Rapport SOA: Projet Uberoo

I. Description de l'architecture et des choix de conceptions

A. Architecture initiale

Dès le MVP, nous avons identifié que les communications entre les différents services risquaient de poser problème car elles sont fréquentes et qu'elles représentent le coeur du métier (la création de commande entre autres). C'est pourquoi nous avions intégré un "bus de messages" (simulé à l'époque). Nous avions pensé l'architecture de la manière suivante (chaque "boîte" correspond à un conteneur) :

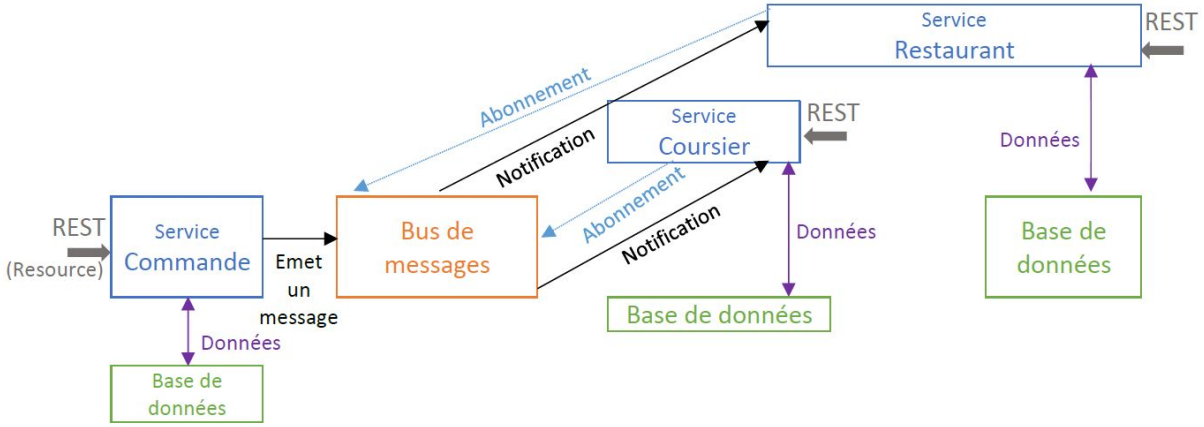

Figure 1 - Architecture initiale du projet

Comme le montre la *figure 1*, chaque service dispose d'une base de données (déployée sur un conteneur qui lui est propre) et d'une API REST lui permettant de communiquer avec une interface utilisateur par exemple. Les échanges entre les services passent tous par le bus de messages qui transmet les messages au service concerné. Le choix de cette architecture (à moitié "Document" et à moitié "Ressource") vient de la résilience que nous offre le bus ainsi que du système de notifications qui va permettre de ne pas envoyer des requêtes en boucle d'un service à l'autre. Pour ce qui est de l'aspect "Ressource", une API REST nous permet d'offrir une interface standardisée vers "l'extérieur" (une application disposant d'une interface utilisateur par exemple), de plus les éléments exposés s'apparentent facilement à des ressources et ne nécessite pas d'être notifié (contrairement à l'évènement d'une nouvelle commande émise du service de "Commande" vers le service du "Restaurant" par exemple).

B. Interaction avec l'API

Disposant d'un bus de messages, notre architecture fait donc coexister différents type de messages, avant l'apparition des nouvelles "User Story" (US) (voir partie II) il n'en existait qu'un :

- **NEW_ORDER** : émit par le service de "Commandes", consommé par les services de "Coursiers" et de "Restaurants"

Exemple :

```
{"address":"410 ch de chez moi","food":["plat1","plat2"],"type":"NEW_ORDER"}
```

Il existait initialement les routes suivantes :

- Service de "Commandes" :
  - Permet d'ajouter une nouvelle commande :
    Chemin : /orders
    Méthode : POST
  - Permet de valider ou non une commande en changeant son statut :
    Chemin : /orders/{orderId}/
    Méthode : PUT
  - Permet de trouver tous les plats correspondant à un tag :
    Chemin : /meals
    Méthode : GET
- Service de "Restaurants" :
  - Permet de trouver toutes les commandes à préparer :
    Chemin : /restaurants/orders
    Méthode : GET
  - Permet de marquer une commande comme terminée ou non :
    Chemin : /restaurants/orders/{orderId}/
    Méthode : PUT
- Service de "Coursier" :
  - Permet de valider ou non une livraison en changeant son statut :
    Chemin : /deliveries/{deliveryId}/
    Méthode : PUT
  - Permet de trouver toutes les livraisons à faire :
    Chemin : /deliveries
    Méthode : GET

C. Remarques

Très rapidement nous avons été confronté à des problèmes de performances nous empêchant de continuer à déployer une telle architecture, en effet des conteneurs "mySQL" (hébergeant les bases de données) sont très gourmands en RAM (en particulier à l'initialisation) ce qui provoquaient des "*timeout*" sur d'autres conteneurs qui nécessitent que les base de données soient démarrées (les services). Nous avons donc créé un unique conteneur qui héberge désormais les trois bases de données (chaque service dispose de sa propre base, au sens "SQL" du terme, mais elles sont toutes déployées dans le même conteneur). Il est évident que nous aurions préféré garder le

système précédent (*figure 1*) qui limitait grandement un potentiel goulot d'étranglement sur les accès aux bases de données lors de la montée en charge.

II. Répercussion des US sur l'architecture et interactions entre les services

Au fur et à mesure que le projet avançait, de nouvelles US sont venues compléter les fonctionnalités attendues du projet. Il a été nécessaire de créer de nouveaux types de messages, de nouvelles routes et un nouveau service pour les supporter. Pour ne pas surcharger le rapport les nouvelles routes et services seront décrits dans les flow des scénarios, le listing de l'ensemble des nouveaux messages est le suivant :

- **NEW_FEEDBACK**: émit par le service de "Commandes", consommé par le service de "Restaurants"

Exemple:

{"type":"NEW_FEEDBACK","author":"prenom nom","content":"mon avis","mealName":"Mon plat","restaurantName":"Mon restaurant","restaurantAddress":"25 rue du restaurant"}

- **NEW_MEAL** : émit par le service de "Restaurants", consommé par le service de "Commandes"

Exemple:

{"type":"NEW_MEAL","name":"Mon plat","restaurantName":"Mon restaurant","restaurantAddress":"25 rue du restaurant","price":10.5,"tags":["Mon tag"]}

- **NEW_RESTAURANT** : émit par le service de "Restaurants", consommé par le service de "Commandes" et le service de "Coursiers"

Exemple:

{"type":"NEW_RESTAURANT","name":"Mon restaurant","address":"25 rue du restaurant"}

- **ORDER_DELIVERED**: émit par le service de "Coursiers", consommé par le service de "Restaurants".

  Exemple:

{"type":"ORDER_DELIVERED","restaurantName":"Mon restaurant","restaurantAddress":"25 rue du restaurant","deliveryAddress":"410 ch de chez moi","food":["plat1","plat2"],"date":"date typé en java","account":"numero de compte","amount":10.0}

- **PAYMENT_CONFIRMATION**: émit par le service de la banque, consommé par le service de "Commandes" et "Coursiers".

Exemple:

{"type":"PAYMENT_CONFIRMATION","status":true,"id":-1}

- **PROCESS_PAYMENT** : émit par les services des “Commandes” et des “Restaurants”, consommé par le service de la banque.

Exemple:

{"type":"PROCESS_PAYMENT","account":"numero de compte","amount":10.0,"id":-1}

III. Bilan des user stories implémentées

La majorité des US nécessitent que les bases de données contiennent au minimum un restaurant, un plat ainsi qu’un utilisateur (pour le service de “Commande”) et un coursier (pour le service de coursier). Pour éviter d’avoir trop des étapes qui apparaissent systématiquement nous allons les décrire ici :

1. As a restaurant owner, I create my restaurant in the system :
   1.1. Requête POST, dont le body contient l’adresse et le nom du restaurant, sur la route “*/restaurants*” du service “Restaurant”.
   1.2. Un message “**NEW_RESTAURANT**” est émis par le service “Restaurant” sur le bus.
   1.3. Le service “Commande” consomme le message et crée le restaurant associé dans sa base données.
   1.4. Le service “Coursier” consomme le message et crée le restaurant associé dans sa base de données.
2. As a restaurant owner, I add a meal to my restaurant :
   2.1. Requête POST, dont le body contient le nom, le prix, les ingrédients et les tags du plat. Cette requête est envoyée sur la route (l’id a été obtenu à l’étape précédente) “*/restaurants/{restaurantId}/meals*” du service “Restaurant”.
   2.2. Un message “**NEW_MEAL**” est émis par le service “Restaurant” sur le bus.
   2.3. Le service “Commande” consomme le message et crée le plat associé dans sa base données.
   2.4. Le service “Coursier” consomme le message et crée le plat associé dans sa base de données.
3. As a customer, I create my account :
   3.1. Requête POST, dont le body contient les informations de l’utilisateur, sur la route “*/users*” du service “Commande”.
   3.2. Le service de “Commande” rentre cette commande dans la base de données de “Commande.
4. As a delivery man, I create my account :
   4.1. Requête POST, dont le body contient les informations du coursier, sur la route “*/coursiers*” du service “Coursier”.
   4.2. Le service de “Coursier” rentre cette commande dans la base de données de “Commande.

Semaine 41: “User Stories” initiales

- 1. As Gail or Erin, I can order my lunch from a restaurant so that the food is delivered to my place.

Flow dans l’application :

1. As Gail or Erin, I search the list of the restaurants :
   1.1. Requête GET sur la route “*/restaurants*” du service de “Commande”.
   1.2. Cette requête fait appelle à la base de données du service de “Commande” pour retourner la liste.
2. As Gail or Erin, I decide to look for the meals of a specific restaurant :
   2.1. Requête GET sur la route “*/restaurants/{restaurantId}/meals*” du service “Commande”.
   2.2. Cette requête fait appelle à la base de données du service de “Commande” pour retourner la liste.
3. As Gail or Erin, I choose my meal and send it to the system :
   3.1. Requête POST, dont le body contient l’adresse de livraison, la liste des plats et l’utilisateur qui effectue la commande.Cette quête est faite sur la route “/orders” du service “Commande”.
   3.2. Le service de “Commande” rentre cette commande dans la base de données de “Commande”, calcul l’ETA et renvoie le résultat à l’utilisateur.
4. As Gail or Erin, I decide to accept the ETA :
   4.1. Requête PUT, dont le body est la commande avec le statut passé à “true”, sur la route “orders/{orderId}” du service “Commande”.
   4.2. Un message “**NEW_ORDER**” est émis par le service “Commande” sur le bus.

- 2. As Gail, I can browse the food catalogue by categories so that I can immediately identify my favorite junk food.

Flow dans l’application:

1. As Gail, I search the list of meals for all restaurant using a tag (like a category) :
   1.1. Requête GET sur la route “/meals?tag=categoryName” où “categoryName” est le nom de la catégorie à rechercher.
   1.2. Cette requête fait appelle à la base de données du service de “Commande” pour retourner la liste des plats correspondant à la catégorie demandée.

- 3. As Erin, I want to know before ordering the estimated time of delivery of the meal so that I can schedule my work around it, and be ready when it arrives.

Flow dans l’application:

1. As Erin, I search the list of meals for all restaurant:
   1.1. Requête GET sur la route “*/meals*” du service “Commande”.
   1.2. Cette requête fait appelle à la base de données du service de “Commande” pour retourner la liste.
2. As Erin, I choose my meal and send it to the system :

2.1. Requête POST, dont le body contient l'adresse de livraison, la liste des plats et l'utilisateur qui effectue la commande.Cette quête est faite sur la route "*/orders*" du service "Commande".
2.2. Le service de "Commande" rentre cette commande dans la base de données de "Commande", calcul l'ETA et renvoie le résultat à l'utilisateur.

3. As Erin, I decide to accept the ETA :
3.1. Requête PUT, dont le body est la commande avec le statut passé à "true", sur la route "*orders/{orderId}*" du service "Commande".
3.2. Un message "**NEW_ORDER**" est émis par le service "Commande" sur le bus.

- 4. As Erin, I can pay directly by credit card on the platform, so that I only have to retrieve my food when delivered.

Cette US a fait apparaître un nouveau service, le service qui fait l'intermédiaire entre notre système et la banque. Le nouveau service vient se connecter au bus de manière analogue aux service "Commande", "Restaurant" et "Coursier".

Flow dans l'application:

1. As Erin, I create my order and validate the ETA : Voir US 3.
2. As Erin, I send my IBAN number to the system
2.1. Requête POST, dont le body contient l'IBAN et le somme à débiter, sur la route "*/orders/{orderId}/payments*" du service "Commande".
2.2. Un message "**PROCESS_PAYMENT**" est émis par le service "Commande" sur le bus.
2.3. Le service "Banque" consomme le paiement, le traite et en fonction de sa réussite où son échec émet un message "**PAYMENT_CONFIRMATION**".
2.4. Le service "Commande" consomme le message.
2.5. Le service de "Commande" modifie le statut du paiement dans sa base de données.

- 5. As Jordan, I want to access to the order list, so that I can prepare the meal efficiently.

Flow dans l'application:

1. As Jordan, I search all the orders to prepare for my restaurant :
1.1. Requête GET sur la route "*/restaurants/{restaurantId}/orders*" du service "Restaurant".

- 6. As Jamie, I want to know the orders that will have to be delivered around me, so that I can choose one and go to the restaurant to begin the course.

Flow dans l'application:

1. As Jamie, I want to know the orders that will have to be delivered around me :
1.1. Requête GET sur la route "/deliveries/?latitude=&longitude=".

1.2. Cette requête fait appel à la base de données du service de livraison pour retourner l'ensemble des commandes qui sont à proximité de sa position.

2. I can choose one and go to the restaurant to begin the course :
   2.1. Requête PUT sur la route "*/deliveries/{deliveryId}*" avec en paramètre l'id coursier.
   2.2. Cette requête fait appel à la base de données du service de livraison pour assigner le coursier à la livraison.

Ayant déjà créer un service spécifique aux coursiers, l'implémentation de cette US fut relativement simple. En effet nous avons ajouté des *Query param* à l'url d'obtentions des Deliveries afin de spécifier une latitude et une longitude lesquelles seraient envoyées par l'application du coursier grâce aux coordonnées GPS de son téléphone. Les adresses déjà contenues dans les entités Restaurant et Delivery nous permettent de calculer les latitudes et longitudes (dans notre exemple elles sont mockées mais on a imaginé un appel à un service de géolocalisation).

- 7. As Jamie, I want to notify that the order has been delivered, so that my account can be credited and the restaurant can be informed.

Flow dans l'application:

1. As Jamie, I want to notify that the order has been delivered :
   1.1. Requête PUT sur la route "*/deliveries*".
   1.2. Cette requête fait appel à la base de données pour mettre la livraison dans l'état livrée.
2. So that my account can be credited and the restaurant can be informed :
   2.1. Un message "**ORDER_DELIVERED**" est émis par le service de livraison pour le bus.

Notre architecture mêlant Resource et Document prend véritablement son sens dans cet exemple. Pour cet exemple on voit clairement que plusieurs de nos services doivent récupérer cette information. A la réception de cette action, un message est instancié contenant l'ensemble des informations que les services de restaurant et de la banque doivent avoir. Ce message est ensuite envoyé sur le bus de message que ces 2 services écoutent.
Chaque service prend l'information du message qui l'intéresse et l'interprète. Pour la banque, un second message doit être émis afin de confirmer au coursier que la transaction a bien eu lieu.

Semaine 43: Première évolution

- 8. As Jordan, I want the customers to be able to review the meals so that I can improve them according to their feedback;

Flow dans l'application:

1. As Jordan, I want the customers to be able to review the meals :
   1.1. Requête POST sur la route "*/meals/{mealID}/feedbacks*" du service de commande.

1.2. Cette requête envoie un message sur le bus de type **NEW_FEEDBACK** qui sera consommé par le service des restaurants, et qui mettra à jour sa base de données pour prendre en compte le nouveau *feedback*.

2. so that I can improve them according to their feedback :
   2.1. Requête GET sur la route "*/restaurants/{restaurantId}/meals/{mealId}/feedbacks*" du service des restaurants.
   2.2. Cette requête retourne la liste des *feedbacks* pour un plat donné dans un restaurant donné.

- 9. As a customer (Gail, Erin), I want to track the geolocation of the coursier in real time, so that I can anticipate when I will eat.

Flow dans l'application:

1. As a customer (Gail, Erin), I want to track the geolocation of the coursier in real time :
   1.1. Requête GET sur la route "*/coursiers/{idCoursier}*".
   1.2. Cette requête fait appel à la base de données pour récupérer la position courante du coursier.

Pas de modification d'architecture pour cette User Story. Les consommateurs souhaitent seulement savoir la géolocalisation du coursier. On ajoute seulement les latitudes et longitudes du coursier sur l'entité, on imagine que l'application des consommateurs appelle régulièrement la route permettant d'obtenir les informations sur le coursier. Du côté du coursier, son application appelle régulièrement la route PUT du service coursier afin de mettre à jour ses informations.

- 10. As Terry, I want to get some statistics (speed, cost) about global delivery time and delivery per coursier.
  Cette US a été implémentée mais un problème sur le scénario d'acceptation provoque un timeout sur le service des coursiers. La cause est encore inconnue.

Flow dans l'application:

1. As Terry, I want to get some statistics (speed, cost) about global delivery time and delivery per coursier :
   1.1. Requête GET sur la route "*/coursiers/{idCoursier}/deliveries*".
   1.2. Cette requête fait appel à la base de données du service de coursier afin de récupérer les statistiques d'un coursier pour un restaurant donné.
2. Actions préalables :
   2.1. Pour les requêtes de prise de livraison par un coursier, on enregistre la date de début d'une livraison.
   2.2. Pour les requêtes de validation de livraison, on enregistre la date de fin de livraison.

Premièrement, nous devions enregistrer différentes informations au moment de la livraison d'une commande. Ainsi on enregistre désormais les dates de début et fin de livraison pour l'entité donnée.
Au sein du service coursier, nous avons rajouté une route afin de récupérer les statistiques d'un coursier pour un restaurant donné. En effet, il nous semblait important qu'un directeur de restaurant puissent connaître les résultats d'un coursier pour son restaurant spécifiquement.

Actuellement, on récupère le coursier puis le restaurant, puis on compare les données des livraisons du coursier avec celles du restaurant (comparaison de latitudes et longitudes pour la vitesse par exemple). Cette manière de faire nous semble très peu efficace à présent.
La méthode est beaucoup plus lourde en calcul et en accès à la base de données que beaucoup d'autres.
Pour la suite, on aurait voulu isoler cette partie de statistiques dans un service à part entière. Connaissait l'importance de ce type de données au sein d'entreprise mettant en lien des consommateurs à des inconnus, il semble primordial de pouvoir étudier les performances de ces inconnus rémunérés.

Ce service serait appelé pour chaque annulation ou validation d'une livraison avec les statistiques de la livraison courante. On aurait alors une très grande rapidité d'accès car les calculs seraient déjà effectués. Ces calculs auparavant tous fait au même moment, serait répartis de manière uniforme sur l'ensemble des livraisons.

Semaine 44: Seconde évolution

- ~~11. As Terry, I can emit a promotional code so that I can attract more customer to my restaurant.~~
  Nous n'avons pas eu le temps de faire cette US. Voir US13.

- 12. As Jamie, I want to inform quickly that I can't terminate the course (accident, sick), so that the order can be replaced.
  De manière analogue à l'US10, cette US a été implémentée mais il persiste un problème sur le scénario d'acceptation.

Flow dans l'application:

1. As Jamie, I want to inform quickly that I can't terminate the course (accident, sick) :
   1.1. Requête PUT sur la route "*/deliveries/{deliveryId}*" avec en corps le message d'annulation "CourseCancelMessage".
2. So that the order can be replaced
   2.1. Un message "**COURSE_CANCEL**" est généré et envoyé sur le bus de message.

Pour cette US, le problème d'implémentation vient d'un soucis de compréhension des principes REST. En effet, on envoie une requête PUT sur le service s'occupant des Deliveries avec en corps du message, un message sur les raisons de l'annulation de la commande. En effet, on ne doit mettre dans les corps de requêtes PUT que des objets correspondant aux entités en base donc des Delivery dans ce cas là.

En considérant que dans le frontend de l'application, on aurait précédemment récupéré l'entité par un GET, le frontend n'aurait fait que modifier les attributs désirés avant de les renvoyer.
Les annulations n'étant qu'un phénomène relativement exceptionnel (pas de besoin de création d'un nouveau service), nous aurions créer un nouveau contrôleur au sein du service coursier, lequel permet de récupérer les messages d'annulation.

L'ensemble du contenu des méthodes restant les mêmes, le seul problème est actuellement le non respect des consignes pour respecter le REST.

On passe la livraison dans l'état annulé, en effet, on ne va pas demandé à un autre coursier de venir récupérer la commande auprès du coursier potentiellement blessé ou malade.
On instancie un message destiné à être réceptionné par les autres services pour que le restaurant puissent reproduire la même commande qui sera par la suite donné à un autre coursier.

Semaine 45: Dernière évolution

- ~~13. As Terry, I can emit a promotional code based on my menu contents (e.g., 10% discount for an entry-main course-dessert order), so that I can sell more expensive orders.~~

  Nous n'avons pas eu le temps de faire cette US.

  Cette US qui est l'évolution de l'US11 ne devrait pas entraîner beaucoup de changement dans l'architecture, il suffirait de rajouter une route dans le services des restaurants pour créer le code promotionnel et un nouveau type de message liant le plat et la promotion qui serait consommé par le service des commandes. Ce message serait bien entendu envoyé lors de la création d'un code promotionnel. Il faudrait créer une route qui permet de modifier une commande dans le service des commandes afin de pouvoir passer un code promotionnel optionnel "*route /orders/{orderId}*", méthode PUT).

- ~~14. As Gail or Erin, I can follow the position of Jamie in real time, so that the food ETA can be updated automatically.~~

  Nous n'avons pas eu le temps de faire cette US.

Nous envisagions de créer un service réservé au calcul de l'ETA et à la gestion de la localisation des commandes. Ce changement entraînerait un nouveau type de message produit par le service des coursiers et consommé par ce nouveau service.
Ce message serait de type "**ORDER_LOCATION_UPDATED**" et contiendrait les nouvelles coordonnées du coursier avec l'ETA et les informations nécessaires pour retrouver la commande associée à ces informations.

IV. Tests unitaires et d'acceptation

Chaque service est testé indépendamment des autres dans un premier temps, cela pour deux raisons. Ils ont été découpé justement pour pouvoir fonctionner sans l'aide des autres (dans une certaine mesure évidemment) mais également pour s'assurer qu'un service remplisse sa fonctionnalité avant de commencer à communiquer avec les autres. Chaque service est décomposé en 3 couches :

- Couche de communication : Il s'agit de la couche "externe" du service, elle a pour vocation de recevoir des requêtes et de convertir (unmarshalling) les données récupérés en données utilisables pour la couche suivante.
- Couche métier (dites "service") : Il s'agit de la couche effectuant toutes les fonctionnalités attendues. Elle travaille avec des "DTO" (Data Transfert Object) de manière à faire abstraction de manière dont sont stockés les données. Elle va recevoir les données de la couche de communication, les traiter et les transmettre à la couche inférieure (persistance). Puis récupérer les données de la couche de persistance, les traiter (éventuellement) et les transmettre à la couche de communication.
- Couche de persistance : Comme indiqué précédemment cette couche va uniquement récupérer les données transmises et les enregistrer dans la base données ou inversement chercher des données et les renvoyer à la couche métier.

Pour cette première phase nous avons testé chaque fonctionnalité unitairement et cela pour chaque couche. Nous sommes aidés de "JUnit" et "Mockito", le premier pour faciliter la rédaction de tests en "Java" pour pouvoir faire abstraction des autres couches. Par exemple pour tester une route exposée par la couche communication nous avons "mockée" (simulée) la couche service, de cette manière nous avons pu nous assurer que la couche communication exposée bien la bonne route et effectuée bien la traduction des données sans être dépendant du fonctionnement de la couche service.

Dans un second temps nous avons testé si les fonctionnalités demandées par les US étaient bien fournis par notre projet. Nous avons donc des scénarios d'acceptation écrits en scripts bash (dossier **Scenarii**) qui correspondent aux US décrites précédemment. Après le lancement de chaque script, les données de chaque base de données sont effacées. Ainsi, chaque script commence avec un contexte identique et les *id* des tuples de chaque base de données ne sont pas remis à zéro. Ceci permet de vérifier que cela ne "tombe pas en marche". Car par exemple, les tables SQL "RESTAURANT" dans la base de données des commandes et dans la base de données des restaurants peuvent avoir des *id* qui coïncident.

Avec le recul, nous réalisons que ces scripts ont pris beaucoup de temps à être mis en place. Une solution qui aurait pu être moins chronophage, voire même plus simple lors du *débogage* aurait été de créer des scénarii "Gherkin" qui permettent de faire des tests orientés comportement. Le logiciel *behave*[1] aurait pu convenir car étant en Python, il aurait été plus simple de faire les requêtes vers les services qu'en utilisant des *curl*.

[1] https://behave.readthedocs.io/en/latest/index.html

V. Tests de charge

Dans notre architecture, la meilleure façon de stresser notre système de façon réaliste est de créer des commandes. En effet, le service des commandes serait logiquement la partie la plus sollicitée de notre système s'il était déployé en production. La seconde partie la plus active serait le service des coursiers pour la mise à jour des livraisons et des positions des coursiers, suivi du service des restaurants pour la récupération des commandes à préparer notamment. De plus, créer une commande utilise l'ensemble des services du système.

Afin de pouvoir effectuer ce test de charge, nous avons implémenté un scénario Gatling. Ce scénario simule un utilisateur qui crée une commande de ramen et accepte l'ETA, ce qui a pour effet d'envoyer un message dans le bus de type **NEW_ORDER** consommé par le service des restaurants (afin de préparer la commande) et par le service des coursiers (afin d'affecter un coursier à la commande).

Le test commence avec 10 utilisateurs par seconde puis augmente de manière aléatoire jusqu'à atteindre une injection de 20 utilisateurs par seconde, le tout sur 20 secondes. Donc on arrive potentiellement à un pic d'environ 260 utilisateurs par seconde.
Ce test paraît simple et soutenable pour un serveur à première vue mais il suffit à déstabiliser notre système et provoquer des erreurs sur un petit pourcentage des requêtes.[2] Néanmoins, ce résultat n'est pas complètement fiable étant donné qu'il a été lancé sur une machine avec peu de RAM. De plus, les services étant dockerisés, les performances sont amoindries par rapport à des serveurs qui seraient dédiés à chacun des services.

VI. Problèmes rencontrés

Kafka

Désérialisation des messages

La désérialisation des messages avec Kafka a été problématique. Les messages étant envoyés sous forme de JSON dans le bus, il semblait évident d'utiliser un JsonDeserializer fournit par Kafka dans les propriétés des consommateurs de messages. Or, notre modèle de messages est conçu de la manière suivante :
Nous avons un classe mère Message de laquelle découle tous les autres types de messages. Les classes filles de Message renseignent l'attribut String "type" contenu dans la classe Message afin de se différencier.
Donc à première vue, il semblait possible de renseigner JsonDeserializer<Message> mais cela avait pour effet de nous faire perdre les données que contenait les types de messages des classes filles de Message. Car le type inféré lors de la déserialisation est Message.

[2] https://www.joelcancela.fr/share/loadTesting%20Order+Accept/index.html

La solution trouvée est la suivante : utiliser le StringDeserializer afin de considérer chaque message comme une String (ce qui, au passage empêche une potentielle boucle infinie si le message dans le bus n'est pas un JSON ou est mal formé, problème qui a été expérimenté lors des premières semaines), est d'effectuer la désérialisation à la main grâce à la classe ObjectMapper. Cette solution bien qu'efficace car elle permet de s'assurer de l'intégrité du message, est un peu sale, et si le projet devait continuer il faudrait chercher une alternative un peu plus adaptée à l'utilisation avec Kafka. Ce problème nous a fait perdre pas mal de temps.

Base de données

Duplication des données

Comme expliqué auparavant (voir I.C), l'implémentation des base de données nous a posé pas mal de problèmes. Nous avons longuement réfléchie à la meilleure méthode pour stocker les données de notre projet. Il se trouve que nous pouvions partager les données au sein d'une seule et même base de données ou créer une base de données par service. La seconde est plus séduisante (moins d'accès concurrent, etc...) cependant la première permet de ne avoir de duplication de données.

En effet, en implémentant la seconde possibilité (une base par service) nous aurions pu laisser la responsabilité à la base de données du service "Restaurant" de stocker les restaurants, à la base de données du service de "Commande" les commandes, etc.... Mais nous aurions alors eut un grand nombre de messages différents et énormément d'échanges dans le bus, ce qui aurait grandement nuit à la lisibilité et la maintenabilité du projet (saturation du bus). Nous avons donc décidé de limiter ces échanges en stockant pour chaque service la représentation des éléments des autres services. Par exemple la notion de "restaurant" existe et est stocké dans les base de données des services "Restaurant", "Commande" et "Coursier" (mais diffère légèrement d'un service à l'autre). Malheureusement cela crée beaucoup de duplication de données, car un "restaurant" existe donc trois fois.