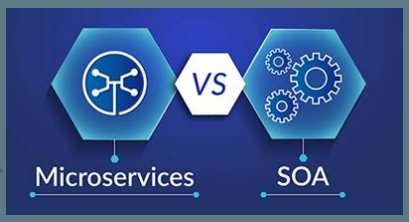
EN DIRECT AVEC LEMARSON

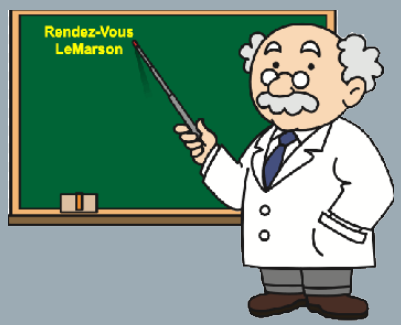
SOA non, MSA oui



Émission animée
par Claude Marson

Le sommaire aujourd'hui
SOA non, MSA oui

- L'urbanisme applicatif, c'est quoi
- Les principes de la réutilisation
- Les constituants d'une architecture urbanisée
- Le malentendu des SOA
- Les microservices et liens avec les containers
- La chaîne de codage des microservices
- Les bonnes pratiques de la programmation réutilisable



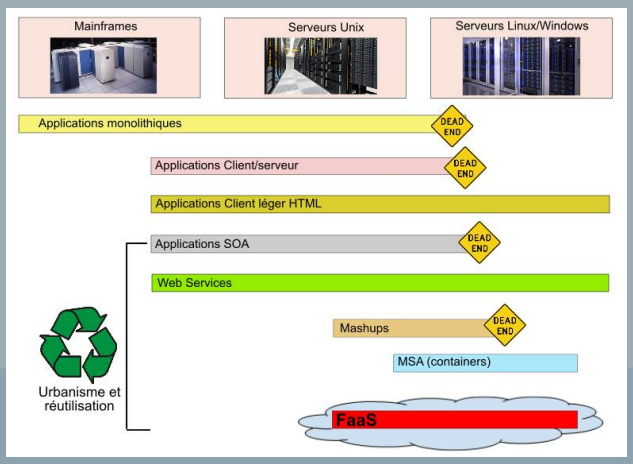
SOA non, MSA oui

L'urbanisme applicatif : c'est quoi ?

- A la fois un comportement et des critères techniques
- L'urbanisme applicatif est un comportement de développeur, qui tend à ce que le code qu'il produit soit modulaire, réutilisable par des tiers internes ou externes, compréhensible, simple.
- L'urbanisme applicatif est aussi un ensemble de méthodes et de règles techniques, qui permettent d'obtenir du code réutilisable
- Les API sont la forme la plus connue de l'urbanisme
- C'est un état d'esprit : le développeur ne programme pas pour lui mais pour la Compagnie qui l'emploie
- SOA, MSA, Web Services, Web Components, FaaS, mashups, sont les formes les plus connues de l'urbanisme applicatif

SOA non, MSA oui

La réutilisation, une histoire mouvementée

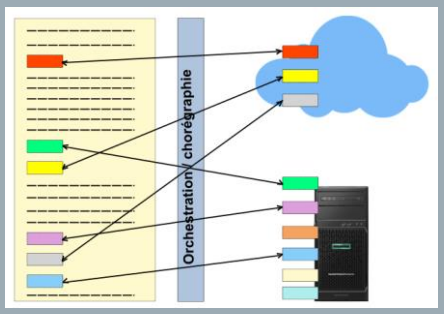


SOA non, MSA oui

Les principes de la réutilisation

- SOA ou MSA : architectures logicielles distribuées, avec un "core" qui sollicite l'exécution de services via des messages, transportés par des protocoles standards.
- Le "core" est à développer ou à modéliser et les services sont exposés dans une bibliothèque de codes prêts à l'emploi.
- L'une des différences entre SOA et MSA, tient à la granularité des services : grande pour les SOA, très petite pour les MSA
- Un service peut être métier ou technique
- L'intégration peut être synchrone ou asynchrone

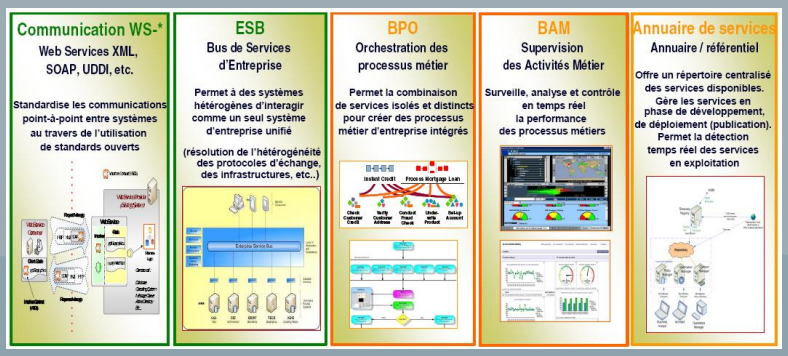
- Encapsulation des services : la logique est dissimulée aux appelants
- Le seul point d'entrée est l'interface : description des traitements, contraintes et données
- Faible couplage pour réduire les dépendances
- Contrats de description et découverte dynamique
- Réutilisation à volonté
- Autonomie des services, qui ne doivent pas dépendre du contexte d'usage
- Composition



SOA non, MSA oui

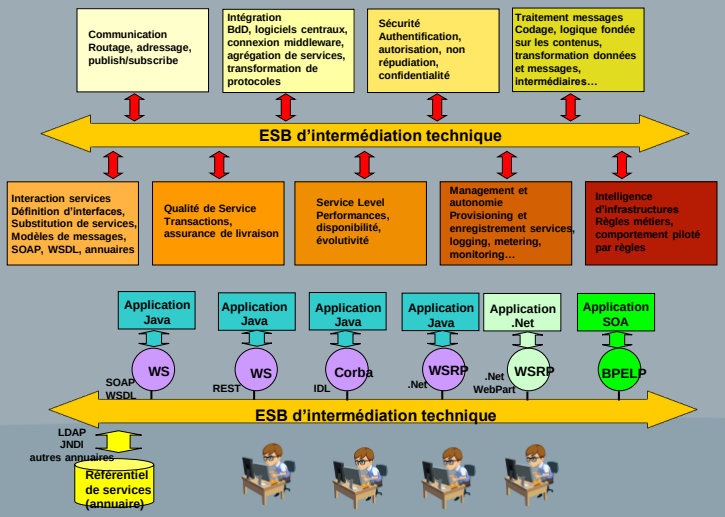
Les constituants d'une architecture urbanisée

- Annuaire des services : UDDI, JNDI (Java)…
- Modélisation de l'enchaînement des composants : BPEL…
- Fichier de présentation de ce que sait faire le service : IDL, WSDN…
- Bus d'intermédiation technique ou chorégraphie (messagerie pour les MSA)
- Protocoles de sollicitation : Rest, Soap, gRPC…



SOA non, MSA oui

Bus : le maillon essentiel de l'intégration

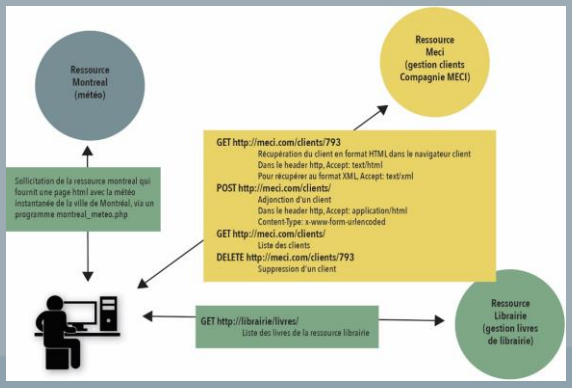


- Un bus est un framework qui comporte les composants nécessaires pour permettre à des applications, composants, clients et serveurs de « se parler », sans que le programmeur ait à développer les aspects spécifiques de l'intermédiation technique
- C'est un chantier délicat, à la base de tous les projets d'urbanisme
- Nécessite de fortes compétences techniques…
- C'est à la fois une architecture de mise en œuvre et une bibliothèque de composants à connaître

Exemples : ESB Blueway, WebSphere Message Broker d'IBM, Sonic ESB (Aurea Software), Artix de Iona, Oracle Enterprise Service Bus, iWay SOA, Active Matrix Business Works (Tibco), Mule (Mulesoft), WSO2…

SOA non, MSA oui

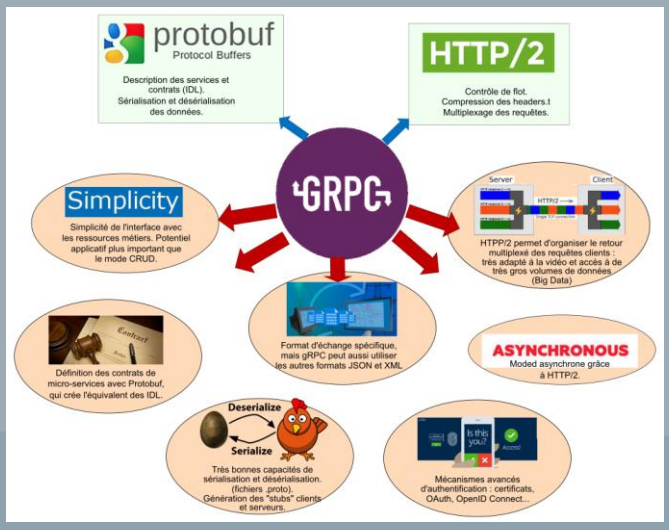
Le protocole REST



- L'accès aux services peut se faire selon plusieurs architectures techniques principales : REST (REpresentational State Transfer), SOAP, gRPC…
  - REST est un style d'architecture : client/serveur, sans état, cache, très simple
  - REST est une extension du principe de lien Hypertexte HTTP, adapté au dialogue entre machines. Il utilise les mêmes fonctions : Get, Put, Post, Delete, pour lire, modifier, écrire, effacer dans une ressource qui peut être décrite en XML ou appeler une commande sur un serveur et lire la réponse.
  - Utilise la méthode Get HTTP pour invoquer un service, la réponse pouvant être XML, RSS, Atom, JSON (un format d'échange ultra léger, pris en charge par de nombreux langages : PHP, Java, C#, Ruby, Python…) ou n'importe quel autre format texte
  - REST est adapté aux projets très ouverts, qui réunissent un grand nombre d'acteurs (Amazon et Yahoo), avec des fournisseurs vis-à-vis desquels le développeur n'a pas de prise.
  - REST est immédiatement opérationnel sans surcoût, il n'y a pas d'installation à effectuer

SOA non, MSA oui
…mais déjà gRPC

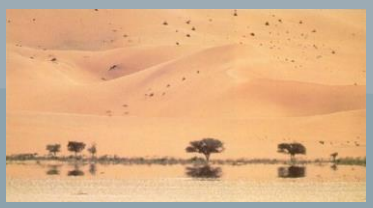
- gRPC de Google, très adapté aux micro-services, grâce à sa faible latence de communications temps réel, point à point, avec diffusion en continu
- Multilangues grâce aux clients Polyglot
- Adapté aux réseaux avec de fortes contraintes (mobiles…), grâce à l'excellente sérialisation de Protobuf, qui fait infiniment mieux et charge moins les liens que ne le fait JSON.



SOA non, MSA oui
Le malentendu des SOA

- En 2004, le Gartner émet l'idée d'un **SI** architecturé autour de services métiers, écrits une fois pour toutes et réutilisables en tout ou partie par les applications futures (100 à 200 par entreprise)
  - Il émet aussi l'hypothèse qu'en 2010, les SI de la planète auront migré à 80 % en SOA et que sur les 20 % restants, 10 % seront des EDA (SI fondés sur des événements)
  - Résultats : en 2020, il existe très peu de **SI SOA** sur la planète
  - Il existe des applications urbanisées, y compris en termes de métiers, mais peu de SOA
- Le malentendu est évident
  - Les directeurs de projets se sont précipités aveuglément
  - L'urbanisation métier a été très incomplète, coûteuse et insuffisamment générique
  - Le taux de réutilisation des services métiers sans retouche, ne dépasse pas 5 %, alors que le palier de crédibilité devrait être de 50 %
  - Dans la pratique, les chefs de projets ont fabriqué des clones
- On a confondu SOA et Web Services
  - Contrairement aux SOA, il existe une multitude d'applications urbanisées à base de Web Services, métiers et techniques et maintenant microservices
  - Ce ne sont pas pour autant des SOA
- Les SOA restent très attractives, mais le basculement global ne se fera sans doute jamais…

SOA : attention aux mirages…



SOA non, MSA oui

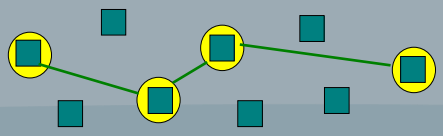
Les SOA

- Un service est un traitement normalisé, mutualisé et référencé au sein de l'entreprise, dont l'interface d'appel est décrite dans un langage neutre (indépendant des technologies) et qui est déployé physiquement sur un serveur local ou dans le Cloud.
- La notion de service correspond à un périmètre fonctionnel destiné à être exposé : sa granularité est d'autant plus grande que celle des tâches est importante
- La granularité des services est établie, de manière à ce qu'ils soient ni trop nombreux – dans ce cas, l'ensemble sera difficile à gérer et à maintenir – ni trop lourds, une architecture se réduisant à quelques services ne présentant plus d'intérêt par rapport aux anciennes applications
- Un service est faiblement couplé
- Synchrone ou asynchrone
- Il est indépendant des autres services et on dit qu'il est sans état (« stateless »)
- Il possède quatre grands propriétés :
  - Il est autonome, il ne dépend pas d'un autre service
  - Il s'expose à travers un contrat : il se « découvre »
  - Les frontières entre services sont explicites et il n'y a pas de recouvrement
  - Certains disent qu'il est à instanciation unique (singleton) : on peut ne pas être d'accord
  - Le service possède un « propriétaire », qui se sent responsable de sa finalité métier

SOA non, MSA oui

Composants et services applicatifs

- On distingue les services des composants
  - Le service a une connotation métier et correspond à une opération qu'est susceptible d'effectuer l'entreprise : la facturation d'un client, la mise à jour d'un compte client…
  - Le composant est un élément basique, qui peut être à la fois technique ou métier, dont la granularité est celle-fois très réduite, une architecture pouvant comporter plusieurs milliers de ces composants
  - Un service sera constitué à l'exécution, de l'enchaînement en séquences ou simultanés, d'un sous-ensemble de ces composants, qui « in fine » produiront le résultat métier attendu.



SOA non, MSA oui

BPEL : la modélisation de l'orchestration en XML

- Destiné à l'enchaînement des composants:
  - Orchestration : quand il y a un service central de coordination (les web services ne savent pas à priori quel sera leur rôle)
  - Chorégraphie : chaque web service sait « ce qu'il a à faire »
- Le modèle BPEL d'un processus métier est un processus en 4 étapes, chacune d'elles étant symbolisée par un document XML:
  - Inventaire de tous les composants impliqués dans le processus métier
  - Définition de l'interface WSDL du processus : sa présentation vis-à-vis de l'extérieur
  - Définition les liens partenaires, c'est-à-dire l'interaction entre le processus BPEL, les services auxquels il a accès et les clients du service BPEL
  - Création du processus métier BPEL, avec la description des variables et l'enchaînement dynamique des opérations.
- Le processus est composé de tâches qui peuvent être primitives ou de structure
  - Primitives : exemples
    - **<invoke>** pour appeler un service
    - **<assign>** pour manipuler les variables
    - **<throw>** pour gérer les exceptions
    - **<terminate>** pour conclure un service
  - Structure : exemples
    - **<sequence>** pour définir l'enchaînement ordonné de services
    - **<flow>** pour définir un ensemble de services pouvant s'exécuter en parallèle
    - **<switch>** pour implémenter des branchements conditionnels
    - **<while>** pour les boucles d'enchaînement de services

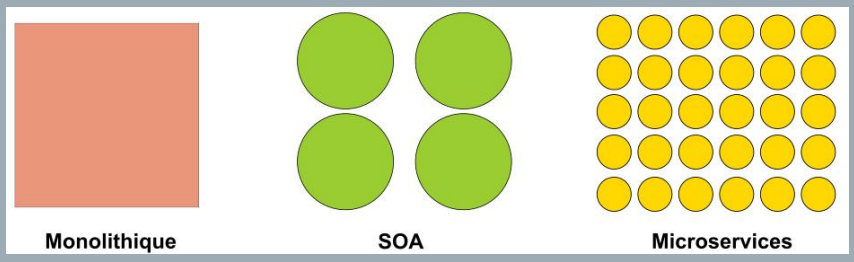
SOA non, MSA oui

Les promesses non tenues des projets SOA

- Pour l'instant le succès n'est **pas** là… sauf chez quelques grandes entreprises : Netflix, LCL…
- Une soi-disant simplicité : c'est faux, une SOA est tout sauf simple
  - La démarche normale n'est pas respectée, qui va des efforts d'urbanisation en amont jusqu'à la réalisation
  - Cela donne des référentiels démesurés et incohérents…il faut tout recommencer
- Peu d'entreprises savent définir des services réutilisables, limités en nombre
- La maturité n'est pas au rendez-vous
- La mutualisation des composants est difficile à maintenir dans le temps
  - Si le contrat d'un service est amené à changer, chaque client doit modifier sa procédure d'appel et la solution est souvent de maintenir plusieurs versions du même service, ce qui va à l'encontre du principe même de l'urbanisme
- On se rend compte qu'une SOA n'a de chance de voir le jour que si la MOA est impliquée : mais très rares sont les cas où ce lien est établi en amont des réalisations
- Le ROI est encore à démontrer
- Les catastrophes sont nombreuses…mais on n'en parle pas…
- Il est probable que la SOA va se limiter à des départements d'entreprises, mais pas à leur globalité, au sens SI
- Les SOA sont une « fausse bonne idée », ce qui ne remet pas en cause l'urbanisme applicatif

SOA non, MSA oui

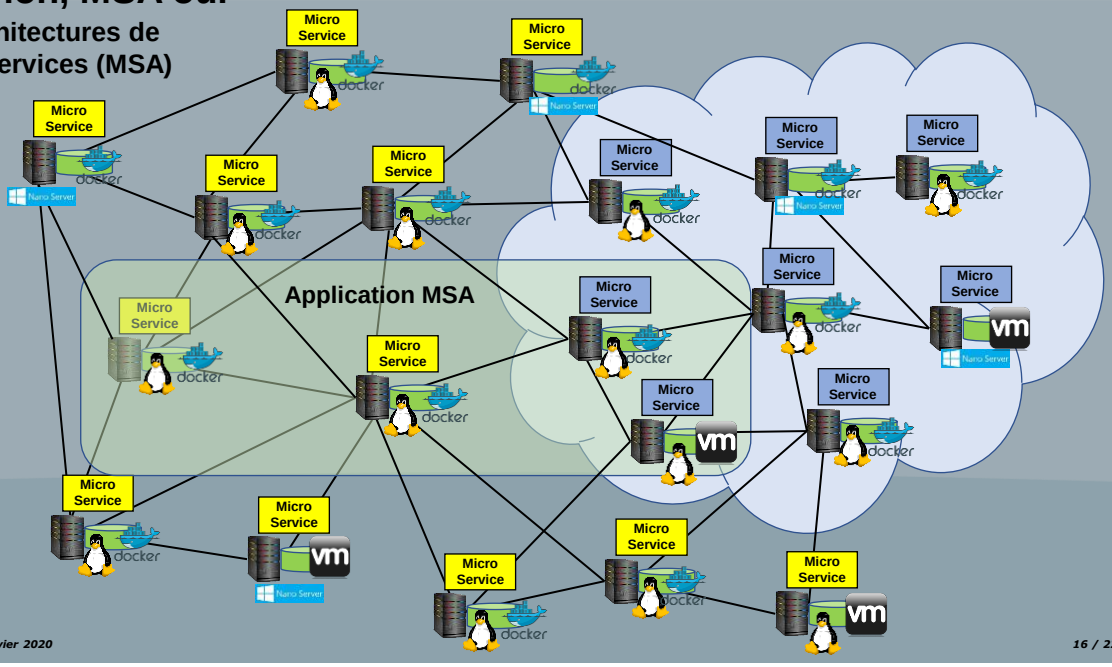
MSA



- D'abord une question de granularité
- Pas de distinction entre métier et technique
- Une entreprise peut s'appuyer sur des milliers de microservices
- Les grands concepts de bus, d'interfaces et d'orchestration sont les mêmes
- Machine virtuelle contre container
- Mais…
  - Le terme micro est relatif, quel est le juste milieu, risque de tomber dans les nanoservices
  - La granularité des SOA pose problème, mais le nombre de microservices en pose un autre
  - Comment être informé qu'un microservice existe : le problème est amplifié par rapport aux SOA
  - Le codage devient une recherche permanente des microservices
  - Difficulté pour maintenir l'indépendance et l'autonomie des microservices : augmentation de la complexité

SOA non, MSA oui

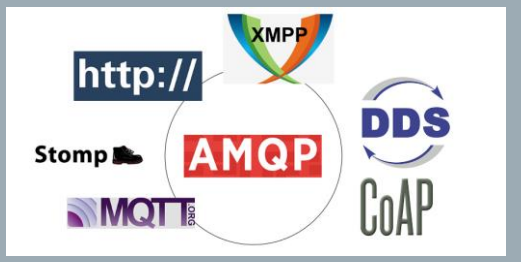
Les architectures de micro-services (MSA)

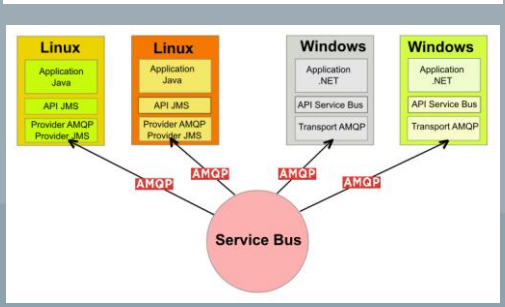


SOA non, MSA oui

Le choix du bus d'intermédiation

- Il faut s'appuyer sur un broker de messages le plus standard possible
- Il en existe un certain nombre, dont AMQP, qui ont été conçus pour un emploi spécifique, plus ou moins éloigné du « brokerage » Publish/Subscribe.
  - HTTP, pour le Web, accessible sur toutes les machines, qui dispose de fonctions de brokerage : maintenant HTTP/3
  - XMPP (eXtensible Messaging Presence Protocol), conçu à l'IETF pour la messagerie instantanée.
  - CoAP (Constrained Application Protocol), est également dû à l'IETF et à son groupe spécialisé CoRE, écrit pour permettre à des capteurs de solliciter des composants Web Service, en mode RESTFul.
  - STOMP (Streaming Text Oriented Messaging Protocol) est un protocole de texte « monté » sur TCP.
  - MQTT (Message Queue Telemetry Transport), de l'OASIS (comme AMQP), dédié aux communications entre capteurs, d'objets IoT et de M2M, créé par IBM et Eurotech.
- AMQP (Advanced Message Queuing Protocol) est [Synapse] la brique applicative la plus pertinente pour élaborer des intergiciels de messagerie.
- AMQP est devenu LE standard : OASIS, ISO, IEC
- AMQP est un protocole applicatif binaire, de bas niveau : contrôle de flux, communications orientées messages avec garantie de livraison, mécanismes d'authentification et de chiffrement fondés sur SASL et TLS.

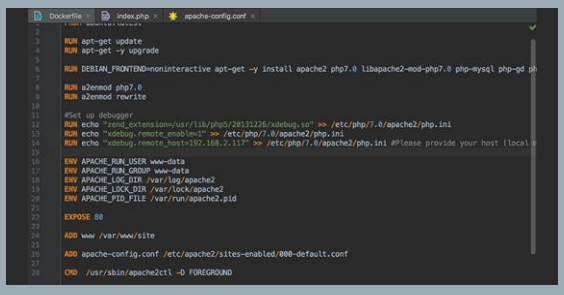




SOA non, MSA oui

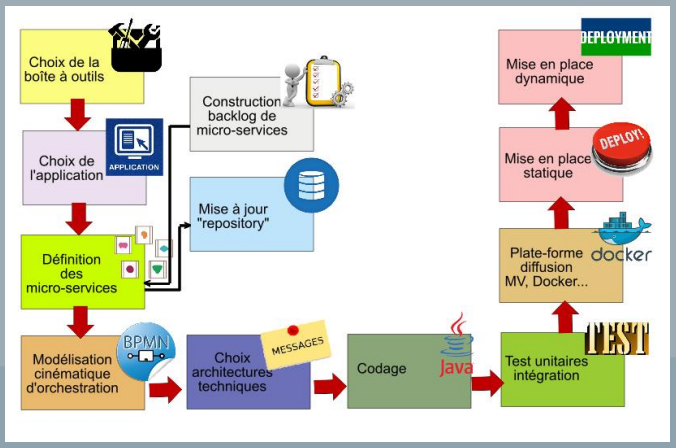
L'urbanisme via Docker, au cœur des architectures MSA)

- L'idée reçue selon laquelle il n'y a rien à faire pour préparer un container, est fausse.
- Le secret de l'opération est le fichier Dockerfile, un fichier texte dans lequel Docker va trouver toutes les commandes pour construire son image, celle qui sera installée et exécutée.
- Le fichier est construit avec des instructions telles que :
  - RUN pour exécuter n'importe quelle commande Linux
  - ENV pour initialiser les variables d'environnement
  - ADD pour copier le nouveau fichier dans le bon répertoire de destination Docker
  - EXPOSE qui précise le port de la machine hôte.
  - mais aussi FROM, MAINTAINER, CMD, LABEL, EXPOSE, ENV, COPY, ENTRYPOINT, WORKDIR, ONBUILD,
  - VOLUME, USER…
- Il y aura autant de Dockerfile que d'environnements de développement et de langages.



SOA non, MSA oui

Le codage des micro-services

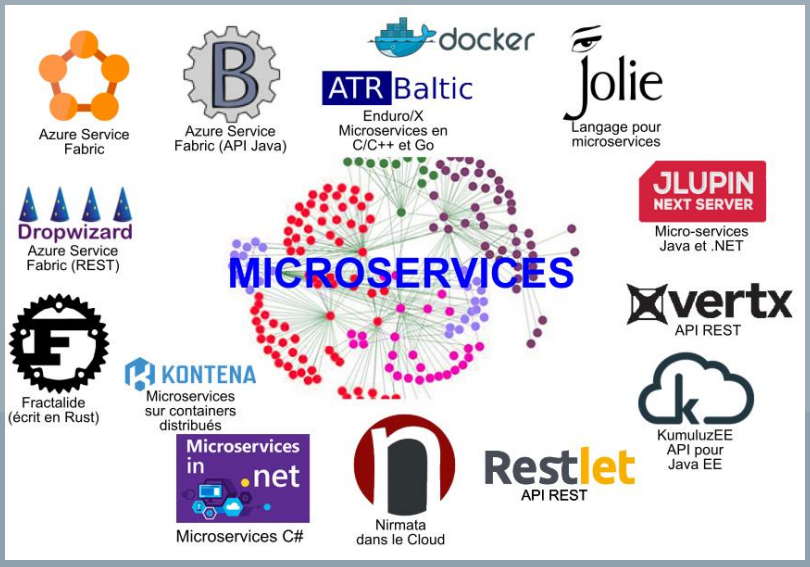


Les problèmes posés par les micro-services ne sont pas éloignés de ceux des SOA :

- la découverte des micro-services, via le repository
- la gestion des exceptions, d'autant plus complexe que l'on est dans un mode distribué
- la gestion des files d'attentes aux micro-services, selon les modes LIFO (Last In First Out) ou FIFO (First In First Out)
- le traitement des formats de messages : XML, JSON, texte, CSV…
- la prise en charge du chiffrement (confidentialité) et identité des micro-services
- la gestion de la dégradation des applications, chaque micro-service pouvant envoyer un état de comportement (heartbeating) au référentiel pour l'informer de sa situation, le micro-service demandeur interrogeant le référentiel pour savoir s'il a des chances d'être servi rapidement et si ce n'est pas le cas, opter pour une solution de repli
- la répartition de charge, qui est une conséquence du point précédent, l'idéal étant de disposer d'une API dédiée qui règlera ce problème, sachant qu'on pourra aussi la prendre en compte dans le codage des micro-services ou par une administration posée sur notre code. Encore un choix à effectuer.
- la tolérance aux fautes, indispensable pour certains micro-services, pour qui on prévoira des procédures de repli prioritaires
- l'enregistrement des évènements de l'application dans un fichier log

SOA non, MSA oui

Les API MSA sont là…



SOA non, MSA oui

Quelle que soit l'architecture :
Les bonnes pratiques de la programmation réutilisable

- Ce n'est pas uniquement un problème technique
- C'est une affaire de discipline
- Il faut réfléchir à ce que doit être le service et anticiper sur ce qu'il pourrait devenir
- On passera de 2 à 3 fois plus de temps que pour un service classique
- Ne pas négliger la documentation et partir du principe que les autres développeurs sont des débutants : faire un effort de pédagogie pour leur faciliter la tâche
- Ne pas refuser le refactoring : un service ne peut être développé sans « retouches » au-delà de 2 à 3 ans, le contraire est une utopie
- Promouvoir le mode « boîte noire » au détriment de la « boîte blanche » et si possible l'imposer
- Faire comprendre que l'important dans une application, ce n'est pas le code mais la conception et qu'il ne faut pas se précipiter dans le codage avant d'avoir compris la finalité du module
- La principale motivation de la réutilisation n'est pas de faire des économies budgétaires
- Penser à la qualité du code : simplicité d'écriture et respect des consignes de codage
- Les méthodes agiles, type SCRUM, peuvent être un bon moyen pour faire passer ces règles : « scrum review » en fin de cycle,
- La réussite d'une approche urbanisée se mesure par le taux de réutilisation du code : quand une fonction, une méthode, une séquence HTML existe déjà, il est interdit de la redévelopper…sauf cas extrême

Nos prochains rendez-vous

Vendredi 24 janvier 2020 :	**RSSI (CISO), se libérer des contraintes artificielles des entreprises**
Vendredi 31 janvier 2020 :	**Les réseaux neuronaux convolutifs**
Lundi 3 février 2020 :	**Actualités du TI**
Vendredi 7 février 2020 :	**La programmation fonctionnelle**
Vendredi 14 février 2020 :	**La fin du scandale des certificats payants ?**
Vendredi 21 février 2020 :	**La justification financière des projets qualité des données**
Vendredi 28 février 2020 :	**ITIL v4, vous avez du temps à perdre ?**

Je vous remercie de votre attention et à bientôt