Technology

There is a better way

Cloud Ready Apps

CONCEVOIR VOS APPLICATIONS POUR TIRER PARTI DU CLOUD

www.octo.com - blog.octo.com

Table des ma+ières

Cloud Ready
Applications

Les applications d'aujourd'hui et leurs futures évolutions doivent être un avantage concurrentiel. Elles doivent s'adapter à un environnement changeant et offrir une qualité de service toujours plus élevée. Le cloud offre des opportunités pour répondre à ces nouveaux challenges.

Pour en tirer profit et concevoir une application cloud ready, nous avons décelé cinq enjeux fondateurs pour les applications de demain.

Pour chacun, nous vous proposons un chapitre pour comprendre comment concevoir l'architecture de ces applications avec des **patterns***, des **pistes de réflexion, des services managés** et des exemples d'implémentations destinés à répondre à cet enjeu. Par souci d'homogénéité, tous les exemples se basent sur Amazon Web Services, mais restent transposables sur toute autre plateforme de cloud.

Que vous conceviez, développiez ou exploitiez des applications sur le cloud, ces éléments vous aideront à comprendre quelles librairies, quels services, quelles architectures utiliser en fonction de vos besoins.

Ces fiches se veulent des pistes de réflexion et non des solutions prêtes à l'usage ; en effet, l'ensemble des patterns* applicatifs décrits sur un schéma ne sont certainement pas tous nécessaires dans votre contexte, De plus les solutions techniques ne sont rien sans l'organisation humaine adaptée. Enfin, parce que **les pratiques Cloud Ready Applications** sont complémentaires avec d'autres pratiques que nous défendons également chez OCTO pour construire des applications : Software Craftsmanship et DevOps notamment.

* Un pattern est une solution répétable et réutilisable à un problème identifié.

XaaS (Anything As A Service) : où vos applications peuvent-elles être déployées ?

En fonction du type de XaaS choisi, votre application dispose de plus ou moins de services sous-jacents :

IaaS : Infrastructure as a Service
Désigne les fondations du cloud computing. Il s'agit d'offres de capacité de calcul, de stockage et de réseau sous une forme comparable à celle d'une machine virtuelle.

CaaS : Container as a Service
Désigne les offres permettant d'héberger des conteneurs de type Docker. Un conteneur de type Docker est, de façon très schématique, une application packagée pour être opérée de façon totalement isolée. Un conteneur est implémenté en isolant des ressources au sein d'un même système d'exploitation.

PaaS : Platform as a Service
Désigne des offres permettant d'héberger des applications, d'offrir un stockage structuré et requêtable (SQL ou NoSQL), de fournir un MOM (Message Oriented Middleware) sous forme de service… Le nombre de stacks prises en charge et l'abstraction vis-à-vis des ressources matérielles sont variables selon les offres.

SaaS : Software as a Service
Désigne le fait de distribuer un logiciel en ligne. Le concept est utilisé depuis très longtemps pour les clients mails. Il a été démocratisé avec la suite bureautique en ligne Google Documents et le logiciel de CRM en ligne SalesForces.

BaaS : Back-end as a Service
Désigne un service sur étagère offrant des fonctionnalités back-end en particulier pour une application mobile.

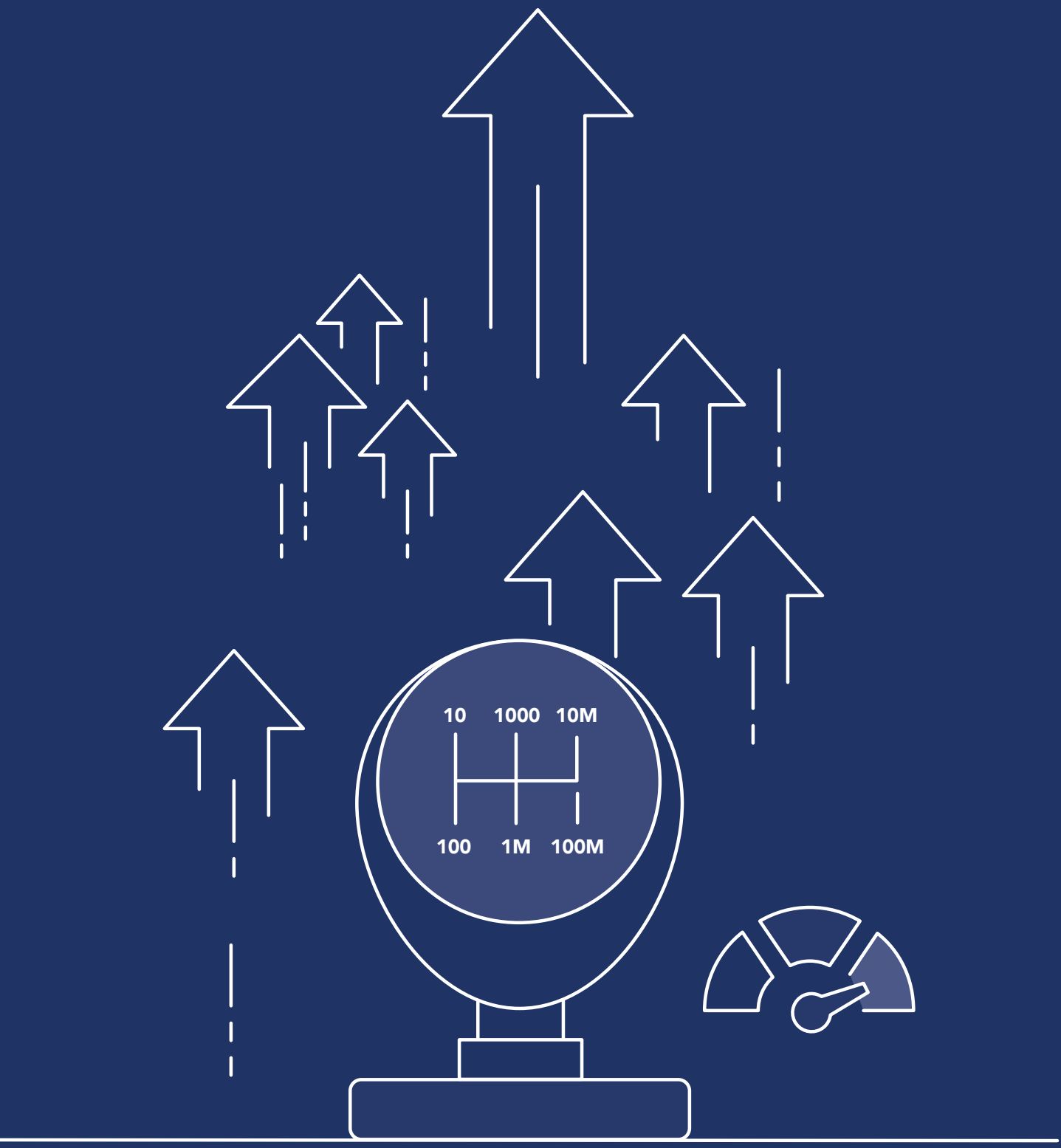
Les principes suivants sont utiles en tout cas pour répondre aux différents enjeux de vos applications.

IaaS	CaaS	PaaS	BaaS	SaaS
Front-end Applicatif	Front-end Applicatif	Front-end Applicatif	Front-end Applicatif	Front-end Applicatif
Back-end Applicatif	Back-end Applicatif	Back-end Applicatif	Back-end Applicatif	Back-end Applicatif
Services (OS, middleware, conteneur)	Services (OS, middleware, conteneur)	Services (OS, middleware, conteneur)	Services (OS, middleware, conteneur)	Services (OS, middleware, conteneur)
Serveur Physique	Serveur Physique	Serveur Physique	Serveur Physique	Serveur Physique
Stockage	Stockage	Stockage	Stockage	Stockage
Réseau DataCenter	Réseau DataCenter	Réseau DataCenter	Réseau DataCenter	Réseau DataCenter

● Développement interne

● Fourni dans l'offre de cloud

Scaler naturellement


10
1000
10M
100
1M
100M

Scaler
Naturellement

S'APPUYER INTELLIGEMMENT SUR LE CLOUD

- Envisager en priorité l'utilisation des services managés, nativement résilients et scalables fournis par les plateformes cloud : PaaS, services de persistance et de cache, BaaS…

Patterns : *SaaS (Ex. Salesforce, AWS RDS, AWS ElastiCache), PaaS (Ex. Heroku, AWS Lambda), BaaS (Ex. Parse, Kumulos).*

CHOISIR LE BON NIVEAU D'ABSTRACTION : SERVEUR, CONTENEUR, RUNTIME…

- Concevoir en faisant abstraction de la localisation physique pour l'exécution, le déploiement en cloud n'offrant pas de garantie sur cet aspect afin d'allouer les ressources là où elles sont le plus rapidement disponibles.
- Adapter la conception au niveau d'abstraction de la plateforme : certaines peuvent imposer des contraintes fortes (temps d'exécution limité, mono-thread…).
- Envisager des produits (système d'exploitation, logiciels, services…) ayant des politiques de facturation à l'usage plutôt qu'au nombre de machines.

Patterns : *Location Transparency, Fast Startup and Graceful Shutdown, Service Discovery.*

CONCEVOIR POUR LA SCALABILITÉ

- Pour simplifier l'élasticité et l'adaptation rapide à la charge, l'application doit embarquer des mécanismes facilitant sa scalabilité.
- La scalabilité horizontale offre une approche plus extensible tout en contribuant à éviter les SPOF, même si elle est plus complexe à mettre en œuvre que la scalabilité verticale.
- Il est préférable de déplacer le traitement plutôt que les données lorsqu'elles sont volumineuses. Le cloud offre en effet la possibilité d'allouer aisément des capacités de traitement en différentes zones.

Patterns : *Load balancer, sharding, shared nothing, microservices.*

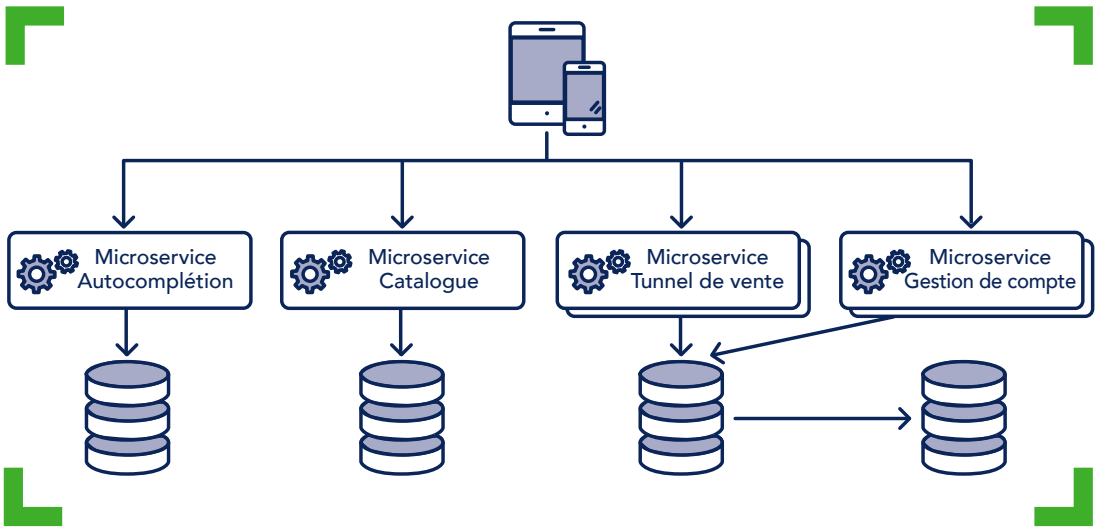
Pattern

Microservices

Approche d'architecture de SI visant à modulariser le SI en privilégiant des services plus petits, portés chacun par une équipe dédiée. Par opposition à une approche plus intégrée/monolithique, une architecture en microservices :

- Clarifie les responsabilités par la matérialisation des frontières.
- Limite la complexité locale et ainsi fiabilise les développements.
- Rend les couplages plus faibles, ce qui facilite les changements et donc l'innovation.
- Permet d'utiliser des technologies différentes.
- Autorise à gérer la scalabilité en permettant de la gérer à un niveau plus fin.
- Autorise à gérer la haute disponibilité à un niveau plus fin.

Cette approche est toutefois plus exigeante en termes d'organisation, du fait de la nécessité de coordonner les changements et les déploiements. Par ailleurs, les liens entre services augmentent la complexité globale et leur mise en place peut être coûteuse.



EXEMPLE

Services offerts par AWS pour la scalabilité

1. Route 53 : Service de DNS qui peut être utilisé pour router le trafic au niveau DNS entre plusieurs régions, en détectant au besoin si une région est indisponible.

2. ELB : Service permettant de répartir la charge entre plusieurs instances AWS, en détectant au besoin si l'une d'elles est indisponible.

3. Cloud Watch : Service de surveillance des instances et applications qui permet notamment de donner des alertes à l'Auto Scaling Group.

4. Auto Scaling Group : Service permettant d'augmenter ou de réduire le nombre d'instances AWS en fonction de métriques CloudWatch.

5. RDS : Relation Data Service. Service managé de bases de données relationnelles.

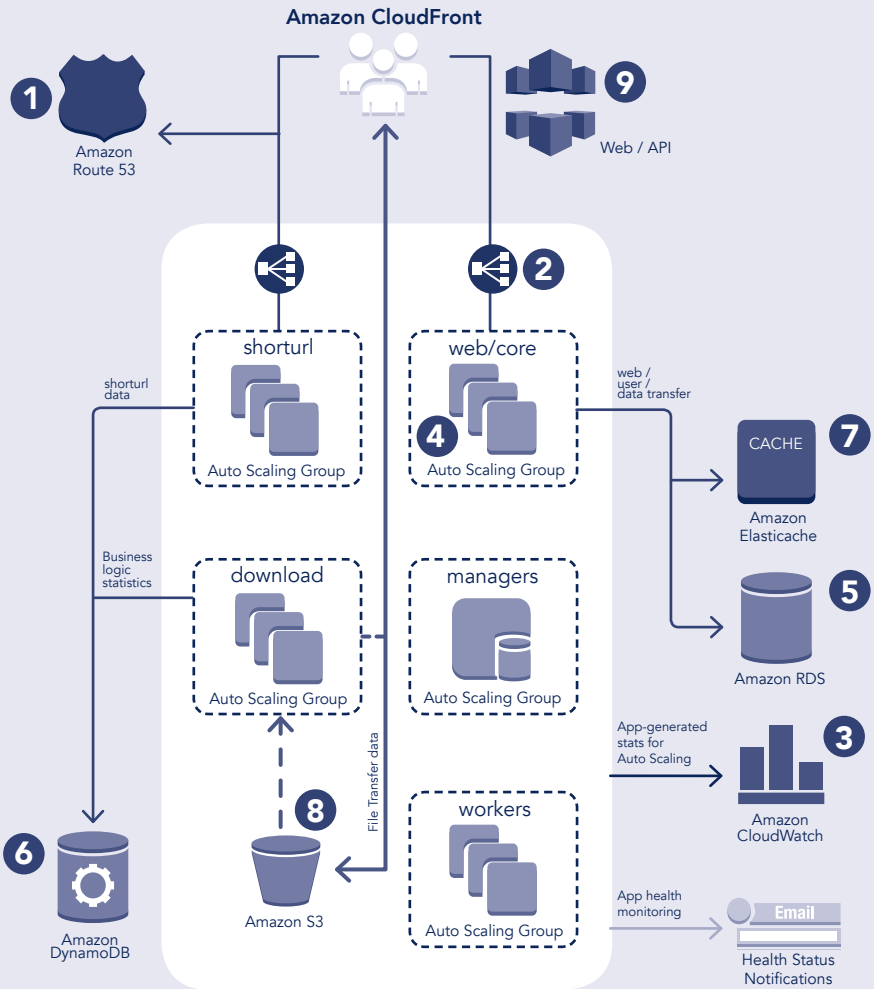
6. DynamoDB : Service managé de base de données NoSQL Clé/Valeur.

7. ElasticCache : Service de cache compatible avec les protocoles Redis et Memcached.

8. S3 : Single Storage Service. Service de stockage objet illimité gérant des fichiers d'une taille allant jusqu'à 5TB (en septembre 2016).

9. CloudFront : Service de CDN (Content Delivery Network) proposant des « points de présence » sur toute la surface du globe ce qui permet de diminuer la latence et de réduire les coûts en servant un cache HTTP depuis une localisation plus proche de l'utilisateur.

Remarque :
La haute disponibilité et la scalablité de ces services sont assurées par la plateforme AWS.


Amazon CloudFront
1
Amazon Route 53
9
Web / API
2
shorturl
Auto Scaling Group
web/core
4
Auto Scaling Group
shorturl data
web / user / data transfer
CACHE
7
Amazon Elasticache
Business logic statistics
download
Auto Scaling Group
managers
Auto Scaling Group
5
Amazon RDS
App-generated stats for Auto Scaling
3
Amazon CloudWatch
File Transfer data
8
Amazon S3
workers
Auto Scaling Group
6
Amazon DynamoDB
App health monitoring
Email
Health Status Notifications

EXEMPLE

Implémentation scalable

Au niveau du stockage de données vous concevez votre application en tenant compte du sharding des données (1). Il s'agit du premier prérequis à la scalabilité. Les systèmes de stockage actuels NoSQL qui abstraient ce concept restent très rares. Leur optimisation nécessite dans tous les cas de maîtriser dès la conception les choix faits au niveau du sharding des données.

Supporter du load balancing (2) permet qu'une requête soit traitée par une instance de l'application, la requête suivante par une seconde instance. De cette façon, vous pourrez augmenter le nombre d'instances pour gérer plus de trafic. Cela impose une gestion très stricte des états dans le code.

Pouvoir s'exécuter sans faire d'hypothèse sur son emplacement physique permet enfin d'aller un cran plus loin en exécutant potentiellement l'instance sur une autre machine, voire dans un autre data-center. C'est ce qu'on appelle la location transparency (3). « Je veux accéder à mon fichier local, je veux lire les logs de cette instance », ces réflexes doivent être perdus.

Enfin, une application qui peut s'exécuter sur une machine de faible puissance, avec de multiples cœurs disponibles, et qui sait les utiliser au mieux vous permet de choisir la ressource la moins chère disponible à ce moment précis. Pour cela, vous devez penser votre application pour qu'elle soit capable d'être arrêtée et redémarrée en plein traitement. AWS offre par exemple un mécanisme d'enchères avec ses instances Spot qui autorisent à consommer de la puissance supplémentaire tant que son prix est inférieur à un seuil fixé. Lorsque le prix remonte, la machine est arrêtée sous 2 minutes. La capacité de fast startup & graceful shutdown (5) est ici indispensable.

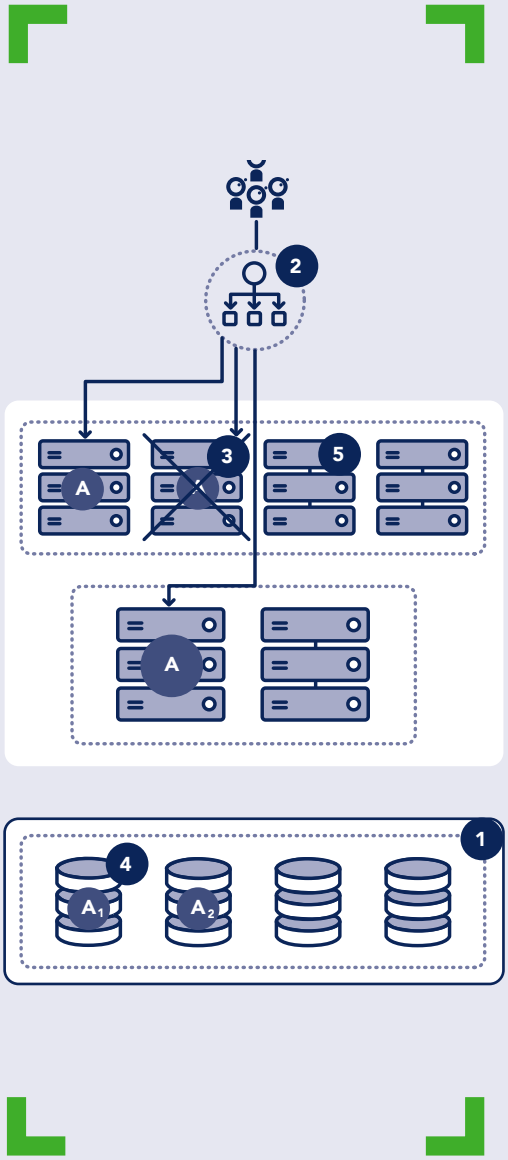
1. Sharding :
Le sharding décrit ainsi un ensemble de techniques qui permettent de répartir les données sur plusieurs machines pour assurer la scalabilité de l'architecture.

2. Load balancer :
Composant réseau ou applicatif qui répartit les requêtes sur différentes instances de façon à équilibrer la charge. Amazon ELB est par exemple un service de load balancing fourni par AWS.

3. Location transparency :
Mécanisme de découplage permettant à l'émetteur de ne pas connaître la localisation du récepteur. Par exemple DNS, reverse proxy…

4. Shared Nothing :
Pattern d'architecture dans lequel les nœuds d'un système distribué ne partagent ni disque, ni mémoire et qui ne présente aucun point unique de défaillance (Single Point of Failure).

5. Fast startup and graceful shutdown :
Capacité de l'application à servir rapidement des requêtes après le démarrage et à réaliser automatiquement les opérations nécessaires pour son arrêt (sauvegarder les résultats intermédiaires, diffuser un état partagé…). En effet dans le cloud, une instance peut être démarrée ou arrêtée à tout moment. Voir http://12factor.net/



Assurer sur la qualité de service

Assurer
sur la qualité de service

MESURER, APPRENDRE, AMÉLIORER

- Collecter des mesures détaillées à tous les niveaux (système, middleware, application) pour connaître l'expérience client (UX), l'analyser et l'améliorer.
- Rendre la performance visible et accessible à tous les acteurs, dans le cadre d'un programme de performance en continu aux objectifs partagés et explicites.

Patterns : *Centralized monitoring, Correlation ID, Continuous performance.*

CONCEVOIR POUR LE SERVICE

- Rechercher la simplicité et la maintenabilité : évolutivité, exploitabilité, déployabilité.
- Appliquer les patterns en fonction de besoins avérés.
- Tenir compte des forces et faiblesses de l'environnement (infrastructure, middlewares, services cloud…).
- Penser à l'utilisateur : soigner l'UX en s'attachant au ressenti du client, prévoir un mode dégradé en cas d'imprévu.

Patterns : *KISS, Graceful degradation, Microservices.*

SE HISSER SUR LES ÉPAULES DES GÉANTS

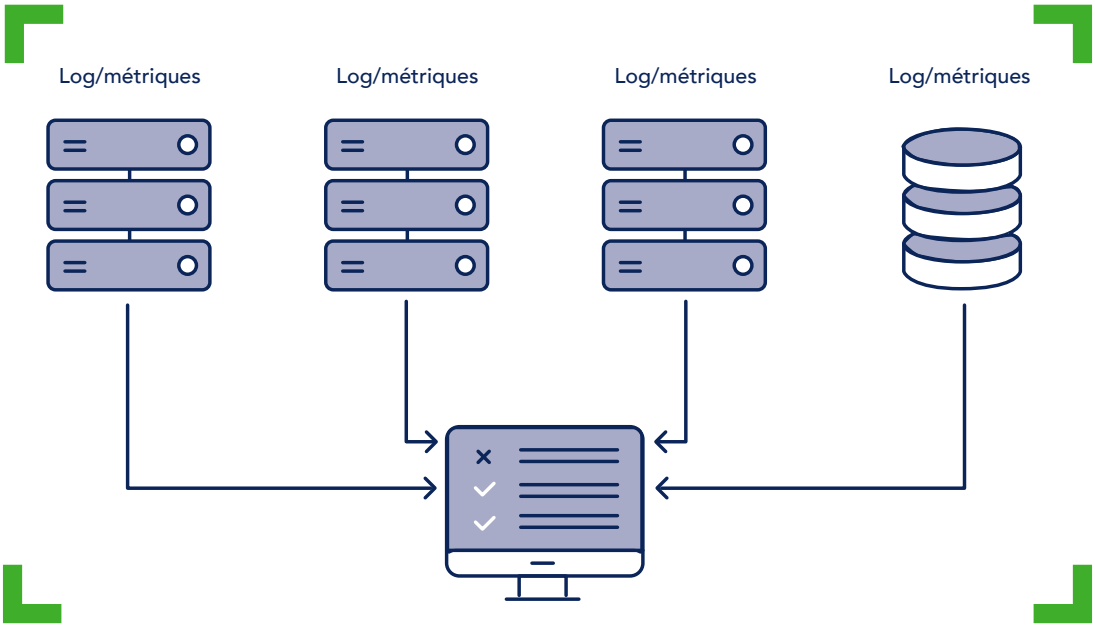
- S'imprégner de l'état de l'art, pour savoir s'appuyer sur les nombreux patterns élaborés par la communauté notamment sur la concurrence, le stockage et l'accès aux données, la réduction de la latence et la robustesse.

Patterns : *lambda architecture, request queuing, request collapsing, request redundancy, timeouts, circuit breakers, bulkheads, idempotence.*

Patterns

Centralized monitoring

Avec la possibilité d'ajouter et de retirer une instance de serveur à tout moment, il est difficile de consulter les logs locaux sur un grand nombre de machines. De plus, ces fichiers locaux peuvent disparaître à tout moment avec les instances associées, rendant la tâche totalement impossible. Il est donc nécessaire désormais de centraliser l'ensemble des logs. Par exemple dans une stack Elastic (Elasticsearch, Logstash et Kibana).

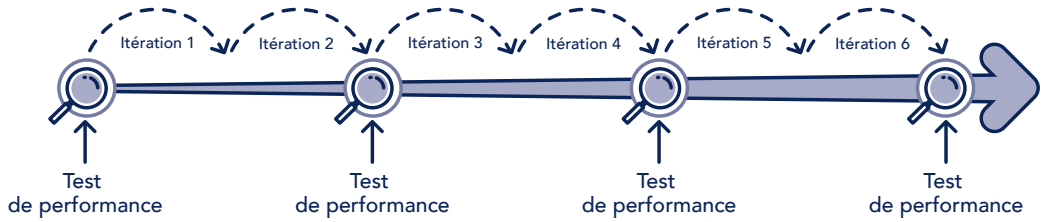


Attention : certains patterns induisent de la complexité, il convient donc d'en peser le pour et le contre.

Continuous performance

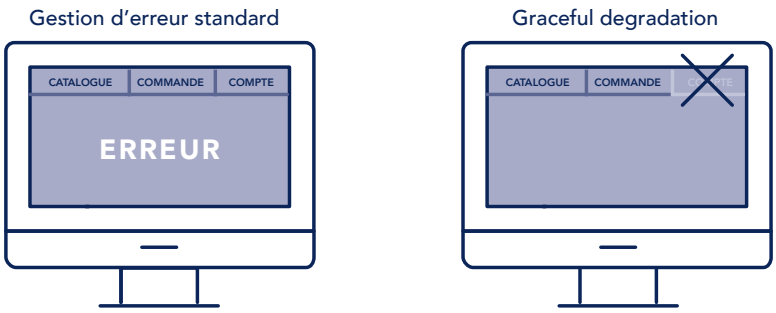
Processus visant à tester et mesurer la performance du système de manière continue, du développement à la production.

Cycles d'itérations agiles



Graceful degradation

Adaptation automatique de l'application à une situation dégradée (perte d'une partie de l'infrastructure, taille d'écran de l'utilisateur réduite, fonctionnalité non supportée par le navigateur cible, bande passante réduite, etc.).

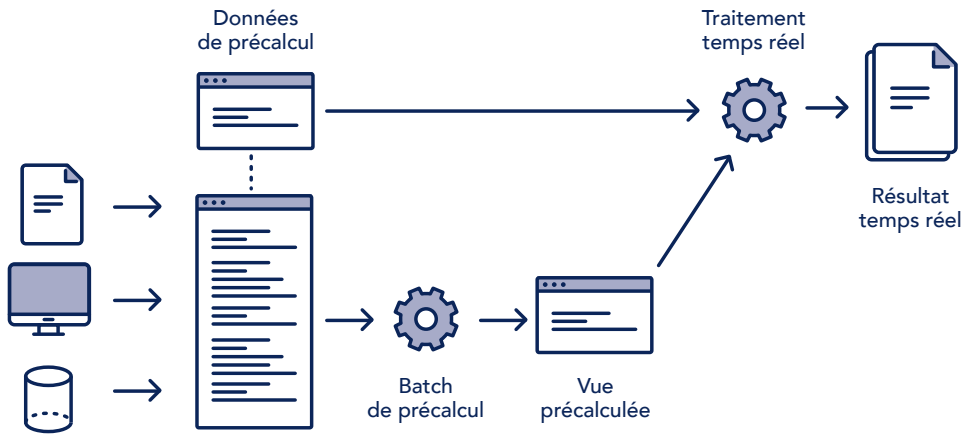


Lambda architecture

Une architecture lambda combine deux types de traitements sur les données :

- Traitements « batch », générant à intervalles réguliers des résultats précalculés.
- Traitements « temps réel », générant des résultats à la demande.

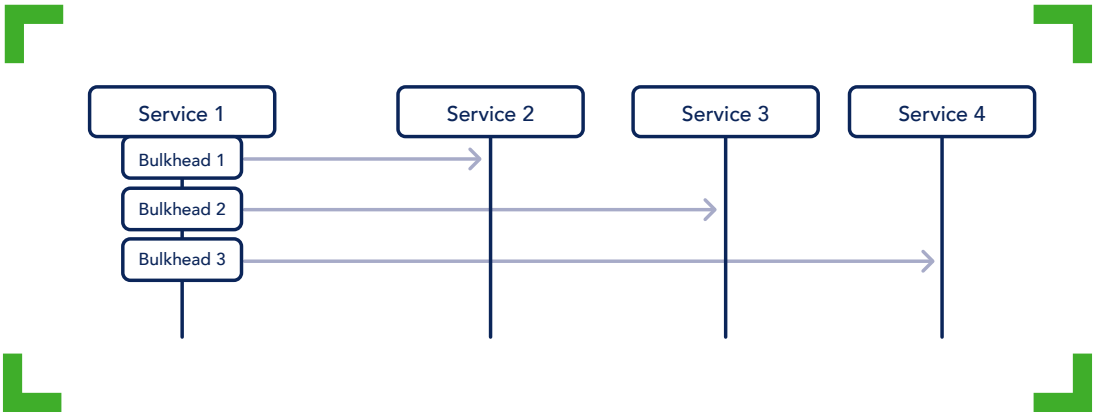
Cette combinaison permet de fournir des résultats actualisés malgré des calculs potentiellement coûteux et des données d'entrée changeant rapidement.



Bulkhead

La défaillance d'un système externe peut rapidement mettre en péril l'ensemble du service. Pour lutter contre cela, le pattern bulkhead permet d'isoler les appels. À la manière des caissons étanches de bateau, il permet de protéger le système d'une défaillance localisée qui se transformera en défaillance globale.

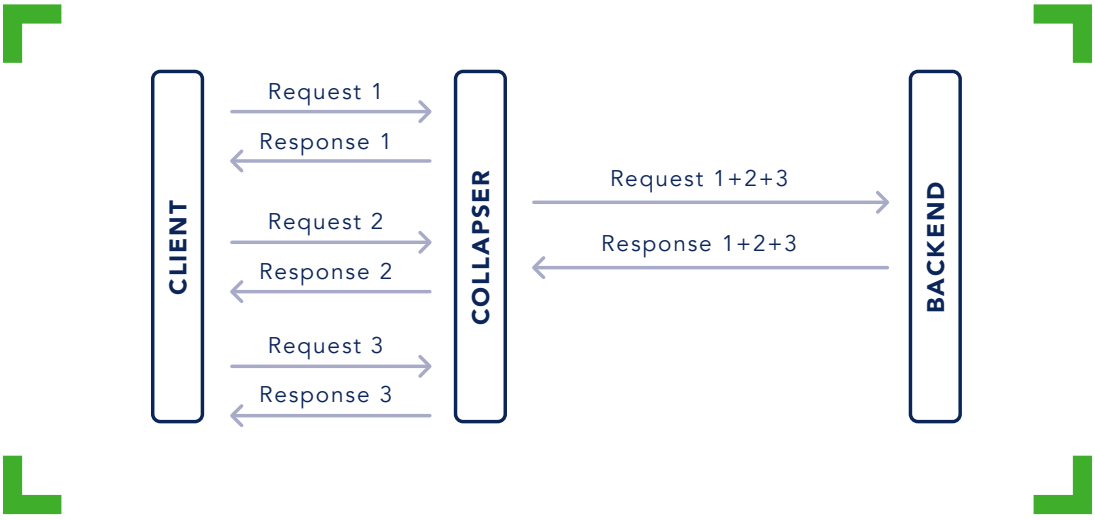
Un exemple d'implémentation de ce pattern est l'utilisation de deux pools de connexions pour qu'une saturation de l'un n'impacte pas l'autre.
(https://github.com/Netflix/Hystrix/wiki/How-it-Works#isolation)



Request collapsing

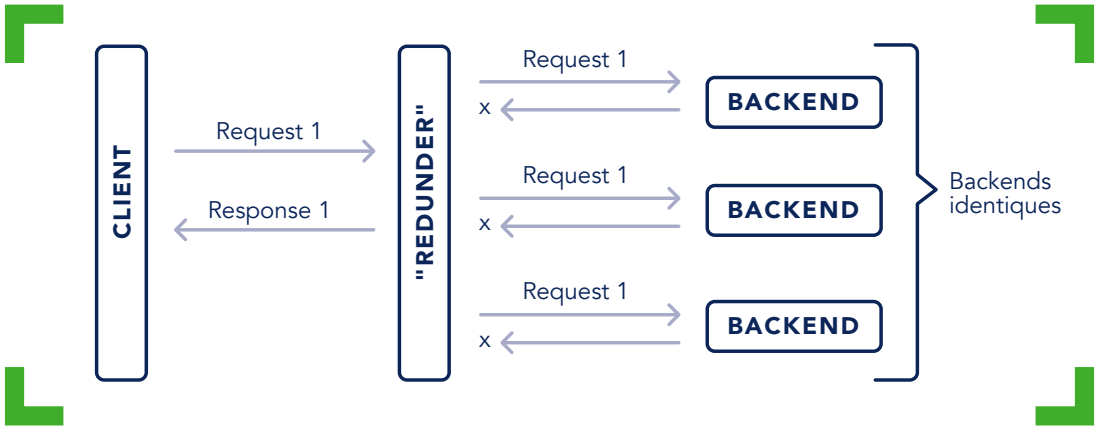
Pattern applicatif permettant de limiter le nombre d'appels distants en regroupant plusieurs requêtes pour n'en faire qu'une. Netflix Hystrix implémente ce pattern par exemple. Cela peut servir à limiter :

- la charge réseau,
- l'impact d'une latence réseau élevée,
- les coûts dans le cadre de licences à la requête.



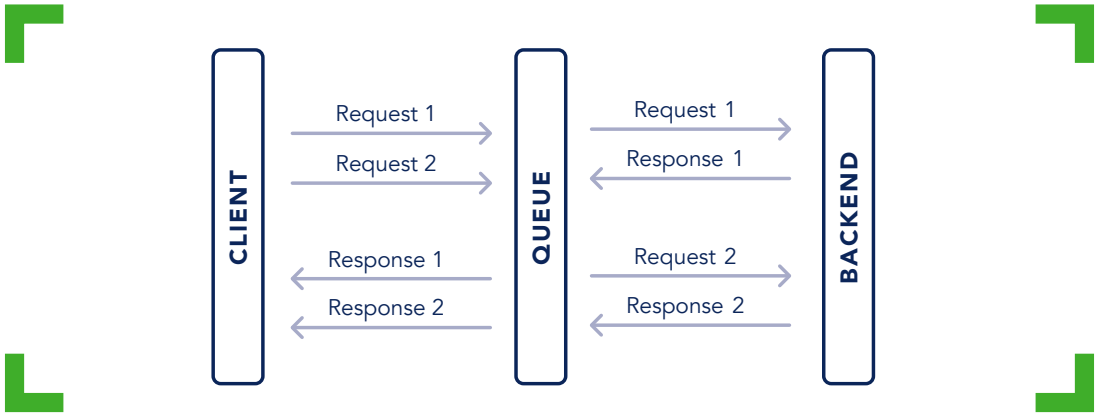
Request redundancy

Pattern qui consiste à envoyer une même requête à plusieurs systèmes distincts afin d'utiliser le résultat de la réponse la plus rapide. Le but est d'optimiser le temps de réponse. Google l'utilise par exemple pour son moteur de recherche.



Request queuing

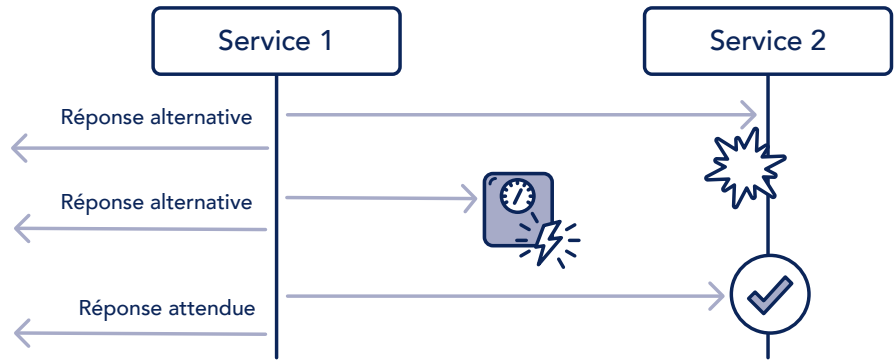
Pattern applicatif consistant à mettre les requêtes en attente afin d'éviter une surcharge du système. Netflix Hystrix, par exemple, le permet côté client. Ce pattern permet un couplage asynchrone, donc plus faible, des deux parties. Ainsi, les deux parties sont moins couplées en termes de disponibilité, ce qui peut permettre d'absorber des défaillances ou de faciliter le déploiement de mises à jour.

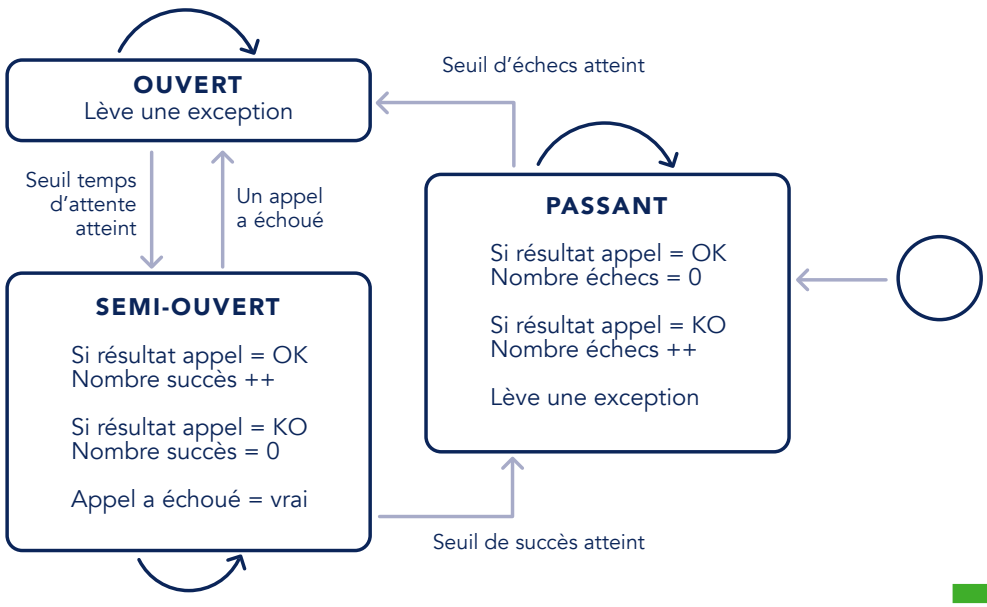


Circuit breaker

En fonction d'un certain nombre de critères d'erreur (timeout, nombre d'erreurs), ce pattern permet de désactiver l'envoi de requêtes au Service 2 et de renvoyer immédiatement une réponse alternative.

Il agit comme un proxy implémentant une machine à états (Ouvert, Passant (fermé), Semi-ouvert) pour l'apprentissage de l'état du service.

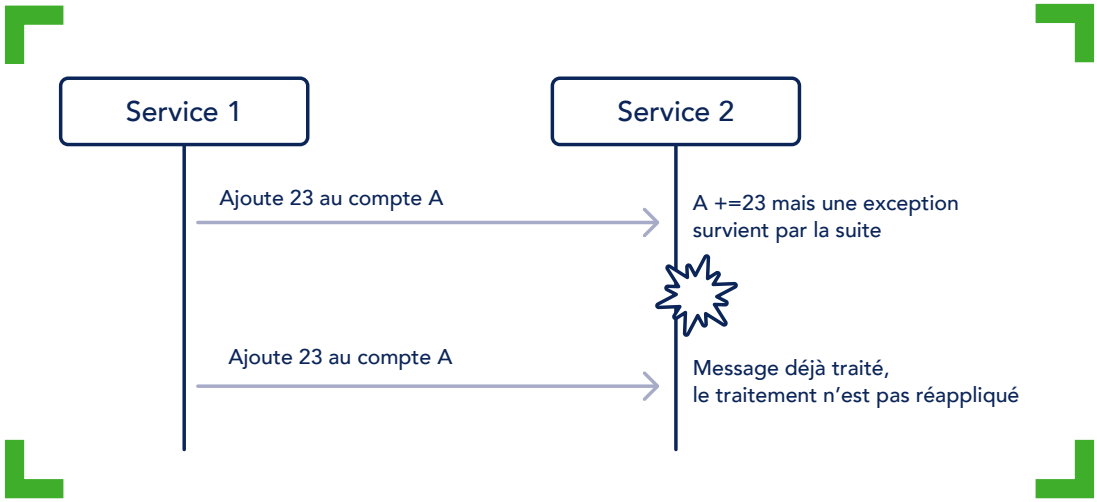




Idempotence

Une fois le service rétabli, la requête en échec peut potentiellement être relancée. Ceci n'est possible que si le Service 2 est capable d'être idempotent : « n » envois successifs de la même commande conduiront toujours au même état.

Il peut par exemple être implémenté en conservant un identifiant unique pour chaque requête et en ignorant les futures occurrences des requêtes avec le même identifiant.



Kiss (Keep it simple, stupid)

Principe qui préconise la simplicité dans la conception, notamment pour réduire les coûts, faciliter la compréhension, augmenter la fiabilité et limiter la maintenance.

Attention : certains patterns induisent de la complexité, il convient donc d'en peser le pour et le contre.


99,99%
de dispo-
nibili+é


24

99,99 %
de disponibilité

CONCEVOIR POUR LA DISPONIBILITÉ

- Éviter les points uniques de défaillance SPOF (Single Point Of Failure).
- Diviser l'application en composants faiblement couplés.
- Concentrer les états persistants dans quelques composants scalables.
- Prévoir et gérer les erreurs pour maîtriser leur propagation tout en conservant les traces pour analyse.

Patterns : *APIs, Microservices, Asynchronous data exchanges, Bulkheads, Idempotence, Health check.*

COMPOSER AVEC LES DÉFAILLANCES

- Accepter qu'en environnement fortement distribué, la probabilité d'erreur ne soit plus négligeable.
- Être capable de détecter un composant défaillant, et instancier à tout moment le ou les composants de l'application pour répondre aux pannes et retrouver un comportement nominal ou un niveau dégradé acceptable.

Patterns : *Pets vs. Cattle, Timeouts, Circuit Breakers, Fast Startup and Graceful Shutdown, Design For Failure.*

TESTER, TESTER, TESTER

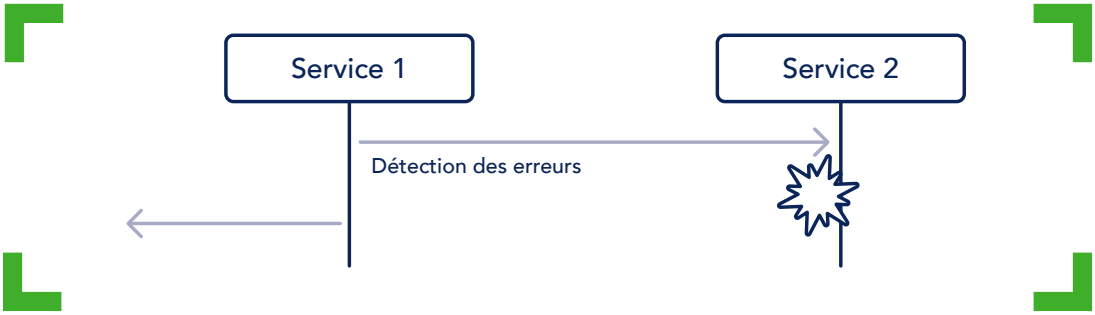
- Éprouver et entraîner sa capacité à gérer les différents incidents de production pour une meilleure réactivité lors des véritables incidents : crash d'un processus, erreur système, perte de serveur, indisponibilité d'une zone géographique du cloud, perte de lien réseau...

Patterns : *Simian Army, Test d'endurance, Test de robustesse.*

Patterns

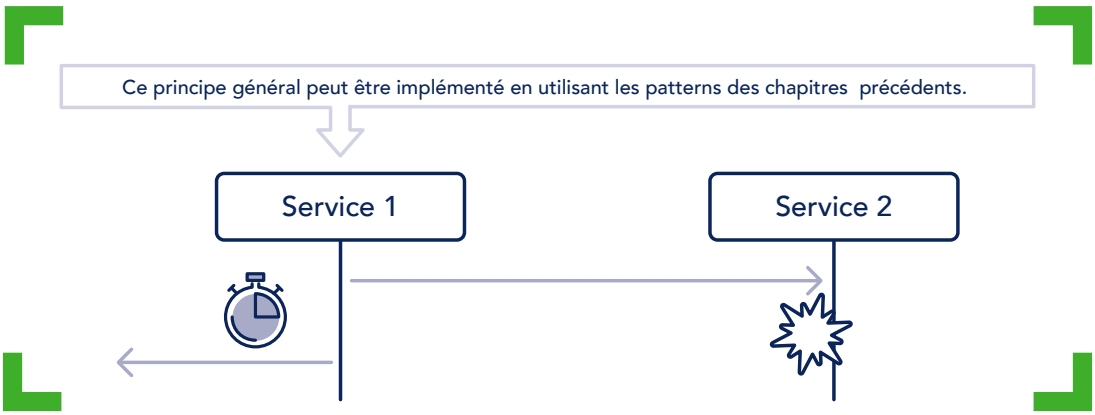
Design for failure

Les traitements applicatifs doivent, dès leur conception, prévoir le cas où les composants qu'ils appellent pourraient tomber en erreur. À leur charge de gérer au mieux ce cas en fournissant si possible un service dégradé à leur client.



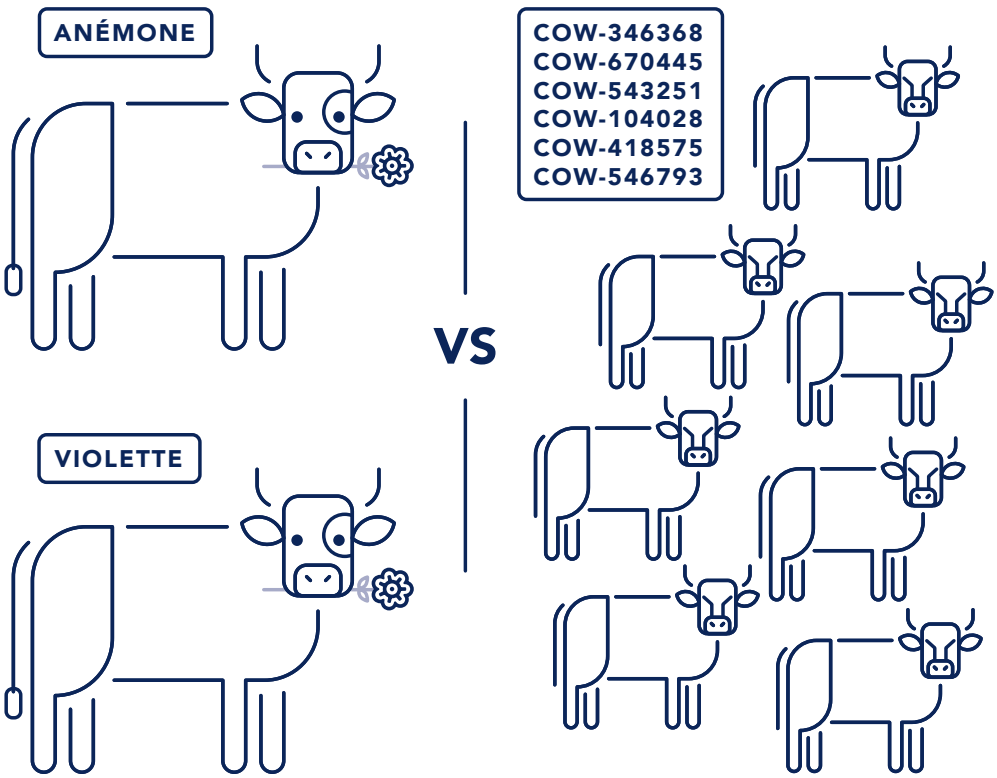
Timeout

Temps d'attente maximal pour chaque appel.



Pets versus Cattle

(lit. animaux de compagnie vs. bétail). Les serveurs ne sont plus nommés « Violette » ou « Anémone » mais par un identifiant unique (cow-xxx). Cette approche industrielle de gestion d'environnements consiste à ne plus nommer, réparer un serveur et le garder longtemps (Pet), mais à systématiquement le détruire et le recréer en cas de problème (Cattle).



Test d'endurance

Test avec la charge cible du système sur une longue durée afin d'identifier les fuites de ressources (ex. fuite mémoire).

Test de robustesse

Test avec la charge cible du système durant lequel on dégrade volontairement une ressource afin d'éprouver la robustesse du système.

EXEMPLE

Exemple d'implémentation haute disponibilité

Votre application est découpée en micro-services, exposant chacun des API que vous avez pris soin de rendre stateless et en adéquation avec les principes REST. Ceci vous permet de pouvoir profiter sans limites des outils de load balancing, de cache…

Vous déployez votre application sur la plateforme IaaS Amazon EC2. Pour atteindre une disponibilité de 99,99 % :

Vous choisissez de déployer votre application sur deux Availability Zones (AZ), qui correspondent à deux zones d'un ou plusieurs datacenters raisonnablement espacés géographiquement pour qu'un incident ne les impacte pas simultanément.

Le composant ELB (3) vous permet de répartir vos requêtes vers les deux datacenters. La haute disponibilité de ce service est directement assurée par AWS. Afin de l'utiliser pleinement, votre application doit cependant exposer une URL de health check (4). En fonction du retour de cette URL, le load balancer pourra bannir l'une de vos instances.

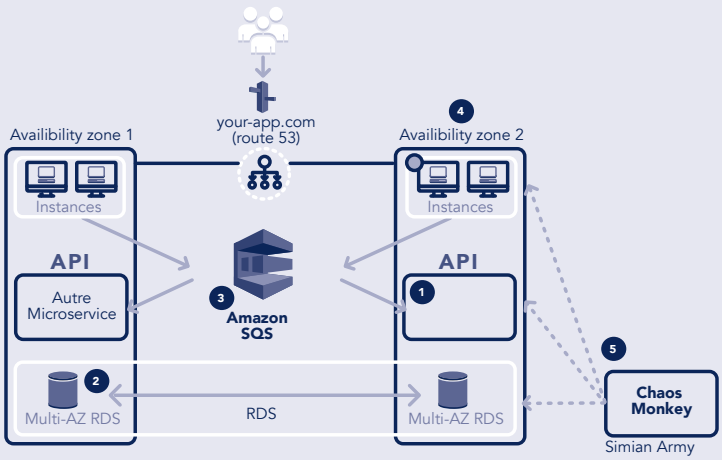
Le composant Route 53 permet d'offrir une garantie supplémentaire en étant capable de rediriger le DNS sur un autre continent où un déploiement similaire peut être réalisé.

Votre base de données est déployée sur RDS (2), une plateforme de Database as a Service, qui offre nativement un mécanisme de réplication.

Afin de garantir le découplage avec votre application, vous choisissez d'implémenter un pattern de communication asynchrone au moyen de Amazon SQS. Celui-ci offre une décorrélation parfaite au prix d'une plus grande complexité.

Après avoir tiré profit au maximum des possibilités des services AWS, l'implémentation de patterns sera nécessaire pour atteindre une disponibilité comme 99,99 %. Timeout et Circuit Breaker seront deux autres patterns applicatifs de haute disponibilité de type quick win.

Enfin, vous testez la robustesse de votre système en lançant un test grandeur nature (par exemple avec Simian Army (5)) qui va aléatoirement supprimer vos instances et s'assurer que l'application est toujours fonctionnelle.



1. API :
Limiter le couplage entre services via une API REST stateless est une bonne pratique car cela permet de tirer parti facilement des outils de haute disponibilité développés au-dessus de HTTP (load balancing, cache…).

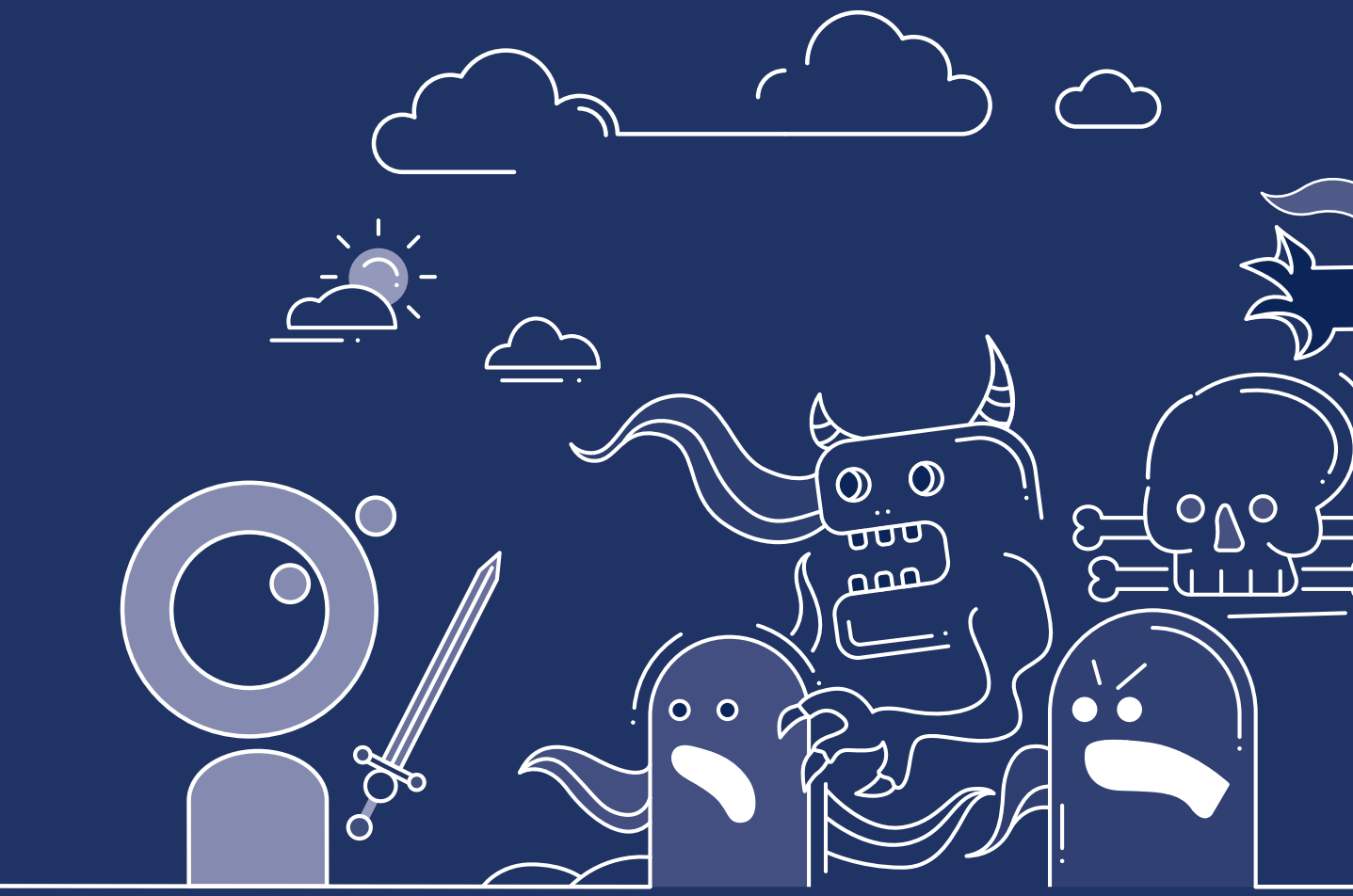
2. Database as a Service :
Les offres de cloud savent désormais proposer des services de haut niveau comme une base de données (service de type PaaS). En matière de haute disponibilité, la réplication des données et les sauvegardes peuvent être prises en charge par le service ce qui évite de devoir les ré-implémenter.

3. Asynchronous Data Exchanges :
Lorsque la sémantique métier l'autorise, reposer sur des échanges asynchrones permet de décorréler les deux services. Ces échanges asynchrones permettent par exemple un mécanisme de reprise sur erreur. Ce type de découplage peut être implémenté en s'appuyant sur un service de message queuing tel que AWS SQS.

4. URL de health check :
Un appel sur une URL de health check va exécuter un test pour s'assurer qu'un composant rend correctement le service qu'il est censé remplir. Une erreur retournée par une URL de health check sera notamment utilisée pour indiquer à un load balancer de ne plus envoyer de requêtes sur ce composant. Metrics propose une implémentation en Java https://dropwizard.github.io/metrics/, healthchecks propose une implémentation pour Express en Node.js https://github.com/broadly/node-healthchecks

5. Simian Army :
Ensemble d'outils open source de Netflix pour éprouver la stabilité de leur plate-forme sur AWS. Le plus connu est Chaos Monkey qui permet de tuer des serveurs de manière aléatoire. https://github.com/Netflix/SimianArmy

Garantir l'intégrité et la sécurité



Garantir l'intégrité et la sécurité

PLACER LA SÉCURITÉ AU BON NIVEAU

- Ramener la sécurité au plus proche du service et s'appuyer sur des systèmes de fédération pour pouvoir garantir la sécurité en environnement ouvert.
- Privilégier la propagation du commettant (principal propagation) ou la délégation d'autorisation à l'utilisation de comptes de services génériques.

Patterns : *Delegated authorization, principal propagation.*

CONSERVER TOUTES LES INTERACTIONS

- Faire de la traçabilité métier et technique une fonctionnalité de premier ordre du système. Enrichir les logs, les normaliser pour faciliter leur analyse.
- Prévoir par exemple des IDs de corrélation dans les échanges pour pouvoir retracer des interactions à travers plusieurs systèmes.

Patterns : *Event sourcing, Correlation ID.*

PENSER FIABILITÉ, INTÉGRITÉ ET SÉCURITÉ

- Utiliser à bon escient les techniques de chiffrement au niveau du transport, des données et des clés.
- Être flexible sur les entrées et strict sur les sorties (loi de Postel).
- Si un modèle de « cohérence à terme » doit être préféré aux transactions, veiller à en maîtriser les impacts sur l'intégrité.

Patterns : *Eventual consistency, Read Your Write consistency.*

Patterns

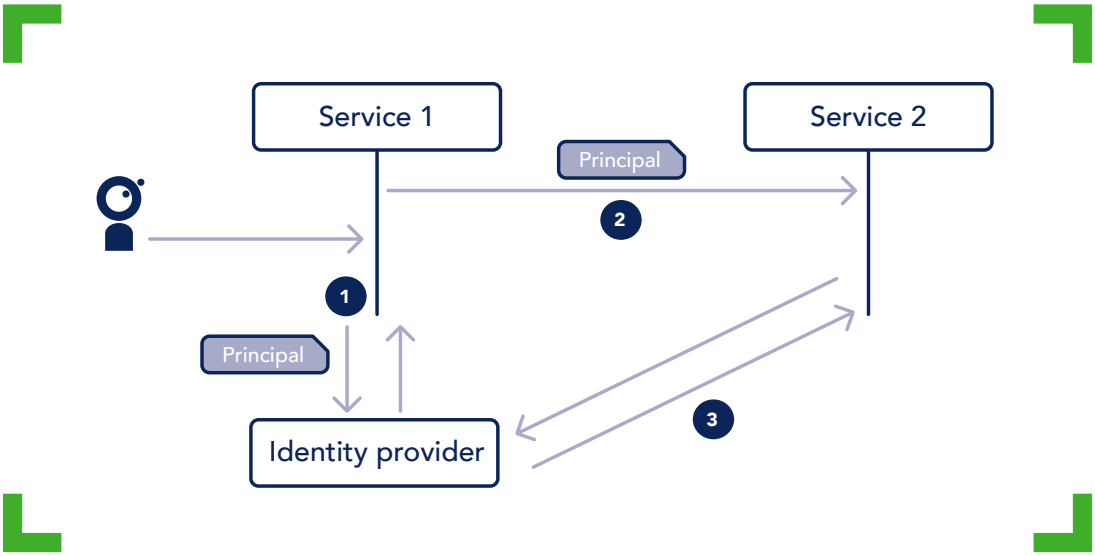
Loi de Postel ou principe de robustesse

Citation de Jon Postel l'un des pionniers d'Internet. « [Les implémentations de TCP] doivent suivre un principe général : être conservateur dans leurs implémentations, être libérale dans ce qu'elles acceptent ». Cela signifie que même si certains messages ne sont pas strictement conformes à la spécification, tant qu'il n'y a pas d'ambiguïté ils doivent pouvoir être traités. Dans le monde du cloud où les services utilisés sont très nombreux, cela réduit le risque d'erreur.

Principal propagation

Mécanisme sécurisé permettant à une application qui a authentifié un utilisateur de transmettre son identité (le principal) à un autre service.

1. Authentification.
2. Propagation du principal.
3. Validation du principal (ici il s'agit d'une délégation d'autorisation, le service 2 valide le principal sur l'identity provider).

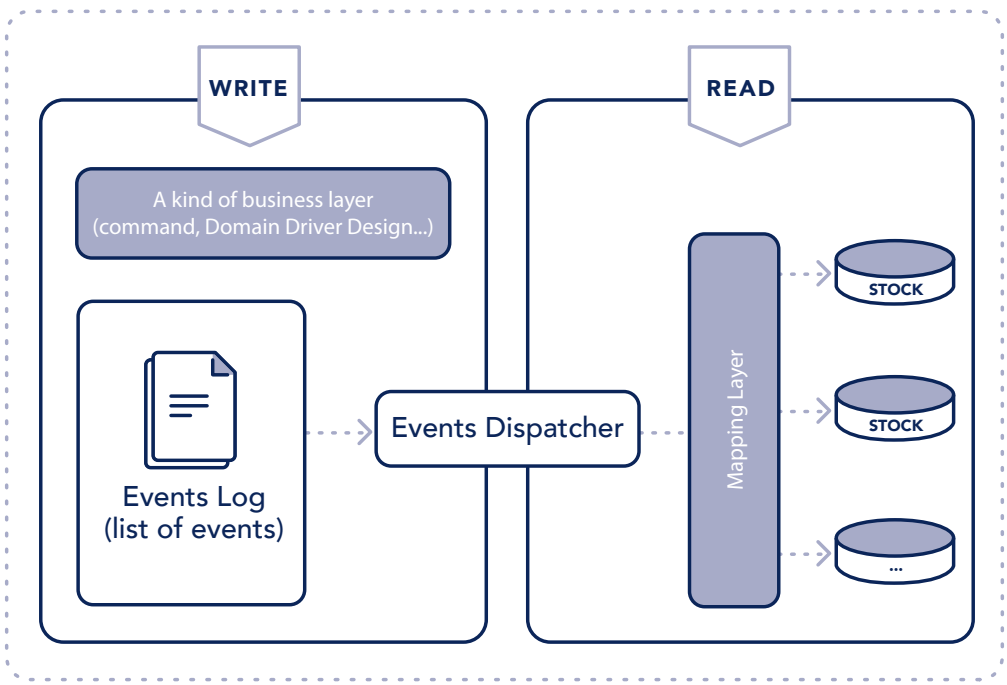


Delegated authorization (ou délégation de droit d'accès)

Mécanisme de délégation de droits (par exemple le droit d'accéder à vos emails) à un service tiers (offrant un nouveau type d'interface) par un mandant (celui qui détient les droits, en l'occurrence vous pour votre compte email). Par exemple, une application de blog propose un bouton pour tweeter sous votre nom. OAuth2 est une spécification permettant d'implémenter ce mécanisme.

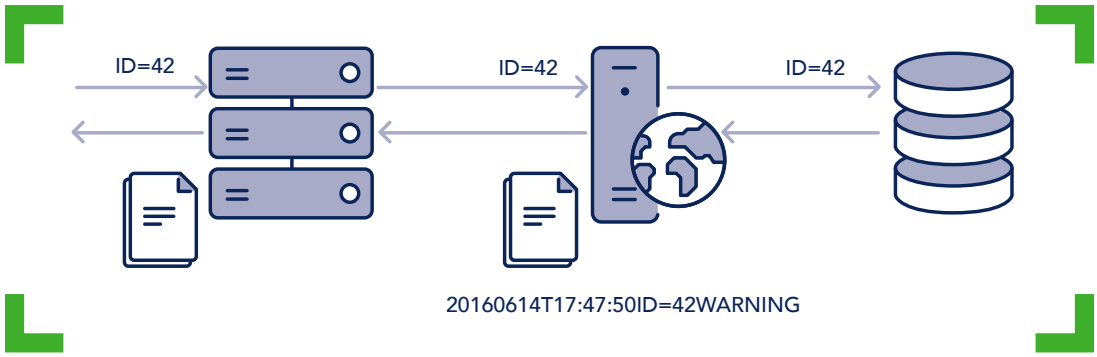
Event sourcing

Pattern applicatif dans lequel l'ensemble des modifications du système sont stockées sous forme d'évènements qui, une fois agrégés, donnent une vision consolidée. Conserver les évènements permet leur rejeu et leur analyse.



Correlation ID

Ajout d'un identifiant dans chaque requête pour suivre une transaction de bout en bout dans le SI.

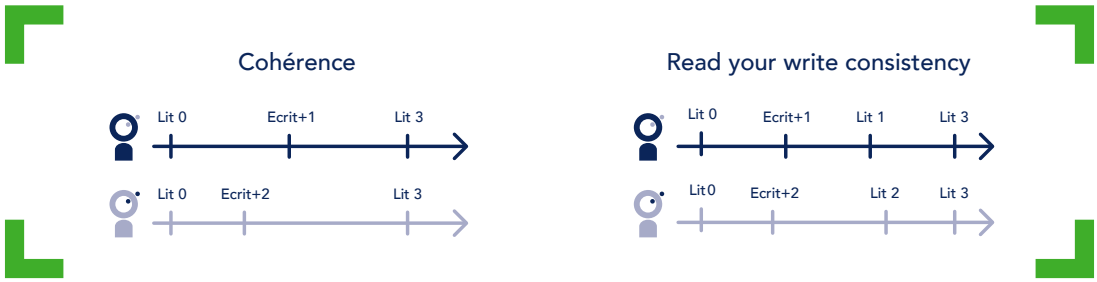


Eventual consistency

En français « cohérence à terme », désigne un modèle dans lequel la cohérence n'est garantie qu'à la « fin » – la « fin » étant relative à votre cas d'utilisation. L'exemple d'implémentation le plus connu de ce modèle est le protocole DNS dans lequel les adresses sont à jour une fois toutes les propagations terminées. Ce modèle est particulièrement important pour un stockage distribué dans lequel la cohérence ne peut pas être définie en permanence du fait des délais de propagation et des potentielles indisponibilités d'un nœud.

Read Your Write Consistency

Modèle de cohérence dans lequel les modifications ne sont visibles qu'à terme globalement, mais qui garantit que l'application à l'origine de la modification en voit les effets immédiatement.



Déployer facilement+ fréquem- men+


24
RELEASE

Déployer
facilement fréquemment

AUTOMATISER POUR ALLER PLUS VITE, SEREINEMENT

- Garantir la reproductibilité du déploiement en limitant l'intervention humaine et en éprouvant les processus sur des environnements comparables à la production, dès le développement.

Patterns : *Infrastructure as code, service discovery, containers.*

DÉPLOYER SANS ACCROC

- Garantir des changements de version plus transparents, jusqu'au « Zero Downtime Deployment » au besoin, pour limiter au maximum les impacts néfastes sur les utilisateurs finaux tout en apportant de la valeur.
- Partager les mêmes méthodes, outils et objectifs du développement à la production.

Patterns : *Tolerant reader, consumer driven contracts, multiversions services & backward compatibility, feature flipping.*

DÉPLOYER TOUT LE TEMPS

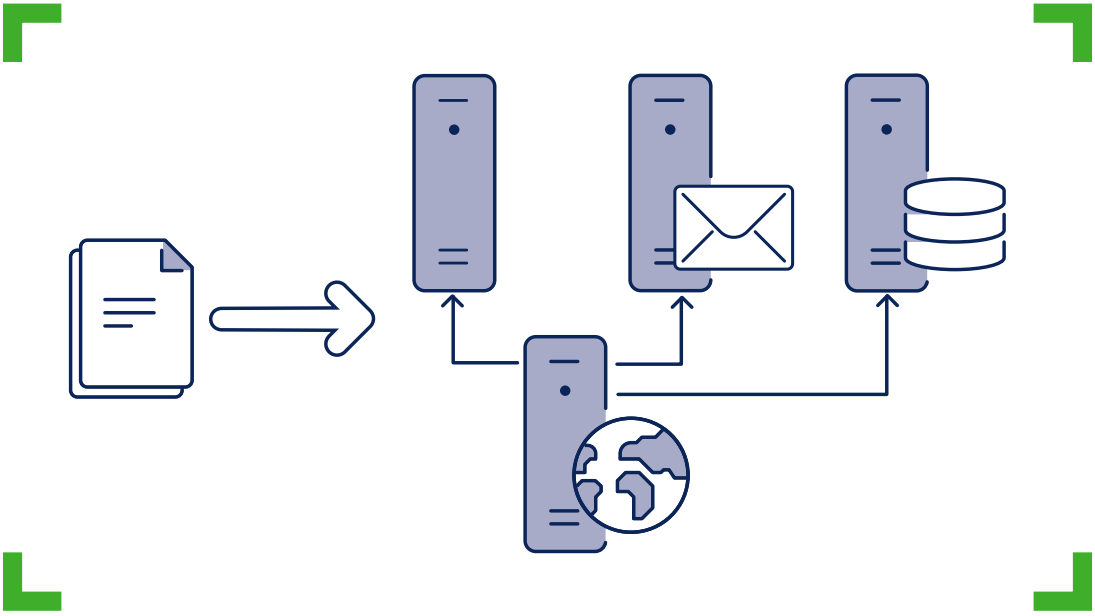
- Réduire le Time To Market et maximiser l'agilité en livrant les fonctionnalités par petites itérations fréquentes.
- Désacraliser les mises en production.

Patterns : *Infrastructure as code, strict separation of build and run stages, service discovery.*

Patterns

Infrastructure as code

Code décrivant l'infrastructure cible de manière répétable et automatiquement sans intervention humaine. Cette pratique offre une approche centralisée, réaliste et versionnable de l'infrastructure. Les fournisseurs de cloud proposent généralement leurs propres outils à l'instar de CloudFormation pour Amazon, qui viennent élargir un panel d'outils open source tels Ansible, Chef, Puppet ou encore Terraform.

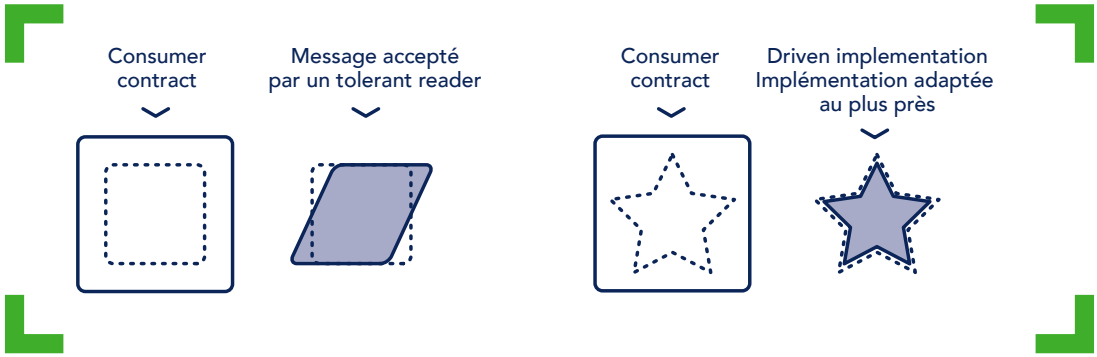


Service discovery

Mécanisme d'enregistrement et de découverte des services. Dans une infrastructure dynamique où les services naissent et meurent pour répondre à la charge, il est important de leur permettre de s'enregistrer dynamiquement pour être connus de tous. Plusieurs solutions sont répandues dans ce domaine telles que Consul, Zookeeper, ETCD ou encore Eureka qui est intégrée au framework Spring Cloud Netflix.

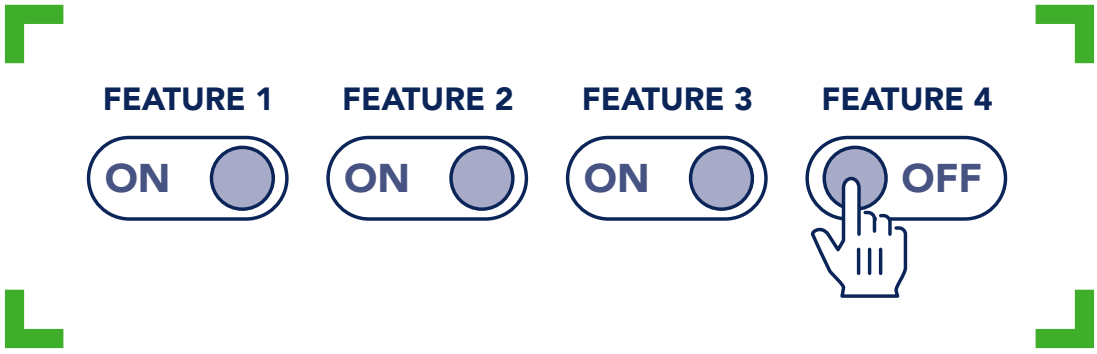
Tolerant reader & consumer driven contract

Le principe est de veiller à ce que le consommateur d'un quelconque système externe soit tolérant à l'évolution du schéma de données qui lui est exposé. Ceci consiste à prévoir des solutions palliatives en cas de valeur non prévue, de valeur absente ou encore ne respectant pas le format attendu (voir loi de Postel). Les navigateurs web par exemple sont très tolérants par rapport à des écarts à la norme HTML.



Feature Flipping

Pattern applicatif permettant d'activer et de désactiver une fonctionnalité à chaud sans interruption de service. Cela permet de déployer en production un composant sans en activer les nouvelles fonctionnalités (pour des raisons légales, pour les tester sur une population réduite, parce qu'elles ne sont pas finalisées, …). Des librairies permettent de faciliter la mise en œuvre telle que FF4J en Java qui apporte notamment une API et IHM pour contrôler l'activation des fonctionnalités.
http://blog.octo.com/feature-flipping/

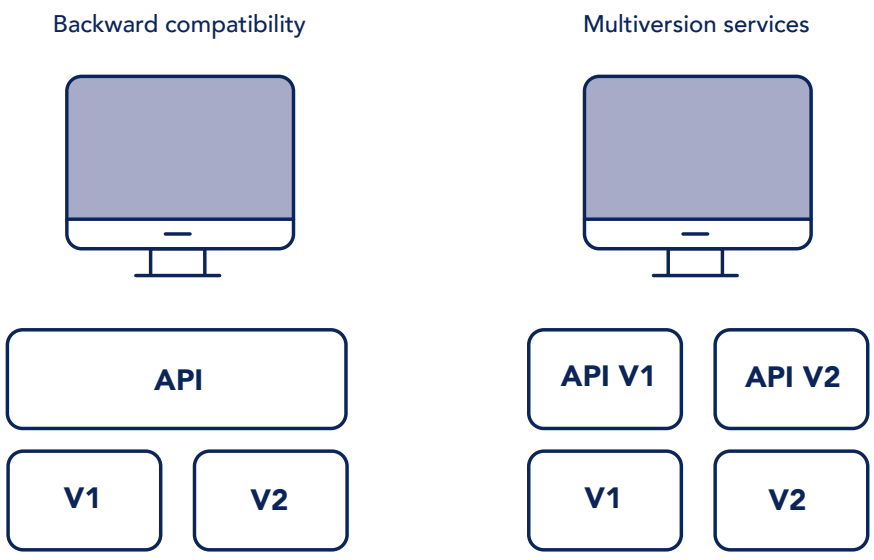


Backward compatibility

Gestion de version consistant à éviter toute modification qui empêcherait l'ancien client de fonctionner. Par exemple, les API utilisées par des clients mobiles natifs doivent souvent maintenir plusieurs versions car les différents devices ne se mettent pas tous à jour simultanément.

Multiversion Service

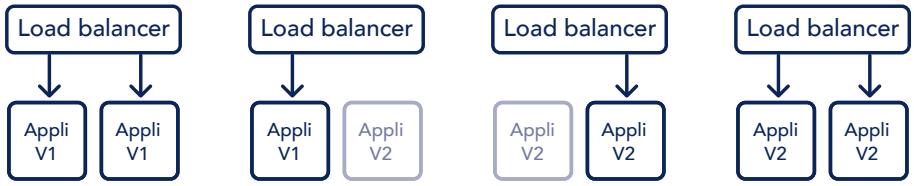
L'enjeu d'un service est d'être capable d'évoluer sans entraîner de régression pour les consommateurs existants. La mise en place d'un versioning sur l'API permet, lors d'évolutions majeures, d'exposer un nouveau service tout en maintenant l'existant. Ce pattern est lourd en termes d'implémentation et il faut veiller à ce que notamment les schémas de données restent compatibles pour les deux API. Attention à prévoir également un décommissionnement des versions au fur et à mesure pour pousser les consommateurs à évoluer et réduire la charge de développement associée.



Zero-Downtime Deployment

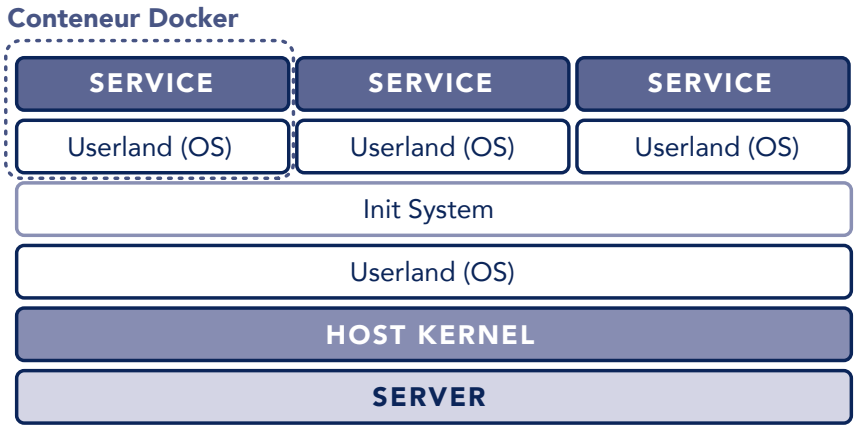
Déploiement d'une application sans interruption de service
http://blog.octo.com/zero-downtime-deployment/.

Exemple de procédé permettant le Zero downtime deployment



Conteneur

Un conteneur est une image d'une application et de ses dépendances. Un conteneur est plus léger qu'une machine virtuelle tout en permettant une isolation entre applications et une façon simple de déployer l'application avec toutes ses dépendances.



Docker Containerisation

Index des Patterns

NOUS APPORTONS *méthode*
et expertise POUR, ENSEMBLE,
concevoir et repenser
LES APPLICATIONS
SUR LE CLOUD
DU CODE A LA PRODUCTION.
NOUS **GARANTISSONS** AINSI,
sérénité et excellence
PAR DES APPLICATIONS **FIABLES,**
PERFORMANTES
ET CONÇUES POUR *tirer parti*
des opportunités du cloud.

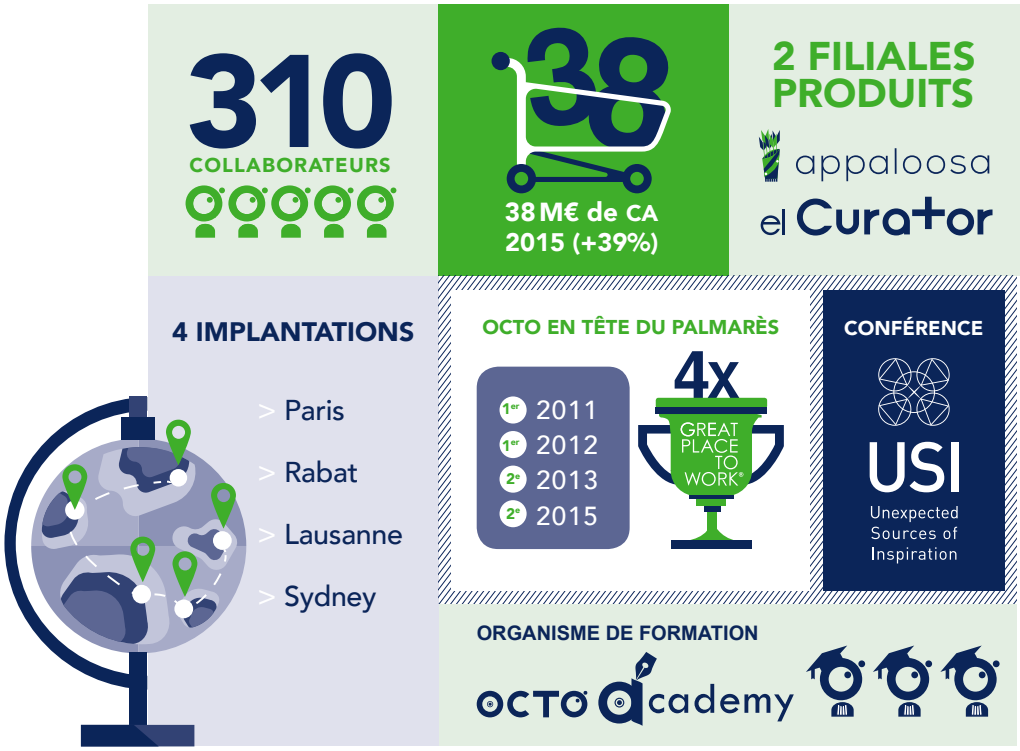
TRIBU CLOUD READY APP CHEZ OCTO TECHNOLOGY

OCTO Technology

CABINET DE CONSEIL ET DE RÉALISATION IT

« Nous croyons que l'informatique transforme nos sociétés. Nous savons que les réalisations marquantes sont le fruit du partage des savoirs et du plaisir à travailler ensemble. Nous recherchons en permanence de meilleures façons de faire. THERE IS A BETTER WAY ! »

– Manifeste OCTO Technology



Merci aux contributeurs :
Youri Antenor Habazac, Arnaud Bétrémieux, Marc Bojoly, Benjamin Brabant, Antonio Gomes Rodrigues, Edouard Perret, Borémi Toch.

Merci aux relecteurs :
Laurent Brisse, Laurent Dutheil, Christian Faure, Eric Fredj, Damien Joguet, Benjamin Joyen-Conseil, Benoît Lafontaine, Arnaud Mazin, Alexandre Raoul, François-Xavier Vende.

Merci à l'équipe communication :
Caroline Bretagne, Joy Boswell, Nelly Grellier.

Dépot légal : juin 2016

Conçu, réalisé et édité par OCTO Technology.
Imprimé par IMPRO
98 Rue Alexis Pesnon, 93100 Montreuil

www.octo.com - blog.octo.com

PARIS I RABAT I LAUSANNE I SÃO PAULO I SYDNEY