POLYTECH NICE-SOPHIA

Université Nice Sophia Antipolis

Rapport Uberoo

Équipe B

CANAVA Thomas, GARDAIRE Loïc, JUNAC Jérémy, MELKONIAN François

Sommaire

Introduction

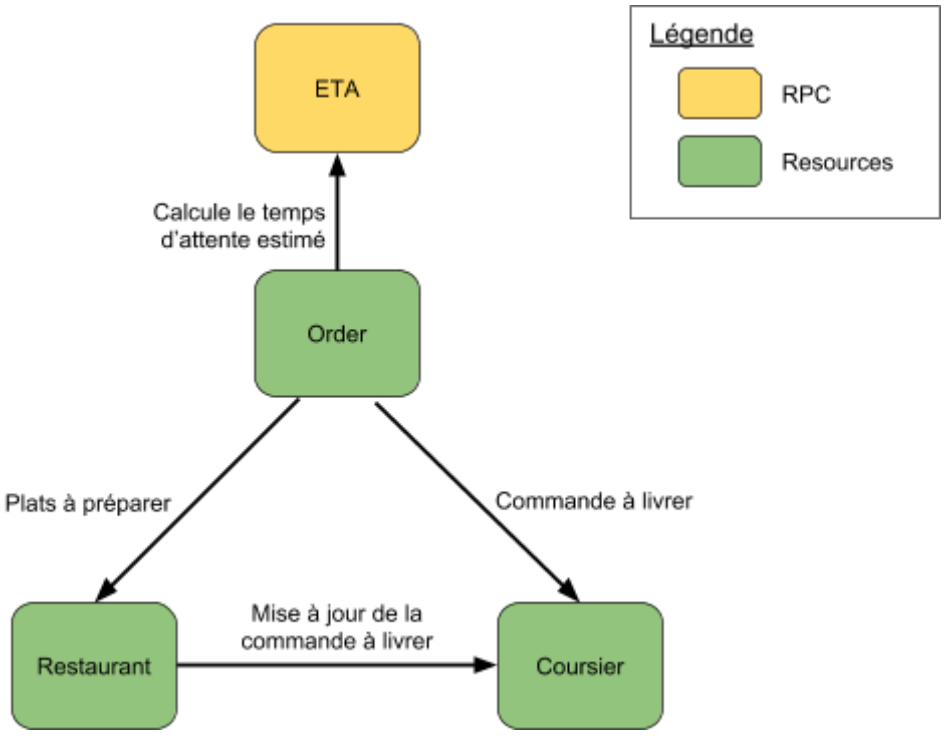

Figure n°1: Représentation de nos services

Nous avons fait le choix de découper notre application en 4 services. Les 3 premiers (*Order*, *Restaurant* et *Coursier*) sont implémentés en orienté *Ressource* et le dernier permettant le calcul de l'ETA est en *RPC*. Ces choix seront expliqués dans la partie suivante.

Le service nommé *Order* est un point d'entrée de notre système pour le client. Il possède 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.

Le service nommé *Restaurant* s'occupe de gérer l'interface utilisée par les différents restaurants partenaires de notre application. C'est ici que ces derniers peuvent consulter les commandes à préparer par exemple. À l'avenir, les restaurants pourront ajouter de nouveaux repas au catalogue via ce service.

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

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

Justification des styles de service choisis

Ressource pour la majorité du projet

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. 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.

RPC pour le service de calcul de l'ETA

À l'inverse, nous avons fait le choix d'avoir un service dédié au calcul de l'ETA malgré son faible rôle pour l'instant. Le premier argument est que ce calcul viendra à se complexifier avec l'évolution du projet et un service lui étant réservé semble donc logique.

De plus, ce service ne manipule pas de ressource à proprement parlé. En effet, l'ETA est un calcul qu'on effectue une seule fois et que nous n'avons pas forcément besoin de stocker (du moins au stade du MVP). Notre choix se porte donc naturellement sur le paradigme orienté procédure.

Le calcul de l'ETA est un protocole que l'on appelle et qui renvoie une information. Il nous permet de faire une demande au service en envoyant directement les informations des repas et la fonction qui doit calculer l'ETA, et récupérer le résultat en sortie. Un second argument en faveur de *RPC* se situe dans le futur ; si le calcul de l'ETA venait à se diversifier, il serait alors intéressant de pouvoir exposer plusieurs fonctions de calcul d'ETA suivant des paramètres précis.

Choix d'implémentation de notre API

Nous avons choisi d'utiliser *Node/Express* pour implémenter ce projet pour plusieurs raisons. En effet, *Node/Express* représentait le parfait compromis entre légèreté et possibilités. Toute l'équipe connaissait cette pile technologique couplée avec *Node*. Nous avons donc évité un coût de formation d'une partie ou de la totalité des membres de l'équipe.

De plus, nous avons choisi ces technologies après le choix de paradigme de nos services. En conséquence, *Node/Express* et *Mongo* sont des candidats de choix pour implémenter des services REST. Ces technologies sont faites pour implémenter du REST et offre un large choix de méthodes dans l'API standard pour faciliter son implémentation.

De surcroît, *Node/Express* n'est pas implémenté avec du multi-threading et passe donc mal à l'échelle verticalement. Pour palier à cela, *Node* a été créé pour permettre d'avoir de très bonnes performances lorsqu'on passe à l'échelle horizontalement.

Nous avons aussi choisi d'utiliser une base de données *Mongo* commune à tous nos services, pour faciliter le développement. Comme les données de chaque service sont indépendantes, nous pouvons facilement créer une base de données par service.

D'autre part, nous avons fait le choix d'utiliser *TypeScript* pour structurer notre code. En effet, *JavaScript* est connu pour être très permissif, ce qui, sur des gros projets, peut conduire rapidement à du code peu lisible et donc peu maintenable. *TypeScript* apporte une rigueur à Javascript qui permet d'éviter ce problème de passage à l'échelle du code. Dans notre cas, cela nous a aussi permis de mettre en commun des interfaces que nous avons utilisées dans plusieurs services, via un modules *commons* importé dans les services concernés. Cependant, cela nous a aussi contraint à uniformiser nos objets métiers entre les différents services, alors que certains n'avaient pas la même représentation (*e.g.* une *Order* ne devrait pas forcément avoir les mêmes champs pour le service *Order* et pour le service *Coursier*).

Enfin, notre gestion des erreurs n'est pas complète. Nous avons essayé de respecter au maximum la sémantique REST en utilisant par exemple le code 201 lors de la création d'une ressource, etc. Mais il reste tout de même des erreurs non gérées, comme par exemple si la base de données est injoignable.

Aperçu de notre API dans les 2 autres styles

Document (alias Messages)

Si nous avions fait le choix d'implémenter notre solution en Document, nous aurions alors différents types d'événements qui permettraient la communication entre les différents services. Chacun des services détaillés ci-dessus consommeraient et produiraient alors ces événements. Voici une liste des événements que nous utiliserions :

- Demande de lister les repas avec des contraintes sur le type de repas (asiatique, italien, etc) et la réponse contenant les repas
- Demande de lister les repas d'un restaurant précis et la réponse contenant les repas
- Demande de calcul de l'ETA et réponse du système contenant l'ETA
- Commande d'un client
- Changement d'état d'une commande
- Demande pour lister les commandes à préparer et la réponse contenant les commandes

Prenons l'exemple de notre service *Order*. Il produirait des événements comme les demandes de calcul de l'ETA et de lister les commandes d'un restaurant, ou un événement permettant de changer l'état d'une commande. Au contraire, *Order* consommerait la réponse d'une demande pour lister des repas avec des contraintes ou la réponse au calcul de l'ETA. Il consommerait aussi un événement de commande de client, étant le point d'entrée du client dans notre système.

Suivant la même logique, le service *Restaurant* consommerait un changement d'état d'une commande qui correspond à la création de la commande pour débuter la préparation du repas. À la fin de la préparation, il produit alors un nouvel événement de changement d'état pour la commande indiquant qu'elle est prête à être livrée. De plus, il consomme les demandes de lister les plats d'un restaurant et produit une réponse à ces demandes. Enfin, il peut aussi produire une demande pour lister les commandes à préparer et consommer sa réponse.

Le service gérant le calcul de l'ETA ne serait concerné que par un type d'événement : il consommerait les demandes de calcul d'ETA et produirait une réponse contenant la valeur attendue.

Enfin, le service *Coursier* serait principalement concerné par les changements d'état des commandes. Il consommerait tous les événements de changement d'état envoyés par les autres services pour mettre à jour les commandes en cours. Au niveau de la production, il ne pourrait produire un événement lorsqu'il reçoit une demande sur l'état d'une commande. Prenons l'exemple où un livreur veut savoir si la commande est prête à être livrée. Il fait une demande à *Coursier* via un événement. Le service le consomme et produit en sortie un événement contenant l'état de la commande.

Cette approche reste limitée dans notre cas, car elle se résume à avoir un événement permettant de faire une demande et en parallèle un autre type d'événement qui correspond à la réponse à cette demande. Étant donné que nous avons peu de communications entre un producteur et plusieurs consommateurs ou inversement, cette approche ne semble pas adaptée à notre problème.

Remote Procedure Call

Dans le cas où nous aurions choisi d'implémenter l'approche RPC, nous aurions alors dû exposer des procédures pour chaque service que nous pouvons consommer.

Pour le *Restaurant*, il pourrait exposer une méthode permettant d'afficher les commandes à préparer, ainsi qu'une méthode permettant d'ajouter une commande à préparer. Une dernière serait disponible pour mettre à jour une commande lorsque celle-ci est prête à être livrée.

Le service *Order*, quant à lui, exposerait des méthodes utiles pour le client. La première permettrait de lister les repas des différents restaurants en y ajoutant des contraintes sur les types de repas (asiatique, italien, etc). La seconde permettrait la création d'une commande.

Enfin, le Coursier aurait lui aussi deux méthodes à exposer : une permettant d'assigner un coursier à une commande pour la livraison de cette dernière et une seconde permettant de mettre à jour le statut d'une commande lorsque celle-ci est en cours de livraison.

Cette approche semble correspondre à nos besoins. elle remplit les critères nous permettant d'implémenter le MVP demandé. Cependant, notre choix ne s'est pas tourné vers l'orienté *RPC* car il reste dépendant des différentes API. En effet, pour fonctionner, chaque service voulant communiquer avec un autre doit connaître son API (le nom des méthodes, les paramètres à passer, etc). Donc chaque modification sur un service se répercute sur tous les services liés à ce dernier. Cet inconvénient nous a dissuadé d'utiliser cette approche.

Résultat des tests de charge

Nous avons fait des tests de charge sur une partie de notre application.

En effet, la partie de notre application la plus critique est la consultation de la liste des plats et le calcul de l'ETA. Cette partie de l'application sera celle soumise au plus grand nombre de requêtes: un client peut consulter les plats et calculer l'ETA sans pour autant commander. On peut imaginer des cas où l'ETA est trop long pour l'utilisateur et il décide donc de ne pas commander.

De plus, étant donné notre architecture orientée service, nous pouvons faire s'exécuter les services sur différentes machines. Donc l'absence des autres services dans les tests de charges ne biaise pas nos résultats, car en production on peut effectivement mettre ces deux services sur une machines à part.

Dans le cadre du MVP, 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 commandent à manger sur une période d'une heure (16.6 étudiants à la minute en moyenne), on peut raisonnablement penser qu'on n'aura pas plus de 100 étudiants à la minute dans un pic de charge. De plus, toujours dans le cadre du MVP, notre première version de l'application fournit un panel de 30 plats pouvant être commandés.

Nous sommes donc partis de ces postulats pour faire nos tests de charge, que nous avons exécuté 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 pour 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 charges.

STATISTICS — Expand all groups | Collapse all groups

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	200	200	0	0%	3.333	2	9	18	28	34	37	12	9
Get meals	100	100	0	0%	1.667	5	18	25	31	37	37	19	8
Compute ETA	100	100	0	0%	1.667	2	4	8	10	14	27	6	4

Figure n°2: Résultats des tests de charge

Nous pouvons tout d'abord voir que toutes les requêtes sont revenues avec un code 2XX, ce qui indique qu'il n'y a eu aucune erreur liée à la charge (timeout, incohérence de base de données, …)

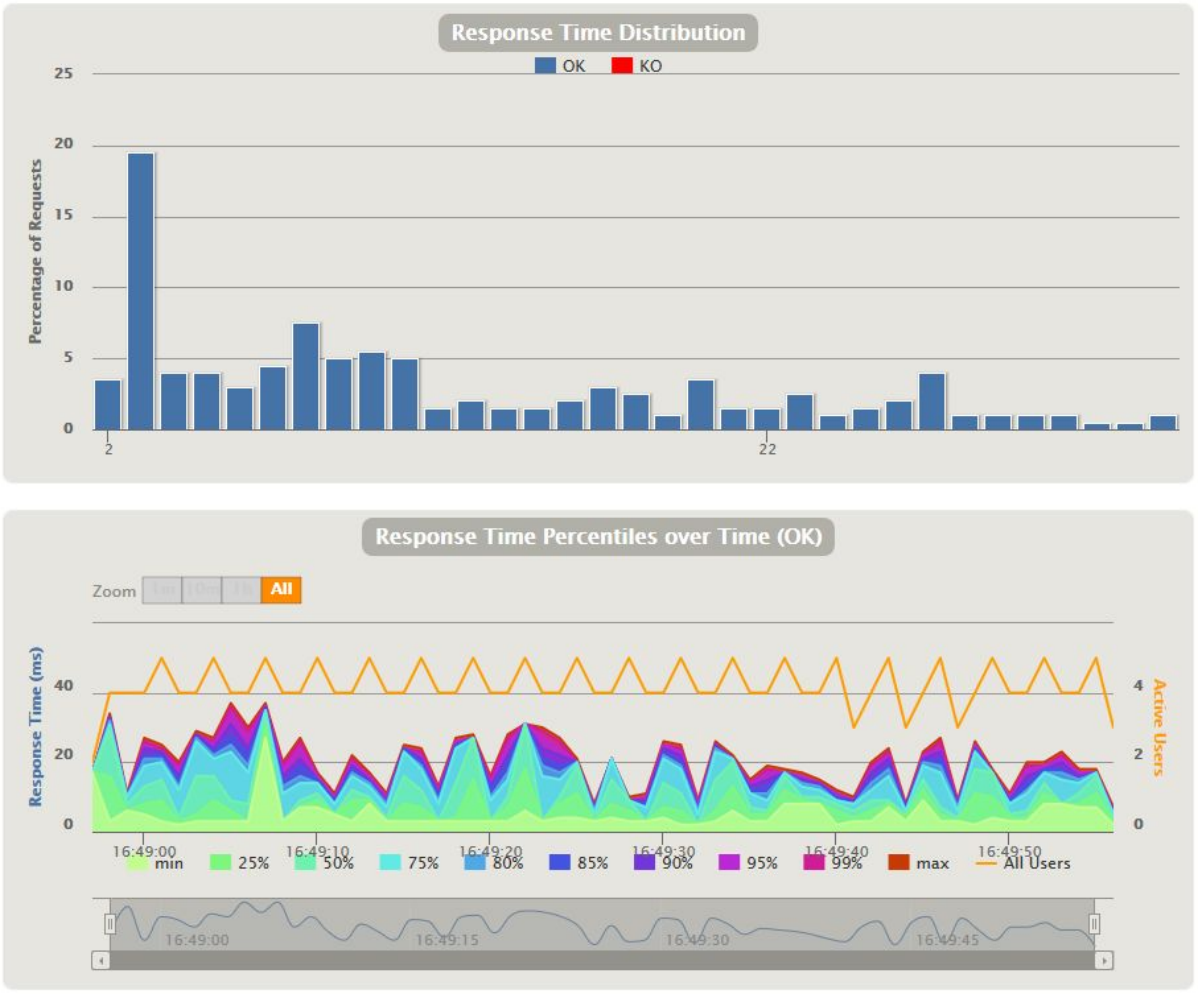

Figure n°3: Graphiques représentant les temps de réponse des requêtes

Nous observons ensuite que les services tiennent parfaitement la charge dans le contexte défini, avec tout de même un petit temps pour supporter la charge complète, et ce malgré le fait que le serveur ait été échauffé. Cependant, le temps d'attente reste inférieur à 40 ms, ce qui est raisonnable, et les services arrivent à gérer toutes les requêtes injectées.

Pour conclure, nous avons vu que, même avec une configuration de machine en deçà de ce que nous aurions en production, nous arrivons à tenir le contexte que nous nous sommes fixés dans le cadre du MVP.