Les nouvelles architectures logicielles

1ère partie

L'informatique repose sur des architectures matérielles et logicielles. Je ne vous apprends rien. Certaines existent depuis quasiment les origines de l'informatique « moderne », dans les années 1970, d'autres sont bien plus récentes. On parle beaucoup du Cloud Computing, mais le Cloud n'est finalement qu'une évolution d'architectures utilisée depuis de nombreuses années.

Les architectures évoluent tout naturellement avec les évolutions techniques, les nouveaux langages, les nouvelles technologies. Ces évolutions ont un impact direct sur les applications et donc sur vous, les développeurs. Ces changements sont parfois masqués, car vous n'êtes pas toujours au contact direct.

Programmez ! vous propose un dossier en plusieurs parties pour explorer quelques architectures. Nous aborderons, en vrac : le Cloud Computing, les architectures dites lambda, les microservices, les conteneurs.

Dans cette 1ere partie : nous parlerons de microservices et Java.

La rédaction.

Les architectures microservices

L'architecture Microservices a fait couler beaucoup d'encre dernièrement : il est le temps de faire le point.

Jérôme Doucet,

et Romain Niveau,

Consultants Xebia

L'idée derrière ce terme est d'isoler chaque fonctionnalité dans un composant qui lui est propre. Les différents services découlant de ce découpage communiquent au travers du réseau plutôt que par des appels de fonctions dans un processus. Une application microservices représente alors un système constitué d'un certain nombre de petites unités indépendantes qui collaborent entre elles, ayant chacune un cycle de vie et une responsabilité propre.
Cette architecture répond à des problèmes récurrents que posent les architectures monolithiques traditionnelles.

LES CONCEPTS

Les Architectures monolithiques et leurs problèmes

L'architecture monolithique est l'une des plus répandues dans les projets informatiques.
Plusieurs raisons expliquent cela. Au début d'un projet, quand celui-ci est encore de petite taille, elle reste très simple à comprendre favorisant la maintenabilité et la testabilité. Le déploiement, facile et rapide à mettre en place, permet de se concentrer sur les fonctionnalités plutôt que sur l'infrastructure. Il permet également de faire de l'intégration continue à peu de frais. Cependant, au cours de l'évolution d'un projet, cette architecture montre fréquemment ses limites.

Maintenabilité du code

Si en début de projet, la simplicité du modèle monolithique favorise grandement la compréhension du code, ceci devient de moins en moins vrai au fur et à mesure que le volume de code augmente, et ce, même avec une organisation rigoureuse de ce dernier. Un nouveau venu mettra davantage de temps à être pleinement opérationnel.
Parce qu'il n'y a pas de frontières fortes entre les modules d'une application monolithique, elles auront tendance à s'effacer, affaiblissant la modularité, complexifiant les relations de dépendance. Le code devient de moins en moins lisible et testable, la productivité et la qualité baisse et enfin la dette technique s'accumule.

Un autre effet pervers de l'architecture monolithique est de rendre une application dépendante à certaines technologies, obligatoirement compatibles entre elles. Choisir Java limitera les évolutions futures aux technologies compatible avec la JVM, C# aux technologies Microsoft, etc. Si un framework utilisé par l'application devient obsolète, la migration vers un nouveau peut nécessiter la réécriture complète de l'application avec les risques que cela comporte.

Déploiement en continu

Une application monolithique semble simple à déployer. Puisque toute l'application est destinée à s'exécuter sur un même serveur, celle-ci est très souvent contenue dans une archive déployée sur celui-ci. Une usine logicielle pourra rapidement et facilement être configurée avec les taches de déploiement adéquates. Cependant, déployer toute l'application, quelles que soient les modifications réellement effectuées, pose divers problèmes. Premièrement, le temps de déploiement s'allonge, tendant à limiter leur nombre. En effet, la taille de l'application augmentant, le container prend plus de temps pour effectuer les démarrages applicatifs.
En conséquence, le déploiement continu

devient plus complexe, ce qui engendre une augmentation des risques associés à chaque déploiement et une perte de productivité.

Scalabilité

Lorsque la charge d'une application augmente, la scalabilité de l'application devient un point critique. Une application monolithique n'ayant qu'une capacité assez limitée (car celle-ci ne se fait que dans une dimension), on ne peut avoir que des copies de l'application entière. Or les différents composants de l'application n'ont vraisemblablement pas les mêmes besoins en ressource. Certains vont faire un usage intensif du processeur, d'autres de la mémoire, d'autres des entrées / sorties. Scaler une application monolithique peut entraîner une augmentation de la consommation des ressources conséquentes de manière inutile.

Les principes de l'architecture microservices

Une nouvelle échelle

Ce qui est le plus visible de prime abord est le changement d'échelle qui s'opère avec les microservices. L'application devient un système d'informations, dont le microservice constitue l'unité élémentaire. Ces unités sont strictement indépendantes les unes des autres. Elles ont chacune leur propre base de code de donnée et pile technologique. Chaque service peut ainsi utiliser les technologies les plus adaptées à son besoin. Il devient aussi possible d'allouer les ressources souhaitées à chaque type de service. Il est également envisageable de choisir un nombre d'instances différent par service, améliorant grandement l'efficacité de la scalabilité. Un service devant supporter une forte charge possède plus d'instances d'exécutions qu'un service faiblement sollicité. Selon le même principe, les données d'un service peuvent être mises à jour, migrées, traitées sans risquer d'impacter les autres fonctionnalités du système. Dans une architecture microservices, chaque service est différent évoluant selon son propre rythme.

Évolutivité et automatisation

Lorsqu'une fonctionnalité précise du SI doit être modifiée, l'évolution en question ne porte que sur le code d'un service en particulier. Les autres services ne seront pas impactés. Ainsi, la modification du code ou des données d'un service provoque uniquement la mise à jour et le redéploiement de celui-ci car il s'exécute dans un processus isolé. Il n'y a aucune obligation d'embarquer des mises à jour d'autres fonctionnalités développées en parallèle, celles-ci impactant d'autres services dont le code est séparé. De la même façon, la présence d'une anomalie sur une fonctionnalité ne bloque pas l'évolution ou le déploiement d'une autre. Il est possible, grâce à ces caractéristiques, d'envisager de prototyper, déployer et tester une nouvelle fonctionnalité sans remettre en cause l'ensemble des services. Dans cette architecture, l'échelle de la modification est donc, non pas un ensemble de fonctionnalités assemblées dans une "application", mais la fonctionnalité, portée par le service.

Services et communication

Pour que le système fonctionne correctement, il est donc nécessaire que les services communiquent entre eux. Le vecteur de cette communication a pour unique rôle d'assurer une transmission fiable des messages. Le système suit la règle suivante : *"Smart end point, dumb pipe"*. L'intelligence du système est contenue dans les services, pas dans le vecteur. Les deux modes de communication les plus courants sont celles de type REST et bus. Lorsqu'un service dépend de messages fournis par d'autres services en amont, la communication est généralement asynchrone et une architecture microservices va tendre vers une architecture réactive, à base d'évènements déclenchés et écoutés par les services.

Tolérance et monitoring

Une des grandes forces des architectures microservices est sa robustesse et sa tolérance aux pannes. En effet, plutôt que de se protéger à tout prix de ce qui pourrait poser problème dans le système, l'approche adoptée par les microservices est l'adaptation. Un service est conçu dans l'optique que le reste du système puisse être disponible, qu'une donnée nécessaire puisse être absente ou au contraire qu'un message puisse contenir des données superflues. Le service est alors capable de fonctionner à la fois en mode nominal et en mode dégradé. De plus, ses interfaces, suffisamment souples, supporteront les écarts.
Malgré cette tolérance, le système doit être surveillé en permanence. Les pannes peuvent compromettre le fonctionnement du système global. Ainsi, il devient nécessaire d'être capable de détecter rapidement une panne pour pouvoir intervenir. Il est également important de surveiller le système et les services d'un point de vue fonctionnel. Son bon fonctionnement passe par la surveillance d'indicateurs métiers : transactions validées, en erreur, quantité de commandes, envois de messages en erreur, nombre d'inscriptions, etc. Tous ces indicateurs métiers sont les premiers révélateurs d'un problème.

LA MISE EN OEUVRE

Quels outils ?

Cloud

Les architectures microservices ont besoin d'une infrastructure souple et robuste à la fois. Un nouveau serveur doit pouvoir être provisionné en quelques minutes pour permettre à un service de monter en charge rapidement. Il doit également pouvoir être décommissionné facilement si celui-ci n'est plus utile. Le Cloud représente le candidat privilégié pour répondre à ces besoins. Des solutions de Cloud public comme AWS d'Amazon ou Azure de Microsoft permettent de réaliser ces opérations pour un coût moindre par rapport à une infrastructure privée. Ils proposent de nombreux services allant du provisionning de CDN aux solutions d'API management. Du coté des Clouds privés, OpenStack se place comme la solution permettant de construire son propre Cloud. La solution est sponsorisée par des sociétés comme Google ou encore Red Hat.

Conteneurs

Issue de travaux anciens sur le système d'exploitation Unix (les C-groups), la conteneurisation arrive aujourd'hui à maturité, le projet Docker en est l'exemple le plus significatif. Cette technique consiste à isoler l'utilisation des ressources de type processeur, mémoire et disque par application sur une même machine. L'utilisation de ces technologies dans une architecture microservices offre un avantage indéniable. Chaque service correspond à une image, livrée sur un «catalogue» d'entreprise. Celle-ci peut être construite et mise à disposition directement par l'usine logicielle. Ainsi, il n'est plus nécessaire de s'adapter à des principes de déploiement propre à chaque écosystème logiciel (JEE, Play, Vert.x, Node.js, Python, etc.). Le déploiement est uniformisé et simplifié : il s'agit de déployer un conteneur par microservices. La conteneurisation facilite également les tests d'intégration, qu'ils soient réalisés sur le poste du développeur ou sur un serveur dédié. En effet, il est beaucoup plus facile de recréer de toutes pièces des environnements à base de conteneurs que de créer une infrastructure physique supportant un monolithe et ses dépendances, à l'image de la production. La boucle de feedback, ainsi accélérée, est portée directement par l'usine logicielle et ne nécessite plus de déploiement sur un environnement identique à la production décorrélé du cycle de développement.
On touche ici à une problématique à la limite

des mondes Dev et Ops. Le choix de cette technologie, impactant toutes les couches du SI, doit se faire de manière concertée.

Supervision

Les bases de données de type Time Series permettent de stocker, avec une grande fiabilité, des métriques à intervalles réguliers. Les équipes opérationnelles ont l'habitude de travailler avec ce type de bases qui permettent de récolter les informations minimales utiles à la détection des pannes. Il s'agit en général des métriques bien connues comme le CPU, la mémoire, l'espace disque, les entrées / sorties, etc. Les développeurs y portent un intérêt croissant, en particulier pour stocker des métriques applicatives, techniques voire métiers. Graphite, très populaire ces dernières années, a atteint ses limites en termes de scalabilité. On lui préférera donc des projets plus récents, embarquant nativement la notion de clustering tels que InfluxDb, entièrement compatible avec Graphite, talonné de près par OpenTSDB. Le stockage des données est important, l'exploitation et la visualisation le sont tout autant. Aucune des bases de données nommées ci-dessus ne propose de système d'alerte. En revanche, elles s'intègrent très facilement avec des outils classiques d'exploitation, comme Nagios, Shinken, Icinga ou Sensu. Les outils de dashboarding basés sur Graphite sont nombreux et leurs évolutions récentes permettent de s'intégrer facilement avec InfluxDb. Le projet le plus populaire est Grafana.

Framework

Les services des architectures microservices sont par définition légers en code et en fonctionnalités. Il doit donc en être de même pour les frameworks utilisés au sein de ces services. Par exemple, Vert.x apporte plusieurs briques nécessaires dans la communication entre services comme un bus de messages ou un serveur de websocket. Léger, son coté polyglotte permet de choisir le bon langage pour le bon service tout en ayant la même base, ce qui est un plus en termes de cohérence pour le système. Spring Boot permet quant à lui de développer rapidement des services grâce à ses différents profils d'application prêts à l'emploi. L'objectif de Spring Boot est de pouvoir déployer facilement des applications standalone embarquant elles-mêmes le serveur choisi (tomcat ou jetty), le tout fournissant par défaut des métriques permettant un suivi de l'application en production. Enfin, sans en être un framework, node.js représente un exemple type d'outil utilisé dans une architecture microservices. Sa légèreté, sa rapidité d'exécution couplée à la rapidité du développement lié au langage Javascript en font un très bon candidat pour des microservices. Le modèle i/o non bloquant de node.js lui permet de tenir particulièrement bien la charge.

Quelle organisation ?

La mise en place d'une architecture microservices dans des équipes implique une certaine réorganisation. Une architecture microservices induit des micro-équipes, en interaction constante, connaissant parfaitement le métier et la technique des quelques services à leur charge. Les services correspondent à une fonctionnalité issue d'un besoin métier autour duquel s'organise une équipe (Feature team) qui a la maîtrise du produit.
De la multiplicité des services découle le besoin d'automatisation des déploiements. Il n'est plus envisageable lorsque les services se comptent en dizaines de les déployer manuellement. En conséquence, les équipes doivent accueillir de nouveaux processus d'automatisation dans lesquels l'outil occupe une place importante. Le mouvement DevOps est la continuité logique de la mise en place d'une architecture microservices, permettant réactivité et souplesse en production.
Enfin, du point de vue des évolutions, il est important de garder un contrôle et une vision d'ensemble, grâce, par exemple, à une cartographie de son SI. Celle-ci permet de ne pas développer des fonctionnalités identiques et de connaître le spectre technologique dont est composé son SI. Ainsi, on veille à appliquer les correctifs de sécurité sur l'ensemble de son parc technologique et ne pas s'exposer aux attaques dirigées vers des versions vulnérables.

LES LIMITES

De nouveaux problèmes ?

Les microservices proposent des solutions à des problématiques connues et héritées d'architectures orientées services déjà en place. Néanmoins, toute nouvelle solution apporte des problèmes qu'il est possible d'identifier afin de s'en prémunir.

Taille des services

Le plus grand principe des microservices concerne la taille de ces services. Celle-ci n'est pas régie par des règles précises du fait de l'absence (souhaitable) de norme ou de spécification. Plusieurs pièges sont toutefois à éviter. Il est fréquent lors du développement de services de vouloir regrouper plusieurs "petits" services en un seul "macroservice" afin de faciliter la communication entre celui-ci et le reste du SI. Ceci va regrouper la complexité des services en un seul qui va de facto, devenir plus difficilement maintenable. Un autre écueil est la tentation du "nanoservice". Transformer chaque méthode d'une application monolithique en microservice générera une quantité importante de services. La communication de ses services n'en sera que plus complexe pour un apport limité en termes de souplesse. Il est donc important de trouver un juste milieu entre des services imposants et des services trop petits. Une règle communément admise est: une personne qui découvre un service doit pouvoir en comprendre le code et le fonctionnel en une journée.

Manque d'automatisation

La philosophie des microservices est de pouvoir ajouter rapidement et simplement des services à un système. Si le temps de déploiement ou le redémarrage d'un service nécessite une intervention humaine coûteuse en temps, alors le système se confrontera à un problème. Pour l'éviter et permettre un système évolutif et résilient, il faut automatiser au maximum les déploiements d'applications mais également l'ajout de nouveaux serveurs. Pour cela, le Cloud permet de bénéficier de capacités machines quasiment illimitées et des outils, comme Docker, facilitent l'automatisation des déploiements et la réplication des services.

Système de supervision défaillant

La multiplicité des services entraîne une complexification de la supervision du système. Que la communication entre services se passe par bus de messages ou par APIs, cette supervision doit être automatisée au maximum. En effet, le nombre de services croît régulièrement au fur et à mesure de la vie du système. Il devient vite impossible pour un humain de suivre tous les services manuellement. Pour pallier ce problème, chaque service doit fournir ses propres métriques au système de monitoring. Elles doivent être techniques mais également business afin de pouvoir mesurer en temps réel l'apport métier d'un service. Des règles d'alertes automatiques doivent être mises en place afin de savoir lorsqu'un service commence à ne plus fonctionner correctement. L'idéal est de disposer d'un système autonome capable de relancer un serveur défaillant de manière automatique, toujours sans intervention humaine.
Un autre problème lié à cette multitude de services est l'exploitation des fichiers de logs. Pour un service unique déployé sur plusieurs

serveurs, celle-ci peut s'avérer être fastidieuse. Une bonne solution consiste à centraliser les logs des services au même endroit, dans le système lui-même avec des solutions comme Flume ou ELK ou bien grâce à des services SaaS comme papertrail ou logmatic.

Profusion des technologies

Même si les microservices permettent l'utilisation de différents langages, il est important de veiller à ne pas multiplier les langages inutilement. Cette profusion de technique rendra le passage d'un service à un autre compliqué pour les développeurs et complexifiera le système inutilement.
Le même constat peut être fait avec les bases de données. Il est possible, et même conseillé, de choisir un type de base de données correspondant à chaque service. Il faut toutefois éviter de multiplier le nombre de bases différentes. Garder un type de base relationnelle et un type de base NoSQL peut être un bon compromis.

Sur le chemin des microservices

Avec l'émergence des architectures microservices, beaucoup de projets se sont lancés dans l'aventure sans toujours rencontrer le succès escompté. En effet, comme nous l'avons vu précédemment, les microservices répondent très bien aux problématiques de charge et de scalabilité mais peuvent compliquer le système dans son ensemble.
Fort de ce constat, Marin Fowler, un des précurseurs de ce type d'architecture propose deux concepts permettant un cheminement vers les microservices : Monolith First et Sacrificial Architecture.

Monolith First

Deux observations s'imposent aux yeux de Martin Folwer :

- La plupart des succès de ces projets viennent d'un monolithe devenu trop gros et qui a éclaté en microservices;
- La plupart des projets ayant démarré directement en microservices ont connu de sérieux problèmes en cours de route.

Plusieurs raisons expliquent ces observations. Les microservices sont efficaces si leurs domaines fonctionnels sont bien isolés. Au début d'un projet, il est parfois complexe de délimiter précisément chaque service. Ceci va entraîner plusieurs phases de refactoring qu'il est bien plus facile d'entreprendre dans un monolithe que dans plusieurs services communiquant via des APIs ou un bus.
Lors de l'initialisation d'un projet, une architecture microservices sera plus longue à mettre en place. Pour une startup ou un projet sans visibilité forte du marché potentiel, ce coût de démarrage peut être un problème quand le but est d'arriver rapidement à un produit fini.

Sacrificial Architecture

Le deuxième concept proposé menant sur le chemin des microservices est la Sacrificial Architecture. Ce concept part du constat suivant : le code écrit aujourd'hui pour un projet ne sera plus adapté dans deux ans.
En effet, il est très difficile voire impossible de savoir comment va évoluer un projet. Le succès sera peut être immédiat nécessitant une évolution rapide du code. A contrario, le projet ne rencontrera peut être pas son public et un investissement trop lourd dès le départ aura des conséquences néfastes pour l'entreprise. Au lieu de prévoir l'évolution à long terme de l'architecture du projet, il est préférable d'initier une architecture jetable permettant de sortir rapidement un produit. Ensuite seulement, en fonction de l'évolution du projet, il sera possible de poser une architecture plus solide, absorbant mieux la charge et permettant une évolutivité plus simple. Le rapprochement avec le concept de monotith first est évident. Suivant le projet, il peut être préférable de partir sur une architecture simple et rapidement opérationnelle afin de sortir le plus tôt possible un produit puis d'adapter (voire même de jeter) l'architecture pour évoluer vers un système plus maintenable et robuste.

CONCLUSION

Les architectures orientées services ont piloté la mise en place des SI de ces 10 dernières années. Le constat majoritaire est une capacité à évoluer qui diminue avec le temps du fait de la difficulté d'isoler les évolutions dans le système. En effet, les applications dont il est composé sont en majorité des monolithes qui réunissent un assemblage hétérogène de fonctionnalités de plus en plus nombreuses avec le temps. À ces problématiques, les microservices proposent une réponse simple : diminuer l'échelle de l'architecture. Cette échelle traduit une isolation nécessaire des fonctionnalités à différents niveaux : code, données, déploiement et exécution. Cette réduction de la granularité a des revers et exige un certain investissement au niveau technologique, méthodologique (Craftsmanship, DevOps) mais aussi et surtout au niveau organisationnel (Agilité, Feature Teams). L'émergence de nouveaux outils facilite également le passage aux microservices. Le Cloud ou les technologies de conteneurisation sont les instruments idéaux pour leur mise en place.
Toutefois, les microservices ne constituent pas non plus une recette universelle. Un système peu complexe sans problématique de charge ou dont le rythme d'évolution est faible ne bénéficiera pas de l'approche microservices. Ils ne garantissent pas une simplification du système global mais permettent la simplification de son évolution. Certaines briques du système ne sont en effet pas constamment bousculées par des besoins d'évolution. On peut donc tout à fait envisager de faire cohabiter le cœur monolithique du système, plus constant, avec en périphérie des briques conçues en microservices pour les besoins les plus mouvants.

⊞