Concepts & Technologies

Les concepts et technologies

Timing : 4h

Overview technologique

Comment un cloud tourne

Que fait-on tourner dans un cloud.

Concepts souvent associé au cloud:

Haute dispo, Scaling, Failover, ...

On va faire quoi

Comme on l’a vu, le cloud peut se résumer à du hosting externe accessible sur un réseau et à plusieurs types de services.

Ok … mais finalement ca ce n’est pas le plus important.

Ce qui nous intéresse c’est ce que l’on peut faire dans le cloud avec nos applicatifs et comment cette technologie nous impacte.

On va faire quoi

Nous allons donc voir deux aspects distincts :

- Comment marche un cloud
- Quel est l'impact du cloud sur mon application

La première partie sera plus d'un point de vue fournisseur de cloud.

La seconde concerne plus les société se basant sur du cloud pour réaliser leurs solutions.

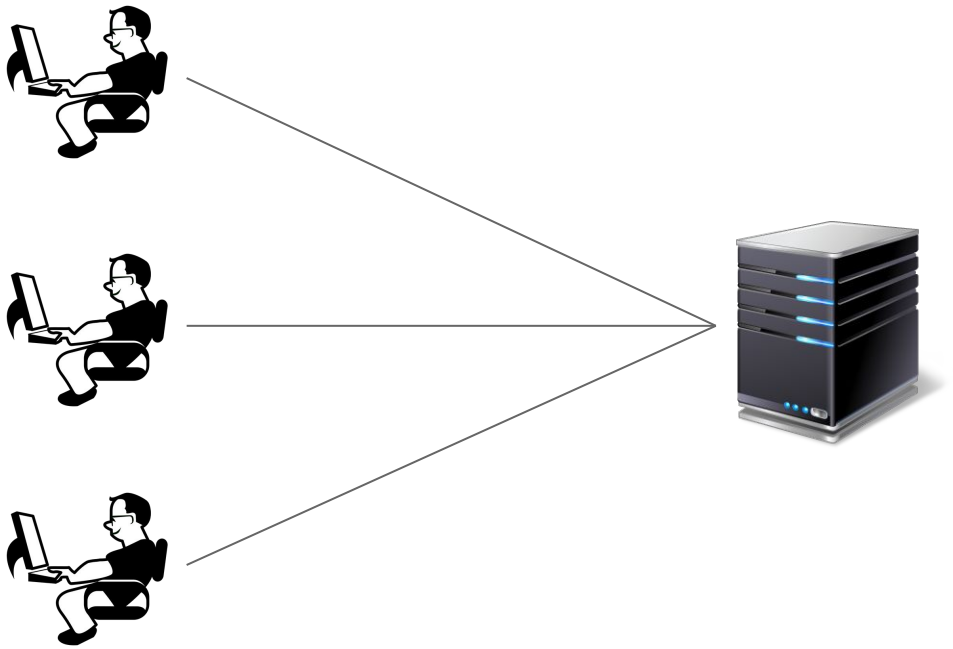
Comment marche un cloud

Rappels sur la virtualisation

EPSICloud !

Ouvrons nos (notre) serveurs à la location et louons le comme un PaaS.

Un user c’est pas génial, on va pas faire beaucoup d’argent



Euh c’est pas très safe ca …

EPSICloud !

Bon va falloir investir en machine pour se faire de l’argent.

On va investir et faire un plan d’investissement pour rentabiliser notre achat d’actif.

On est pas très effectif comme ca …

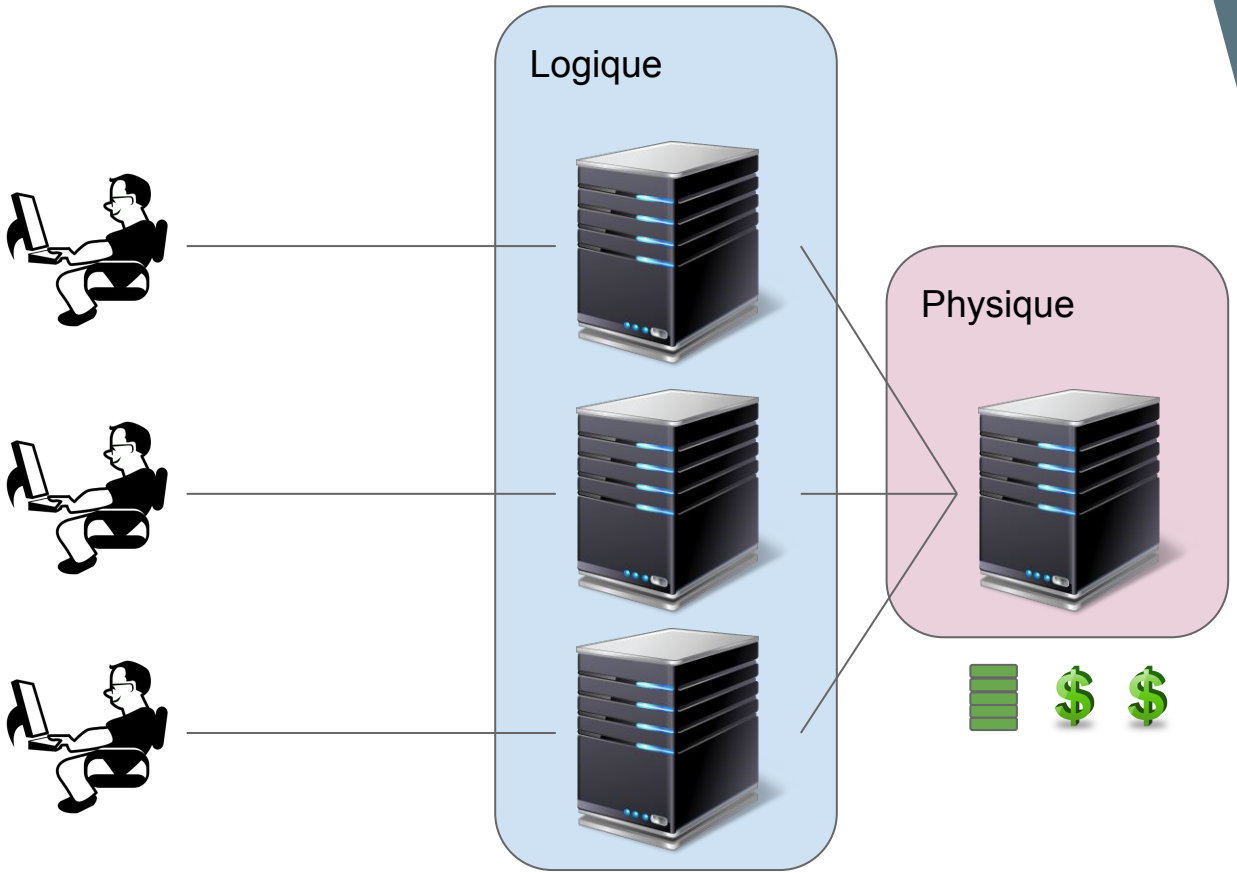
EPSICloud

Constat

- Augmentation du nombre de serveur
- Sous exploitation des ressources
- Coût élevés (maintenance, mises à niveaux, …)

La solution à nos problème :

Virtualisation


Logique
Physique

La virtualisation

En informatique, on essaye toujours de distinguer deux mondes : Le monde du matériel et le monde du logiciel. Le premier fournit les équipements physiques, le second fournit les données et les applications.

La virtualisation a été inventée dans le but de s'affranchir au maximum de la couche matérielle de l'informatique.

La virtualisation

Le but de la virtualisation est donc de permettre à plusieurs OS de s'exécuter de manière isolés et indépendantes sur une même machine.

L’objectif étant de profiter au maximum des capacités des serveurs et de pouvoir adapter chaque environnement aux besoins de celui-ci.

Approche de base

- 4 ressource :
  - CPU
  - RAM
  - DISQUES
  - RESEAU
- Allocation des ressources : statique ou dynamique ?
- Les serveur virtuels ne voient que les ressources qui leurs sont allouées et sont donc isolés les uns des autres
- Un utilisateur ne verra pas que l'OS n'est pas natif

La virtualisation c’est pas simple

Techniquement

- Les os ne sont pas conçu pour être virtualisé
- Les os sont conçu pour communiquer avec le matériel et pas avec d’autres OS

Problème évident : comment l'OS virtualisé va-t-il pouvoir communiquer au plus bas niveau avec le matériel si il existe une couche intermédiaire ?

Les types de virtualisation

Au fil du temps, les concepts de virtualisations ce sont étendus pour aller vers 3 objectifs principaux :

- L’abstraction
- La replication
- L’isolation

Pour répondre à ces problèmes 3 types de solutions

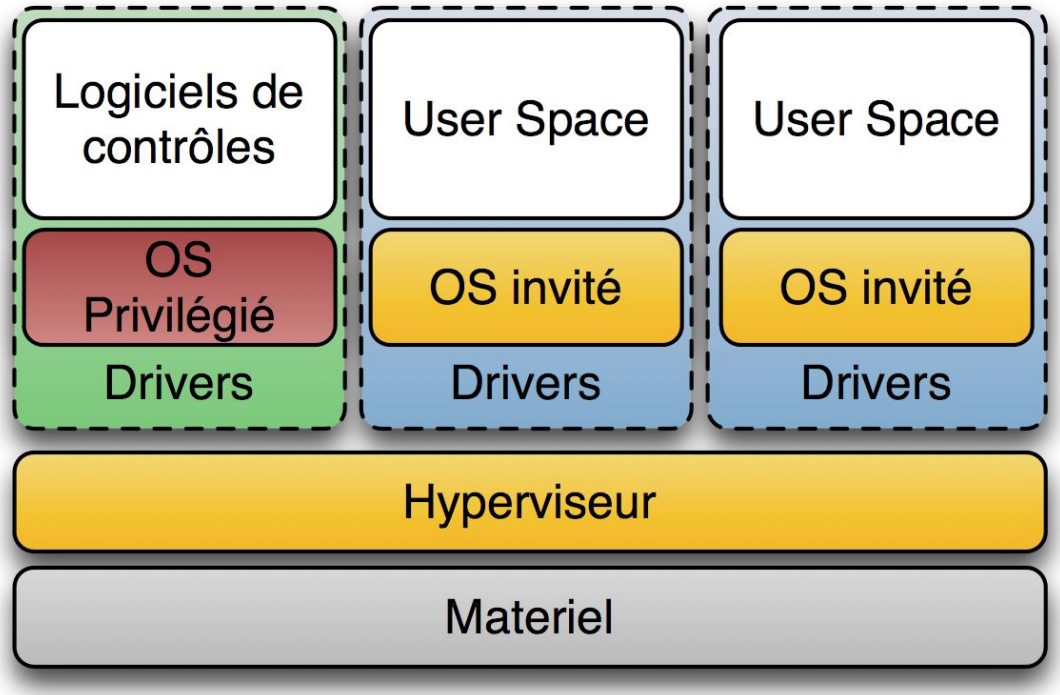
- Les isolateurs
- Les hyperviseurs (type 1)
- Les hyperviseurs (type 2)

Hyperviseurs (type 1)

Les hyperviseurs (type 1)

Un hyperviseur de type 1 est comme un noyau système très léger et optimisé pour gérer les accès des noyaux d'OS invités à l'architecture matérielle sous-jacente.

Si les OS invités fonctionnent en ayant conscience d'être virtualisés et sont optimisés pour ce fait, on parle alors de para-virtualisation



Les hyperviseurs (type 1)

Les hyperviseurs (type 1)

Exemples :

- Xen Server (Citrix)
- vSphere (VmWare)
- Hyper-V Server (Microsoft)
- Parallels Server (Parallels)

Les hyperviseurs (type 1)

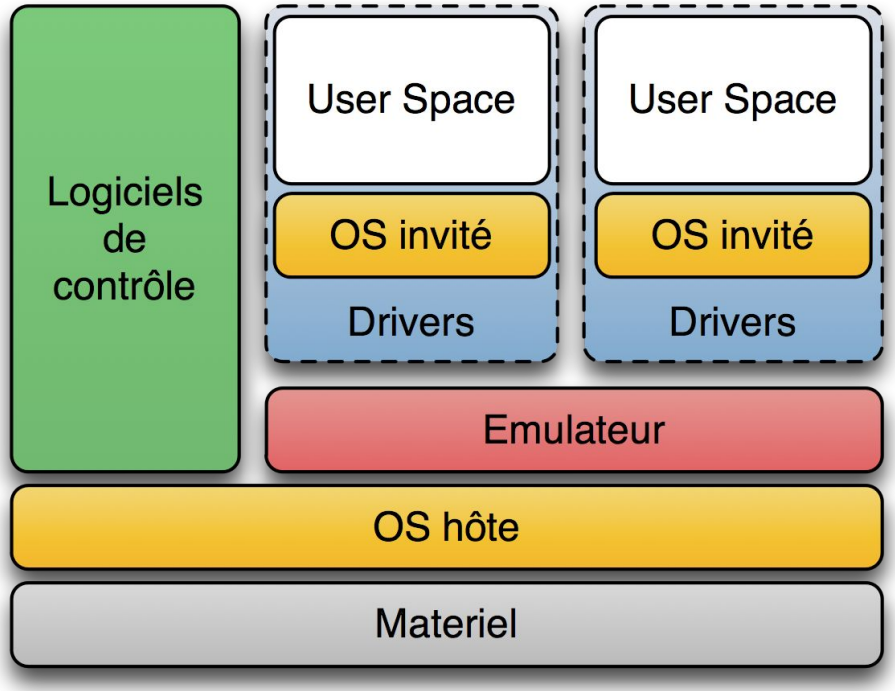
Actuellement l’hyperviseur type 1 est la méthode de virtualisation d'infrastructure la plus performante mais elle a pour inconvénient d’être contraignante et onéreuse, bien que permettant plus de flexibilité dans le cas de la virtualisation d'un centre de traitement de données.

Hyperviseurs (type 2)

Les hyperviseurs (type 2)

Un hyperviseur de type 2 est un logiciel qui tourne sur l'OS hôte. Ce logiciel permet de lancer un ou plusieurs OS invités.

La machine virtualise ou/et émule le matériel pour les OS invités, ces derniers croient dialoguer directement avec ce matériel. Cette solution est très comparable à un émulateur, et parfois même confondue. Cependant le processeur, la RAM ainsi que la mémoire de stockage (via un fichier) sont directement accessibles aux machines virtuelles.



Les hyperviseurs (type 2)

Les hyperviseurs (type 2)

Exemples :

- Qemu (open-source)
- VMWare fusion (VmWare)
- Parallels Desktop (Parallels)
- VirtualBox (Oracle)
- KVM (open-source)
- Virtual Server (Microsoft)

Les hyperviseurs (type 2)

Cette solution isole bien les OS invités, mais elle a un coût en performance. Ce coût peut être très élevé si le processeur doit être émulé, comme cela est le cas dans l'émulation.

En échange cette solution permet de faire cohabiter plusieurs OS hétérogènes sur une même machine grâce à une isolation complète (Vous pouvez mettre à dispo un Microsoft à côté de vos Linux sur la même machine).

Les isolateurs

Les isolateurs

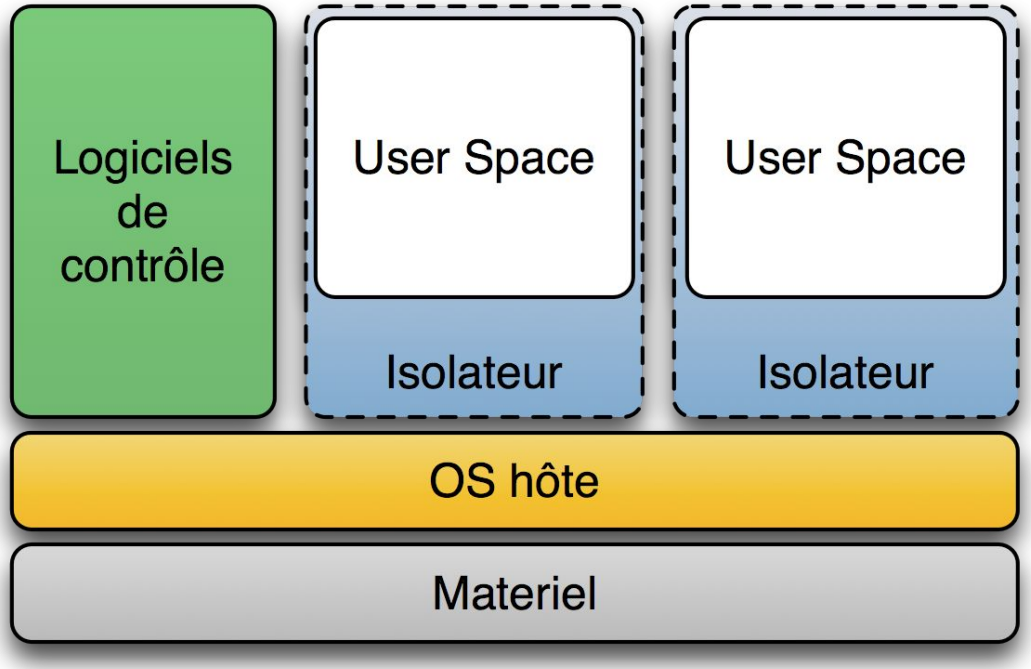
Un isolateur est un logiciel permettant d'isoler l'exécution des applications dans ce qui est appelé des contextes, ou bien zones d'exécution.

L'isolateur permet ainsi de faire tourner plusieurs fois la même application dans un mode multi-instance (plusieurs instances d’exécution) même si elle n’était pas conçue pour ça.

Les isolateurs

Les isolateurs ne sont pas de la virtualisation d’OS mais elles ont un même but : faire tourner des applicatifs de manière isolé et indépendantes sans que ceux-ci sache qu’ils sont exécutés dans une environnement virtualisé.

Le but final étant l'application et pas de faire juste tourner des OS les un à côté des autres il s’agit d’une approche de virtualisation viable.



Les isolateurs

Les isolateurs

Exemples :

- VServer
- chroot
- BSD Jail
- OpenVZ
- LXC

Comme on peut le voir il s'agit de technologies basés sur l'écosystème Unix (Linux, BSD, Solaris, …)

Oubliez les isolateurs dans l'univers Microsoft

Les isolateurs

Cette solution est très performante, du fait du peu d'overhead (temps passé par un système à ne rien faire d'autre que se gérer), mais les environnements virtualisés ne sont pas complètement isolés.

La performance est donc au rendez-vous, cependant on ne peut pas vraiment parler de virtualisation de systèmes d’exploitation, on reste à un niveau applicatif.

Les hyperviseurs vs les isolateurs

Les constructeurs de processeurs comme AMD et Intel ont intégré depuis plusieurs années des fonctions de virtualisation directement dans l'architecture de leurs processeurs.

Cette technologie permet au processeur de faire fonctionner sur une même puce plusieurs systèmes d'exploitation en parallèle. Les VMs peuvent utiliser le matériel directement, sans que l'hyperviseur leur en donne l'autorisation. Les gains de performances sont très importants avec un processeur gérant la virtualisation.

Automation

Cloud = Virtu ?

La virtualisation fait partie des technologies coeurs qui ont permis de mettre en place les premiers services cloud, mais ce qui est déterminant pour fournir un service cloud c’est l’automatisation.

Le principe des services cloud est d’être une ressource à la demande.

Automatisation de configuration

La configuration de nouvelles instances de ressources à mettre à disposition à la demande.

La variété des types de ressources (IaaS, PaaS, SaaS).

La gestion de ses propres ressources.

Pour que ces trois principes vivent ensemble il est nécessaire de mettre en place des outils d’automatisation qui depuis se sont généralisés.

Gestion de configuration

La configuration des serveurs est une problématique qui existe depuis très longtemps. Mais dans le cas d’un cloud il s’agit d’un exercice toujours plus périlleux.

De nouveaux outils permettent de définir les configurations cible pour des ressources et d’automatiser sa mise en place.

Gestion de configuration

Le principe des outils de gestion de configuration est de définir la configuration cible d'une resource.

Une configuration pour un service PaaS :

- Debian 8
- Java 7_56
- Server JEE Wildfly
- Redirection du trafic port 80 -> Server d'app

Cette configuration sera installé automatiquement

Gestion de configuration

Ce genre d'outils peuvent intervenir à plusieurs niveaux selon leurs but.

Exemples :

- Ganeti (VM, OS, Matériel, …)
- Ansible (Configuration logicielles, applications, …)
- Salt (Déployement, …)
- Chef/Puppet (Configuration applicative, …)

Applicative impact

Software as a Service

Dans la suite logique de leurs utilisations, nos applications deviennent et sont déployées comme des services.

Que ce soit des soft entiers (SaaS) ou seulement des briques applicatives, le cloud nous pousse à développer notre application comme un service fournit à d’autres.

Where is my code running ?

Lorsqu'un applicatif est déployé et utilisé sur le cloud, on peut savoir ou il se trouve.

De même plus on s'abstrait du système moins on est capable de maîtriser sa résilience.

Un SLA de 99.9 % ne veux pas dire que votre machine ne crashera jamais …

Let it crash !

Lorsque l’on développe dans le cloud il est très important de prendre en compte ce principe.

Le support de fonctionnement de votre applicatif est éphémère.

Plutôt que de se battre contre cet état de fait, créons des applications qui prennent en compte ce principe.

Quelques concepts

- Stateless | Stateful
- Non Blocking programmation
- High-Availability
- Scalling
- Caching
- Distributed programmation
- Backpressure
- ...
- Twelve Factor

Quelques concepts

- Stateless | Stateful
- Non Blocking programmation
- High-Availability
- Scalling
- Caching
- Distributed programmation
- Backpressure
- …
- Twelve Factor

Petit commentaire

Tous les concepts sont également valables sur des architecture non cloud.

On les prend juste moins souvent en compte.

Stateless / Stateful

Stateless / Stateful

Stateless : Un serveur stateless est un serveur qui traite chaque requête comme une transaction indépendante et sans relation avec une quelconque précédente requête.

Stateful : Un serveur stateful est un serveur qui traite des requêtes en gardants des informations qui pourront être exploitées pour de futurs appels.

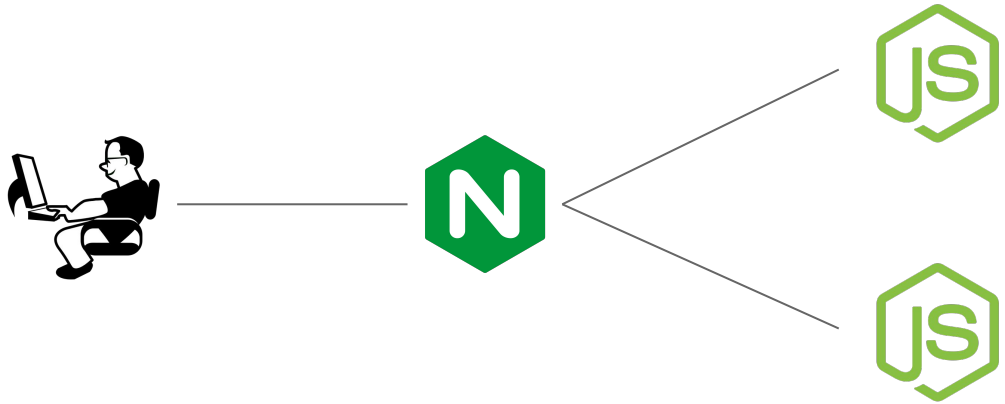
L’exemple le plus classique d’état coté serveur est la session utilisateur.

A votre tour.

Quels types de services doit-on préférer et pourquoi ?

Une autre question

Avant même de parler de cloud, comment peut-on garder une gestion d'état quand nous avons deux serveurs en parallèle ?



Stateful is hard !

La gestion d’état est très difficile dès que plusieurs acteurs entre en jeux.

Si on rajoute la contrainte cloud d’un serveur en lequel je n’ai pas confiance, l’équation devient très difficile.

L’approche la plus simple est de limiter la gestion d’ état aux système conçus pour (Database, Cache, …)

Programmation non bloquante

Non Blocking programmation

La programmation non bloquante est devenue très commune de nos jours.

Elle est particulièrement utilisé dans les applications cloud mais pas que.

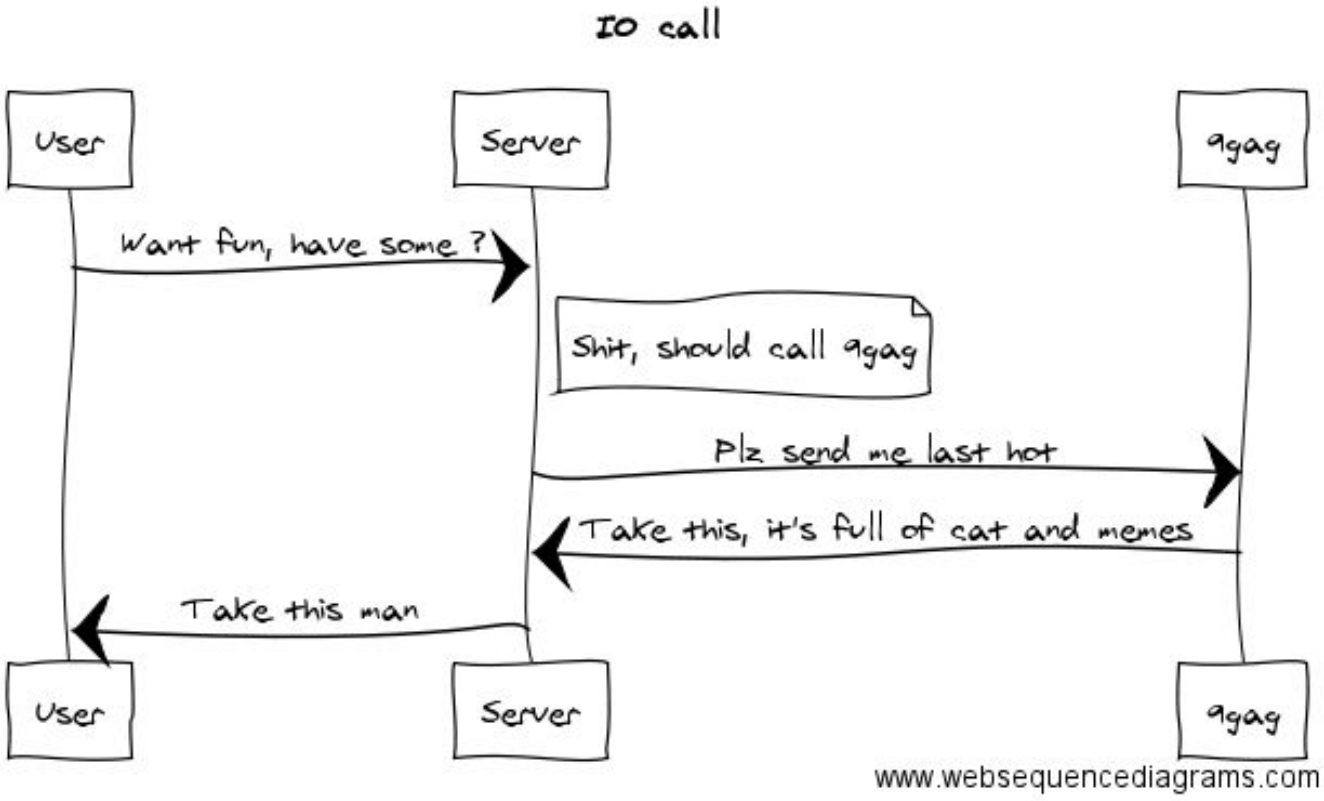
Il s’agit ici de minimiser le blocage de resources (CPU, Memoire, Network, …) pour fournir une efficacité maximale a notre code.

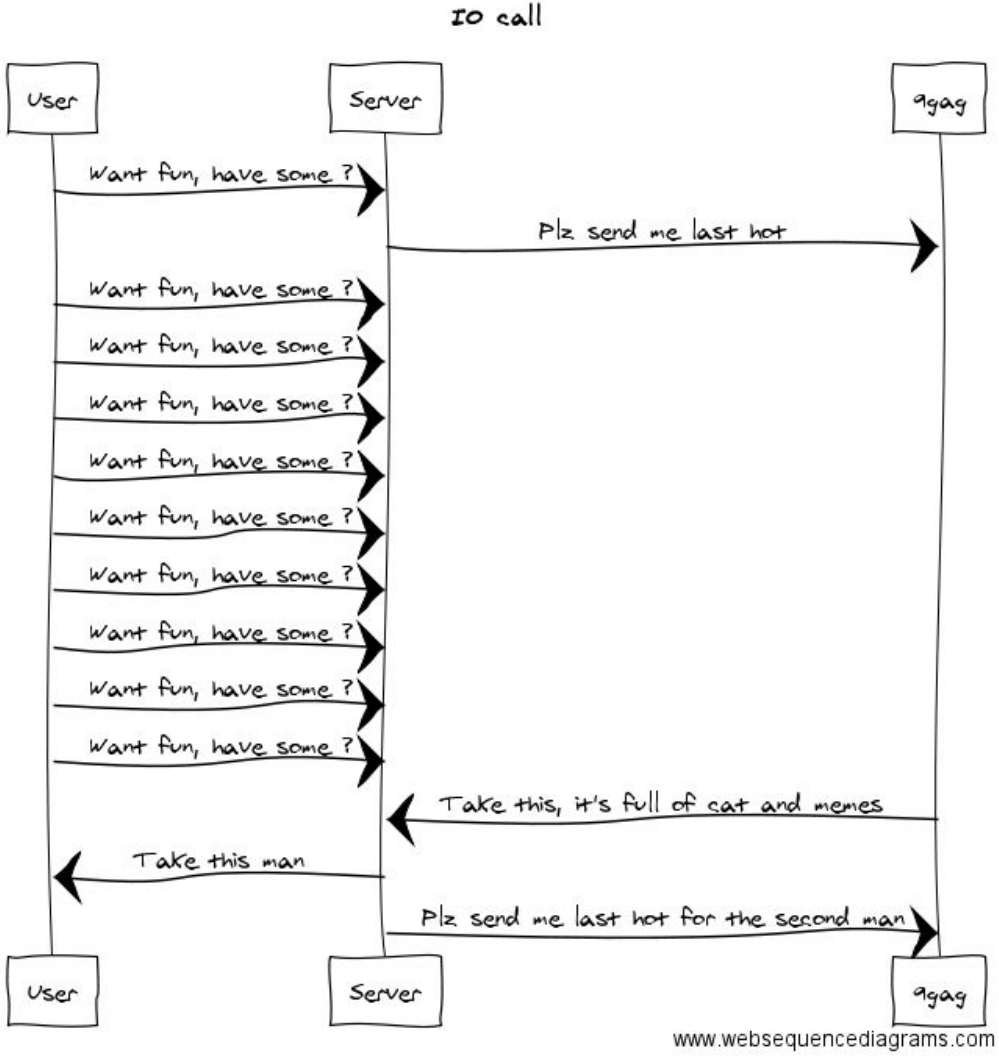
La resource se paye

Dans des univers cloud, la resource se paye.

Si votre programme peut faire la même chose avec ⅙ de mémoire en moins sur un ensemble de service ca fait un gain.

Le principe est d’utiliser des techniques de programmation (asynchone, programamtion concurrente, acteurs, …) pour éviter les blocages de resources.


IO call
User
Server
9gag
Want fun, have some ?
Shit, should call 9gag
Plz send me last hot
Take this, it's full of cat and memes
Take this man
User
Server
9gag
www.websequencediagrams.com


IO call
User
Server
9gag
Want fun, have some ?
Plz send me last hot
Want fun, have some ?
Want fun, have some ?
Want fun, have some ?
Want fun, have some ?
Want fun, have some ?
Want fun, have some ?
Want fun, have some ?
Want fun, have some ?
Want fun, have some ?
Take this, it's full of cat and memes
Take this man
Plz send me last hot for the second man
User
Server
9gag
www.websequencediagrams.com

Des paradigmes et des hommes

Pour régler ce type de problèmes on peut utiliser de nombreux outils :

- Programmation fonctionnelle + async
- Programmation reactive
- Event-loop
- …

Bref creusez un peu vous allez trouver

High availability

High availability

La haute disponibilité est une notion importante pour un service :

Comment m'assurer que mon service réponde à tout moment ?

Comment éviter un SPOF (diapo suivante).

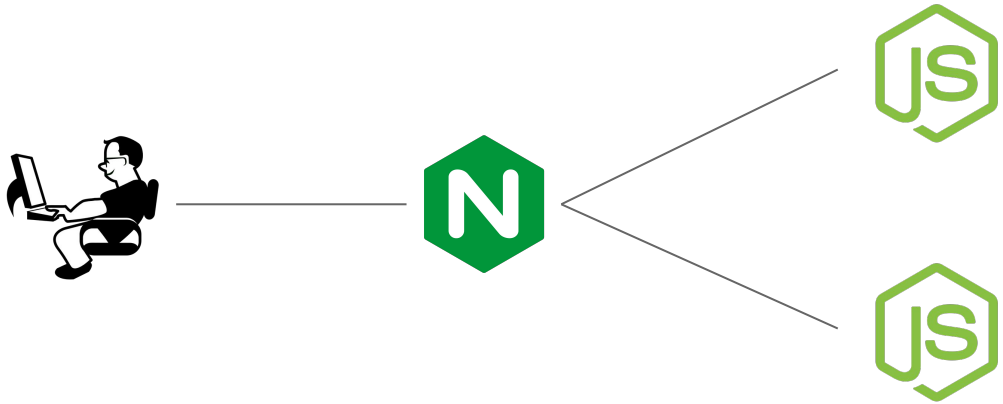
“

Un **point unique de défaillance** (*Single Point of Failure* ou *SPOF* en anglais) est un point d'un système informatique dont le reste du système est dépendant et dont une panne entraîne l'arrêt complet du système.

La notion de point unique de défaillance est fortement liée à celle de service, dans la mesure où un problème sur le point concerné entraîne une interruption de service.

Redondance de service

La solution la plus simple pour gérer la haute dispo est la redondance de l'application.



Si un des serveur nodejs crash, le premier gèrera le trafic le temps de remonter le deuxième

HA à l’infini

Dans l’exemple précédant, le load-balancer devient à son tour le SPOF du système.

Pour avoir une autre disponibilité d’un système il faut :

- Redondance
- Suppression des SPOF
- Monitoring des erreurs

Scaling

Scaling

Deux versions du scaling :

- Scaling vertical

  Si le besoin de ressources augmente, utilisons plus de resources. (Changement de machine, ajout de RAM, changement network, …)

- Scaling horizontal

  Gagner en performance applicative en augmentant le nombre d’instance et pas les capacité de chacunes.

Which type ?

Le scaling vertical nécessite une connaissance approfondie des besoins d’une application et il s’agit d’un process lent.

On parle donc souvent plus de scaling horizontal car il est plus souple mais demande beaucoup plus de préparation lors du développement.

Petit warning

Quand on parle scaling, instinctivement, on pense tous scale-up (augmentation de la capacité).

Le scaling a une portée beaucoup plus importante.

Le scaling est la capacité à gérer le besoin actuel, ni plus, ni moins.

Payer 30 serveurs alors que 2 peuvent faire tourner ma charge actuelle est de l'argent perdu.

Des pré-requis

Le plus grand problème avec le scaling, c’est qu’il est facile à réaliser mais que tous les applicatifs ne peuvent pas être scale automatiquement.

Tous les points vu avant sont soit obligatoire soit recommandés.

La non gestion d’état est surement la condition la plus importante.

Twelve Factor (Methodology)

Twelve Factor

La méthodologie 12 factor est un papier qui est dérivé de l'expérience d'un des plus ancien acteur du cloud : Heroku.

Il s'agit de mettre en place 12 points importants lors de la réalisation d'une application pour qu'elle soit utilisable et déployable comme un service.

https://12factor.net/fr/

Ces twelves factors (1 /2)

I. Codebase

One codebase tracked in revision control, many deploys

II. Dependencies

Explicitly declare and isolate dependencies

III. Config

Store config in the environment

IV. Backing Services

Treat backing services as attached resources

V. Build, release, run

Strictly separate build and run stages

VI. Processes

Execute the app as one or more stateless processes

Ces twelves factors (2 /2)

VII. Port binding

Export services via port binding

VIII. Concurrency

Scale out via the process model

IX. Disposability

Maximize robustness with fast startup and graceful shutdown

X. Dev/prod parity

Keep development, staging, and production as similar as possible

XI. Logs

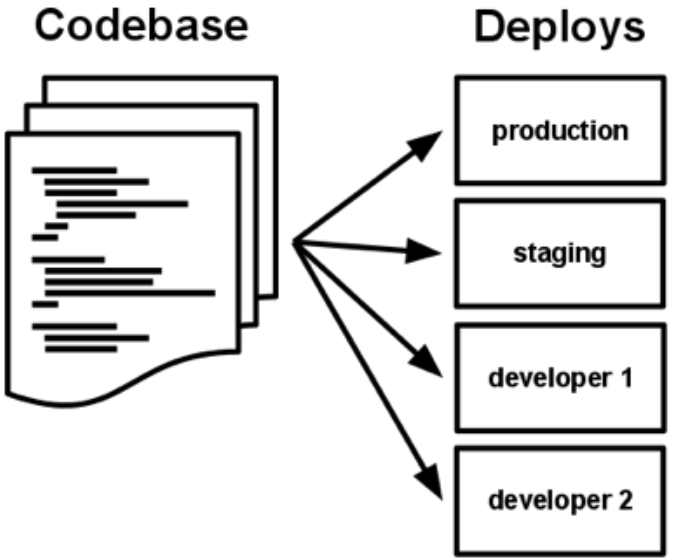
Treat logs as event streams

XII. Admin processes

Run admin/management tasks as one-off processes

I - Codebase

One codebase tracked in revision control, many deploys



- Always track source code
- codebase = 1 repo
- if multiples repo it's not an app but a distributed system
- **Multiple app should not share a same repo**
- There is only one codebase per app, but there will be many deploys of the app.
- A deploy is a running instance of the app.

II - Dependencies

- App never relies on implicit existence of system-wide packages
- **all dependencies should be declared, completely and exactly, via a dependency declaration manifest**
- App uses a dependency isolation tool during execution to ensure that no implicit dependencies “leak in” from the surrounding system
- The full and explicit dependency specification is applied uniformly to both production and development
- One benefit of explicit dependency declaration is that it simplifies setup for developers new to the app
- **Cloud app never relies on the implicit existence of any system tools (curl does not exist on all systems)**

III - Configs

An app's config is everything that is likely to vary between deploys (staging, production, developer environments, etc).

This includes: Resource handles to the database, Credentials to external services, etc.

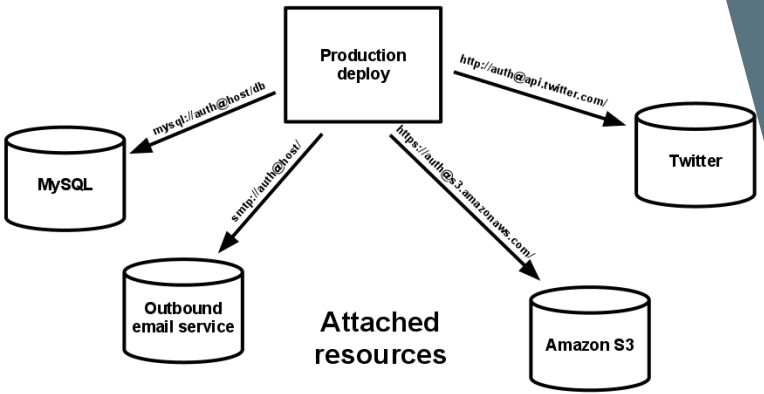
- **Never store config as constants in the code**
- App requires **strict separation of config from code**
- Codebase could be made open source at any moment, without compromising any credentials
- **App stores config in *environment variables***.
  - Env vars are easy to change between deploys without changing any code; unlike config files,
  - there is little chance of them being checked into the code repo accidentally;
  - they are a language- and OS-agnostic standard unlike custom config files, or other config mechanisms such as Java System Properties

IV - Backing service

Treat backing services as attached resources

A *backing service* is any service the app consumes over the network as part of its normal operation:

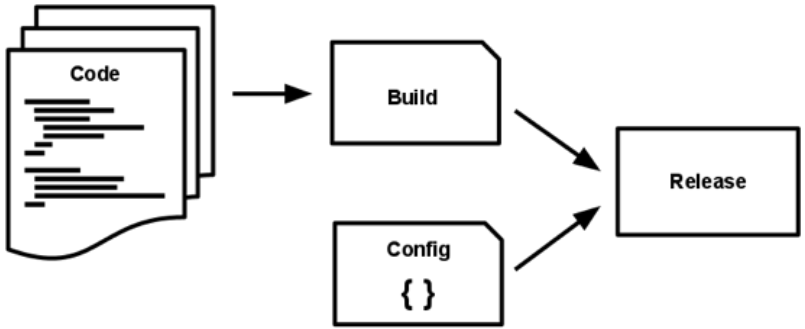
- datastores (such as MySQL or CouchDB),
- messaging/queueing systems (such as RabbitMQ or Beanstalkd),
- SMTP,
- caching systems (such as Memcached)
- etc.



- **An app makes no distinction between local and third party services**
- both are attached resources, accessed via a URL or other locator/credentials stored in the config
- app should be able to swap out a local MySQL database with one managed by a third party (such as Amazon RDS) without any changes to the app's code
- Resources can be attached and detached to deploys at will

V - Build, release, run

Strictly separate build and run stages



- it is impossible to make changes to the code at runtime, since there is no way to propagate those changes back to the build stage.
- Every release should always have a unique release ID (timestamp or incremental number)

VI - Processes

Execute the app as one or more stateless processes

- **processes are stateless and share-nothing. Any data that needs to persist must be stored in a stateful backing service, typically a database.**
- The memory space or filesystem of the process can be used as a brief, single-transaction cache. For example, downloading a large file, operating on it, and storing the results of the operation in the database.
- **app never assumes that anything cached in memory or on disk will be available on a future request or job** – with many processes of each type running, chances are high that a future request will be served by a different process.
- app should do compiling assets during the build stage, such as the Rails asset pipeline, rather than at runtime
- **"sticky sessions"** – that is, caching user session data in memory of the app's process and expecting future requests from the same visitor to be routed to the same process - **should never be used or relied upon**.

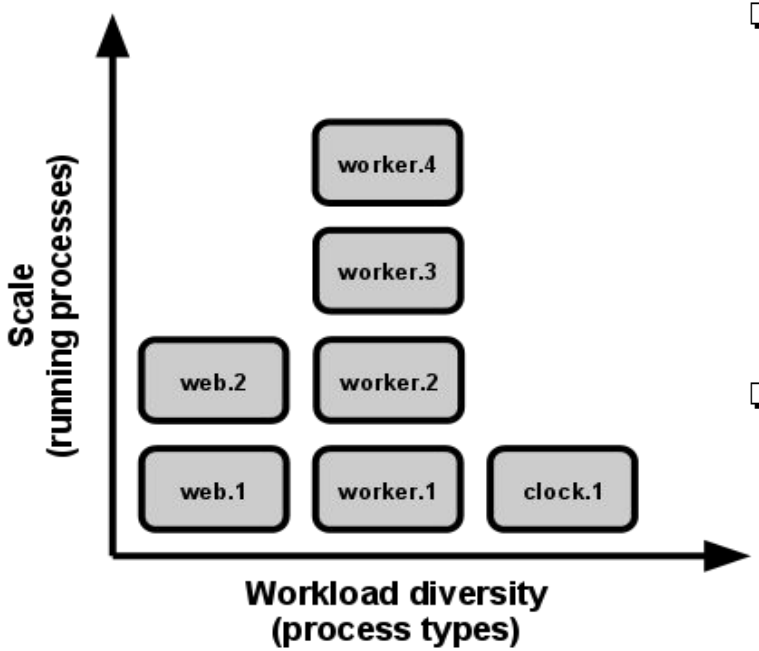
VII - Port binding

Export services via port binding

- **app is completely self-contained** and does not rely on runtime injection of a webserver into the execution environment to create a web-facing service
- web app **exports HTTP as a service by binding to a port**, and listening to requests coming in on that port
- This is typically implemented by using dependency declaration to add a webserver library to the app, such as Tornado for Python, Thin for Ruby, or Jetty for Java and other JVM-based languages
- HTTP is not the only service that can be exported by port binding (XMPP, Redis, etc.)

VIII - Concurrency

Scale out via the process model



- **app, processes are a first class citizen.** Processes in the twelve-factor app take strong cues from the unix process model for running service daemons. Using this model, the developer can architect their app to handle diverse workloads by assigning each type of work to a *process type*. For example, HTTP requests may be handled by a web process, and long-running background tasks handled by a worker process.

- app processes should never daemonize or write PID files. Instead, rely on the operating system's process manager (such as Upstart, a distributed process manager on a cloud platform, or a tool like Foreman in development) to manage output streams, respond to crashed processes, and handle user-initiated restarts and shutdowns

IX - Disposability

Maximize robustness with fast startup and graceful shutdown

- **app's processes are *disposable*, meaning they can be started or stopped at a moment's notice.**
  This facilitates fast elastic scaling, rapid deployment of code or config changes, and robustness of production deploys.

- Processes should strive to **minimize startup time**.
  Short startup time provides more agility for therelease process and scaling up; and it aids robustness, because the process manager can more easily move processes to new physical machines when warranted.

- Processes **shut down gracefully when they receive a SIGTERM** signal from the process manager
  For a web process, graceful shutdown is achieved by ceasing to listen on the service port (thereby refusing any new requests), allowing any current requests to finish, and then exiting

- Processes should also be **robust against sudden death**, in the case of a failure in the underlying hardware.
  A recommended approach is use of a robust queueing backend, that returns jobs to the queue when clients disconnect or time out.

X - Dev/Prod Parity

Keep development, staging, and production as similar as possible

There have been substantial gaps between development and production :

- **The time gap:** A developer may work on code that takes days, weeks, or even months to go into production.
- **The personnel gap**: Developers write code, ops engineers deploy it.
- **The tools gap**: Developers may be using a stack like Nginx, SQLite, and OS X, while the production deploy uses Apache, MySQL, and Linux.

App is designed for continuous deployment by keeping the gap between development and production small

- Make the time gap small: a developer may write code and have it deployed hours or even just minutes later.
- Make the personnel gap small: developers who wrote code are closely involved in deploying it and watching its behavior in production.
- Make the tools gap small: keep development and production as similar as possible.

XI - Logs

Treat logs as event streams

Logs provide visibility into the behavior of a running app. In server-based environments they are commonly written to a file on disk (a “logfile”); but this is only an output format.

- **app never concerns itself with routing or storage of its output stream.** It should not attempt to write to or manage logfiles. Instead, each running process writes its event stream, unbuffered, to stdout.

- During local development, the developer will view this stream in the foreground of their terminal to observe the app’s behavior.

- In staging or production deploys, each process’ stream will be captured by the execution environment, collated together with all other streams from the app, and routed to one or more final destinations for viewing and long-term archival.

- The event stream for an app can be routed to a file, or watched via realtime tail in a terminal. Most significantly, the stream can be sent to a log indexing and analysis system, or a general-purpose data warehousing system.

XII - Admin processes

Run admin/management tasks as one-off processes

Developers will often wish to do one-off administrative or maintenance tasks for the app, such as:

- Running database migrations (e.g. `manage.py migrate` in Django, `rake db:migrate` in Rails).
- Running a console (also known as a REPL shell) to run arbitrary code or inspect the app's models against the live database. Most languages provide a REPL by running the interpreter without any arguments (e.g. `python` or `perl`) or in some cases have a separate command (e.g. `irb` for Ruby, `rails console` for Rails).
- Running one-time scripts committed into the app's repo (e.g. `php scripts/fix_bad_records.php`).

- **One-off admin processes should be run in an identical environment as the regular long-running processes of the app**

Conclusion

Des concepts, des concepts

Le cloud c’est ca !

C’est un ensemble de pratiques et de méthodologies applicable pour réaliser et mettre à disposition des applicatifs.

Mettre en place toute ces choses va vous permettre d’avoir des applications robustes, scalables avec une bonne capacité d’évolution.

Questions ?