les plats concernés. L'avantage ici d'avoir un service dédié au calcul du prix est l'évolutivité. Pour l'instant, nous avons fait le choix d'avoir des codes promotionnels simples (10% sur tout le restaurant). Mais ces codes ont vocation à devenir de plus en plus complexes. Avoir un service dédié à leur application permet donc d'éviter une surcharge des autres services.

La seconde n'a eu que très peu d'impact sur notre système. Lorsqu'un coursier signale un accident, un message est envoyé au restaurant demandant de cuisiner de nouveau la commande. Les statistiques ne sont pas impactées car elles prennent en compte le livreur ayant finalisé la commande. Les plats nouvellement cuisinés par le restaurant seront donc livrés par un nouveau coursier qui sera enregistré comme le livreur de cette commande pour les statistiques.

Ajout de la semaine 45

Cette dernière semaine nous a conforté dans l'idée que nous avons fait le bon choix pour les codes promotionnels avec l'ajout d'une story de code plus complexe que la semaine précédente. Notre système était donc prêt pour cette évolution, et n'a demandé qu'une modification de la structure d'une réduction

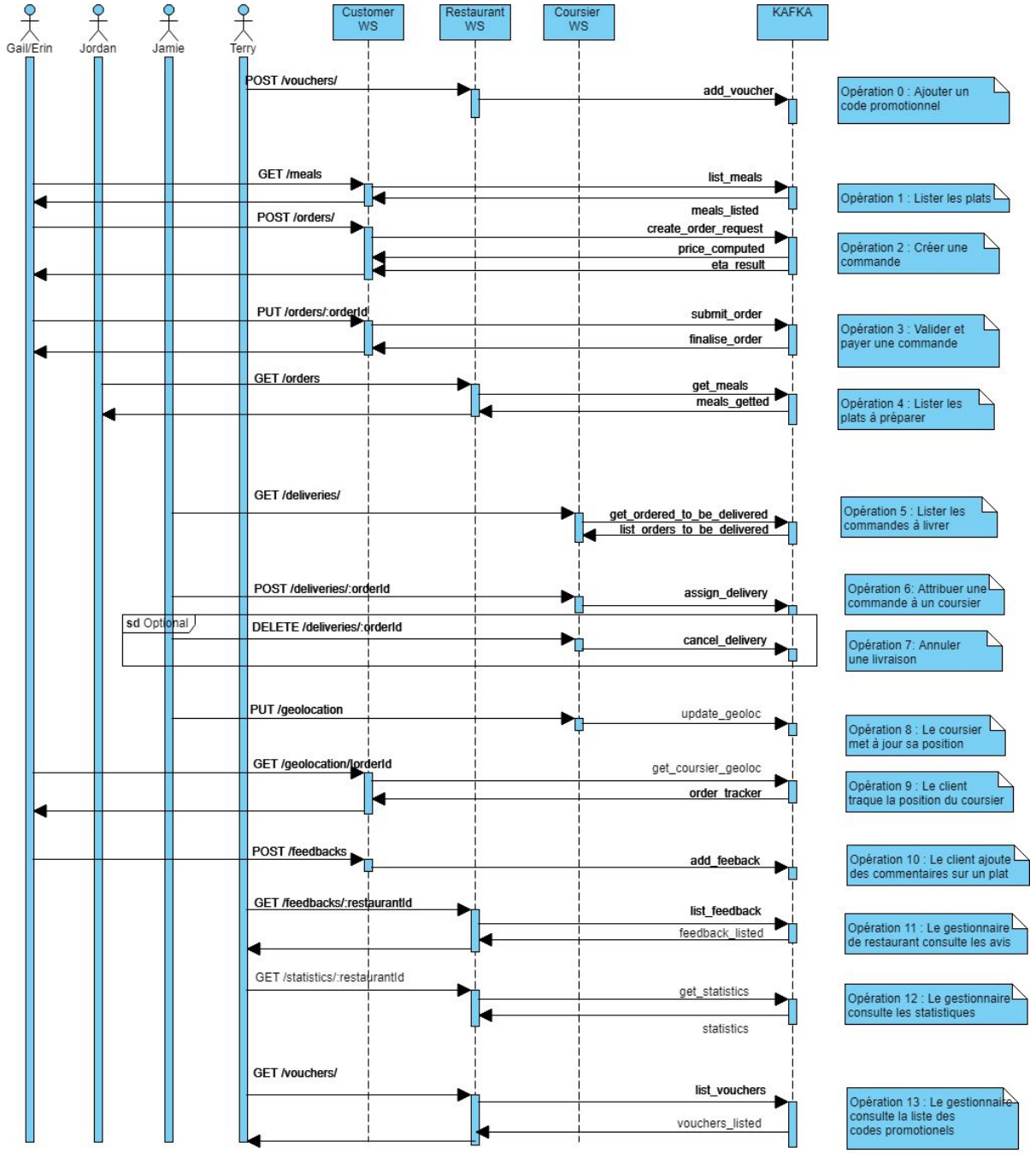
Le second ajout concernait la géolocalisation du coursier. Cette dernière pouvait maintenant permettre de mettre à jour l'ETA de la commande en temps réel. Le système de localisation en temps réel étant déjà en place, il a suffi d'ajouter la position du client dans la requête, permettant ainsi de calculer la distance entre le coursier et le client, qui devient l'ETA de la commande.

Architecture de notre application

Interactions des acteurs avec notre système

Le diagramme ci-dessous présente synthétiquement les interactions entre les acteurs et notre système, en passant par les 3 *Gateway*. Les personae utilisés ici sont identiques à ceux présents dans les demandes du client, c'est-à-dire:

- Gail et Erin sont des consommateurs souhaitant se faire livrer une commande.
- Jordan est un chef cuisinier de restaurant.
- Jamie est un coursier travaillant pour Uberoo.
- Terry est le responsable d'un restaurant qui collabore avec Uberoo.



<u>Figure n°2:</u> Interactions entre les Gateway et notre système

Interactions entre microservices pour chaque opération

Opération 0 : Ajouter un code promotionnel

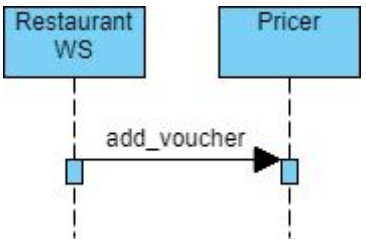


Figure n°3: Message échangé pour ajouter un code promotionnel

Événement	Payload

add_voucher	{
	"restaurantId": "12",
	"expirationDate": "2020-01-01",
	"discount": 0.12,
	"code": "AZERTY50",
	"neededCategories": ["Burger","Dessert"]
	}

Opération 1 : Lister les plats

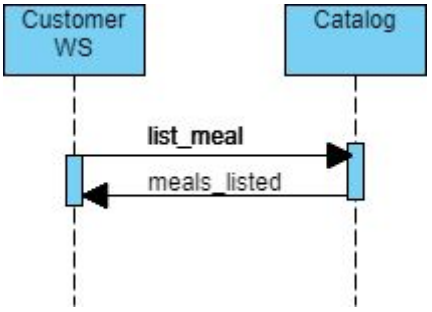


Figure n°4: Messages échangés pour lister les plats

Événement	Payload

list_meal	{
	"categories": ["asiatique","kebab"],
	"restaurants": ["Mac Donalds","Chez Cathy", "Jeremie cookies"]
	}

meals_listed	{
	meals: [
	{
	id: 34,
	name: "Mac first",
	category: "burger",
	type:"main_course",
	eta: 4,
	price: 1.0,
	restaurant: {
	id: 12,
	name: "Mac Donalds",
	address: "4 Privet Drive"
	}
	}
	]
	}

Opération 2 : Créer une commande

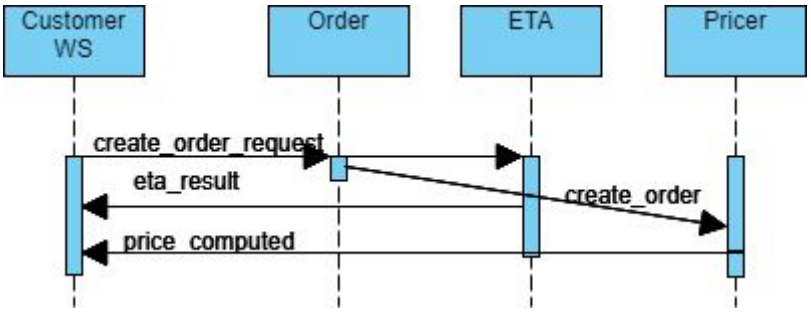


Figure n°5: Messages échangés pour créer une commande

Avant la validation d'une commande, le client souhaite connaître le temps estimé, ainsi que le prix qu'il doit payer.

Événement	**Payload**
create_order_request	{ timestamp: 182038333, meals: [

	<pre>{
	id: 34,
	name: "Mac first",
	category: "burger",
	type:"main_course",
	price: 11.0,
	eta: 4,
	restaurant: {
	id: 12,
	name: "Mac Donalds",
	address: "4 Privet Drive"
	}
	}
	],
	customer: {
	name: "Mario",
	address : "3 Privet Drive",
	},
	voucher: "MCDOPROMO"
	}</pre>

create_order	<pre>{
	requestId: 45433874839,
	orderId: "uuid",
	events: [
	{
	event: "creation",
	time: 36478836478
	}
	]
	}</pre>

eta_result	<pre>{
	orderId: "uuid",
	eta: 34
	}</pre>

price_computed	<pre>{</pre>

	orderId: "uuid", price: 21 }

Opération 3 : Valider et payer une commande

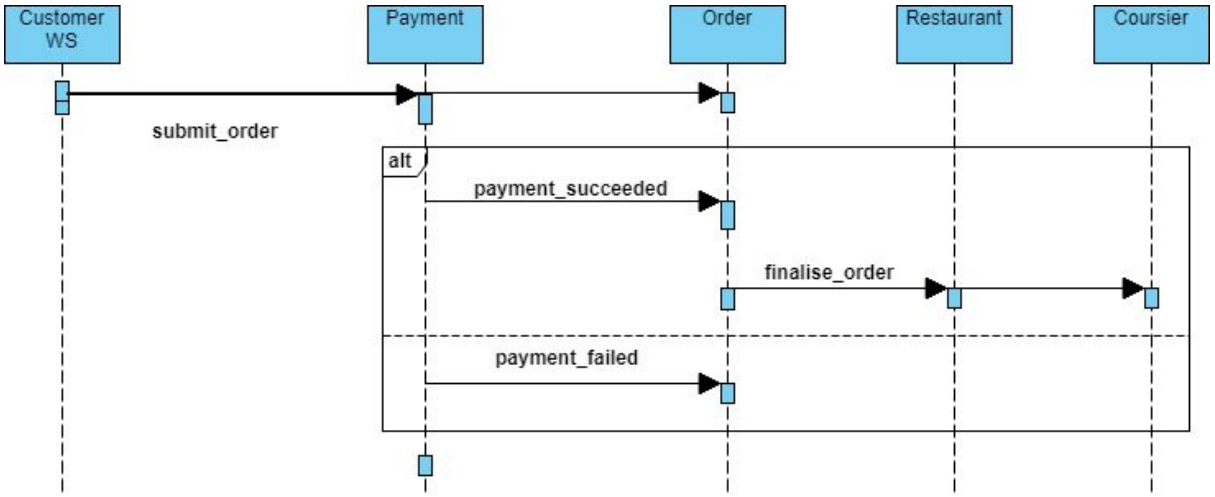


Figure n°6: Messages échangés pour valider et payer une commande

Le client envoie sa commande. Si le paiement est un succès, la commande est finalisée, c'est-à-dire qu'elle peut être envoyée aux restaurants impliqués. Si le paiement est un échec, un message d'erreur apparaît.

Événement	**Payload**

submit_order	{
	timestamp:1234,
	order: {
	id: "uuid",
	meals: [
	{
	id: 34,
	name: "Mac fist",
	category: "burger",
	type:"main_course",
	eta: 4,
	price: 1.0,

	restaurant: {
	id: 12,
	name: "mac do",
	address: "4 Privet Drive"
	}
	}
	],
	customer: {
	name: "Mario",
	address : "3 Privet Drive"
	}
	},
	creditCard: {
	name: "Mario",
	number: 5131637836475957374,
	ccv: 753,
	limit: "07/19"
	}
	}

payment_succeeded	{
	order{
	id: "49678"
	}
	}

payment_failed	{
	orderId:"49678"
	}

finalise_order	{
	timestamp: 1718293,
	order: {
	id: "uuid",
	meals: [
	{
	id: 34,
	name: "Mac first",

	category: "burger",
	type:"main_course",
	eta: 4,
	price: 1.0,
	restaurant: {
	id: 12,
	name: "Mac Donalds",
	address: "4 Privet Drive"
	}
	}
	],
	customer: {
	name: "Mario",
	address : "3 Privet Drive"
	}
	}
	}

Opération 4 : Lister les plats à préparer

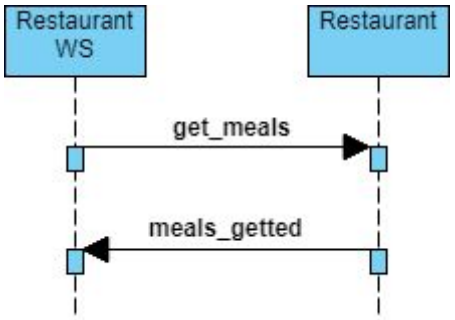


Figure n°7: Messages échangés pour lister les plats à préparer

Événement	**Payload**

get_meals	{
	restaurantId : 12,
	status: "todo"
	}

meals_getted	{

	<pre>{
	  orders: [{
	  id: "number",
	  status: "todo",
	  meals: [
	  {
	  id: 19
	  name: "Mac first"
	  }
	  ]
	  }]
	}</pre>

Opération 5 : Lister les commandes à livrer

Figure n°8: Messages échangés pour lister les commandes à livrer

Événement	Payload

get_ordered_to_be_delivered	<pre>{
	  coursier: {
	  id: "uuid",
	  address:"31 rue des platanes"
	  }
	}</pre>

list_orders_to_be_delivered	<pre>{
	  requestId: 24435323623,
	  orders: [
	  {
	  id: 12,
	  restaurant: {</pre>

	<pre>address:"25 rue des platanes"
	},
	customer: {
	address:"12 avenue des marais"
	}
	
	}
	]
	}</pre>

Opération 6 : Attribuer une commande à un coursier

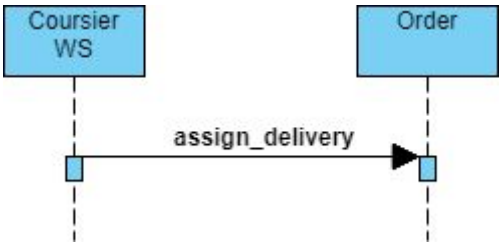


Figure n°9: Message échangé pour attribuer une commande à un coursier

Événement	Payload

assign_delivery	<pre>{
	timestamp: 1718293,
	orderId : "uuid",
	coursierId: "uuid"
	}</pre>

Opération 7 : Annuler une livraison

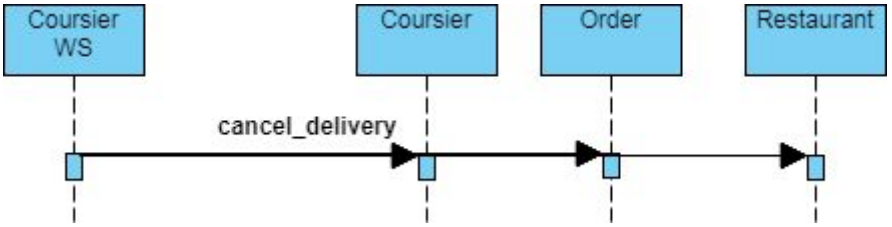


Figure n°10: Message échangé pour annuler une livraison

Événement	Payload

cancel_delivery	{
	coursierId: "uuid",
	orderId: "uuid"
	}

Opération 8 : Le coursier met à jour sa position

Figure n°11: Message échangé pour mettre à jour la position d'un coursier

Événement	Payload

update_geoloc	{
	timestamp: 1718293,
	coursierId: "uuid",
	orderId: "uuid",
	geoloc: {
	long:"43",
	lat:"7"
	}
	}

Opération 9 : Le client traque la position du coursier

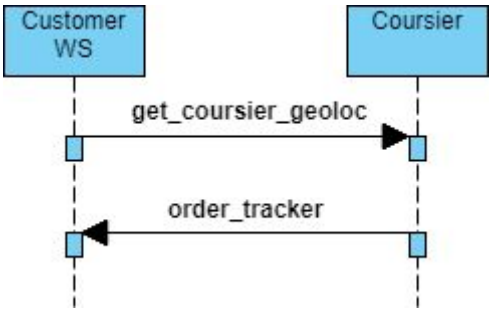


Figure n°12: Messages échangés pour traquer la position d'un coursier

Événement	**Payload**

get_coursier_geoloc	{
	orderId: "uuid",
	geoloc: {
	lat: 12,
	long: 32
	}
	}

order_tracker	{
	orderId: "uuid",
	coursier: {
	id: "uuid",
	geoloc: {
	lat: 12,
	lng: 32
	}
	},
	eta: 37
	}

Opération 10 : Le client ajoute des commentaires sur un plat

Figure n°13: Message échangé pour ajouter un commentaire sur un plat

Événement	Payload

add_feedback	{
	mealId: "uuid",
	rating: 4,
	customerId: "uuid",
	desc: "Plat de très bonne qualité"
	}

Opération 11 : Le gestionnaire de restaurant consulte les avis

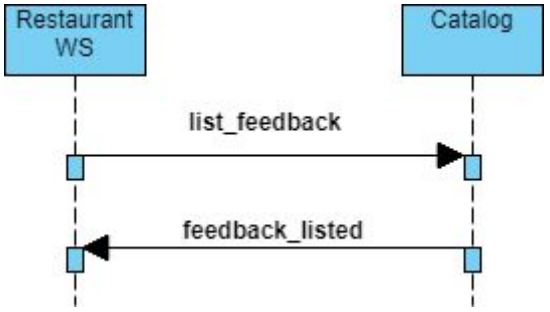


Figure n°14: Messages échangés pour consulter les avis d'un restaurant

Événement	Payload

list_feedback	{
	restaurantId : "uuid"
	}

feedback_listed	{
	meals: [
	{
	id: "uuid",
	name: "Mac first",
	category: "burger",
	type: "main_course",
	feedback: [
	{
	rating: 4,
	customerId: "uuid",
	desc: "Très bon repas"
	}
	]
	}
	]
	}

Opération 12 : Le gestionnaire consulte les statistiques

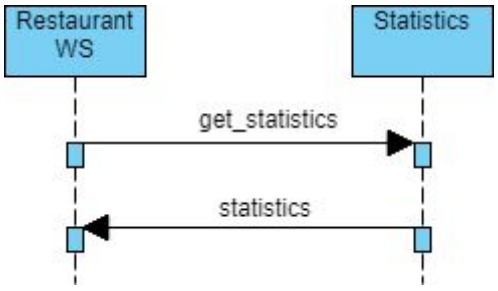


Figure n°15: Messages échangés pour consulter les statistiques d'un restaurant

Événement	Payload

get_statistics	{
	restaurantId: "uuid"
	}

statistics	{ coursiers: [{

	<pre>{
	  id: "uuid",
	  orders : [
	  {
	  time: 121,
	  date: "ISO formatted date",
	  meals: [{
	  id: "uuid"
	  }]
	  }
	  ]
	  }, ...
	  ]
	}</pre>

Opération 13 : Le gestionnaire consulte la liste des codes promotionels

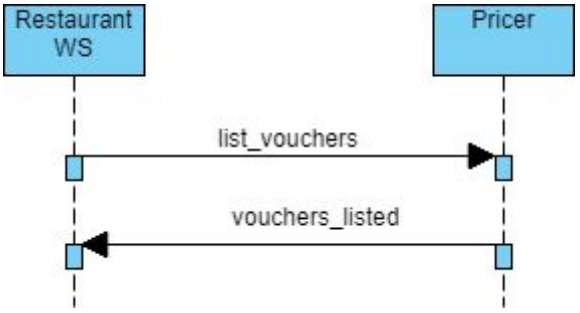


Figure n°16: Messages échangés pour consulter la liste des codes promotionnels

Événement	Payload

list_vouchers	<pre>{
	  restaurantId: "uuid",
	  requestId: "uuid"
	}</pre>

vouchers_listed	<pre>{
	  resturantId: "42",
	  vouchers: [
	  {
	  restaurantId: "42",
	  code: "AZERTYUIOP",</pre>

	<pre>discount: 0.2, // -20%
	expirationDate: 07/11/1996 ISO
	},{
	restaurantId: "42",
	code: "QSDFGHJ",
	discount: 0.6, // -60%
	expirationDate: 07/11/2000 ISO,
	neededCategories:["burger","dessert"]
	}
	]
	}</pre>

Nota Bene : Chaque message partant d'une *Gateway* contient une *requestId* non affichée dans les *payload* ici, permettant de lier le message de réponse.

Gestion d'erreur

La seule erreur métier traitée est le fait qu'un client ne peut pas se faire livrer si son paiement a échoué.

Nous sommes conscients qu'il aurait fallu passer plus de temps à traiter la gestion des erreurs et des messages malformés dans notre système. C'est clairement un point noir de notre projet.

Voici la liste des erreurs que nous aurions traitées avec plus de temps :

- Une commande peut ne jamais être assignée et/ou délivrée.
- Les restaurants sont ouvert 24 heures sur 24 : les clients peuvent commander même si le restaurant est fermé
- Globalement, les requêtes des clients qui n'attendent pas de réponse peuvent ne pas fonctionner et nous n'en informons pas le client.
- Nous supposons que le coursier ne fait pas d'erreur: nous n'avons aucun moyen d'annuler une action émise.
- Si le client ferme la connexion avant la réponse de notre système, la passerelle peut se fermer inopinément.
- Dans le cas où un des microservices n'est pas disponible, notre système ne marchera plus dans les fonctionnalités utilisant le microservice. L'erreur ici est que le client n'est jamais renseigné et les messages seront stockés dans *Kafka* jusqu'à ce que les services soient disponibles.

- Notre système n'est pas idempotent : certaines requêtes envoyées en double peuvent déclencher des situations non voulues. Avec notre système d'identifiant unique au système, cela rend une grande partie de notre système idempotent. Cependant, nous avons 3 évènements non idempotents :
    - l'ajout de commentaire sur un repas
    - la création d'une commande crée un doublon dans notre système
    - le cas où le coursier notifie 2 fois de la fin de livraison d'une commande entraîne un double comptage dans les statistiques

Justification de nos microservices

Lors de la première version de notre système, nous avions un service *Order* qui avait trop de responsabilités. Il était un point central du système. Nous avons donc dû rééquilibrer les responsabilités. Au fur et à mesure de l'arrivée de nouvelles fonctionnalités, cette différence entre les services s'est réduite, grâce à la diversité des story. Nous avons su créer les services nécessaires pour que chacun ait une responsabilité propre (la gestion des statistiques, la mise en place des codes promotionnels, etc).

De plus, nous avons mis en place de l'*event-sourcing* au niveau des commandes. Nous trouvons que c'est la meilleure solution pour avoir un historique des changements de statut des commandes et permettre l'extensibilité sur les décisions à prendre en fonction de l'état d'une commande. Le fait que le code métier soit regroupé dans ce service permet aussi de modifier facilement les décisions prises par notre système sur les commandes comme par exemple, si le client nécessite un nouveau mode de paiement ou du paiement différé, ou avec l'ajout de nouvelles étapes dans le cycle de vie d'une commande.

Du côté des statistiques, il serait simple d'ajouter un nouveau type de statistiques grâce à l'implémentation du service.

De même, le service gérant les comptes des coursiers a peu de fonctionnalités actuellement. Il peut donc facilement accueillir de nouveaux outils, comme du monitoring, ou de la gestion de compte.

Le coursier enregistre uniquement les données qui sont nécessaire pour le livreur : en effet, ce service enregistre uniquement les informations sur l'emplacement du restaurant et du client, ainsi qu'un moyen pour identifier une commande.

Nous avons pu voir au cours du développement que lorsque ce composant n'était pas en ligne, le reste des fonctionnalitées de notre système continuaient de fonctionner. C'est aussi le seul composant en charge de la localisation de nos coursier.

Avec l'ajout de fonctionnalités du client, nous avons dû implémenter un système permettant à un coursier de trouver les commandes à livrer qui sont proches de lui. Notre implémentation est naïve pour l'instant, mais est prêt à se complexifier pour permettre au coursier de contrôler au mieux son efficacité de livraison.

Au début du projet le calcul du prix d'une commande était réalisé dans le service de paiement, mais avec les nouvelles demandes du client, notamment les codes promotionnels nous avons réalisé que le calcul du prix était une responsabilité à part entière, qui ne devait pas dépendre de l'état du paiement.

Le paiement a la responsabilité d'effectuer un transfert d'argent et de vérifier que tout c'est bien passé. Il ne calcule pas lui même le prix, il le reçoit d'un autre composant. Cela peut permettre de faire évoluer les types de paiement, indépendamment du calcul du prix. Par exemple si on décide d'ajouter la possibilité de payer à la réception de la commande.

Le service calculant l'ETA est complètement indépendant et peut aussi évoluer selon les demandes du client. Par exemple, il pourrait prendre en compte les heures de pointe ou de trafic élevé, ainsi que la surcharge éventuelle du travail des coursiers.

Enfin, nous avons construit un système de façon à avoir le minimum d'impact en cas d'ajout d'un nouveau type de coursier (drone par exemple), ou même l'ajout de restaurant et de plats au sein de ces restaurants.

Cependant, un des points modifiant le plus notre système serait de laisser la possibilité au client de faire une commande sur plusieurs restaurants. Dans ce cas, cela entraînerait de profondes modifications de nos services *Pricer, Coursier*, *Order* et généralement tout notre flow d'application. Une des solutions pourrait consister à modéliser ce type de commande comme étant plusieurs commandes en faisant une commande par restaurant. Cette solution permettrait de limiter l'inpact des modifications suite à cette fonctionnalité, mais entraînerait alors des modifications dans notre service *Statistics* et *Payment.*

Notre gestion de projet

Chaque semaine, lors de l'arrivée des nouvelles demandes du client, nous prenions la séance de TD pour discuter de l'implémentation de ces dernières : l'impact qu'elles allaient avoir sur notre code, les services à modifier, les services à ajouter si nécessaire, … Cela permettait d'avoir l'avis de tous les membres du groupe et de partager les idées plus simplement. Nous avons gardé une trace de toutes ces réflexions que nous avons eu au cours de ces séances, permettant de suivre l'évolution du projet semaine après semaine.

De plus, nous avons pris soin de détailler les messages envoyés par chaque service dans un document tenu à jour chaque semaine avec l'arrivée des nouvelles fonctionnalités. Cela a grandement accéléré le travail de développement car ce document contenait les contenus exacts de chacun des messages. Il était donc facile de les retranscrire en code et de savoir le contenu d'un message lors de sa réception depuis *Kafka*.

Une fois arrivés sur une solution qui satisfaisait tout le groupe, nous nous répartissions le travail pour la semaine à venir de façon à implémenter ces nouvelles fonctionnalités. La répartition se faisait majoritairement selon qui avait déjà travaillé sur quels services. Une personne ayant déjà codé dans le service pouvait plus facilement ajouter des fonctionnalités car il avait déjà connaissance du code.

Nous faisions aussi un point en milieu de semaine pour voir l'évolution du travail de chacun des membres. Ce type de réunion permettait de savoir les points bloquants et de rééquilibrer les tâches en cas de besoin.

Nos tests de charge

Nous avons fait des tests de charge sur plusieurs parties de notre application.

En effet, nous avons identifié plusieurs scénarii indépendants dans leurs déroulement, mais qui seront effectués en parallèle par les différents acteurs du système.

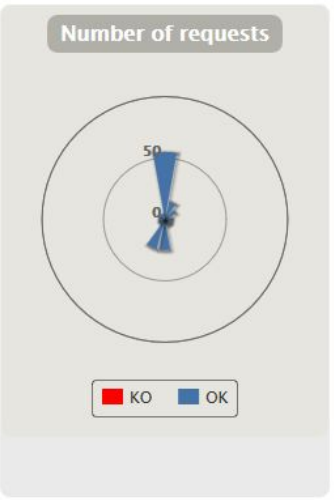
Nous avons tout d'abord des utilisateurs qui regardent la liste des plats et les avis, sans spécialement commander. Ensuite, nous avons les coursiers, qui listent les commandes autour d'eux, afin de pouvoir prendre des courses. Puis nous avons les cuisiniers, qui listent en permanence les plats qu'ils ont à faire. Enfin, nous avons les utilisateurs qui commandent, avec tout ce que cela implique, c'est-à-dire le paiement, la préparation des plats, le traçage du coursier, ... Dans le système, ces 4 scénarii sont donc indépendants, et nous devons donc supporter la charge de tous.

Nous avons choisi Polytech comme contexte. En effet, dans un premier temps, nous pourrions fournir un accès anticipé pour les environs 1000 étudiants de Polytech Nice-Sophia. En envisageant le pire des cas, c'est-à-dire où tous les étudiants veulent commander à manger sur une période d'une heure (16.6 étudiants à la minute en moyenne), on peut raisonnablement penser que nous n'aurons pas plus de 50 étudiants à la minute dans un pic de charge. Cependant, tous ces étudiants ne vont pas commander à manger (en tout cas pas dans la même minute). Nous sommes donc partis sur 10 commandes d'étudiants par minute. Toujours dans le contexte de Polytech, il nous semblait raisonnable de commencer avec 5 restaurants et 5 coursiers.

Pour résumer, voici les scénarii que nous avons utilisés pour nos tests de charge:

- Consultations de la liste des plats / avis: 50 utilisateurs / seconde
- Commande complète (de la liste des plats à la livraison): 10 utilisateurs / seconde
- Consultations de la liste des plats à faire par les restaurants: 5 utilisateurs / seconde
- Consultations de la liste des commandes proches par les coursiers: 5 utilisateurs / seconde

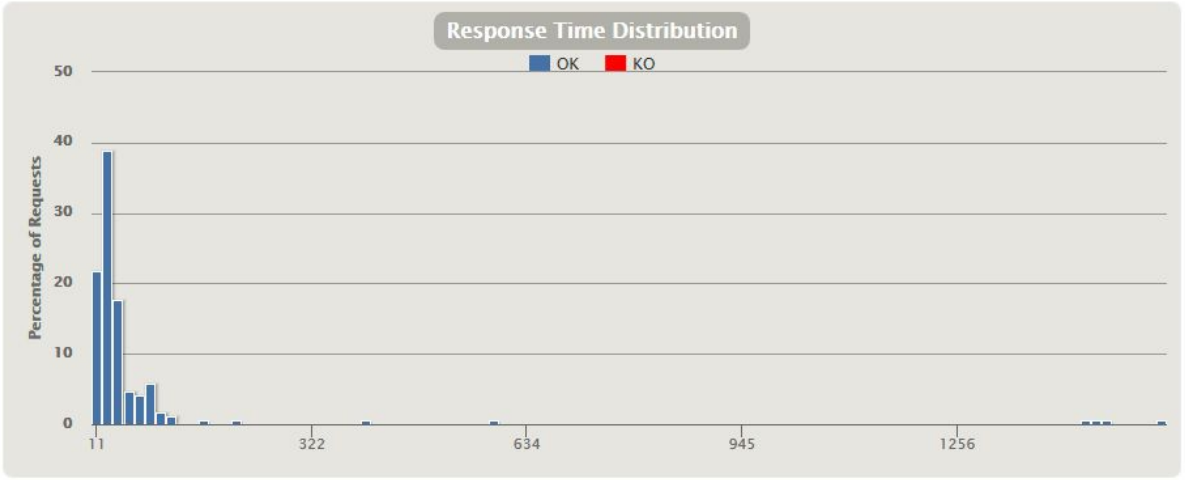
Nous sommes donc partis de ces postulats pour faire nos tests de charge, que nous avons exécutés avec *Gatling*. Nous avons effectué ces tests sur une machine avec un Intel Core i5-6300U et 8Go de RAM sous Fedora 28. Cette configuration est inférieure à une machine que nous pourrions utiliser en production, mais permet de nous donner une vague idée des performances de notre application. Le serveur a été échauffé avant d'effectuer les tests de charge.



Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	175	175	0	0%	2.5	3	31	46	116	1457	1560	75	223

<u>Figure n°17:</u> Répartition du temps des requêtes et des codes d'erreurs

Tout d'abord, nous pouvons voir que même si quelques requêtes mettent du temps à être traitées, nous n'avons pas d'incohérence dans le système (pas de code autres que 2XX) et nous n'avons pas de timeout non plus.



<u>Figure n°18:</u> Distribution du temps de réponse des requêtes

De plus, nous pouvons voir que les requêtes qui prennent du temps ne sont qu'une minorité des requêtes. On aurait donc 4 requêtes sur 175 qui prennent un temps peu raisonnable (supérieur à 1200 ms), et le temps de traitement du reste des requêtes est aux alentours de 40 ms. Le résultat le plus intéressant est cependant le suivant.

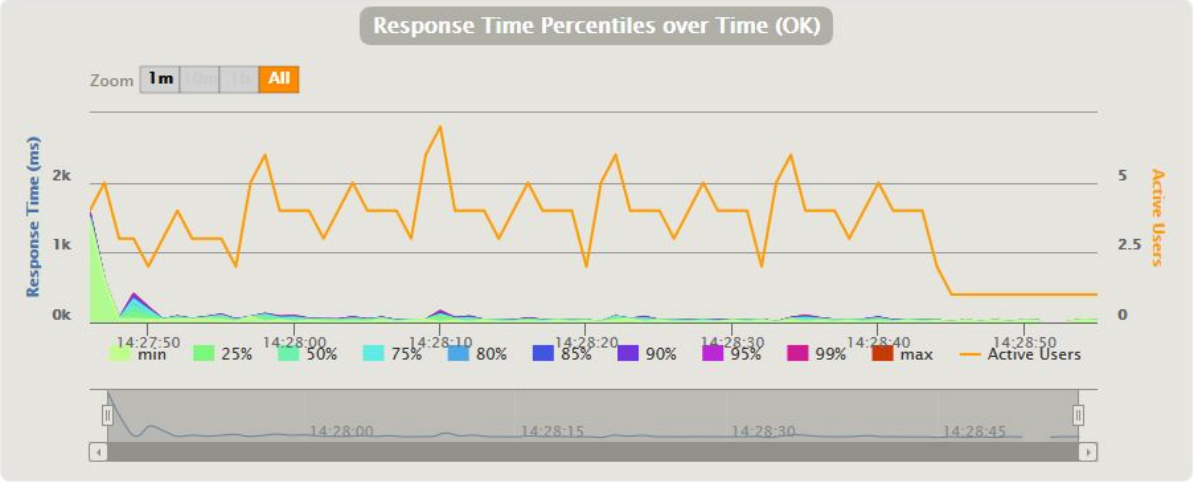


Figure n°19: Distribution du temps de réponse des requêtes

On peut voir que le temps de réponse des premières requêtes est énorme par rapport aux autres. Le plus étonnant, c'est que tous les microservices ont été chauffés en utilisant le scénario transversal. On pense que Kafka met du temps à traiter les 4 premières requêtes, ce qui est sûrement lié à la configuration de la machine que nous avons utilisé. En effet, nous avons tout de même 12 microservices, Kafka et 8 bases de données Mongo (une par service qui a besoin de persistance). Tout ceci doit entraîner un ralentissement au lancement des tests de charge. On observe cependant qu'une fois ce ralentissement passé, le système tient parfaitement la charge, définie comme le pire des cas de notre contexte.

Pour conclure sur les tests de charge, nous avons choisi un contexte réduit, mais qui nous permettait d'être testé avec les moyens à notre disposition. En effet, il est peu probable que nous tenions 1000 requêtes par seconde dans ce contexte de déploiement. Dans celui-ci, nous n'utilisons pas la puissance des micro services, à savoir la possibilité de passer à l'échelle horizontalement. Par exemple, pour faire un contexte de déploiement "réaliste", il aurait fallu lancer 1 microservice par machine (au moins), avec chaque base de données sur une autre machine et voire même *Kafka* sur plusieurs machines. Au vue des tests de charge que nous avons effectués sur une unique machine avec une petite configuration, nous sommes assez confiants que sur un contexte de déploiement "réaliste", nous pourrions tenir une charge conséquente.

Conclusion

Pour conclure, ce projet était, pour nous, le premier contact avec l'approche micro services. Nous avons donc fait face aux nombreuses difficultés que celle-ci engendre. De plus, nous avons eu des problèmes au niveau de la gestion du code sur ce projet.

D’une part, nous avons fait face à des problèmes liés à la distribution des responsabilités entre les différents microservices. En effet, pour essayer nos services, nous avions besoin de les déployer dans un environnement où étaient présents tous les autres services. Nous avons passé du temps pour déterminer de quel services les problèmes venaient, rendant le débuggage plus compliqué que sur un projet synchrone REST par exemple.

D’autre part, nous nous sommes rendus compte que nous avons négligé certains principes de base et nous l’avons payé en temps. Tout d’abord, nous avons négligé pendant un certain temps la mise en place de tests, permettant de valider que nous avions bien développé ce que nous voulions développer. Ces tests ont longtemps consisté en un scénario manuel, qui était une extension du scénario de bout en bout fait pour le premier rendu, en plus de tests avec *Postman*. Le manque de test unitaire s’est fait ressentir vers la fin du projet, lorsque les nouvelles fonctionnalitées nécessitaient de modifier celles déjà présentes. Nous aurions dû mettre en place des tests unitaires ou bien une liste de scénarii permettant de définir les contrats de nos services, et des tests de non-régression.

Par contre, outre ces problèmes, nous avons maintenu une gestion de projet intéressante durant tout le projet. Les séances de TD permettaient d’avoir des discussions constructives sur les possibilités au niveau de l’implémentation des nouveautés. De plus, le maintien d’une documentation interne à jour nous a procuré un gain de temps et d’efficacité hors du commun, en évitant toute confusion entre les membres du groupe.

En conclusion, même si nous avons fait face à de nombreux problèmes, tant liés aux microservices qu’à la gestion de code, nous en avons surmonté une grande majorité et nous avons progressé grâce à cette expérience, en approchant les problématiques liées aux architectures microservices.