TECH/TRENDS

#7

1ÈRE ÉDITION

PERCEVOIR LE FUTUR

AJUSTER FLUIDIFIER VIVRE

TechTrends - Publication de Xebia IT Architects

Xebia

SOFTWARE DEVELOPMENT **DONE RIGHT**

Les TechTrends sont l'expression de notre savoir faire ; forgé sur le terrain, auprès de nos clients dans le cadre des projets que nous menons avec eux.

Fruit d'un travail collaboratif de nos consultants, vous y trouverez, nous l'espérons, les nouvelles tendances technologiques et méthodologiques ainsi que l'état de l'art de notre profession.

Nous tentons, dans le cadre de ces publications, de vous dispenser des conseils directement opérationnels afin de vous guider dans les décisions stratégiques que vous avez à prendre.

Distribués à plusieurs milliers d'exemplaires tous les ans, la collection des TechTrends s'étoffe régulièrement de nouveaux ouvrages.

Luc Legardeur, Président

AJUSTER

FLUIDIFIER

VIVRE

INTRODUCTION

Le système d'information, encore trop souvent perçu comme un centre de coût, est devenu depuis longtemps un organe vital de l'entreprise. Non seulement, il apporte de la valeur au quotidien, mais il reflète également **l'état de santé de l'entreprise** : un système d'information malade est synonyme de problèmes qui ne se limitent pas à l'informatique[1].

Afin de garder son entreprise compétitive, il est essentiel de faire évoluer les technologies de son SI et d'éviter d'entrer dans une spirale négative d'obsolescence de ses composants. Les utilisateurs sont de plus en plus exigeants sur **la rapidité de circulation de l'information** qu'ils doivent pouvoir consommer au travers de multiples canaux (web, téléphone, tablette, etc.). Nous vivons une accélération continue des besoins qui défie les capacités de nos organisations.

Avec de telles évolutions, le système d'information doit pouvoir s'adapter, et cela suppose d'assimiler ce que nous offre l'informatique d'aujourd'hui. C'est dès maintenant qu'il faut **préparer le SI de demain**.

À fonctionnalités égales, la mise en place d'une **architecture dite Microservices** et d'une organisation, ainsi que des choix technologiques qui s'y associent, devient un atout face à la concurrence en proposant plus de réactivité face aux nouvelles opportunités mais également plus de résilience.

Nous vous proposons un éclairage sur un ensemble de pratiques et de technologies "Back-end" qui vous permettront de faire évoluer votre système d'information pour répondre aux besoins d'aujourd'hui et le préparer à ceux de demain :

- **Ajuster** vos moyens à vos besoins sans perdre de flexibilité.
- **Fluidifier** les interactions de vos différents services pour améliorer votre résilience.
- **Vivre** sereinement la production avec les outils d'aujourd'hui.

---

1. Loi de Conway

AJUSTER

Un SI est un système complexe qui se développe et se transforme.

À la manière d'un être vivant, c'est un assemblage de plus petites unités qui fonctionnent de concert et se répartissent des rôles clés au sein d'un ensemble plus global. De nombreux styles d'architecture permettent de réaliser de tels assemblages. Néanmoins, le style monolithique est un triste standard de facto.

Pourquoi votre système d'information a-t-il besoin d'un régime ?

Les applications monolithiques gèrent un périmètre fonctionnel complet, couvrant l'ensemble des cas d'usage de toute une frange métier de l'entreprise.

Leur base de code, d'une taille parfois alarmante, en augmentation constante, freine grandement leur capacité à évoluer. Les changements, difficilement testables et dont les impacts globaux sont imprévisibles, sont perçus comme un risque. L'intégration des monolithes au sein du SI est souvent une gageure.

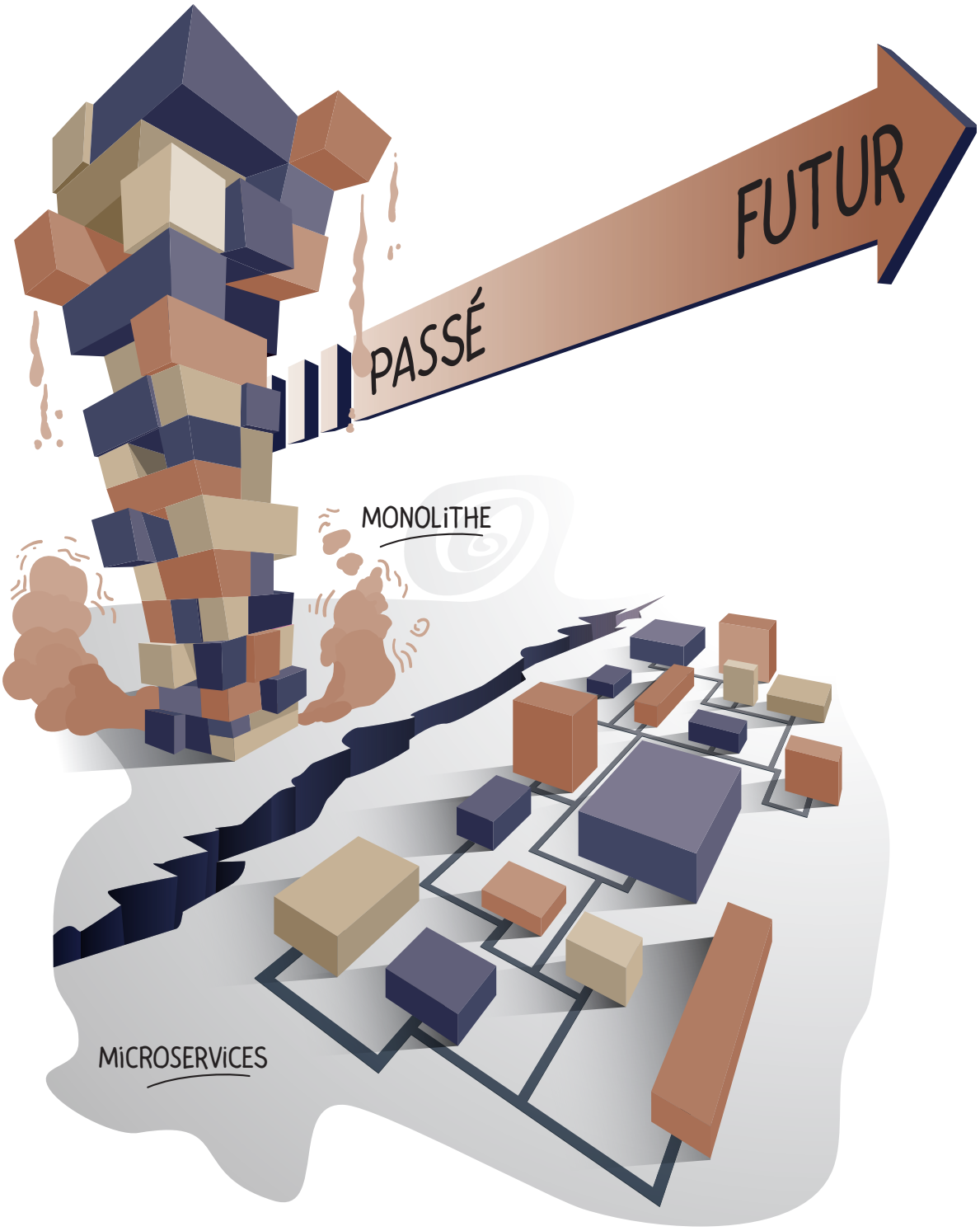
Ce type d'architecture engendre bien des problèmes :

- **La perte de la connaissance métier :** les projets au coeur du SI sont généralement des projets de longue haleine. Plus le périmètre est étendu, plus la difficulté pour un développeur à être opérationnel rapidement est grande. Dans un écosystème où les équipes se renouvellent à un rythme soutenu, conserver la maîtrise fonctionnelle, sans tomber dans le syndrome du développeur super-héros est un réel enjeu.
- **Des difficultés à refactorer :** une base de code unique qui accumule de la dette technique induit rapidement une incapacité à la refactorer dans de bonnes conditions. Le temps de développement d'une correction suit une courbe exponentielle au fur et à mesure de l'avancée du projet, pouvant aller jusqu'à la sclérose et à l'incapacité de réaliser les modifications demandées.
- **Jeter l'existant pour tout refaire :** dans les cas de blocages extrêmes, la tentation est forte pour repartir sur une base plus saine et réintroduire de l'innovation. Mais à quel prix ? Certes cette solution est envisageable, mais son coût est généralement extrêmement élevé et son issue plus qu'incertaine. Revenir au même niveau de fonctionnalités est souvent le but de ces refontes Big Bang, sans même se poser la question de l'adéquation avec les besoins métiers actuels. Durant la refonte, les évolutions fonctionnelles structurelles seront renvoyées aux calendes grecques et l'image retournée sera celle d'un projet qui n'avance pas et qui coûte cher. La concurrence n'aura peut être pas les mêmes difficultés et continuera à innover, captant de nouveaux clients qui vous échapperont, avec en sus le risque de ne pas obtenir un résultat plus pérenne à l'arrivée, et de refaire un tour de roue dans cinq ans.

- **Une augmentation des coûts de recette :** le code des applications de type monolithe ressemble souvent à un joli plat de spaghetti. Il est difficile de modifier une fonctionnalité sans risquer d'en casser une autre. Il faut donc réaliser des tests de non régression sur de larges pans de l'applicatif pour valider chaque évolution. Là encore, la courbe suivie est généralement exponentielle. Il est choquant de voir des recettes durer plus longtemps que les développements qu'elles sont censées valider.
- **Une incapacité à moderniser l'applicatif :** les bases de code uniques rendent plus difficile l'introduction de nouvelles technologies. Les langages et les frameworks techniques évoluent en permanence : changer ou introduire une technologie au niveau d'un monolithe peut facilement devenir une tâche insurmontable et entraîner la mise au placard d'évolutions essentielles à la vie du produit ou de la brique SI concernée. Cela peut rendre compliquée la « simple » mise à jour d'un framework, mais tourne au cauchemar lorsqu'il s'agit de remplacer purement et simplement une brique obsolète.
- **Des difficultés de mise en production :** une application monolithique, par nature, déploie en une fois un grand ensemble de fonctionnalités en production. Le cycle de vie de chacune de ces fonctionnalités est alors couplé au cycle de vie global de l'application. La mise en production est rarement sereine, car le périmètre et les impacts d'une potentielle erreur sont colossaux.
- **Des difficultés d'innovation :** l'ensemble des problématiques très techniques citées précédemment conduit insidieusement à un problème bien plus important pour l'entreprise : l'incapacité générale à répondre rapidement et de façon pertinente aux changements du marché. Développer et déployer une fonctionnalité urgente ou « coup de poing » (comme une opération marketing ponctuelle) conduit à prendre des risques insensés, ainsi qu'à créer une accumulation de verrues, parfois développées directement en production, dont le suivi et la réintégration dans la base de code est une douce utopie. Les Time To Market s'allongent et les responsables métiers dénigrent « ces techniciens qui les empêchent de faire correctement leur travail ».

Alors que faire ? Ajustons ! C'est-à-dire ne pas jeter pour tout refaire, mais **faire évoluer par petits morceaux**. Déconstruire l'existant pour le reconstruire autrement : identifier les différentes fonctionnalités pour les extraire et les isoler. Cette approche permet d'améliorer les capacités d'évolution et d'adaptation du code tout en garantissant une **meilleure maintenabilité et donc une meilleure qualité.**

Les architectures Microservices et les concepts qui les accompagnent soutiennent particulièrement bien cette approche.



Architecture Monolithique vs Microservices

Travailler avec une taille réduite

Le terme « Microservice » porte en son nom une idée de taille, en l'occurrence particulièrement réduite. Cette propriété nous guide vers un principe très important : **une taille raisonnée pour une complexité limitée**. La taille d'un Microservice n'est pas régie par des règles précises du fait de l'absence (souhaitable) de norme ou de spécification. Toutefois, un Microservice peut être défini comme une unité logicielle qui correspond à une **fonctionnalité précise, logique et cohérente** du système.

Un Microservice est donc **autonome**, assurant en totalité la fonctionnalité qu'il réalise. Il a **son propre code**, gère ses propres données et ne les partage pas – pas directement en tout cas – avec d'autres services. Il peut être, par exemple, un simple service d'auto complétion, un composant de gestion des comptes utilisateurs ou bien encore un service de notification.

> *Le Microservice repose sur un principe très important : une taille raisonnée pour une complexité limitée.*

Séparer les responsabilités

Chaque service doit avoir une responsabilité réduite. Une granularité plus fine induit un plus grand nombre de services, à la responsabilité limitée et surtout clairement identifiée. Cette séparation correspond à **un des principes SOLID : le "Single Responsibility Principle"**.

> *Chaque service doit avoir une responsabilité réduite.*

Gérer un cycle de vie simplifié

Chaque service s'exécute dans son propre processus. Ainsi, la modification du code ou des données relatives à une fonctionnalité provoque uniquement la mise à jour et le redéploiement d'un service car celui-ci s'exécute dans un contexte isolé.

Il n'y a pas d'obligation d'embarquer des mises à jour de fonctionnalités développées en parallèle car le code est séparé. Chaque service a **un cycle et une durée de vie qui lui sont propres**. Ainsi, la présence d'une anomalie relative à une fonctionnalité ne bloque pas l'évolution ou le déploiement d'une autre fonctionnalité. Selon le même principe, les données d'un service peuvent également être mises à jour, migrées ou traitées sans risquer d'impacter les autres briques du système.

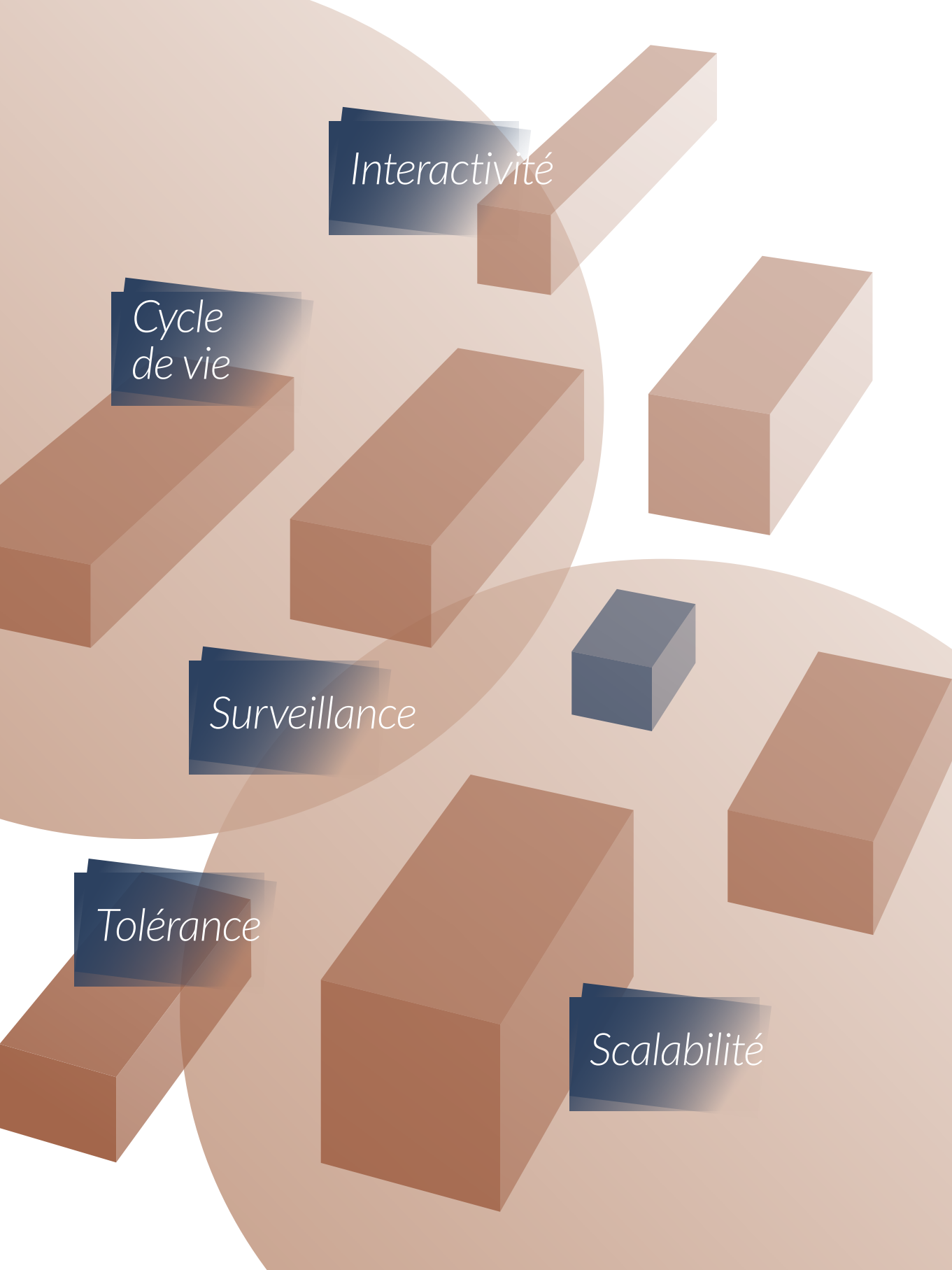
Avec des responsabilités limitées et des cycles de vie différenciés, il devient simple de se séparer d'un composant qui n'est plus utile ou devenu inadapté. Un service peut ainsi être suspendu ou bien réactivé indépendamment des autres, pour peu que l'indisponibilité de ce service soit prévue et gérée par les composants qui en dépendent (nous traiterons ce point dans la suite du document).

Innover sans risque

Le système est construit à une échelle qui permet d'en modifier une fonction sans avoir peur de créer des effets de bord sur d'autres fonctions. Dans cette architecture, l'échelle de la modification porte donc non pas sur un ensemble de fonctionnalités assemblées dans une application, mais sur la fonctionnalité portée par le service. Il devient alors plus facile de **tester localement** (par exemple via des techniques d'A/B testing) plusieurs scenarii fonctionnels sans mettre l'ensemble du projet en risque. De la même façon, introduire un nouveau processus (par exemple, mettre en oeuvre un paiement sécurisé type 3D secure, en remplacement d'un paiement classique) est une opération quasi anodine d'un point de vue technique, en dépit des lourds impacts fonctionnels qu'elle engendre.

Être scalable

S'agissant de processus isolés, l'augmentation ou la diminution des ressources allouées à un service peut se faire de manière indépendante. On peut choisir individuellement les services dont on veut modifier les capacités. L'unité de mise à l'échelle est bien réduite au service et non pas à une application ou un ensemble technologique de fonctions.


Interactivité
Cycle
de vie
Surveillance
Tolérance
Scalabilité

Rationaliser les communications

Les services sont autonomes mais doivent tout de même travailler de concert, car ils forment un système fonctionnel cohérent dans sa globalité. Les services doivent donc échanger des messages afin de déclencher, réguler ou inhiber les autres services.

La multiplication des services s'accompagne également d'une multiplication des canaux de communication. Prendre la voie des Microservices est une bonne occasion de rationaliser ces interactions à l'aide de solutions simples.

Aujourd'hui, les **architectures de type REST** supplantent le protocole SOAP en proposant une notion d'accès aux ressources plutôt que d'appels distants à des services. De même, la richesse des patterns de routage et de transformation des ESB a souvent donné lieu à des incompréhensions et, paradoxalement, à de grandes difficultés d'intégration. Aujourd'hui, on préfère s'appuyer directement sur un simple serveur de messaging pour se recentrer sur le besoin initial : transmettre des messages en offrant un découplage fort entre les communicants.

Améliorer la tolérance à la panne

> *Tout ce qui est susceptible de mal tourner, tournera nécessairement mal.*
>
> Loi de Murphy

Des services seront indisponibles, des données seront manquantes, peut-être même incohérentes. Des services peuvent être volontairement éteints, remplacés ou mis à jour. Tous ces aléas font la vie d'un SI. Ils se produiront. Plutôt que de chercher à se protéger, par le rejet, le modèle d'architecture Microservices **privilégie la tolérance et l'adaptation**. On parle de **"Design for Failure"**.

Chaque service est membre d'un écosystème où ses partenaires peuvent être indisponibles. Un service doit pouvoir adapter son fonctionnement à l'état du système dans lequel il évolue. Cela se traduit par exemple par une remontée d'alertes, un fonctionnement en mode dégradé ou encore une reprise sur erreur lors du retour à la situation nominale.

Cette acceptation de la panne va au-delà des dépendances entre services. Chaque service doit être conçu pour tolérer des formats de données qui évoluent. Un service doit pouvoir traiter

des messages qui comportent des données dont il n'a pas besoin. Si le contrat de communication entre deux services doit évoluer de façon incompatible, on peut envisager la coexistence de versions différentes du service pendant la période de migration des clients.

Mettre son système sous surveillance

La migration d'un SI vers une architecture Microservices doit être accompagnée de la mise en oeuvre d'un système de surveillance, d'autant plus important que les services sont nombreux.

L'ensemble doit être **en permanence monitoré**. Même si les services sont tolérants aux pannes, celles-ci peuvent compromettre le fonctionnement global. Il est donc important d'être capable de les détecter rapidement pour pouvoir intervenir efficacement : corriger une défaillance, redémarrer un service sur une autre machine, augmenter le nombre de ressources lorsqu'il y a un pic de trafic, une lenteur ou une saturation d'un service.

Il est également vital de **surveiller le système et les services d'un point de vue fonctionnel**, via le monitoring d'indicateurs métiers : transactions validées, en erreur, volume de commandes, nombre d'inscriptions, abandons de panier, etc. Tous ces indicateurs métiers sont les premiers révélateurs d'un problème. Ils peuvent également aider à distinguer les fonctions du système les plus sollicitées de celles qui sont inutilisées.

> *You build it, you run it!* [2]

Monter des équipes multidisciplinaires

Tout comme un service se concentre sur la réalisation d'une fonction, les équipes s'organisent autour de fonctionnalités métier et non pas de technologies. Cette organisation s'apparente aux **Feature Teams** : les équipes sont pluridisciplinaires et se complètent pour la réalisation d'un objectif commun, faire fonctionner un système.

La surveillance que l'on évoquait précédemment fait partie intégrante des responsabilités de cette équipe. Il ne s'agit pas d'un rôle porté par un département distinct et dédié. Le service est avant tout un produit, issu d'une entité qui se l'approprie, le réalise et l'opère – au sens "run".

2. Werner Vogels, CTO d'Amazon

TAKE AWAY BACK TRENDS

AJUSTER

- Faire un état des lieux des forces et faiblesses de votre architecture actuelle.
- Appréhender les principes des Microservices.
- Accompagner votre nouvelle architecture Microservices d'un système de surveillance et d'une équipe pluridisciplinaire.

FLUIDIFIER

La mise en œuvre d'une approche Microservices permet de fluidifier la réalisation d'une application, de sa conception à sa maintenance.

Cette mise en œuvre passe aussi par l'adoption de nouvelles pratiques et l'abandon de certaines autres devenues désuètes et contre-productives.

Penser à un découpage fonctionnel cohérent

Une organisation par domaines fonctionnels permet d'obtenir un découpage propre et logique d'une application. Par exemple, dans le cas d'un journal numérique, vous pourrez aisément diviser le programme en services ayant chacun une responsabilité, tels que :

- un service d'alimentation des informations à la une,
- un service de gestion des comptes utilisateurs,
- un service de notification des news,
- un service de résultats sportifs,
- un service de données météo pour alimenter les cartes,
- un service de gestion des petites annonces,
- etc.

Un (Micro)service correctement dimensionné regroupe **un ensemble de fonctionnalités d'un domaine**, dont on peut se faire un schéma mental de fonctionnement sans avoir à poser les détails sur papier.

Dans le cadre d'une nouvelle application, ce découpage pourra se faire dès le début du projet à travers **une approche DDD (Domain Driven Design)**.

Dans le cadre de la migration d'un existant, il faudra extraire les fonctionnalités existantes les unes après les autres pour les réorganiser par domaines fonctionnels et les migrer progressivement sur l'architecture Microservices cible. L'idéal est de commencer par les fonctionnalités les plus simples à isoler. Les services périphériques de l'application sont en général de bons candidats. Pendant cette phase d'extraction/migration, il est préférable, autant que faire se peut, d'arrêter l'ajout de fonctionnalités dans les monolithes existants.

> *Une organisation par domaines fonctionnels permet d'obtenir un découpage propre et logique d'une application.*

Isoler les fonctionnalités

Se doter d'outils adaptés

Une fois votre découpage fonctionnel réalisé, le choix des bons outils, nécessaires à la construction de votre système, est déterminant.

Par nature, une architecture Microservices ne nous contraint pas au choix d'une pile technologique unique. En effet, l'isolation permet de choisir les outils les plus pertinents à utiliser pour chacun des services. Cette flexibilité permet d'optimiser le développement, en utilisant les meilleurs outils à la disposition du développeur.

Cela va néanmoins à l'encontre de la volonté de standardisation exigée dans certaines entreprises qui considèrent que le développement doit être fortement cadré.

Les middlewares mastodontes tout-en-un (peu performants, complexes à déployer et coûteux à scaler) sont à délaisser, au profit **d'outils toujours plus légers, pragmatiques et répondant à un besoin ciblé**. À date, nous pouvons citer **Spring Boot, Vert.x et Node.js** pour leur légèreté, mais également **DropWizard** pour ses qualités de monitoring et d'exploitabilité, **Akka** pour sa composabilité ou encore **Finagle** (de Twitter) qui propose une normalisation des communications entre les différents composants applicatifs.

Concevoir des applications réactives

Fluidifier une architecture à base de Microservices implique une réflexion sur les principes de fonctionnement des programmes afin de les rendre réactifs. **Le Reactive Manifesto**[3] définit une application réactive comme étant :

- **Disponible :** un composant doit réagir rapidement et efficacement aux problèmes et répondre avec une faible latence.
- **Résiliente :** chaque composant est responsable de sa gestion d'erreurs, le système global doit continuer à fonctionner en cas d'erreur.
- **Élastique :** le dimensionnement des ressources est ajustable au besoin, ce qui facilite le côté « disponible » et permet d'éviter les points de contention.
- **Orientée messages :** une communication événementielle permet d'avoir beaucoup moins d'adhérence entre les composants. Le découplage temporel, par l'intermédiaire de files d'attente permet d'être plus résilient aux latences réseaux ainsi qu'aux indisponibilités.

3. reactivemanifesto.org

Développer selon les principes de ce Manifesto engendre des contraintes : par exemple, pour rendre les applications éligibles à l'élasticité, il faut en premier lieu s'astreindre à concevoir des applications sans état (stateless). Il est également nécessaire de s'éloigner des modèles d'applications multi-threadées pour favoriser des patterns de type Reactor qui permettent de multiplexer les traitements d'événements. Cela permet ainsi le traitement d'un grand nombre de demandes simultanées tout en diminuant la pression sur les ressources machines telles que la mémoire et le CPU.

Assurer l'interopérabilité et l'exposition des ressources

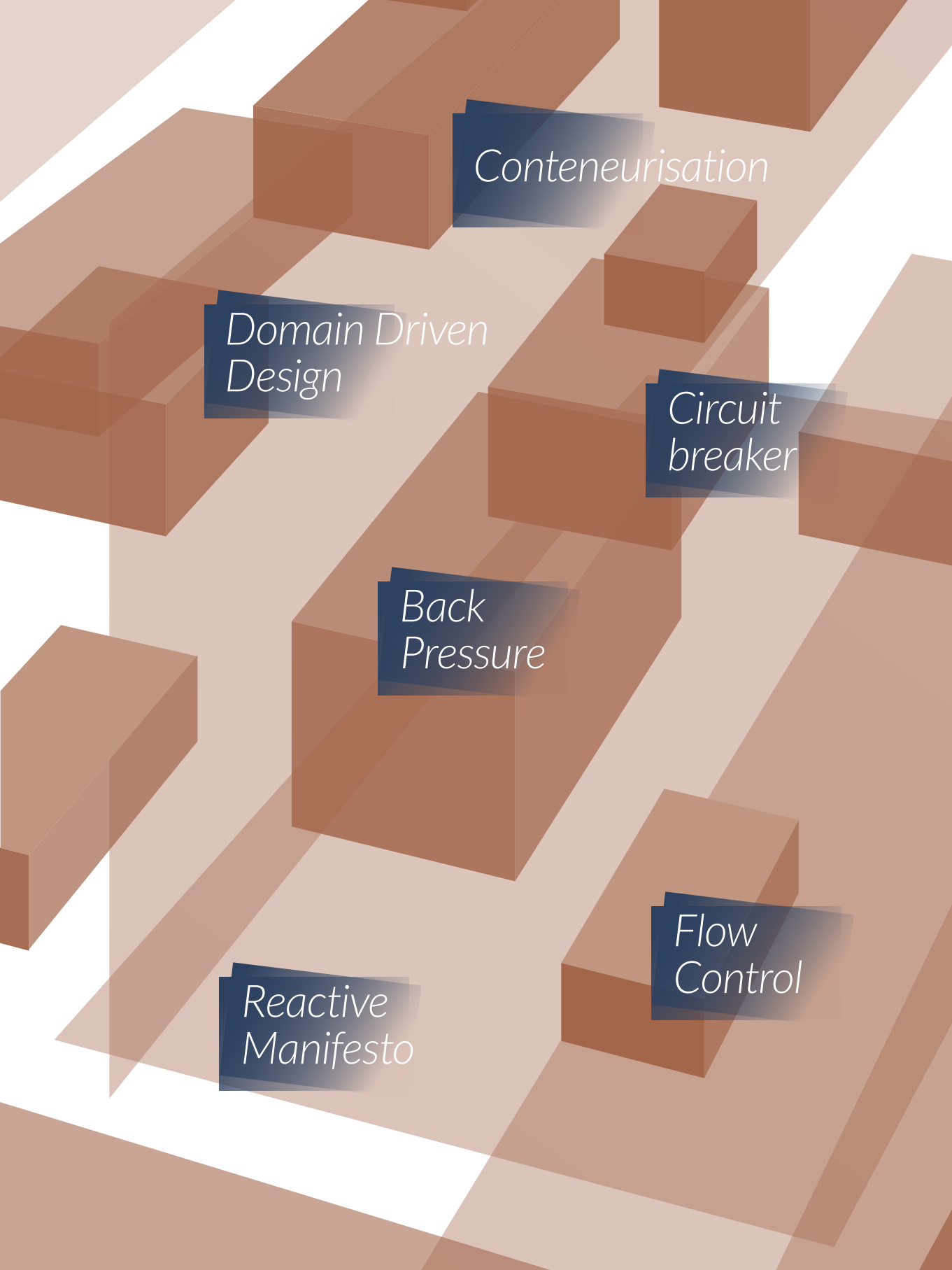
Les architectures issues du Web proposent des moyens de communication et d'échange de données adaptés à différentes situations. Ils ont été conçus pour être scalables, interopérables et ouverts. Les ressources qu'ils exposent sont adressables et découvrables. Au final, les besoins d'une entreprise ne diffèrent pas de ceux auxquels répondent ces **architectures (WOA, pour Web Oriented Architecture**). Ces dernières ayant fait leurs preuves de longue date, il semble cohérent d'aligner nos développements sur ces standards.

Ainsi, les applications gagneront à favoriser les formats de données tels que **JSON** (au détriment du XML), qui a l'avantage d'être parfaitement intégré avec les navigateurs Web. L'interopérabilité en est simplifiée.

Les **standards ouverts du Web** proposent des technologies viables dans le temps. Les intégrer en tant que fondation des développements applicatifs est la garantie de concevoir des applications faites pour durer.

Assurer le recensement, la documentation et la découverte des services

Dans une architecture à base de services, le référentiel de services appartient à une famille de composants destinés à ce que l'on appelle généralement **la gouvernance**. Il en est un élément incontournable. Traditionnellement, et de façon macroscopique, un référentiel de services dans une architecture distribuée comprend deux grandes catégories de fonctionnalités :


Conteneurisation
Domain Driven Design
Circuit breaker
Back Pressure
Flow Control
Reactive Manifesto

- Les fonctions de registre sont destinées à l'exécution des services et visent à faciliter le fonctionnement de l'infrastructure de services.
- Les fonctions d'annuaire, quant à elles, visent à consolider la connaissance des domaines fonctionnels.

Portés en son temps par les éléphants de la SOA, les services de référentiels trouvent une nouvelle jeunesse à travers de nouveaux outils tels qu'**etcd**, **Marathon** ou bien **Consul**. Habituellement, associés avec **CoreOS et Docker**, ils implémentent les notions de registre et d'annuaire qui permettent aux applications de « découvrir » comment accéder à un service donné ou bien les paramétrages à appliquer, voire même de s'abonner à certaines informations pour réagir à des changements opérationnels.

Maîtriser votre Flow Control

Rendre une **architecture asynchrone** apporte de nombreux avantages et tend à rendre chaque composant indépendant. Ceci n'est qu'en partie vrai : le système est dynamique et connecté, et s'adapte à ses différentes contraintes. Cependant les composants restent couplés (même si cela est minimal).

Quand un consommateur d'événements est trop lent vis-à-vis du producteur, un phénomène d'engorgement apparaît au niveau de la file d'événements. La file s'allonge continuellement jusqu'à atteindre un plafond (mémoire ou disque), impactant ainsi tous les composants qui en ont besoin. Le producteur finira donc lui-même par cesser de fonctionner. Pour pallier ce problème, il devient nécessaire de faire appel aux notions de **Flow Control et de back pressure** (un composant soumis à une forte charge doit avertir, via une boucle de feedback, les composants qui le maintiennent sous pression).

Plusieurs approches sont possibles. La première consiste à ralentir artificiellement l'émetteur pour soulager la pression. Ce mécanisme est bien connu des outils de messaging et des protocoles réseaux. ActiveMQ peut ralentir l'émission de messages lorsque la mémoire ou l'espace disque dépasse une certaine limite. RabbitMQ va même plus loin, en incluant par défaut ce contrôle de flux à toutes les étapes de ses traitements internes, quitte à brider les performances localement et ponctuellement pour maintenir un débit stable.

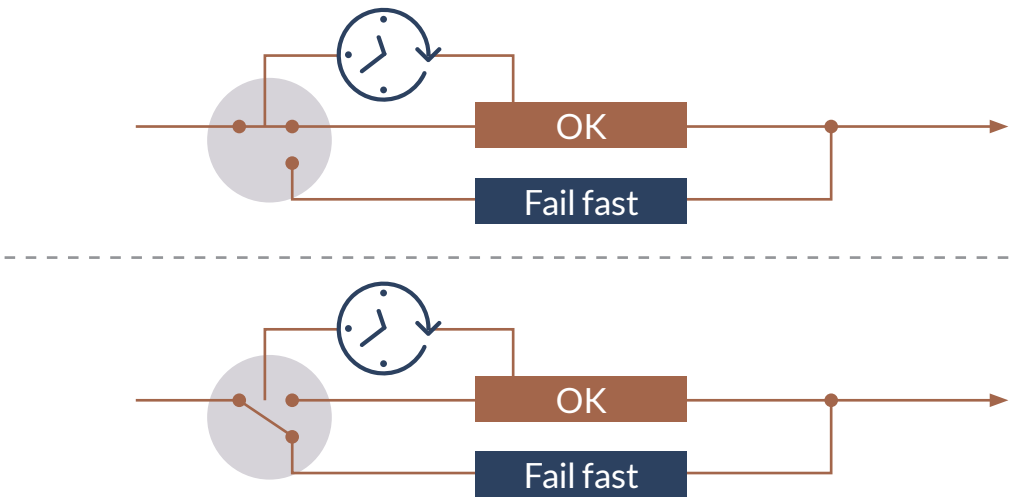
De son côté, l'API Reactive Streams promeut les notions de **back pressure asynchrone**. Typesafe, Red Hat, Twitter, Netflix ou encore Oracle participent activement à cette initiative. Elle a pour ambition de fournir un cadre concret à ces problématiques. Les concepts de Flow Control sont

ici plus riches que de simplement ralentir l'émission (ce qui pourrait avoir pour conséquence de reporter le problème plus en amont). Il existe entre autres plusieurs stratégies (par priorisation, abandon de message, débordement sur disque, etc.) pour éliminer ou déprioriser une partie des messages.

Il faut avant tout retenir que dans des systèmes de plus en plus distribués où les flux de données jouent un rôle central, il est nécessaire d'avoir une vision globale du trafic entre les différents composants. Les problématiques bien connues au niveau des réseaux (engorgement, pertes de packet, buffering, etc.) se retrouvent également au niveau applicatif. Il est important de les prendre en considération dans l'architecture du SI.

Se protéger avec un circuit breaker

Les architectures fortement distribuées, si elles sont bien pensées, améliorent la capacité de résilience du système mais peuvent engendrer un surcroît d'activités opérationnelles. Les exécutions de plans de reprise d'activité (PRA) deviennent alors des opérations très délicates. Lors d'incidents de production, on se repose sur des actions humaines pour analyser et prendre des décisions (retirer un serveur, le redémarrer, désactiver un service, faire passer un batch, etc.). **Sans automatisation, les procédures se multiplient et deviennent difficilement testables**. Certaines, obsolètes, perdureront, alors qu'en parallèle la connaissance se perdra.



Concept du circuit breaker

Pourtant, les applications sont elles-mêmes les meilleurs juges des choix à faire lorsqu'un partenaire adopte un comportement suspect ou disparaît. Par exemple, quand une latence apparaît sur un back-end, l'application peut décider de ne plus l'appeler pendant un certain temps et proposer un mode dégradé ou retourner une erreur à son propre appelant.

Hystrix, un outil de Netflix, implémente ce type de pattern : basculer sur des services de compensation, lorsque certains critères ne sont pas respectés. Cela force le développeur à considérer les situations hostiles à l'application et à garantir sa stabilité, quitte à ne pas être complètement fonctionnel.

À un niveau plus opérationnel, des **load balancers comme HAProxy** permettent de sortir ou de faire revenir des instances en fonction des réponses des health checks (pour un service Web, on utilise une url pour vérifier que le service est "up"). Moins intrusive au niveau code, cette solution n'évite pas les effets domino. Quand une instance tombe, la charge bascule sur d'autres instances qui à leur tour pourraient se mettre à bégayer et finir par impacter l'appelant et ainsi de suite.

Travailler en amont des projets sur ces sujets permettra d'anticiper bien des problèmes et l'automatisation de certaines opérations apportera de la sérénité.

Conteneuriser ses Microservices

Issue de travaux anciens sur le système d'exploitation Unix (les C-groups), la conteneurisation (qui permet de limiter et d'isoler l'utilisation des ressources de type processeur, mémoire, disque, par différentes applications sur une même machine) arrive aujourd'hui à maturité, avec la mise au pinacle du projet **Docker**.

L'utilisation massive de ces technologies dans une architecture Microservices offre un avantage indéniable. À chaque service correspond une **image qui est livrée sur un « catalogue » d'entreprise.** Cette image 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 Microservice.

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 est ainsi accélérée. Elle est **portée directement par l'usine logicielle** et ne nécessite plus un déploiement sur un environnement iso 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, qui impacte toutes les couches du SI, doit donc se faire de manière concertée.

TAKE AWAY BACK TRENDS

FLUIDIFIER

- Isoler les fonctionnalités de vos applications.
- Adopter de nouvelles pratiques d'ingénierie.
- Concevoir des applications réactives.
- Utiliser les standards du Web.
- Conteneuriser.

VIVRE

À l'heure où le nombre d'utilisateurs et la diversité des périphériques utilisés pour se connecter aux services sont en constante augmentation, les systèmes doivent être conçus pour être disponibles à tout moment.

La mise en œuvre d'une approche Microservices ne dispense pas de se préparer à l'exploitation du système en production afin de la vivre sereinement.

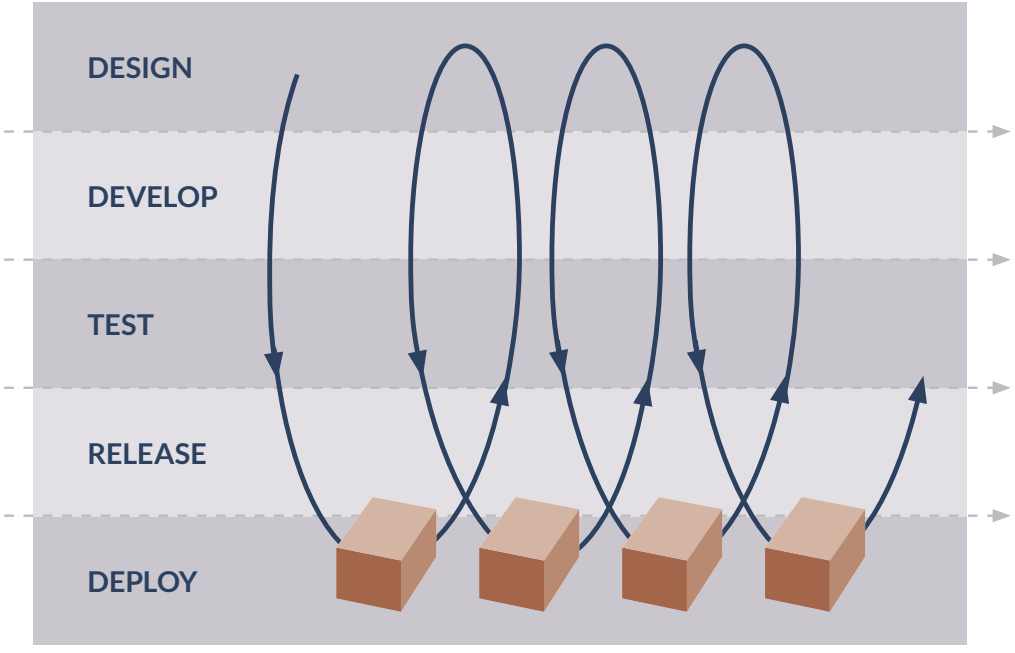
Moderniser le déploiement

Multiplier les composants, les technologies et les services a nécessairement un impact sur la livraison et le déploiement des applications. C'est pour cette raison, entre autres, que la mise en place d'une architecture Microservices doit s'accompagner de la mise en œuvre d'une **chaîne complète et intégrée de livraison et de déploiement en continu** (Continuous Delivery & Continuous Deployement).

En effet, là où par le passé les opérationnels ne déployaient que quelques artefacts, une application architecturée en Microservices nécessite le déploiement d'une myriade de composants isolés. Dans ce contexte, la répétition de tâches manuelles, déjà fastidieuse et source d'erreurs, n'est plus envisageable. Il faut donc viser la suppression de toute action humaine des processus de déploiement.

Cette automatisation des procédés de Delivery et de Deploy passe par :

- **L'intégration de profils Ops** au sein des équipes pluridisciplinaires afin que le Delivery et le Deploy fassent partie intégrante des cycles de développement.
- **L'utilisation de conteneurs** afin de standardiser les déploiements indépendamment des technologies retenues pour chaque brique.



L'intégration continue

Adapter la capacité

Estimer la capacité nécessaire au bon fonctionnement des différents services du SI ou d'une application est une étape incontournable et doit intervenir le plus tôt possible. Dans le cadre d'un système Microservices, l'application de règles théoriques (telles que « la loi de Little ») atteint ses limites.

Une fois la scalabilité du système assurée, le moyen le plus sûr de garantir le bon dimensionnement de la capacité reste la mise en œuvre de **tests de performance** et de charge intégrés au cycle de développement.

Assurer la scalabilité

Dans le cadre d'une application réactive, la scalabilité est adressée par **la notion d'élasticité** : il faut pouvoir provisionner des ressources à la demande.

Côté « salle machine », le **Cloud** a montré la voie : qu'il soit public, privé ou hybride, le provisioning de serveur « à la demande » est à la base de ces infrastructures élastiques.

Au niveau architecture, la réflexion doit être étendue aux middlewares sur lesquels elle repose. En effet, un middleware qui n'est pas scalable limitera mécaniquement la scalabilité de l'ensemble.

Mettre en place des tests de performance et de charge

Les tests de performance permettent de **mesurer la robustesse et les temps de réponse** d'un système applicatif en fonction de sa sollicitation. Ils chronomètrent chacune des étapes grâce à des sondes techniques positionnées sur les composants de l'architecture (base de données, serveur d'application, accès disque, réseau, etc.). Les informations ainsi recueillies permettent de corréler les temps de réponse utilisateurs avec les temps de réponse techniques : affichage navigateur, réseau, serveur(s), etc.

Pour les implémenter, il n'est pas nécessaire d'avoir une application complètement opérationnelle. Il suffit, dans un premier temps, d'isoler les services qui seront les plus utilisés ou qui auront le plus de valeur ajoutée, de les bouchonner si besoin et d'observer leur comportement sous forte charge.

Pour ce faire, **Gatling**, écrit selon les principes réactifs, est capable de générer des charges importantes avec peu de ressources. De plus, il s'intègre parfaitement aux outils du développeur Java (Maven, Jenkins, etc.) et donc dans l'usine logicielle. Pour compléter, il est possible de pratiquer du **microbenchmarking**. Cette technique consiste à tester directement les performances d'un snippet de code source. JMH, un outil OpenJDK récemment apparu, vient ainsi compléter notre trousse à outil.

C'est généralement sous charge que l'on mesure le mieux la résilience globale d'un système. Des **campagnes régulières de tests de charge**, simulant un nombre progressif d'utilisateurs en fonction de scenarii précis, afin de valider le comportement de l'application pour une charge attendue, fourniront des informations précieuses et permettront de détecter au plus tôt de mauvais comportements. Ils **mettront en évidence les points sensibles et critiques** de l'architecture technique.

Rendre votre application résiliente

La résilience est la **capacité d'un système à résister aux différentes pannes** qui ne manqueront pas de survenir. L'idée n'est pas de se défendre à tout prix contre la panne (d'un service, d'une application, d'un serveur voire même d'un datacenter), mais plutôt de l'accepter et de prévenir en conséquence.

Netflix possède d'ailleurs à ce sujet un raisonnement à l'extrême : puisque la panne surviendra, et que l'architecture doit s'adapter en conséquence, pourquoi ne pas la provoquer volontairement pour tester, en production, la résilience du système ? C'est l'idée qui préside à la destinée du projet Chaos Monkey.

> *L'idée n'est pas de se défendre à tout prix contre la panne, mais plutôt de l'accepter et de prévenir en conséquence.*

Résilience des services d'un journal numérique conteneurisés

Créer un système distribué et consistant

> *À la notion de répartition s'ajoutent deux concepts essentiels : la réplication et le partitionnement.*

Résister à la panne physique ou logique d'un élément passe souvent par une distribution et une redondance des processus, à tous les niveaux de l'infrastructure : machine, cluster et datacenter.

Au niveau du processus sur la machine, **les processus légers**, **monothreadés**, qui peuvent redémarrer rapidement en cas de crash, sont à privilégier. On citera par exemple le populaire Node.js.

Au niveau du cluster de machines, il existe deux types d'approches :

- **Le classique master/slave** a l'avantage de la simplicité. Il est actuellement en perte de vitesse dû à un temps de recovery allongé et une limitation de la scalabilité.
- Le plus **récent multi-master** (ou masterless) a pour lui la scalabilité et la robustesse, mais induit une plus grande complexité de mise en place, en particulier au niveau des échanges réseau nécessaires pour garder la cohérence globale du cluster.

Au niveau du datacenter, il est possible de mettre en place des **load-balancers globaux**, ou des CDN, qui permettent de se prémunir de la chute d'un bâtiment entier. La solution est globalement coûteuse car elle oblige à maintenir a minima deux infrastructures complètes. Les hébergeurs cloud, qui ont pour la plupart des solutions multizones, permettent de régler plus rapidement ces difficultés.

Ces trois niveaux ne s'excluent pas l'un l'autre mais au contraire s'additionnent en fonction du niveau d'exigence sur la performance, la disponibilité et la robustesse de l'ensemble.

À cette notion de répartition s'ajoutent deux concepts essentiels permettant d'affronter les différentes perturbations (comme la perte d'un ou plusieurs noeuds par exemple) :

- **La réplication** : les données sont répliquées d'un noeud sur N autres noeuds pour garantir la disponibilité de ces données en cas de panne.
- **Le partitionnement** : lorsqu'un seul noeud ne peut porter toute la donnée, on la partitionne sur différents noeuds, chacun formant alors un sous-ensemble logique des données.

En parallèle, définir un gendarme garant de l'état du système reste indispensable. Celui-ci sera capable de détecter les changements de topologie et de les restituer à l'ensemble des noeuds tout en étant lui-même hautement disponible et résilient. On parle de **Single Point of Truth**. Zookeeper est à date l'une des solutions les plus connues et les plus utilisées pour répondre à ce

besoin. Apparus plus récemment, etcd et Consul sont porteurs d'une alternative crédible basée sur l'algorithme de consensus distribué Raft.

Le choix des outils permettant d'implémenter cette résilience dépendra des stratégies choisies pour répondre aux exigences du système :

- **Failover / Failback** : quand un noeud tombe, est-ce que le flux est routé sur les autres instances ? Lorsque le noeud revient, l'instance est-elle à nouveau utilisée ?
- **Split brain** : lorsqu'un ensemble minoritaire de noeuds ne peut plus communiquer avec les autres noeuds, les écritures sont-elles bloquées ou existe-t-il des processus pour fusionner a posteriori les ensembles ?
- **Recovery disaster** : existe-t-il des processus pour récupérer, même partiellement, un ensemble de données perdues et quelle sera la durée d'indisponibilité ?

Les réflexions autour de ces stratégies doivent être menées suffisamment tôt dans le cycle de vie d'un projet car elles impactent fortement les choix technologiques, l'architecture globale et les procédures de maintenance.

Gérer les caches

Lorsqu'une application est amenée à gérer une grande quantité de données transientes, les systèmes de cache permettent de soulager la mémoire des processus ainsi que la charge CPU, et d'atteindre un meilleur taux de disponibilité. Attention néanmoins à ne pas déplacer le problème et devenir dépendant de la disponibilité des caches : vos caches doivent être un moyen de soulager les applications, pas de les contraindre.

Pour les caches, le **choix du produit est important**. Certains logiciels open-source, comme Memcached, Varnish ou Redis sont particulièrement adaptés. Attention à ne pas retomber dans le travers de l'outil universel : la tentation de détourner l'utilisation des bases de données NoSQL, comme MongoDB, est souvent grande. Cependant un tel détournement est dangereux, notamment sur des scenarii d'écritures massives.

> *Le stockage des données est bien sûr capital, mais leur exploitation et leur visualisation le sont tout autant.*

Superviser

La multiplication des composants amène un autre défi de taille. Bien que l'architecture soit prévue pour être hautement disponible, scalable et auto-adaptative, elle ne résistera pas éternellement. Il faut donc être en mesure de détecter les dysfonctionnements et d'intervenir.

Utiliser des Time Series Databases

Les bases de données de type Time Series sont conçues pour **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 de métriques bien connues comme **le CPU, la mémoire, l'espace disque, les I/O**, etc. Les développeurs y portent un intérêt croissant, en particulier pour stocker des métriques applicatives, techniques voire métier.

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 (100 % retro compatible avec Graphite), talonné de près par OpenTSDB.

Le stockage des données est bien sûr capital, mais leur exploitation et leur visualisation le sont tout autant. Aucune des bases de données nommées ci-dessus ne proposent de système d'alerting. En revanche, elles se couplent très facilement aux outils classiques d'exploitation, comme Nagios, Shinken, Icinga ou Sensu.

Les **outils de dashboarding** basés sur Graphite sont légion et leurs évolutions récentes permettent de se brancher facilement sur InfluxDB. Notre préférence actuelle va au projet Grafana.

Monitorer les temps de réponse utilisateurs

Tracer les temps de réponse d'un point de vue utilisateur est capital pour améliorer l'efficacité d'un système, notamment dans le cadre d'un site web grand public. En effet, l'expérience utilisateur est directement liée au temps d'affichage d'une page. Le **RUM (Real User Monitoring)** est un outil de prédilection pour une mise en situation. Il est souvent utilisé dans le cas de l'A/B testing pour comparer concrètement différentes alternatives.

Plusieurs solutions clés en main, comme New Relic, existent sur le marché pour mettre en place le RUM sur votre système.

Cependant, dans le cadre d'une architecture Microservices au sein de laquelle une chaîne de traitement pour la centralisation et la normalisation des logs est déjà en place, il est aisé **d'implémenter son propre système**. Les informations idoines contenues dans les "access log" des serveurs http seront acheminées vers les outils d'analyse déjà en place. Des frameworks comme Zipkin (inspiré de Dapper) aident à implémenter un monitoring distribué.

L'objectif est d'obtenir une vision claire du temps de réponse utilisateur et de sa décomposition en entrées / sorties des différents composants.

Sécuriser

La sécurisation des systèmes et la protection des données associées sont plus que jamais **un enjeu crucial** à l'heure où les attaques sur les SI sont toujours plus nombreuses et sophistiquées. Au-delà de la mise en oeuvre de bonnes pratiques classiques, les architectures Microservices sont une occasion de **renforcer la sécurité des systèmes de par l'isolation des différents composants** du système.

Isoler via la conteneurisation

La conteneurisation participe à la sécurisation des systèmes car elle renforce l'isolation entre les différents composants. D'autre part, le fonctionnement dans un conteneur **réduit les surfaces d'attaques** car le nombre de binaires disponibles sur le système, le nombre de processus qui s'y exécutent et le nombre de ports ouverts sont minimaux. Enfin, la conteneurisation facilite **la mise à jour des environnements** et permet de préparer à l'avance les mises à jour pour un déploiement rapide.

Sécuriser avec HTTPS

L'actualité récente prouve une fois de plus que la sécurisation des données et des échanges n'est pas une option. De nombreux acteurs ont déjà franchi le pas et décidé de **ne proposer que du HTTPS** à leurs clients.

Bien entendu, passer au tout HTTPS n'est pas sans coûts ni difficultés. Mettre en place des terminaisons SSL pour de gros volumes de connexions requiert une **puissance de calcul plus importante** et nécessite donc des investissements machines pour aligner les capacités avec les ambitions. Là encore, ce coût CPU peut être déchargé sur des infrastructures Cloud. Par exemple, la solution ELB (Elastic Load Balancer) d'Amazon Web Services prend en charge le traitement des terminaisons SSL et offre une scalabilité théoriquement infinie.

Les configurations SSL doivent évidemment être maintenues à jour le plus régulièrement possible. Récemment, des failles majeures telles que HeartBleed ont prouvé que le chiffrage est un rempart important, mais pas inviolable. Les équipes doivent être sensibilisées au sujet et appliquer les bonnes pratiques relatives à la sécurité.

Respecter OWASP

La communauté maintient et partage plusieurs référentiels de bonnes pratiques visant à sécuriser les systèmes, applications et données échangées. Appliquer ces bonnes pratiques permet de réduire le spectre des risques.

Par exemple, le projet OWASP[4], pour **Open Web Application Security Project**, propose **un top 10 des failles de sécurité** ou faiblesses les plus courantes et explique comment les traiter. Il met également à disposition de nombreuses ressources, aussi bien du code que des guides pour améliorer la sécurité des applications.

Mettre en place le Passwordless Authentication

Quels que soient les systèmes techniques de sécurité mis en œuvre, un des premiers vecteurs d'attaque sur les systèmes reste l'utilisateur et ses mauvaises habitudes : utilisation de mots de passe faibles, utilisation d'un même mot de passe sur plusieurs systèmes, mauvaises réactions face au phishing, etc.

Une façon radicale de protéger l'utilisateur contre lui même consiste à supprimer complètement l'utilisation de mots de passe. C'est ce qu'on appelle le Passwordless. Le principe est simple : **envoyer un mot de passe à usage unique à chaque connexion sur une application**, via email, SMS ou dongle dédié.

Bien entendu, intégrer de telles pratiques impose de réinventer certains fonctionnements applicatifs, mais ils bénéficieront aussi bien aux utilisateurs de vos applications qu'à l'image de vos produits.

---

4. owasp.org

TAKE AWAY BACK TRENDS

VIVRE

- Se préparer aux mises en production.
- Intégrer les grands principes de supervision, de résilience et de sécurité.
- Monitorer et tester votre architecture.
- Sécuriser son système.

CONCLUSION

Transformer et moderniser les applications de nos SI est un challenge qu'il faut relever pour pouvoir répondre à des attentes toujours plus grandes. Choisir de ne pas moderniser son SI et laisser de côté les innovations destinées à le rendre plus réactif est un jeu dangereux. Il est donc essentiel **d'adopter le principe d'innovation continue** en acceptant de s'éloigner de certaines pratiques qui ont servies à concevoir les solutions actuelles.

La capacité à faire évoluer les fonctionnalités de nos applications peut être acquise via la mise en place de solutions à base de Microservices qui vont permettre de **découper la complexité en plus petites unités plus simples à comprendre**, à maintenir et à faire évoluer.

Ce découpage sur une architecture Microservices engendre certes une **complexité de déploiement et d'exploitation**, mais elle est compensée par **l'adoption de nouvelles pratiques** associées, à l'image de la conteneurisation, qui permet de standardiser le déploiement de composants et facilite la répétabilité d'un composant à l'autre, en plus d'optimiser l'usage des serveurs. La mise en place d'outils d'exploitation de nouvelle génération permet, par ailleurs, **d'anticiper et de résoudre des problèmes** en amont plutôt que d'attendre qu'ils ne surviennent, pour les détecter et les traiter.

L'adoption des **principes du Reactive Manifesto,** complémentaires à ceux des Microservices, permet de rendre les applications à la fois plus scalables et plus résilientes. Elle offre aux utilisateurs des systèmes robustes et plus performants. Ces ingrédients sont aujourd'hui indispensables pour apporter un niveau de service optimum.

Enfin, ces changements, associés à une bonne **intégration des principes des architectures Web**, permettent de construire des systèmes plus facilement interopérables avec les outils en ligne aujourd'hui omniprésents, mais également de s'appuyer avantageusement sur des **infrastructures Cloud** pour offrir des niveaux de provisioning, d'élasticité et de disponibilité difficiles à atteindre via nos propres infrastructures.

L'intégration de la transformation continue n'est pas un risque ou bien une charge, mais au contraire l'assurance de toujours être en mesure d'adapter ses outils et systèmes informatiques pour rester en phase avec les besoins utilisateurs et métiers.

TAKE AWAY BACK TRENDS

AJUSTER

- Faire un état des lieux des forces et faiblesses de votre architecture actuelle.
- Appréhender les principes des Microservices.
- Accompagner votre nouvelle architecture Microservices d'un système de surveillance et d'une équipe pluridisciplinaire.

FLUIDIFIER

- Isoler les fonctionnalités de vos applications.
- Adopter de nouvelles pratiques d'ingénierie.
- Concevoir des applications réactives.
- Utiliser les standards du Web.
- Conteneuriser.

VIVRE

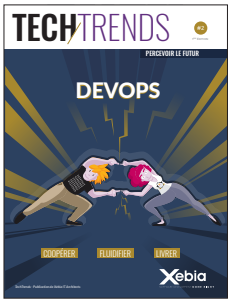
- Se préparer aux mises en production.
- Intégrer les grands principes de supervision, de résilience et de sécurité.
- Monitorer et tester votre architecture.
- Sécuriser son système.

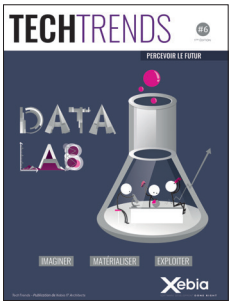

TAKE AWAY
TAKE AWAY
Back
TRENDS

Merci à

Marina Tracco, Julien Renault, Telma Moreira et Louis Laroche.

À lire et à relire





Les précédents numéros des TechTrends sont disponibles en téléchargement (pdf et epub) sur xebia.fr. Si vous souhaitez recevoir une version papier, nous vous invitons à envoyer un mail à : marketing@xebia.fr

Les auteurs

Christophe Heubès

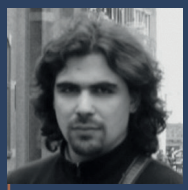
Pablo Lopez

Anne Beauchart

Alexis Kinsella

Nicolas Jozwiak

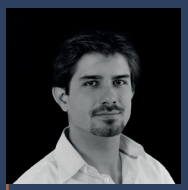
Romain Niveau



Jérôme Doucet

Guillaume Arnaud

Sergio Dos Santos



William Montaz

Design graphique : micheletmichel.com

TechTrends Back- 1ère édition - Mai 2015

SOFTWARE DEVELOPMENT **DONE RIGHT**

Xebia France
156 bd Haussmann, 75008 Paris
+33 (0)1 53 89 99 99
info@xebia.fr

Toutes les informations sur :
xebia.fr