Notes sur le module : *Cloud Natif.*

Armel Pitelet

27 février 2022

Table des matières

Remarque : *Sur la dénomination Cloud.*
Terme marketing.

Remarque : *Sur internet.*
Ensemble de réseaux qui se base sur des protocoles de communication (http, bas débit, ...). L'ensemble de ces protocoles fait internet qui est un système de communication (pas que le web). Le but premier est de transmettre de l'information via des protocole (irc pour le chat par exemple, smtp pour les mail,...).

Remarque : *Sur la notion d'IP et l'histoire d'internet.*
Internet protocol. Le nom de version fait référence au nombre d'octet (ipv4 $\rightarrow$ 4 octets par adresse), ce qui fait un nombre important (4 milliards) mais limité d'ip. Sans adresse ip, pas de communication via internet(au sens large donc plus que juste le web). L'ipv6 est apparue pour balancer ce problème (6 octets par adresse).Début des 90 apparition du web : déblocage de l'accessibilité à l'information. Puis la partie mercantile apprait rapidemment. La notion de qualité logicielle apprait alors (aka un service qui fonctionne) et avec les fournisseurs de serveur (accès à internet)/service aka ceux qui possèdent l'infrastructure et les points de connexion. Puis la fibre succède au cable cuivre. Les technos (et marchand associés) sont de plus en plus dépendant de cette connexion.

Note :
NAT $\rightarrow$ Network Address Translation, PAT $\rightarrow$ Port Address Translation.

Remarque : *Sur le* ***DNS****.*
Domain Name Service. Service distribué qui alias un ip avec un nom lisible humainement (et fait l'inverse également).

Remarque : *Sur la notion de Cloud.*
Il s'agit en fait de l'ébergement massif et centralisé de serveur destinée à la location. Contrairement à l'ancienne manière de faire le serveur n'est pas monté en physique chez un client mais est simplement loué prêt à l'emploi. Le Cloud fournit via API des ressources utilisables pour le déploiement d'une infrastructure. En générale les services Cloud sont des acronymes qui finissent en *aas* (as a Service). Le Cloud se base sur la virtualisation : une machine physique est capable d'émuler une/plusieurs machines virtuelle(s).

Remarque : *Iaas.*
Infrastructure as a Service. C'est à dire le démarage d'une machine virtuelle par un Hyperviser (la machine physique qui fait tourner la machine virtuelle). On alloue ensuite de la RAM, du CPU,

de la mémoire à ces machines virtuelles (dimensionnement du serveur).

Remarque : *Sur la virtualisation.*
Le gros interêt est de pouvoir déplacer des VM de serveur en serveur sans couper le service de VM en question. Autre point d'interêt : deux VM sont isolés l'une de l'autre (en terme de système, pas de réseaux).

Remarque : *Faas.*
Function as a service. Cette fois ci au lieu de déployer une application entière on ne déploie qu'une partie, une fonctionnalité (comme une base de donnée par exemple ou une fonction de traitement d'image).L'application se retrouve être un ensemble de fonctionnalité distribué à travers le Cloud. Cela évite d'avoir une machine (serveur) mobilisé en permanence.

Remarque : *DNSaaS.*
Principe du DNS mais avec la possibilité de rajouter des noms de domaines à la volé. Cela revient à mettre à disposition un serveur dns de manière dynamique.

Remarque : *CaaS.*
Container as a Service. Kubernetes par exemple. Docker est la technologie de containerisation et Kubernetes est le service de gestion de ces container.

Remarque : *SaaS.*
Software as a Service. Permet d'éviter le déploiement sur des machines d'un Software donné. Par exemple : gmail, github,...

Remarque : *PaaS.*
Platform as a Service. Contrairement à l'Iaas ou l'on à tt la machine virtuelle (avec installation et configuration à faire), le Paas met à disposition une plateforme déjà configurée, sécurisée, mise à jour, mise à l'échelle... pour faire tourner du code directement.

Remarque : *NaaS.*
Network as a Service. Permet de créer des réseaux virtuelle (pas forcément accessible de l'extérieur mais plus des réseaux internes).

Au finale le principe du Cloud est d'automatisé tt ces services (depuis une interface). Il existe aussi du DBaaS, du VPNaaS, Maas (Metric as a Service pour des stats.)

Remarque : *Sur les Cloud publiques.*
Ce sont des services de Cloud ouverts, qui centralise les services Cloud au sein d'une seule entreprise (AWS de Amazon, Azure de Microsoft, Ovh en france, Scaleway, Clever Cloud, Alibaba Cloud en Chine). Selon les localisation les loies sont différentes ! Par exemple le Cloud Act américain permet au gouv US d'accéder aux données hébergées par une entreprise Américaine quel que soit l'endroit physique ou sont hébergées les données.

Remarque : *Sur le Cloud Native.*
Une application est dite Cloud Native si elle est concue dans l'idée de tourner dans le Cloud.

Remarque : *Private Cloud.*
Le Cloud n'est pas partagée, il est centralisé (infra et logiciel) chez le client. VMWare, Openstack (solution python open source), Proxmox (plus léger que le reste pour créer des VM à la de-

mande/volée), Kubernetes.

Remarque : *Hybrid Cloud.*
L'idée est de séparer les services dans différents Cloud (provider de Cloud). Au lieu de tt centralisé, tt est décentralisé chez plusieurs providers.

Remarque : *Le MulciCloud.*
L'application est partagée sur plusieurs Cloud différents (chez différents Cloud providers).

Remarque : *Sue les* ***12 factors****.*
Un ensemble de bonne pratique qui permettent de définir une application comme *Cloud Native Complient*. Voir https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/definition et ci dessous :

1. Code Base : Il faut un code source partageable et versionné ! (et disponible en remote). L'idée est d'avoir une démarche qualitative. Idéalement il faut même avoir une base de code par microservice (au moins un repos par application). De plus le code ne doit pas être dépendant de ou il doit être déployé (debug, release, local, ...).
2. Dependencies : Il faut garder à jour la liste des dépendances (via package manager : quelle lib, quelle version), les déclarer explicitement et les isoler (dans un fichier de descriptions des packages : .yaml, .json). Il ne faut pas garder le code sources des dépendances au même endroit que le code source de l'application. De plus il faut considérer une dépendance comme du *readonly*. Le fait de lister les dépendance permet aussi de les auditer (pour les licences par exemple).
3. Configurations : tt ce qui peut varier entre différents déploiement ne doit pas être hardcodé dans l'application (Pour des raisons de déploiement mais aussi de sécurité). Ex : adresse i, username. La meilleur façon de passer de la configuration est d'utiliser les variables d'environnement. Il est aussi possible d'utiliser du yaml ou du json mais ce n'est pas l'idéal car pas facilement éditable. On peut également mettre la configuration dans un service à part dédié à la gestion des configurations (Consul, etcd). Attention à ne pas passer des *secret* dans la configuration (comme des password). **Vault** permet de centraliser et stocker les *secret*. La config se base en générale sur un système de clef valeur.
4. Backing Service : Tt ce dont la base de code à besoin peut être considérer comme un système externe (I/O, base de donnée,...) et doit donc être isolé. L'appli est un service, la base est un service, l'authentification est un service.
5. Build, Release, Run : Build est la phase de transformation du code source en code exécutable (avec dll et ressources embarquées si besoin). Attentiojn ce n'est pas forcément une compilation, cela peut mener à une image docker par exemple. Release : mise à disposition du binaire pour qu'il soit utilisable par qq un. Run : mettre à disposition la release et la faire tourner (c'est le déploiement) dans un environnement particulier en utilisant une configurations donnée (attention sur le site en lien ils parlent d'une Relase comme d'un build accompagnées d'une config). La limite entre Dev et Ops se situe entre Release et Run.
6. Process : Plusieurs instances d'une application doivent s'éxécuter en différents process indépendant les uns des autres (par instance). Cette isolation peut se faire en lancant chaque instance sur une VM différente (potentiellement dans la même *Hypervisor*) ou bien via des containers (docker orchestré par kubernetes). L'idée est de sécuriser sa machine par rapport à une possible défaillance d'une des instances de l'application (ou une CVE exploitable selon les technologies utilisées dans l'application).
7. Port Binding : est le principe de mapper un port d'écoute sur un autre (Ex : docker écout le 80 mais il est mappé par la machine sur le 5000 car les 1 à 2024 sont réservé à root). On passe par

du PAT (Port Adress Translation) pour rediriger le port déclarer vers un autre. L'intéret est surtout lié à la sécurité. Cela permet également d'assurer l'unicité des ports mappés dans les dockers (car ces dernier peuvent écouter sur le même port mais de deux container différent et la machine host doit tt de même pouvoir les distinguer. On map donc chacun des ports [identique] sur un port différent via PAT). En clair il faut que la configuration puisse donner la possibilité de mapper un port.

8. Concurrency : Il faut pouvoir déployer l'application plusieurs fois sans que les différentes instances ne bloque l'accès aux données des autres (ex d'un fichier partagé qui ne doit pas être bloqué par une des instance [sauf au moment de l'utilisation].) Dans la même veine il est possible d'utiliser un **Redis** pour gérer l'état d'une session de connexion ou bien d'utiliser des système d'authentification sans état (au lieu de devoir stocker l'état coté serveur, qui peut changer si l'état du serveur change). Voir également **jwt** pour gérer de l'authentification. Au final on doit pouvoir démarer autant de fois que l'on souhaite l'application en parallele sans qu'une instance ne *lock* une autre.
9. Disposability : Le code est jetable. Il ne faut pas hésiter à faire, refaire, jeter, recommencer, patcher pour ensuite nettoyer. De plus le process correspondant doit démarer vite, à volonté et s'arréter correctement sans problème.
10. Dev/Prod Parity : on se doit d'être au plus proche de la production lors du développement. Ex : si en prod on utilise du MySql pour la base en v5, on doit utiliser la même chose en local lors du développement. Pour cela **docker compose** peut aider : cela permet de monter plusieurs containers avec des spec spécifiques (on passe par un fichier docker-compose.yml). Il faut faire attention au Mock dans ces cas là par exemple. Il faut aussi faire attention aux dépendances utilisées lors du dev.
11. Logging : il faut des logs sur différents niveaux. Voir remarque en dessous.
12. Admin Processes : Si une opération d'administration est nécessaire (migrations, purge, bash d'insertion) il faut développer un process spécifique qui va s'occuper de sa et le séparer de l'application en elle même. Il faut donc un/des services dédiés à ce genre de chose. Les opérations de maintenances doivent également être traité de cette manière.

Remarque : *Sur le versionning.*

Pour accompagner les bonnes pratiques sur le Code Base et les Dependancies : voir le **semantic versionning** (SEMVER). Par exemple le systeme version = Major.Minor.Patch. Une mise à jour de patch est juste un update de type fix qui n'affecte pas le fonctionnement de l'application. Si l'on fait une update Du Minor (en ajoutant une fonctionnalité) il faut garder la rétrocompatibilité avec les mineurs précédentes (engament dans le SEMVER). Cette rétrocompatibilité n'est pas obligatoire lors d'un bump de la Major.

Note :

Une **CVE** est une faille de sécurité connue et référencée (exemple : log4j)

Remarque : *Sur les secret.*

Il faut absoluement séparer les *secret* du reste des données de configurations. En théorie (selon la loi) les mots de passe doivent être hasshé.

Remarque : *Sur le hash.*

Voir **bcrypt** et **sha256** qui sont des algo d'encodage/décodage et de hash

Remarque : *Sur les SPOF.*

SPOF : Single Point Of Failure : quand une seul erreur fait s'écrouler tt le système. A éviter le

plus possible.

Remarque : *Un peu plus sur la notion de Concurrency et de Scaling.*
On peut scale deux deux manières. Verticalement, cela implique de simplement augmenter la capacité en charge d'un process (ou une machine). Horizontalement, cela implique de multiplier le nombre de process (ou de machine). Le problème du scale verticale est que l'on ne peut pas faire d'application concurrente, de plus le scale vers le haut à forcément une limite (par exemple on sera forcément plus limité avec une connexion par rapport à plusieurs en parallele sur différente machine). Le choix entre l'un et l'autre peut aussi faire sur des critères financiers. La concurrency est le principe de pouvoir faire tourner plusieurs instances simultannée du même programme. Pour cela il faut parfois déporter certains service en dehors de l'application mais rend les applications plus facilement scalable (voir localisée à plusieurs endroits).

Remarque : *Sur le* ***SLA***.
Service Level Argeement : taux d'occupation annuelle des machines. En gros 100% est équivalent à un serveur qui fournit du serveur tt le temps.

Remarque : *Sur les niveaux de log.*

- Debbug : précis et complet, pour le dev en dev.
- Information : comportement global du programme (un suer demande une info par exemple).
- Warning : problème ne provoquant pas d'erreur du programme mais signalant un disfonctionnement (ex : userName not found).
- Error : il s'est passé qq chosd e problématique sans pour autant crasher l'application. (ex : cannot send Email pour une appli de mail). L'application ets toujours fonctionnel mais dégradée
- Fatal error : crash totale de l'appli (ex : DB Connection fail). Doit faire une alerte, ouvrir un ticket (déclencher l'alarme générale).

Note :
Au niveau des log il faut commencer par les plus graves puis rajouter les autres au fur et à mesure.

Remarque : *Sur a notion de* ***Daemon***.
Il s'agit d'un process qui se lance puis continue de tourner. C'est l'inverse du script à usage unique.

Note :
Pour faire du Cloud gratuitement voir **OpenStack**.

Remarque : *Pour un peu plus d'infos sur un modèle en réseau.*
Voir : **OSI** (un peu daté mais toujours pertinent).

Remarque : *Sur les différents types d'API.*
En vrac SOAP, XMLRPC, JSONRPC, GRAPHQL, REST, ... qui sont des modèles d'architecture d'API.

Remarque : *Sur une alternative au CRUD classique.*
Voir le **CQRS** : les read et write sont différents (avec un modele de données différent pour chaque).

Note :

Pour faire du microservice il faut à tt prix être bien outillé. L'automatisation est essentielle (déploiement, mise à jour, test). De plus il faut éviter le monolityque distribué également.

Remarque : *Sur les microservices.*

Un micro Service = un process cad gère un métier (dans l'idéal). Dans un application en microservice cela permet de réutiliser des microservices déjà implémenter. Le Trick est de les rendre suffisemment générique. Faire du micro service c'est passer par de l'itération continue. Pour des outils **kafka** (payant) ou bien **RabittMq** (gratuit). Attention : il n'y a aucune raison de faire du microservices si ces derniers ne sont pas indépendants les uns des autres (le plus possible en tout cas).

Définition : **Sur le signe SGBD**.

Système de Gestion de Base de Donnée.

Note :

NoSql signifie Not Only Sql.

Remarque : *Sur la notion de Canary Deployment.*

Il s'agit en fait de ne déployer une version d'une appli que chez un certains pourcentages d'utilisateur. Les micro services permettent de faire du canary déployment sur de nombreuse feature.

Remarque : *Sur la relation entre base de donnée et containrisation.*

Il n'est en générale par recommandé d'avoir une base de donnée dans un microservice. La base à tendance à prendre tt la place que l'on lui donne, si l'on veut le containeriser il vaut mieux le séparer dans son propre container qui doit être un container spécialisé dans la base de donnée (pour des raisons de performances, sur les IOPS (Input Output Per Second)). Le mieux au final est d'avoir une base de donnée en externe, à laquel les microservices vont venir se connecter en récupérant simplement une configuration.