
Architectures Web-Services, Micro-Services et API

Table des matières

I - Les mutations du S.I.....	5
-------------------------------	---

II - Concepts SOA et Urbanisation SOA..... 14

1. Architecture SOA (présentation).....	14
2. SOA et Intégration.....	19
3. Paradigme "tout est service".....	20
4. Principales technologies SOA.....	21
Attention: il ne faut pas se sentir obligé d'utiliser toutes les technologies "SOA" (quelquefois très complexes) pour la simple raison qu'elles existent.....	21
Les besoins simples pourront être gérés en utilisant simplement des "web services" (SOAP ou REST).....	21
Les annuaires de services sont facultatifs et rarement utiles.....	21
La sécurité doit idéalement être découplée du "pur fonctionnel" (via des intercepteurs) et est souvent prise en charge par des spécialistes.....	21
Les "ESB" peuvent être vus comme une infrastructure à mettre en production et on peut souvent en faire abstraction en phase de développement.....	21
L'orchestration sophistiquée (avec "BPEL" ou bien "BPMN2 + java") n'est réellement utile que dans des contextes asynchrones ou bien lorsque l'on recherche une bonne traçabilité entre les modèles et les portions de code à maintenir et faire évoluer.....	21
5. Gains de l'approche SOA et enjeux.....	22
6. Quelques typologies de services à assembler.....	23
7. Cadre classique pour services élémentaires.....	25
.....	25
8. Modélisation des services élémentaires.....	26
9. Services fonctionnels (définition de la catégorie).....	28
10. Notions essentielles d'urbanisation.....	30
11. Urbanisation fonctionnelle.....	32
12. Urbanisation applicative.....	34
13. Business Modeling et "Business Uses Cases".....	35
14. Cas d'utilisation d'une partie du SI (soa).....	37
15. Modélisation des processus métiers.....	38
16. Pièges à éviter :.....	39
17. Considérations sur la granularité des services.....	50
18. Composants SOA.....	50
19. SOA bien dimensionné (selon moyens et besoins).....	53
20. Cycle de vie d'un projet SOA.....	54
21. Activités de modélisation SOA (enchaînement).....	58

III - Architecture technique SOA (SOAP, ESB, BPM)..... 59

1. Deux grands types de WS (REST et SOAP).....	59
2. Protocole SOAP.....	61
3. WSDL (Web Service Description Language).....	65
4. Détails sur WSDL.....	68
5. Tests d'un service web "soap" (soap-ui).....	73
6. Quelques API et technologies "web services"	75
7. API WS SOAP JAVA : JAX-WS.....	76
8. EAI et ESB.....	78
9. EAI (Enterprise Application Integration).....	83
10. ESB (Enterprise Service Bus).....	86
11. Notion de "moteur de services"	89
12. Catégories d'ESB.....	90
13. SOA et mode asynchrone.....	93
14. BPEL (présentation).....	94
15. BPEL en mode asynchrone.....	101
16. jBPM et activiti (présentation).....	103
17. UserTask.....	105
18. Considérations diverses sur l'architecture SOA :	105

IV - L'approche REST..... 108

1. Caractéristiques clefs web-services "REST" / "HTTP"	108
2. Web Services "R.E.S.T."	109
3. Quelques API et technologies "web services REST"	114
4. Appels de WS REST (HTTP) depuis js/ajax.....	119
5. Limitations Ajax sans CORS.....	124
6. CORS (Cross Origin Resource Sharing).....	125
7. Web services "REST" pour application Spring.....	128
8. WS REST via Spring MVC et @RestController.....	128
9. Test de W.S. REST via Postman.....	140
10. Test de W.S. REST via curl.....	141
11. Notion d'Api REST et description.....	143
12. Swagger.....	146
13. Config swagger2 / swagger-ui pour spring-mvc.....	146
14. HATEOAS.....	149

V - L'approche micro-services..... 151

1. Architecture "micro-services" (essentiel).....	151
---	-----

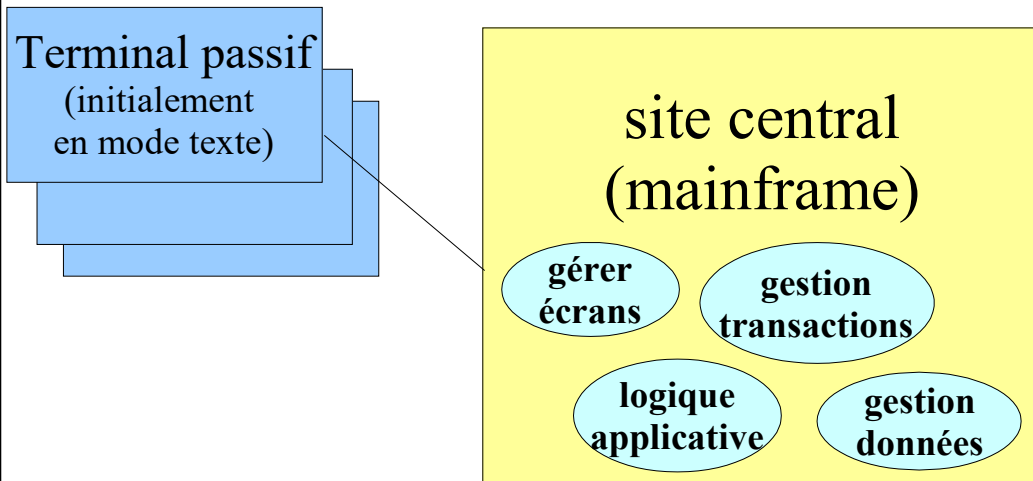
VI - Design/Gestion d'API..... 167

1. Design d'une api REST.....	167
2. Api Key.....	171
3. Token d'authentification.....	172
4. OAuth2 (présentation).....	176
5. Eventuelle Monétisation d'une API REST.....	181
6. Monitoring d'API REST.....	181
7. Gateway API.....	182
8. Portail développeur.....	184
VII - Annexe – Cloud computing (essentiel).....	186
1. Le "Cloud Computing" (présentation).....	186
VIII - Annexe – DevOps.....	189
1. DevOps.....	189
IX - Annexe – Docker (présentation / essentiel).....	196
1. Docker : Vue d'ensemble.....	196
X - Annexe – WS-REST (NodeJs/Express).....	205
1. Ecosystème node+npm.....	205
2. Express.....	205
3. Exemple élémentaire "node+express".....	206
4. Installation de node et npm.....	207
5. Configuration et utilisation de npm.....	208
6. Utilisation basique de node.....	210
7. WS REST élémentaire avec node+express.....	210
8. Avec mode post et authentification minimaliste.....	212
9. Autorisations "CORS".....	214
XI - Annexe – Essentiel XML/XSD (pour SOAP).....	216
XII - Annexe – BPMN.....	222
1. BPMN (essentiel).....	222

I - Les mutations du S.I.

1.1. Evolution générale des architectures

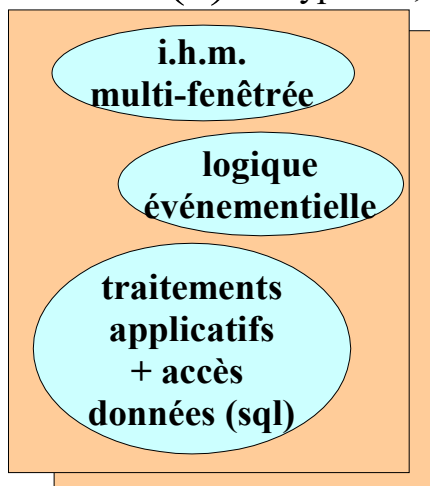
Architecture centralisée / monolithique



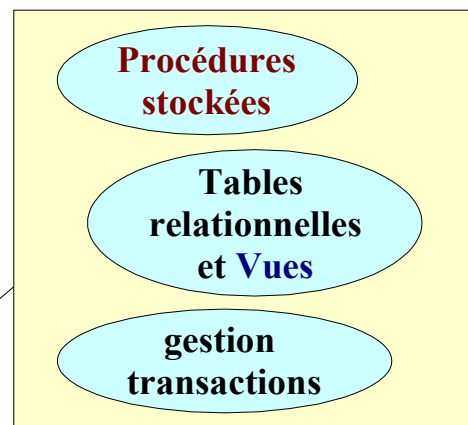
- ++ back office efficace (avec machine puissante)
- interface homme/machine limitée
- solution propriétaire

Architecture client/serveur (années 90)

Client(s) (ex: L4G de type VB,...)

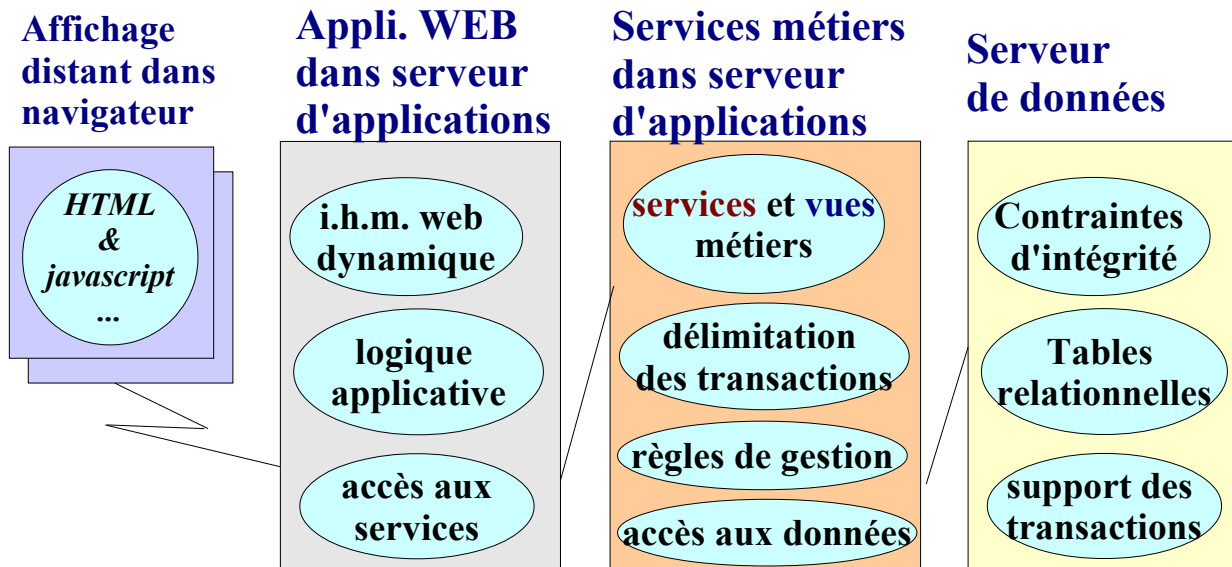


Serveur(s) (ex: SGBDR de type Oracle, ...)



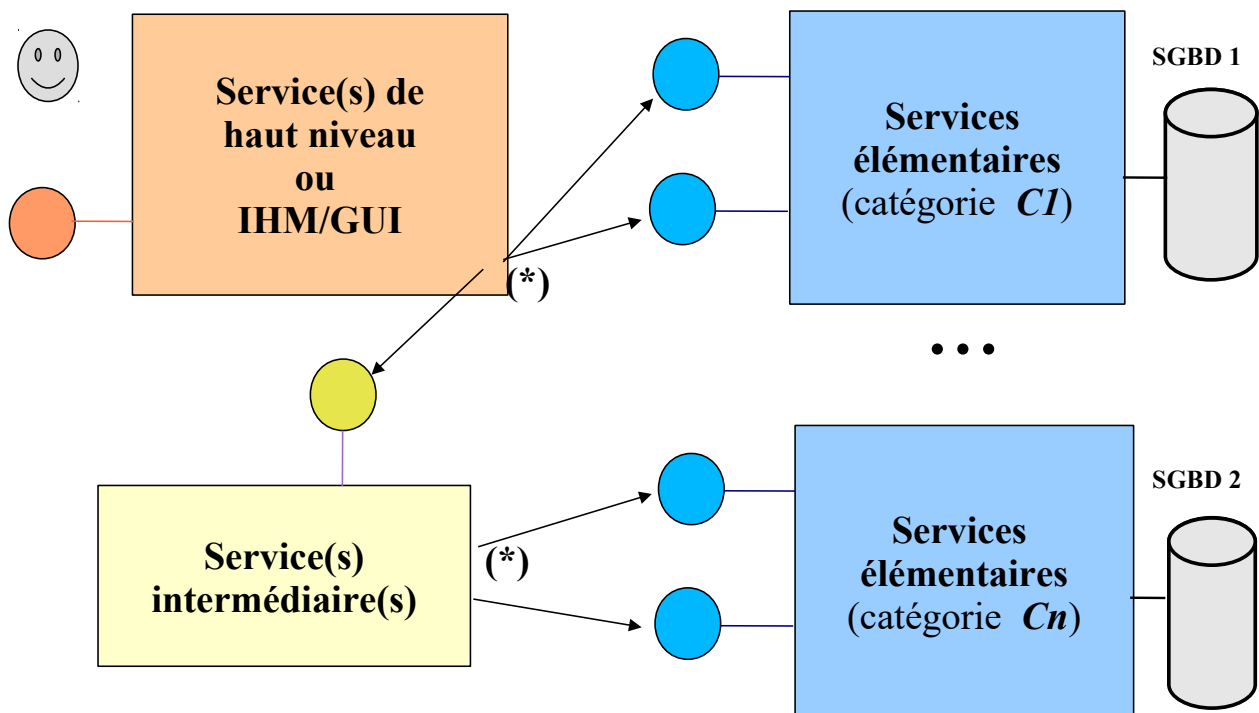
- ++ *interface graphique sophistiquée*
- *modularité limitée*
- *prolifération des clients lourds et déploiements délicats*

Architecture n-tiers (début années 2000)

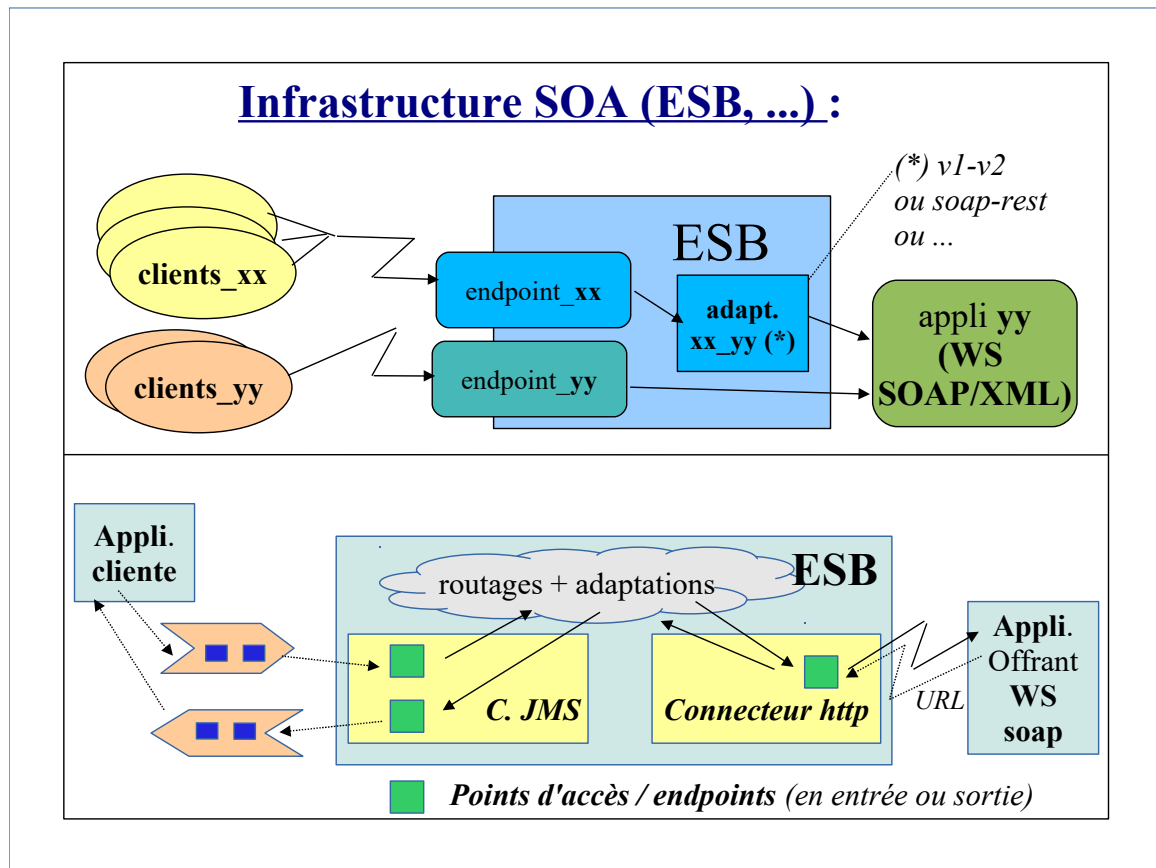


++ modularité , déploiements simplifiés , ihm correcte
- complexité de l'ensemble

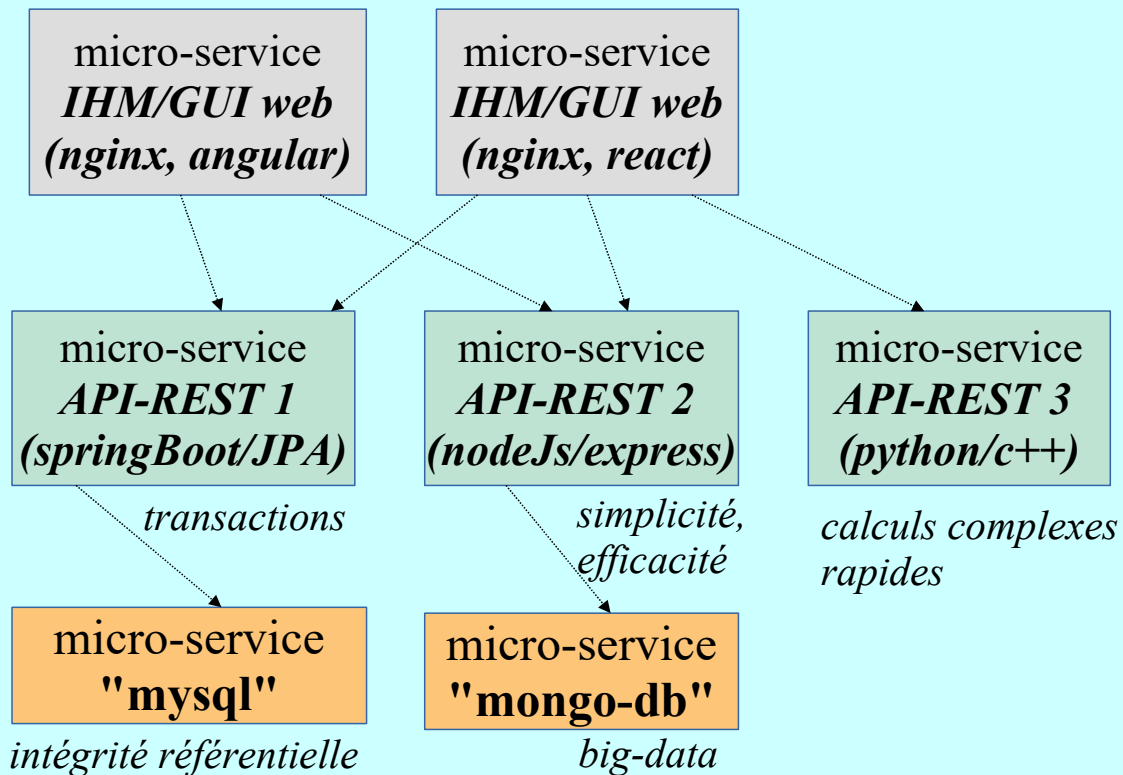
architecture orientée services (soa)



(*) Orchestration de services .

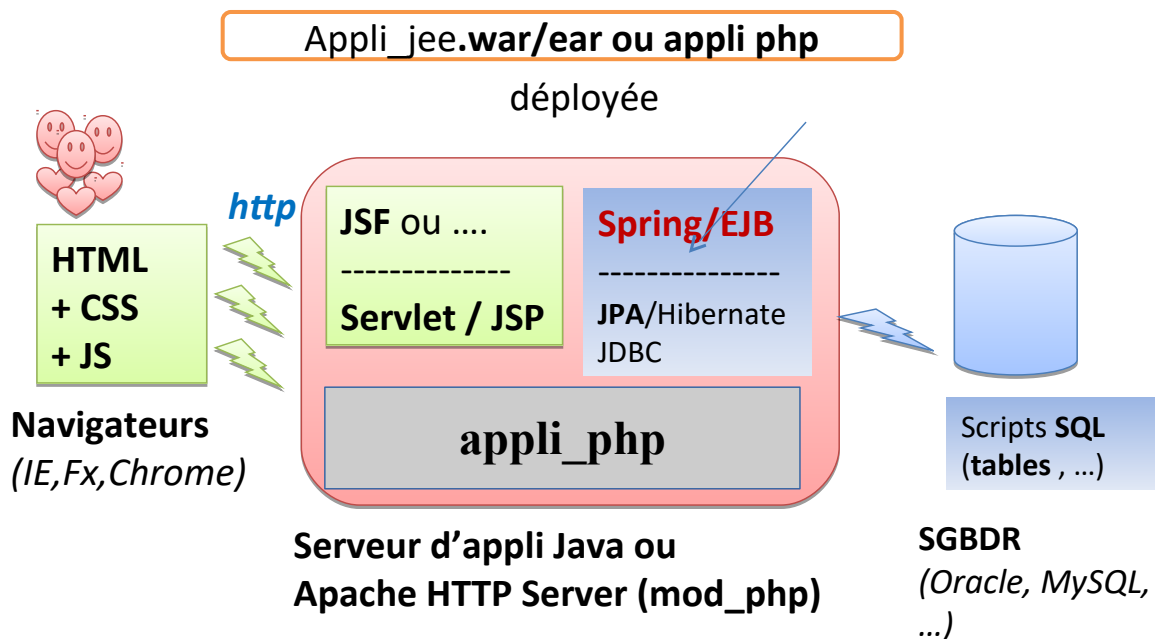


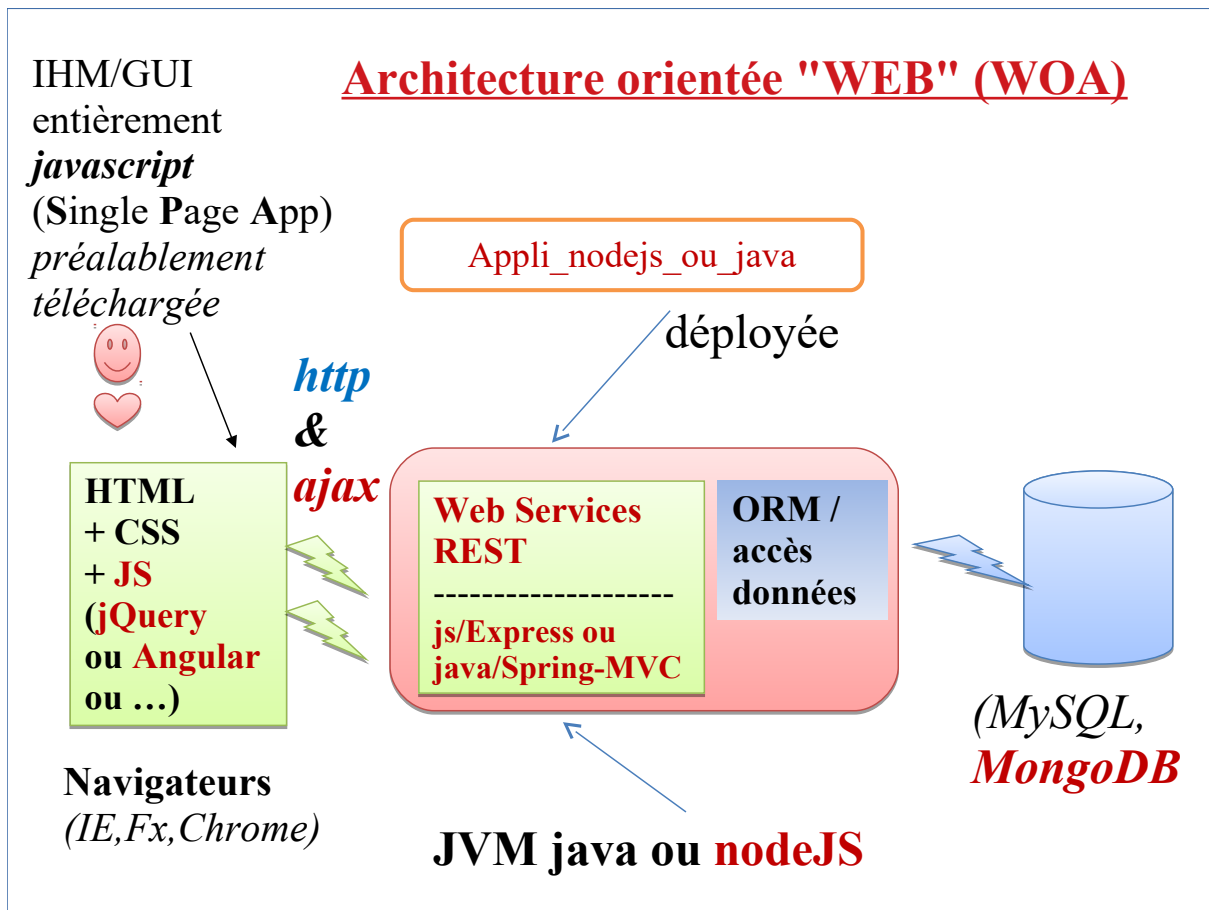
Architecture micro-services (avec micro-conteneurs)



1.2. Evolution et généralisation de l'architecture web

pages générées côté serveur (has been)





1.3. Diversification/multiplication des clients/devices

Multiplications et diversifications des "clients/devices"



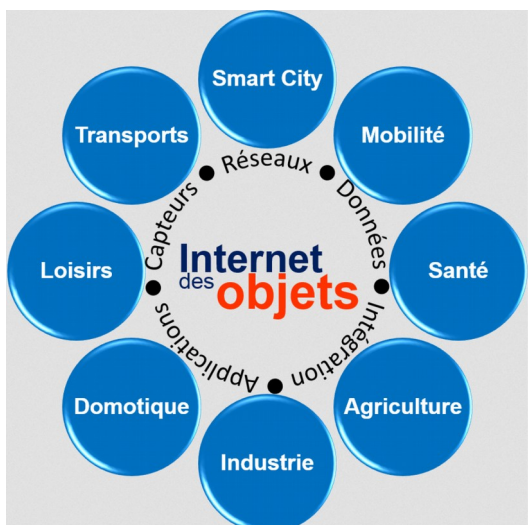
tablettes/ smartphones / desktop / ...

android / iphone / ...

windows/linux/...

--> *tout ça peut invoquer des WS REST/HTTP/JSON*

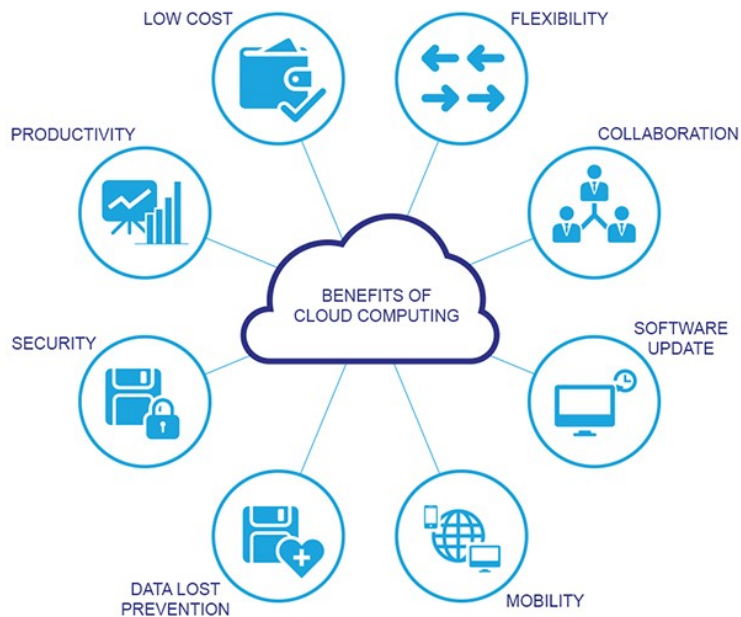
L'internet des objets (connectés)



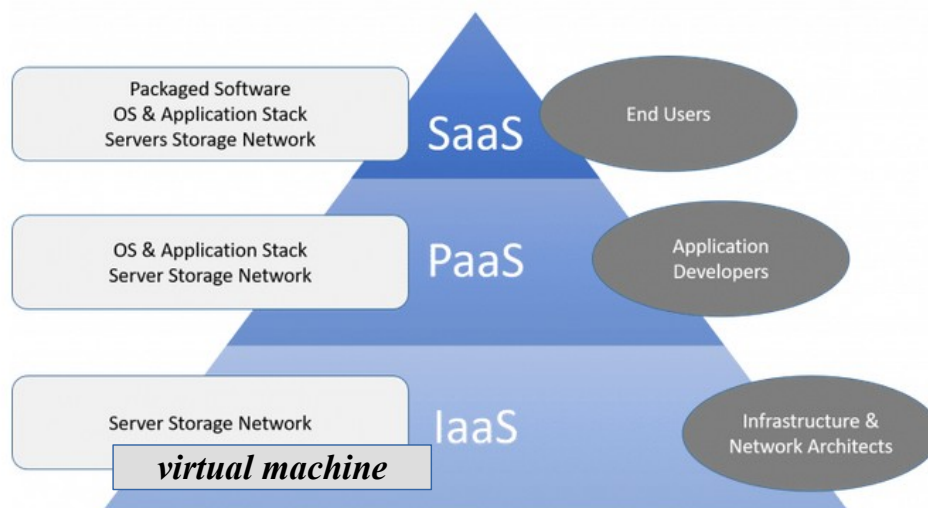
Contraintes : légèreté , simplicité , ...

1.4. Cloud computing

Cloud computing (apports)



Cloud computing (hébergement , délégation)



SaaS : Software As A Service

PaaS : Platform As A Service

IaaS : Infrastructure As A Service

1.5. A grande échelle

Les solutions trouvées/apportées par les géants du WEB

Besoin de solutions fiables/performantes à grande échelle
--> recherche / mise au point --> solutions maintenant accessibles
(cloud public ou cloud privé ou ...)

Amazon --> **Amazon Web Services** (EC2, S3, ...)

FaceBook --> React, ...

Twitter --> bootstrap CSS

Google --> **Angular** , **Kubernetes** , ...

Netflix --> **Netflix oss**

pour big data --> mongo , cassandra , ...

1.6. Transformation digitale

La transformation digitale

Le terme à la mode de "transformation digitale" désigne les changements liés à l'intégration de la technologie digitale dans la société humaine et se base sur plusieurs grands piliers :

- la mobilité
- le temps réel
- l'internet des objets
- le big data
- l'universalité d'internet.

Mouvement/tendances associé(e)s :

- agilité
- pratiques "DevOps"
- travail collaboratif

II - Concepts SOA et Urbanisation SOA

1. Architecture SOA (présentation)

1.1. Agilité visée dans l'architecture SOA

Architecture **SOA** (*orientée services*)

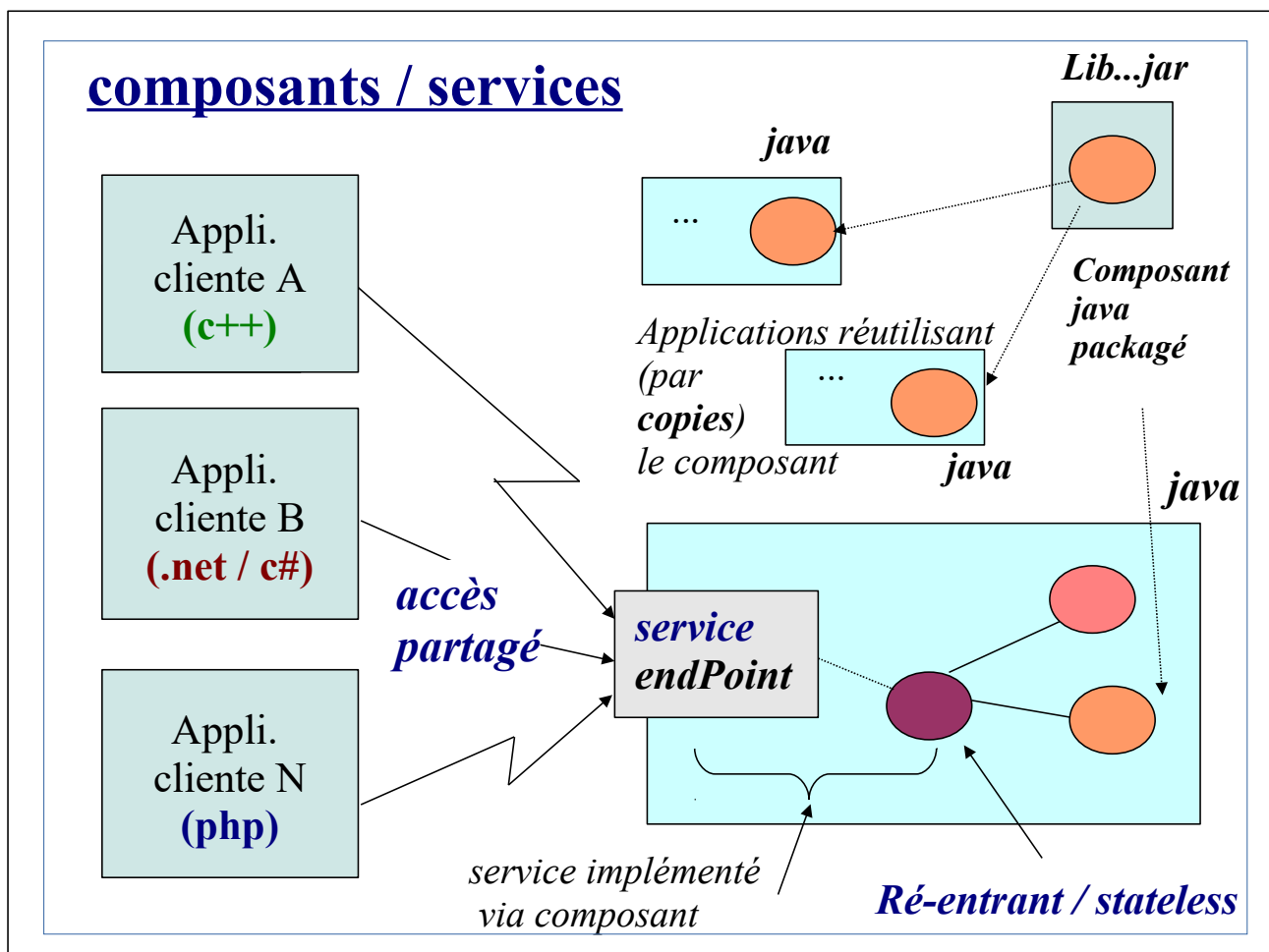
Grands traits de l'architecture SOA:

- Interopérabilité entre **langages différents** (*c++ , java, php , js , ...*)
- découplage entre invocation et implémentation (avec éventuels *intermédiaires reconfigurables* [adaptateur, ESB, ...])
→ Système informatique **souple/agile** .
- **Services de "haut niveau"** appelant des **sous services intermédiaires ou élémentaires** pour **partager** des données , des traitements , règles métiers , ...
- **Intégration**/adaptation de services rendus par des applications externes (de support, ...).

Une architecture "SOA" vise essentiellement à mettre en œuvre un S.I. Agile :

- souple
- modulaire
- facilement reconfigurable

1.2. Service = "traitement partagé"



Une fois **publié** (*mis en production*), un service est potentiellement accessible (en mode "partagé") depuis une multitude d'applications clientes.

On peut ainsi partager des "paquets de traitements" et indirectement accéder (en lecture et/ou écriture) à des données partagées (en arrière plan).

Les accès partagés sont effectués en mode "appels distants" / **RPC** ("Remote Procedure Call") .

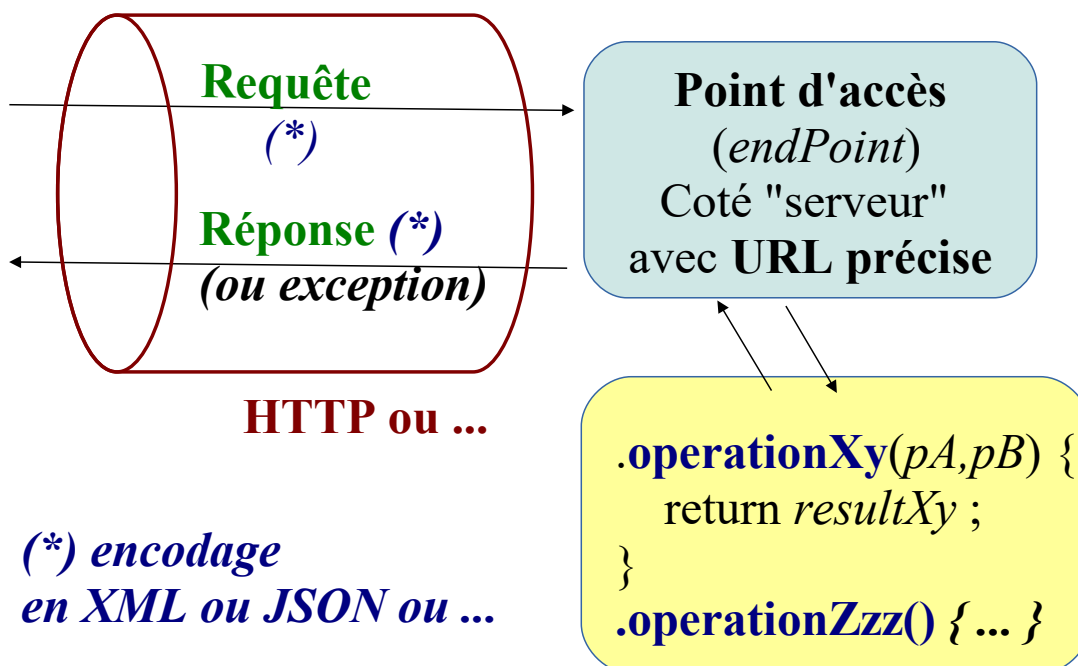
Via des protocoles d'accès plus ou moins précisément normalisé, ces appels distants peuvent être effectués depuis des applications clientes développées dans des langages de programmation divers et variés (C/C++ , java , C# , php ,)

1.3. Service "WEB"

Un "service WEB" est un service qui est invocable (appelable) via une (ou plusieurs) des technologies WEB standardisées suivantes :

- **XML** (norme officielle "W3C")
- **HTTP** (protocole d'accès aux pages HTML , protocole dit de "transport" , norme du W3C)
- **JSON** (Javascript Object Notation) : alternative plus simple vis à vis de XML.

Service "Web" = service mise en œuvre avec technologie(s) "web" (*http et/ou xml*)



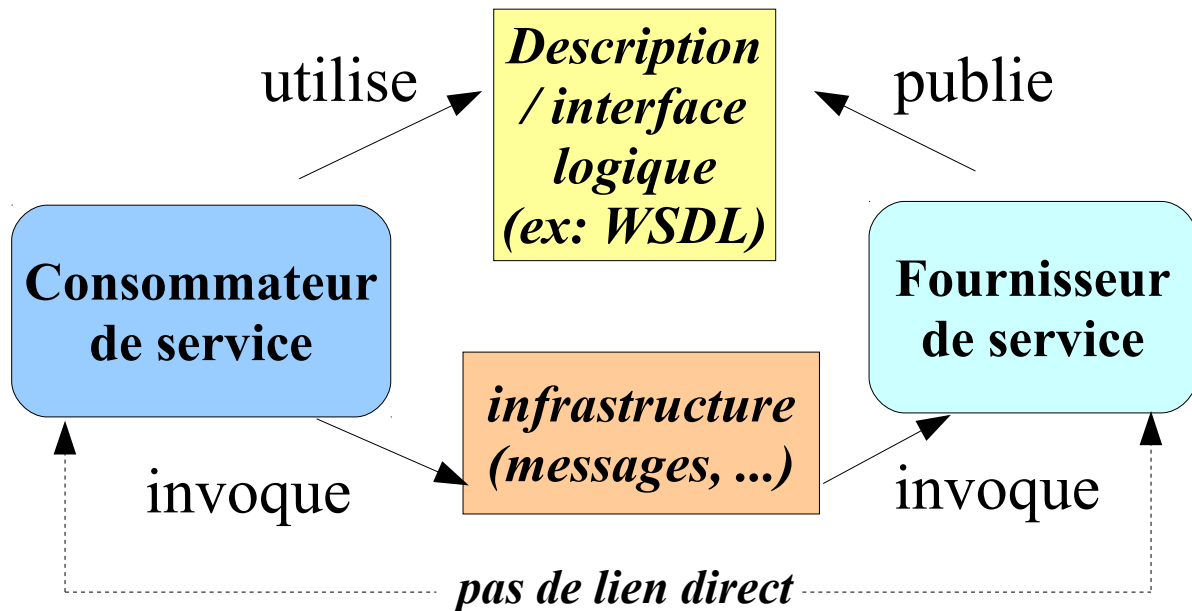
Au sein de l'architecture "SOA" , certains services sont quelquefois programmés avec d'anciennes technologies et l'accès est alors souvent effectué indirectement (via un ESB par exemple).

Autrement dit, dans l'architecture SOA, tous les services ne sont pas systématiquement des "services web" mais par exemple des services "Cobol" , "RMI" , "JMS" , "Corba" , "COM/DCOM" ,

Lorsque l'on développe aujourd'hui de nouvelles applications qui offrent des "services" , on a tout intérêt à utiliser une des technologies "web services" : SOAP ou REST pour éviter des intermédiaires inutiles ou bien pour simplifier les paramétrages d'éventuels intermédiaires facultatifs.

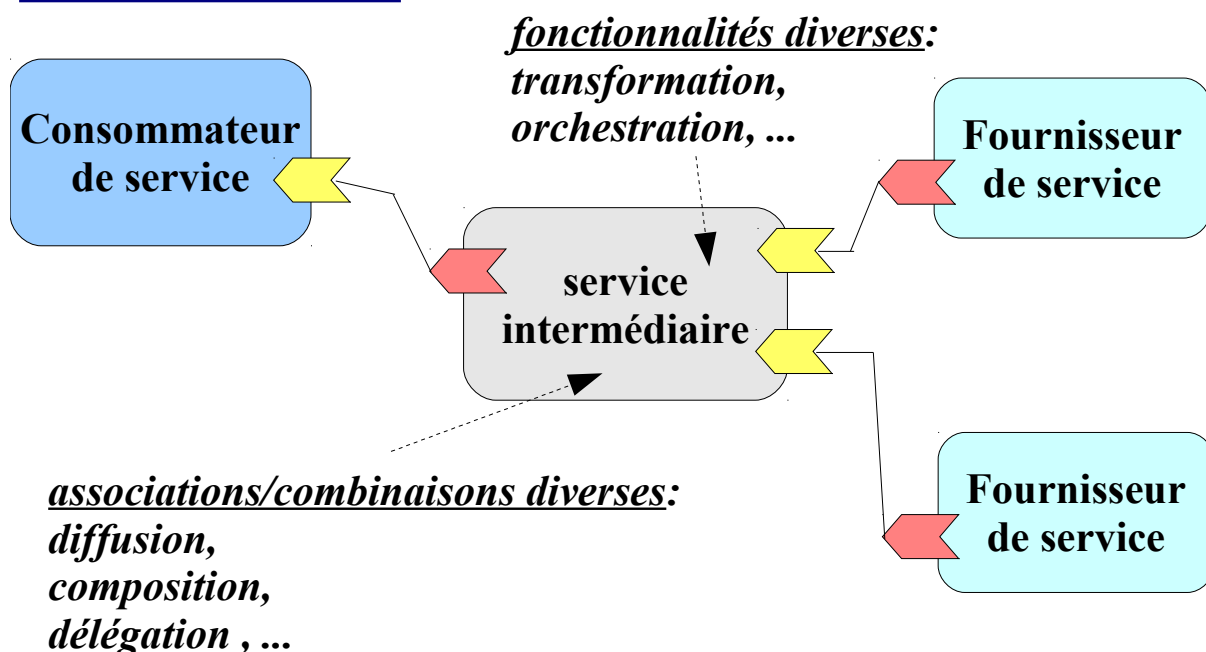
1.4. Découplages entres invocations et implémentations

SOA : Combinaison de services sur le mode "fournisseurs/consommateurs faiblement couplés"

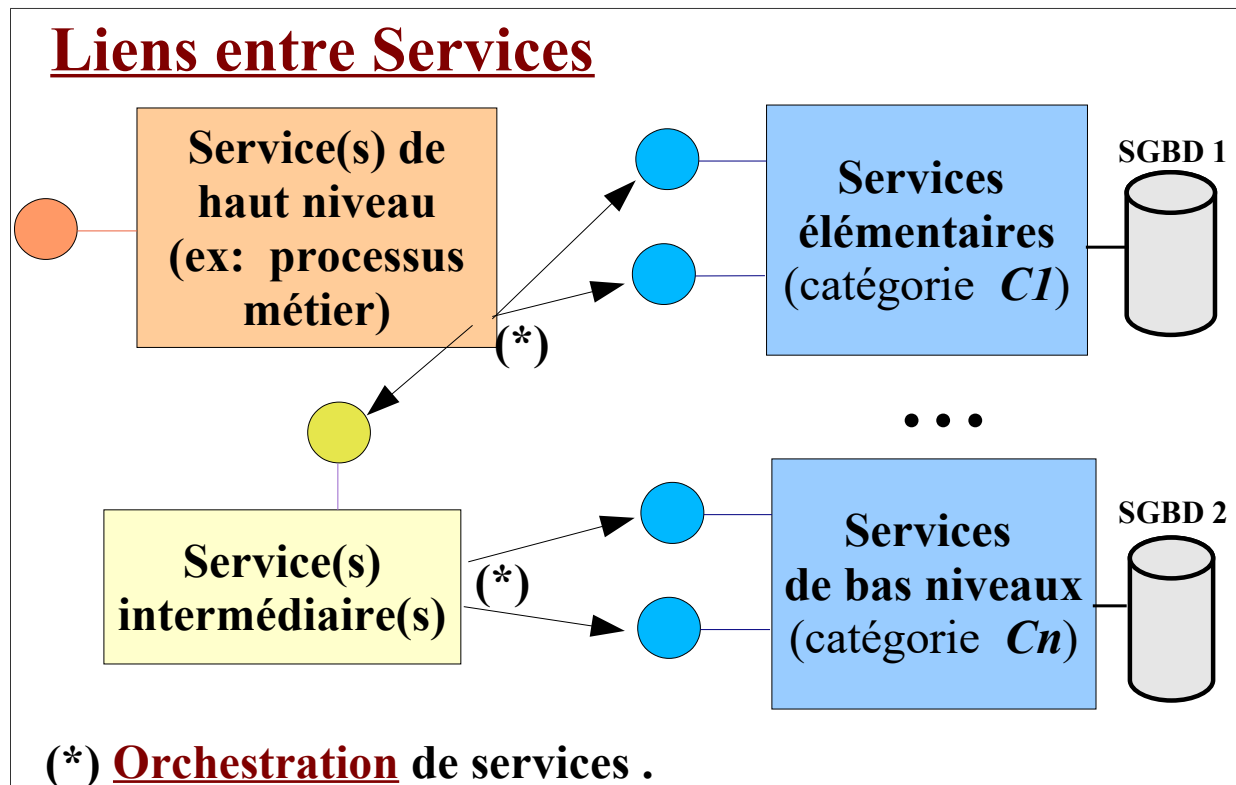


1.5. Consommation et offre de services

SOA : Services intermédiaires = fournisseurs et consommateurs.



1.6. Différents niveaux de services



- Un **service élémentaire** n'appelle pas d'autres services en arrière plan.
- Un service élémentaire est potentiellement associé à une base de données (avec transactions courtes)
- Un **service intermédiaire** (ou bien de haut niveau) appelle toujours au moins un autre service en arrière plan .
- Un service de haut niveau peut quelquefois déclencher des opérations longues (en mode asynchrone).

1.7. SOA au niveau du Middle office , middle-ware

Beaucoup d'éléments de l'architecture SOA (*ESB, adaptateurs, processus BPEL, ...*) sont utilisés sur la partie "**Middle-ware**" d'un SI d'une assez grande entreprise .

Autrement dit ,

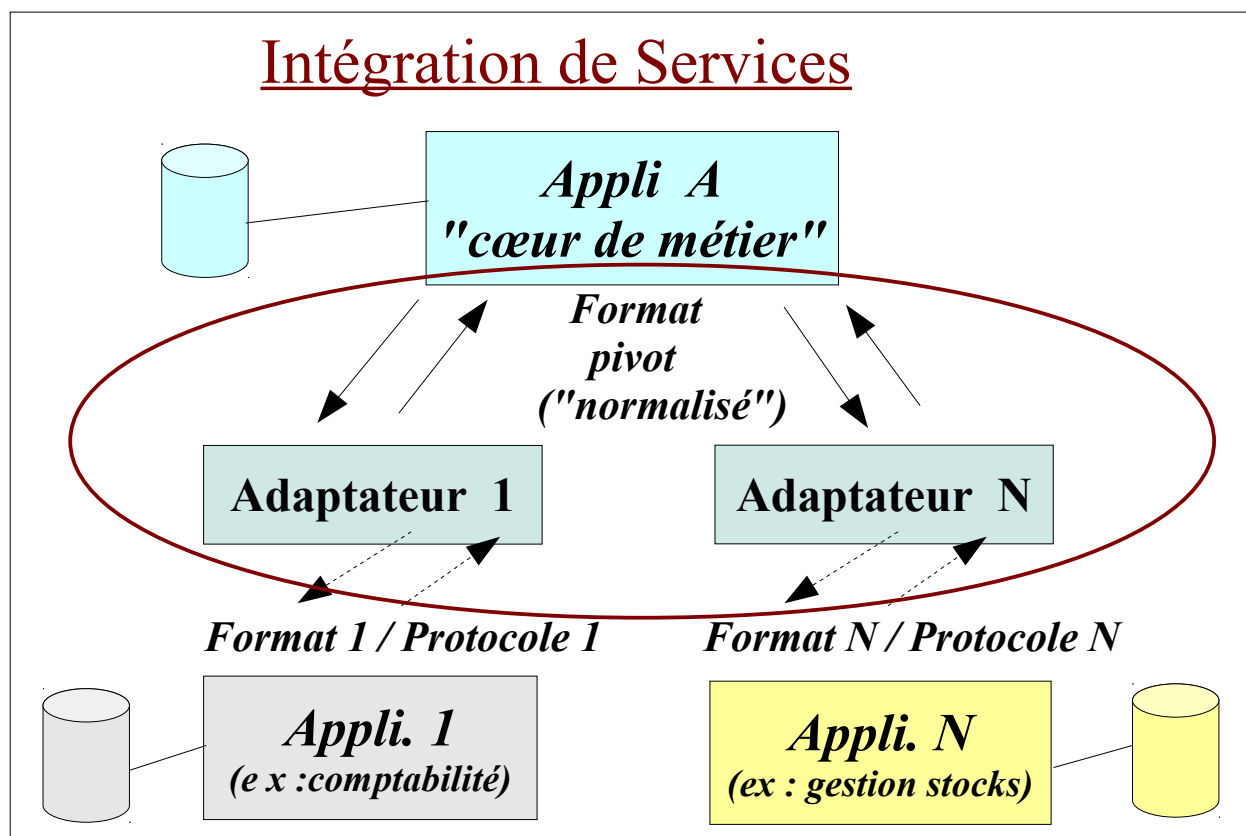
- Mise à part les web services REST, SOA n'est pas souvent directement lié à la partie IHM/web (c'est le rôle de la partie front-office)
- SOA n'est que rarement lié aux SGBDR (c'est le rôle de la partie "back")
- **SOA est avant tout à voir comme un système fonctionnel .**

2. SOA et Intégration

L'architecture SOA est souvent utilisée pour **intégrer** dans un **SI spécifique au cœur de métier** des **applications existantes (non développées en interne mais achetées)** qui **rendent des services transverses ou génériques** (ex: Compta , gestion de Stock, ...).

Pour faire communiquer entre elles des applications provenant de différents éditeurs , on a besoin de tout un tas d'éléments **intermédiaires** fournis par l'**infrastructure SOA** (*ESB* , *Adaptateurs*, ...).

Dans le cas d'une application "clef en main" achetée auprès d'un éditeur de logiciels, on ne maîtrise absolument pas le format exact des services offerts (liste des méthodes , structures des données en entrées et en sorties des appels). Via des paramétrages de transformations "ad-hoc" effectuées au niveau d'un ESB, il sera (en général) tout de même possible d'invoquer cette application depuis d'autres applications internes au SI de l'entreprise.



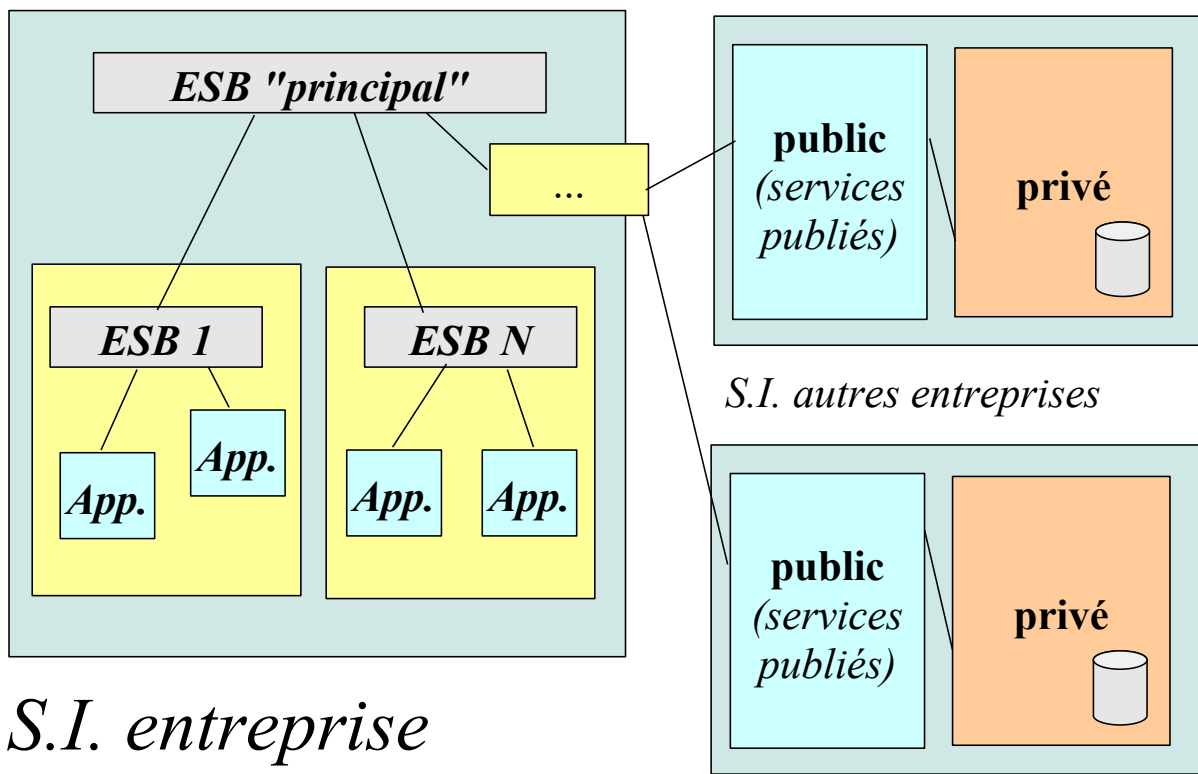
Une ancienne norme SOA du monde Java s'appelait JBI (Java Business **I**ntegration)

Attention, attention :

Les "Web-Services", ESB et "Processus BPM" ne constituent que l'*infrastructure technique*.
Un vrai projet SOA d'entreprise doit absolument comporter en outre un volet "**MODELISATION**" (avec UML et/ou BPMN) qui est vraiment **fondamental** !!!

SOA c'est un peu "**programmer/configurer le SI à grande échelle**".
Vue la grande portée de SOA, l'aspect modélisation est très important .

Très grande portée de SOA



3. Paradigme "tout est service"

Au sein d'une architecture SOA, on peut (dans une première approche conceptuelle) considérer que presque tous les éléments d'un système informatique sont des services :

- services de persistances (lectures & mises à jour dans une base de données ou ...)
- services d'habilitation (vérifier les privilèges d'accès, ...)
- services de vérifications, contrôles,
- services métiers,

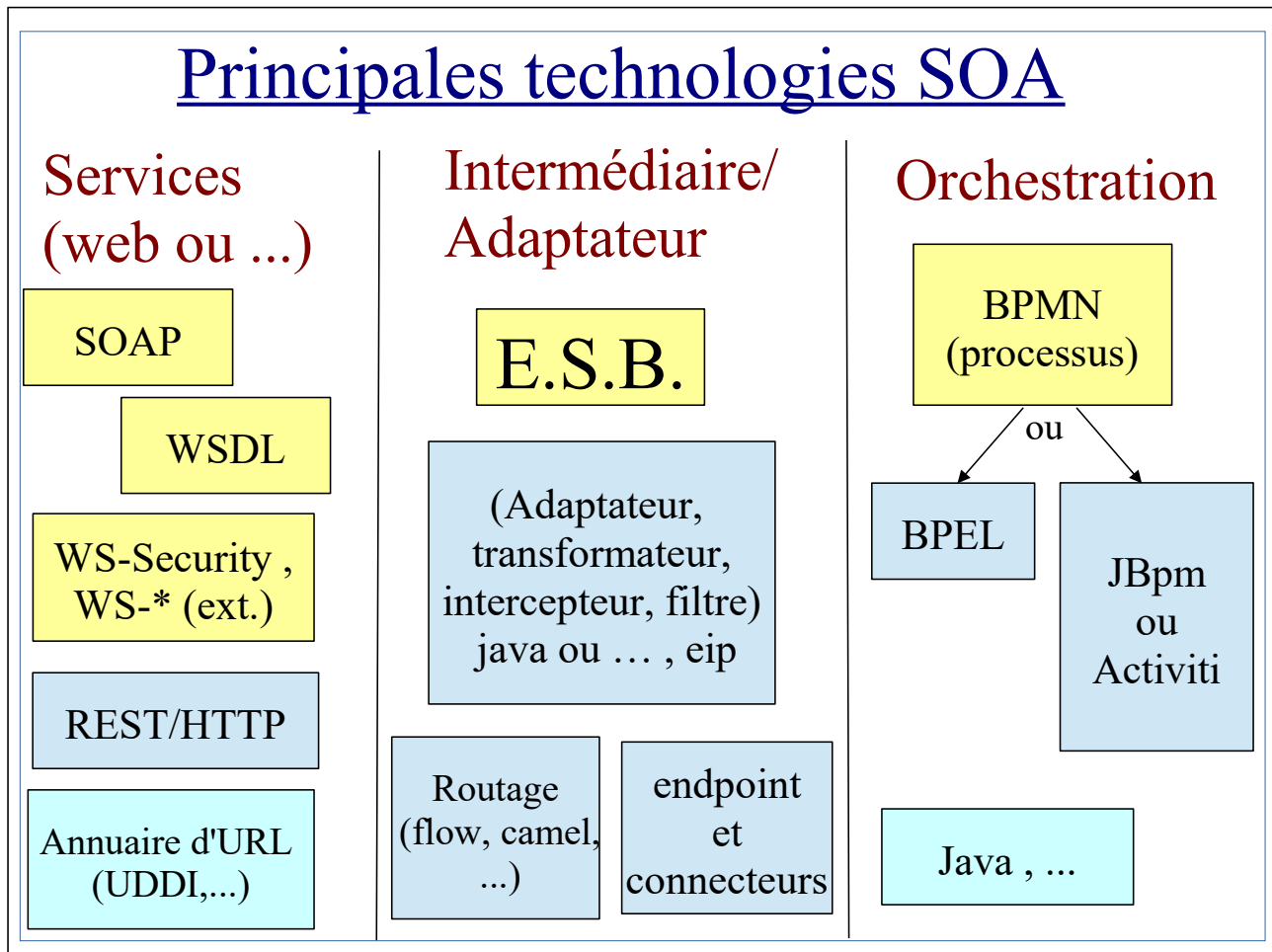
Mise à part quelques contraintes techniques dont il faut tenir compte (transactions courtes proches des données, performances correctes, sécurité, ...), la plupart des services (ou sous services) peuvent se mettre en œuvre de manières très diverses :

- prise en charge directe par l'entreprise ou bien sous traitance (SAAS,) .
- service local, proche ou lointain (géographiquement) .
- service programmé sur mesure ou bien "acheté + adapté"
-

==> Il convient donc d'effectuer une première phase de modélisation SOA en prenant du recul et en ne se focalisant que sur des considérations purement sémantiques/fonctionnels .

Beaucoup d'autres aspects ("organisationnel", "logiciel", "géographique", ...) sont idéalement à modéliser comme des aspects séparés (que l'on règle par paramétrages ou via des implémentations dédiées) .

4. Principales technologies SOA



Attention: il ne faut pas se sentir obligé d'utiliser toutes les technologies "SOA" (quelquefois très complexes) pour la simple raison qu'elles existent.

- Les besoins simples pourront être gérés en utilisant simplement des "web services" (SOAP ou REST).
- Les annuaires de services sont facultatifs et rarement utiles.
- La sécurité doit idéalement être découplée du "pur fonctionnel" (via des intercepteurs) et est souvent prise en charge par des spécialistes.
- Les "ESB" peuvent être vus comme une infrastructure à mettre en production et on peut souvent en faire abstraction en phase de développement.
- L'orchestration sophistiquée (avec "BPEL" ou bien "BPMN2 + java") n'est réellement utile que dans des contextes asynchrones ou bien lorsque l'on recherche une bonne traçabilité entre les modèles et les portions de code à maintenir et faire évoluer.

5. Gains de l'approche SOA et enjeux

5.1. Principaux apports de l'approche SOA :

- **interopérabilité** (java , .net , c++ , php , ...)
- **flexibilité/agilité** (relocalisations et/ou reconfigurations possibles des services via intermédiaire de type ESB)
- **partage de traitements métiers** (règles métiers , services d'entreprise , ...)
- **partage de données** (accessibles via les services)
et donc moins de besoin(s) de "re-saisie" ou de "copie/réplication" .
- **urbanisation plus aisée** (chaque partie peut évoluer à son rythme tant que certains contrats fonctionnels sont respectés en utilisant s'il le faut des adaptateurs).
- ...

Ces apports sont tout de même **conditionnés par** :

- le besoin quasi-impératif d'une **bonne modélisation** SOA (à caractère *transverse*)
- *l'utilisation maîtrisée de technologies SOA* (WS,ESB,...) fiables et performantes.

5.2. Enjeux pour l'entreprise :

- **Potentielles économies d'échelle** (si partage effectif de services mutualisés).
- **Meilleur réactivité** (du fait d'un SI plus souple/flexible/agile) .
- **Meilleurs communications externes** (B2C , B2B) via des services publics visibles à l'extérieur de l'entreprise.
- ...

Et selon la qualité de la modélisation et de la mise en œuvre :

- ***Peut être une maintenance plus aisée***
(il est plus facile de maintenir ou faire évoluer plusieurs petites applications assez indépendantes et qui communiquent via SOA que de maintenir une énorme application monolithique).
- Des performances à peu près aussi bonnes (si ESB bien choisi et bien paramétré).

5.3. Enjeux stratégiques (liés au cloud-computing)

Le concept de **cloud-computing** (à dimensionnement flexible) est actuellement à la mode et considéré quelquefois comme un enjeu stratégique .

La mise en œuvre du "**cloud-computing**" s'appuie essentiellement sur :

- **SaaS** (software as a service)
- **Paas** (platform as a service)
- **Iaas** (infrastructure as a service)

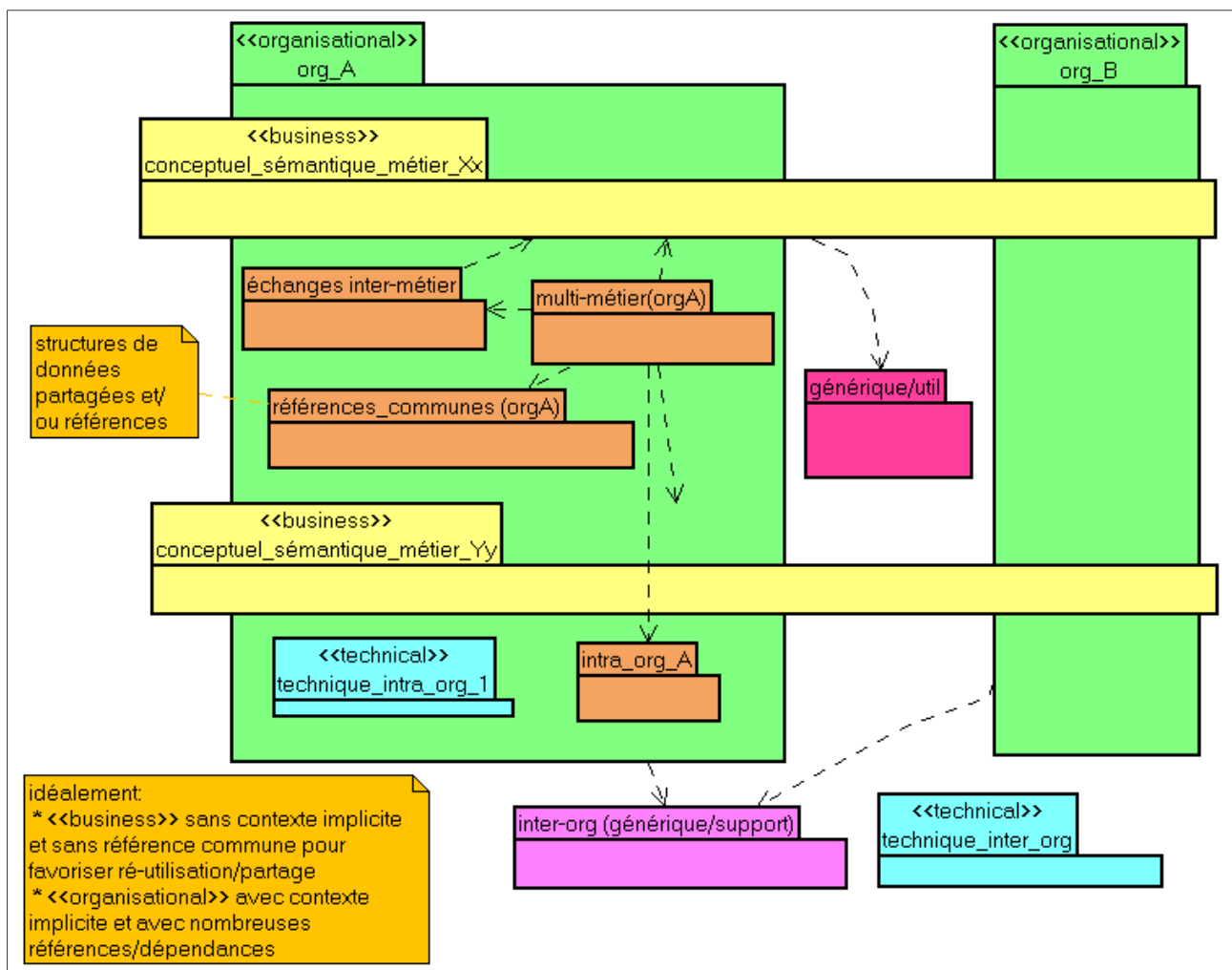
6. Quelques typologies de services à assembler

6.1. techniques ou métiers ou organisationnels

<i>Types de services</i>	<i>Principales caractéristiques</i>
Service purement "métier"	Lié à un domaine métier précis (ex : comptabilité , facturation , ...) et idéalement réutilisable
Service "organisationnel"	Service spécifique à une organisation/entreprise , s'appuie souvent sur un ou plusieurs services "métier"
Service "technique/utilitaire"	Service utilitaire (plutôt technique) . Ex: impression , mailing , ...

Distinguer au besoin :

- domaines purement métiers (comptabilité , R&D , marketing , ...) avec services métiers élémentaires
- domaines organisationnels (départements , centres de gestion ,) avec contextes
- domaines purement techniques (facilement réutilisables , sans contexte , ...)



A également généralement distinguer : "**cœur de métier**" et "**support**"

6.2. Niveaux de complexité des services :

Services élémentaires (de bas niveaux) :

- Accès (potentiel) à des données , des règles métiers , ...
- Souvent liés à un seul secteur (ou une "catégorie") métier
- Quelquefois techniques (Authentification / Habilitation , Impression, ...)
- Services métiers distants (avec transactions courtes) [ex : Spring , EJB]
- ...

Services intermédiaires techniques :

- Adaptateurs (format, ...), ...

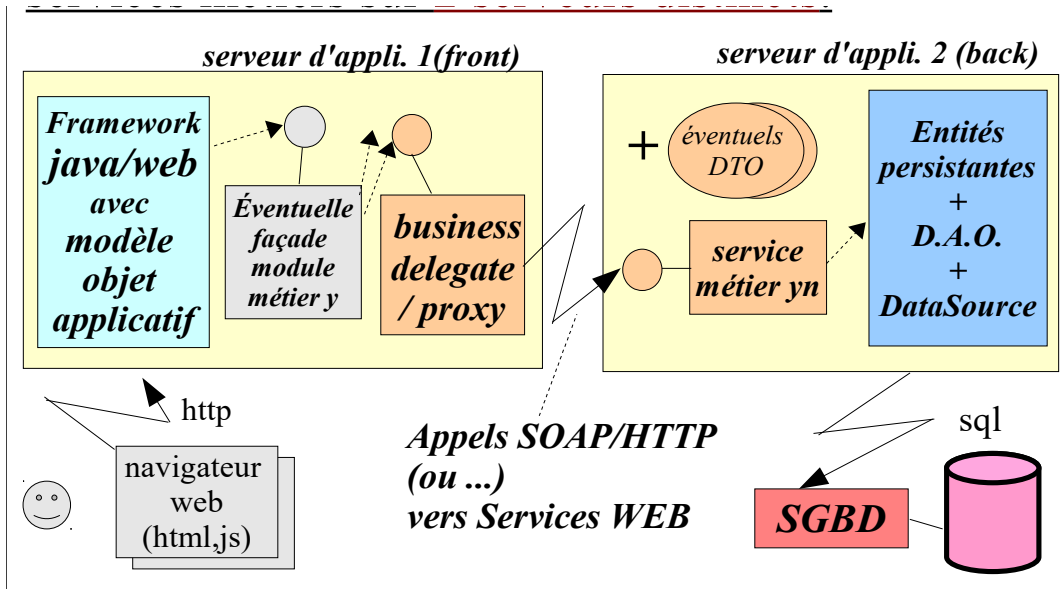
Services intermédiaires fonctionnels:

- recombinaisons fonctionnelles (composition, assemblage , délégation ,)
- orchestrations simples (en mode synchrone) , ...

Services d'orchestrations évoluées (de hauts niveaux) :

- Processus métier (avec orchestration) → technologies **BPEL** ou **activiti** ou **jBpm**
modélisation **BPMN** , **BPMN2**
- processus potentiellement longs , opérations asynchrones,

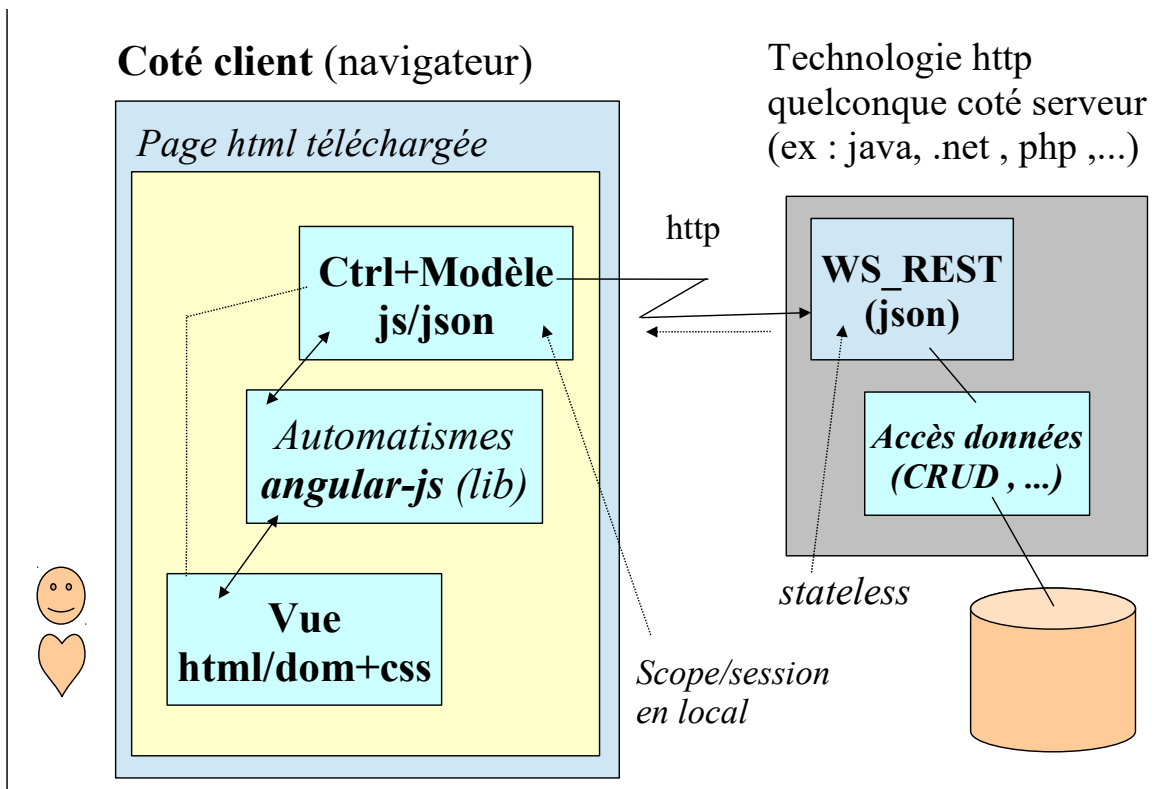
7. Cadre classique pour services élémentaires



Le schéma ci-dessus montre une architecture logicielle classique (ici basée sur Java/JEE) où les communications inter-applications sont centrées autour d'appels de méthodes distantes (sur des Web-Services "SOAP").

L'application qui fournit le service est souvent prise en charge par des technologies traditionnelles (Java/JEE, .net, ...) prenant en charge les transactions courtes et l'accès aux SGBDR.

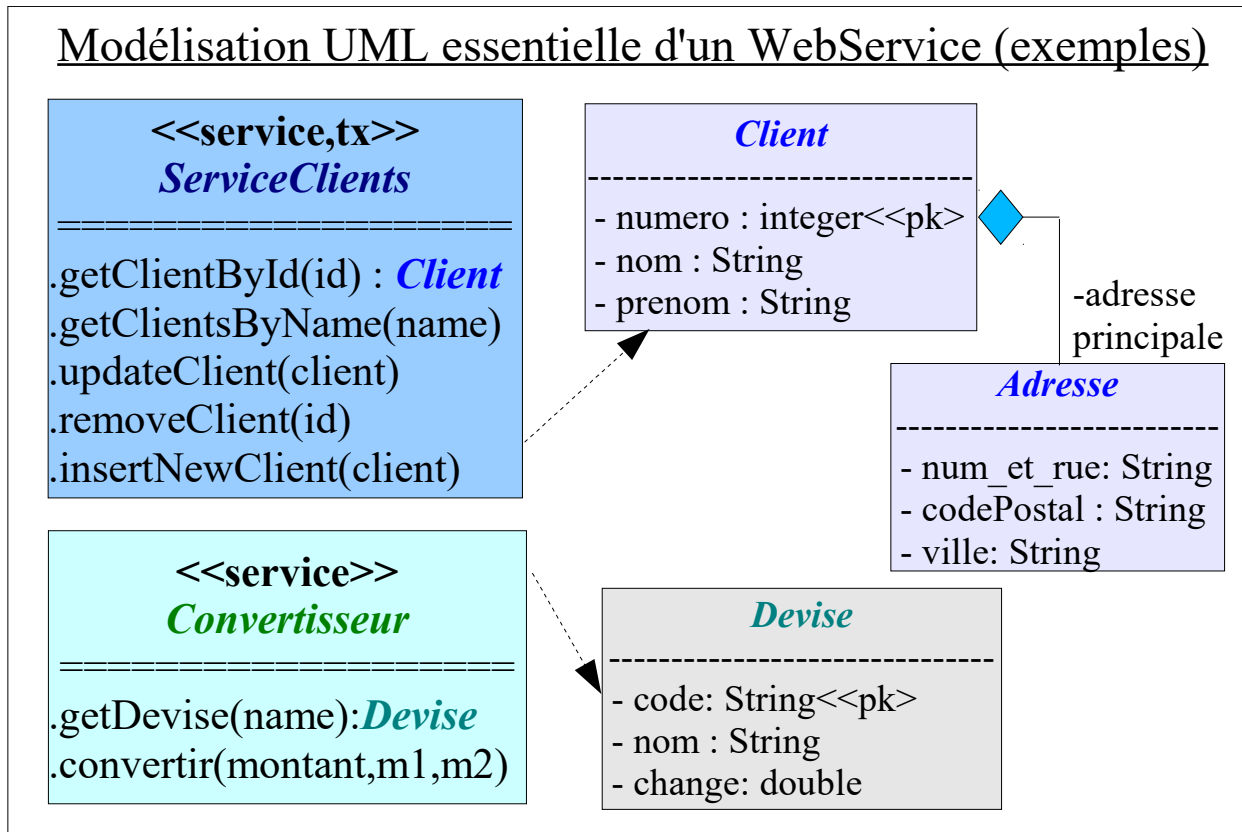
Pour ne pas limiter l'illustration à des technologies purement "RPC", voici un autre exemple (ou type) de service élémentaire :



Message de réponse = représentation (UML → JSON) d'une ressource téléchargée.

8. Modélisation des services élémentaires

8.1. Modélisation essentielle visée



Les **services élémentaires** sont essentiellement modélisés comme des **classes (ou interfaces) UML**. Un service (en vision RPC) est avant tout une **liste d'opérations (méthodes distantes)**.

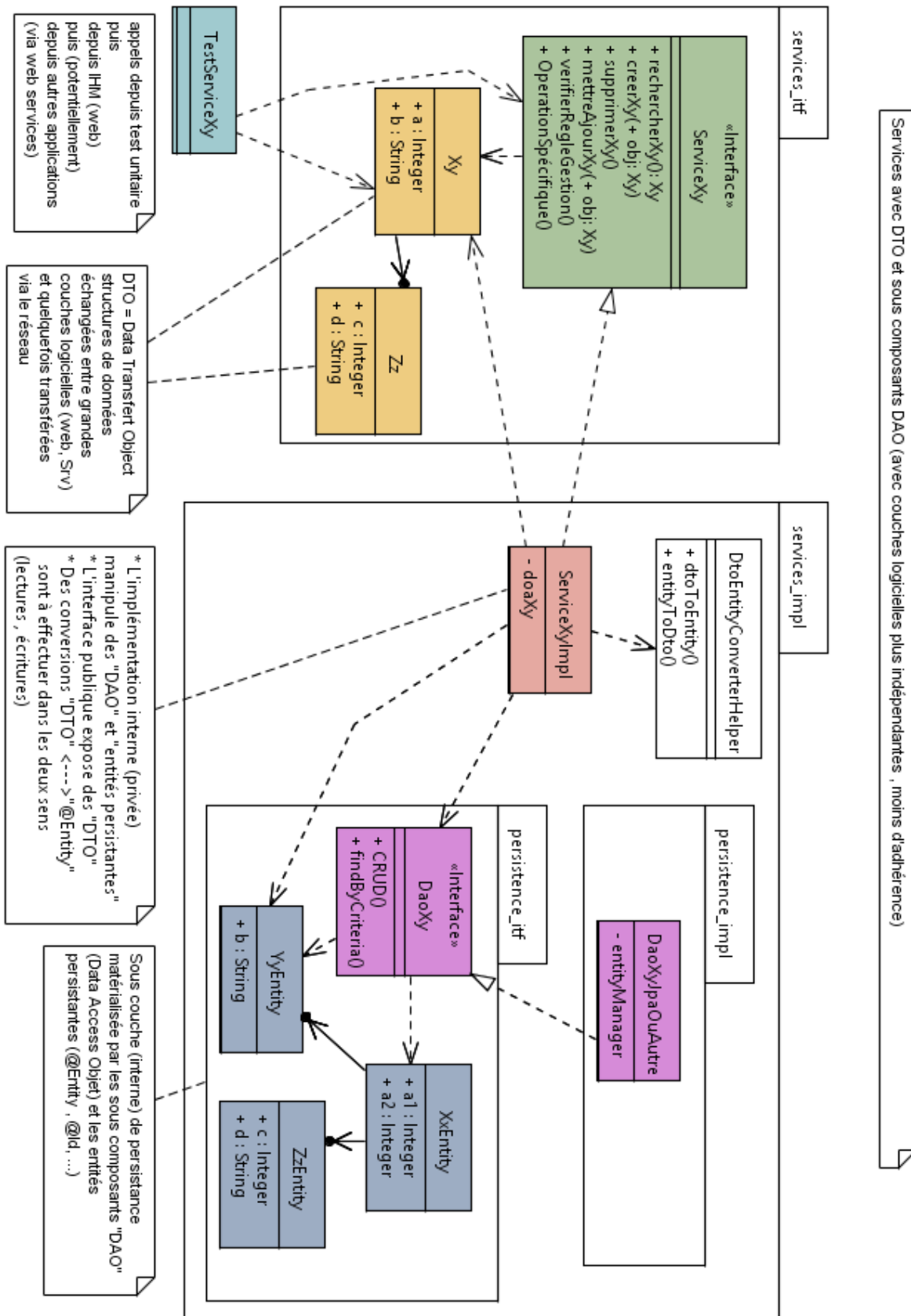
Pour que la **modélisation** du service soit **précise** il faut **indiquer** tous les **types** de données (pour les **paramètres d'entrées** et les **valeurs de retour**).

Dans de très nombreux cas les types des paramètres sont des structures objets complètes (*ici "Client", "Adresse" et "Devise"*) qui se modélisent comme d'autres classes UML.

Autre précision quelquefois importante: "**package englobant UML**" → namespace UML

8.2. Dto exposés par les services

Dans la cadre de la modélisation SOA , les structures de données exposées (en entrées et en sortie) par les méthodes d'un service sont à considérer comme des données d'échanges (représentation externes quelquefois différentes du format d'implémentation interne) .



9. Services fonctionnels (définition de la catégorie)

Un **service** (purement) **fonctionnel** est un **service intermédiaire** (non élémentaire) qui s'appuie à son tour sur d'autres services .

Autrement dit : "un service peut en cacher un autre " . Ce qui est très fréquemment le cas dans une architecture SOA.

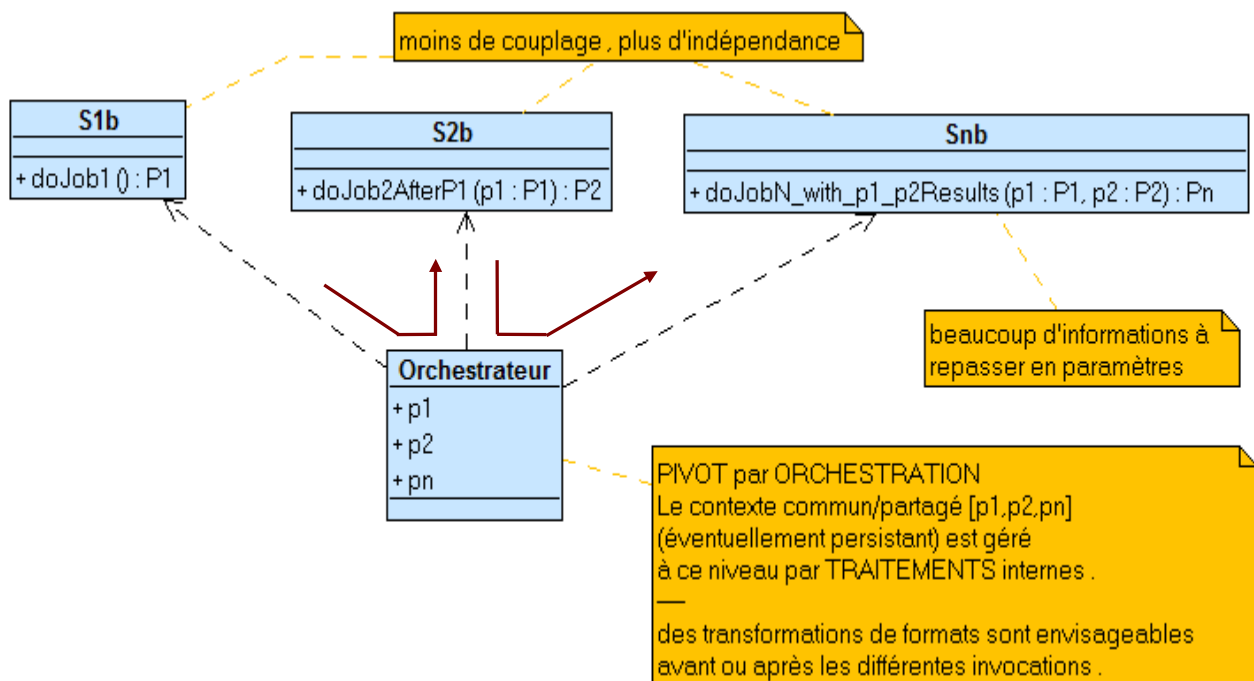
Contrairement aux services évolués d'orchestrations (souvent avec des parties asynchrones) , **un service fonctionnel reste simple** (*traitements synchrones et orchestration élémentaire*) .

Un service fonctionnel doit pouvoir être programmé et mis en œuvre avec des technologies simples (ex : classe java) . Pas besoin de bpmn / bpel .

Un service fonctionnel effectue essentiellement des **recombinaisons fonctionnelles** (*composition, assemblage , délégation , appels conditionnés ,*)

9.1. Données "pivots" (échangées)

Données "pivot"



Récupérées comme résultat d'un appel sur un premier service et ré-envoyées en entrée d'un autre service, certaines données pivotent (telles quelles ou retransformées) au niveau de l'orchestrateur.

...

9.2. Modélisation des services fonctionnels

La modélisation d'un service fonctionnel comporte **trois aspects complémentaires importants** :

- **structure du service rendu**
(même modélisation que pour un service élémentaire)
- **dépendances vis à vis d'autres services**
(via "dependency UML" au sein d'un diagramme de classes ou ...)
- **logique / algorithme des sous-appels internes**
(via "**diagrammes de séquences UML**" ou bien "**diag d'activités UML**" ou bien ...)

10. Notions essentielles d'urbanisation

10.1. Urbanisation du S.I. (présentation)

L'objectif principal consiste à *faire évoluer les différentes parties du SI de façon convergente* (en respectant les mêmes grandes lignes directrices) et à *faire cohabiter les parties existantes* au sein d'une *structure globalement découpée en zones/quartiers/blocs*.

Ce découpage en **modules** et sous modules **autonomes** vise à :

- garantir une certaine liberté d'implémentation
- clarifier les zones d'échanges (communications) entre les différentes parties du SI.

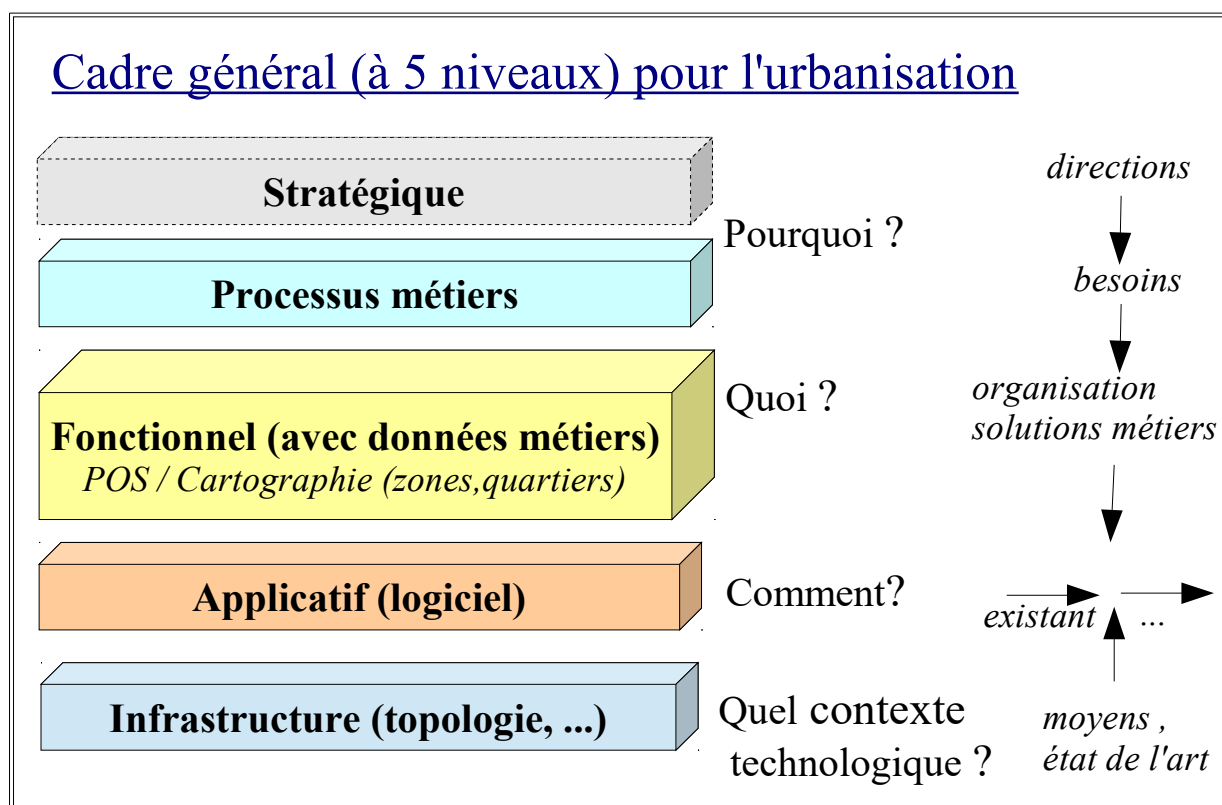
Plus particulièrement, l'urbanisation vise :

- à renforcer la capacité à construire et à intégrer des sous-systèmes d'origines diverses,
- à renforcer la capacité à faire interagir les sous-systèmes du SI et les faire interagir avec d'autres SI ([interopérabilité](#)),
- à renforcer la capacité à pouvoir remplacer certains de ces sous-systèmes (interchangeabilité).

et de manière générale pour le SI à :

- favoriser son évolutivité, sa pérennité et son indépendance,
- renforcer sa capacité à intégrer des solutions hétérogènes ([progiciels](#), éléments de différentes plate-formes, etc.).

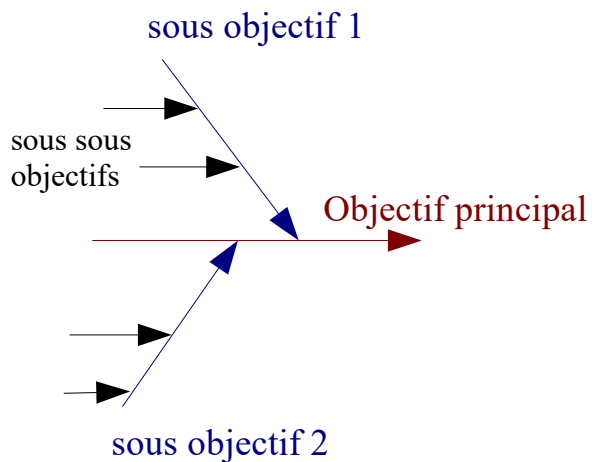
10.2. Cadre général / classique à 5 niveaux



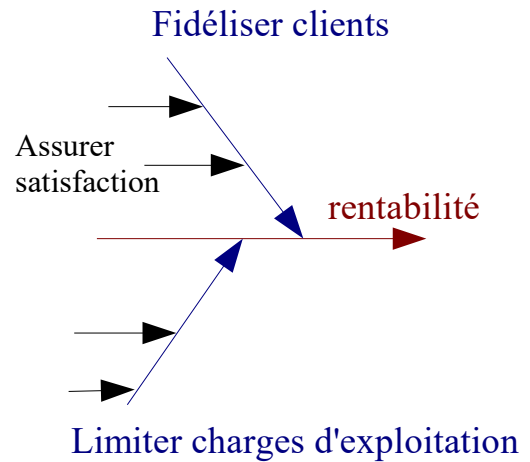
10.3. Niveaux "stratégiques" et "processus"

Expression des directions stratégiques

Diagramme d' Ishikawa



exemple



Processus métiers

Objectifs métiers
(et *sous objectifs*)

éventuellement
schématisés via

*Business
Uses Cases*

à atteindre en développant des

Processus métiers
(et sous processus)

généralement modélisés via des

Diagrammes d'activités
(et sous diagrammes)

UML

ou
bien

BPMN
(*Business Process
Model and Notation*)

11. Urbanisation fonctionnelle

Objectif principal : **Bien ranger , bien s'organiser** et surtout "**éviter les doublons**".

11.1. Zones , quartiers , blocs

L'urbanisation consiste à découper le SI en modules autonomes, de taille de plus en plus petite :

- les **zones**,
- les **quartiers** (et les **îlots** si nécessaire),
- les **blocs** (blocs fonctionnels).

Exemple :

- zone production bancaire
 - quartier gestion des crédits
 - îlot gestion des crédits immobiliers
 - bloc fonctionnel gestion d'un impayé

Entre chaque module (zone, quartier, îlot, bloc) se dessinent des **zones d'échange d'informations** qui permettent de *découpler* les différents modules pour qu'ils puissent évoluer séparément tout en conservant leur capacité à interagir avec le reste du système.

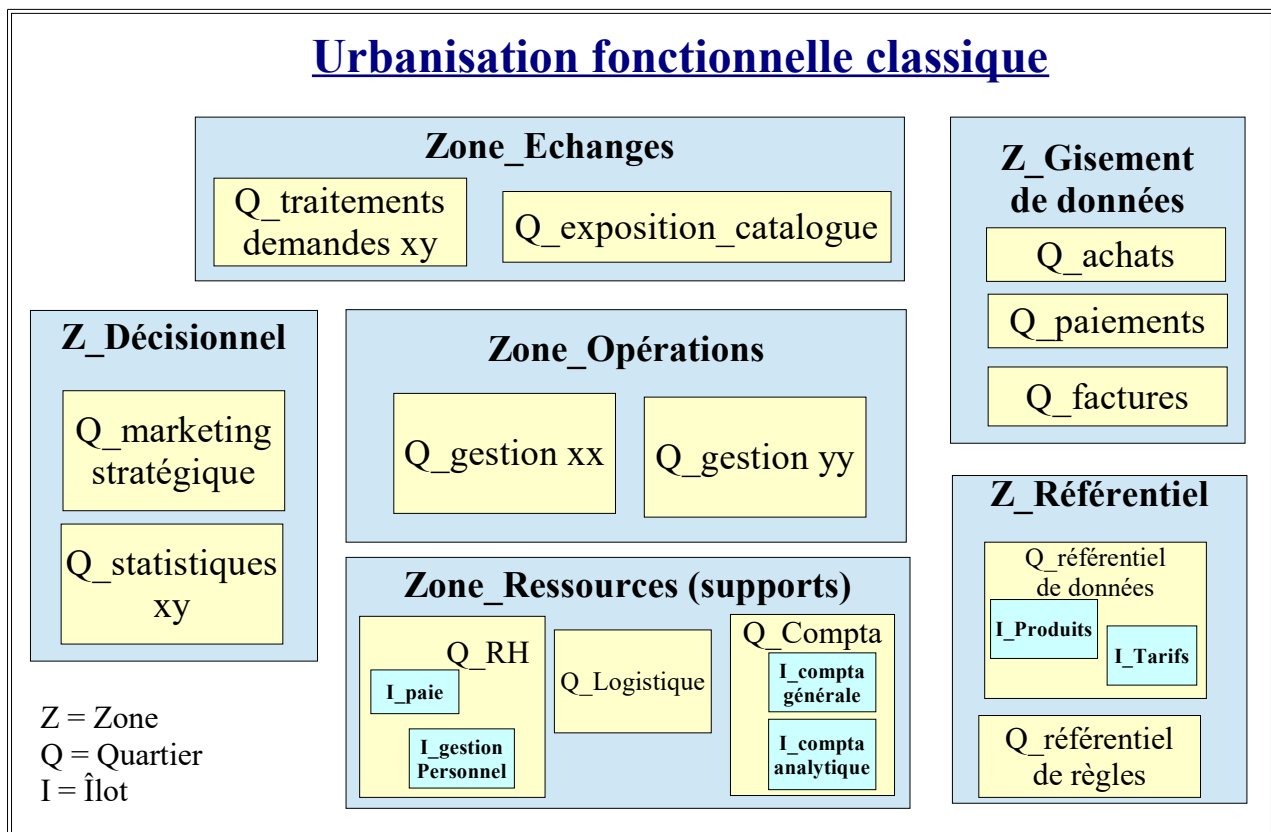
Eléments de cartographie fonctionnel du SI

Domaines (types de Zones)	<<échanges>>, <<référentiel et gisement>> , <<opérationnel>> , <<ressources/support>>, <<décisionnel>>
Zones (fonctionnelles)	Regroupement homogène (métier , organisation/ entreprise , ...)
Quartiers (fonctionnels)	Secteur métier (ou para-métier)
Îlots (fonctionnels)	Souvent via une application (ou une partie d'un ERP , ...)

11.2. Les principaux types de zones

Zones	Caractéristiques
<i>échanges</i> avec l' <i>extérieur</i> du SI	<u>acquisition</u> /émission de/vers les <u>partenaires</u> : <u>clients</u> , <u>fournisseurs</u> , etc. (prise du S.I.) Zone incluant les "IHM" exposées ,
<i>activités opérationnelles</i> (<i>cœur de métier</i>)	Zone(s) "cœur de métier" (incluant par exemple des quartiers de types gestion des opérations bancaires, gestion des opérations commerciales, gestion des opérations logistiques internes, etc.)
gestion des <u>données de référence</u> communes à l'ensemble du SI	les <u>référentiels</u> de <u>données structurées</u> (<u>données clients</u> , <u>catalogue</u> de <u>produits</u> et <u>services</u> , etc.) (référentiel de données stables)
gestion des <i>gisements de données</i> (également à voir comme un référentiel commun)	ensemble des <i>informations produites quotidiennement</i> , communes à l'ensemble du SI (données de production, etc.) (référentiel de données qui évoluent rapidement)
<i>Ressources (activités de support)</i>	<u>comptabilité</u> , <u>ressources humaines</u> , etc.
aide à la <u>décision</u> et le <i>pilotage</i>	<u>informatique décisionnelle</u> .

Urbanisation fonctionnelle classique



à évidemment adapté en fonction :

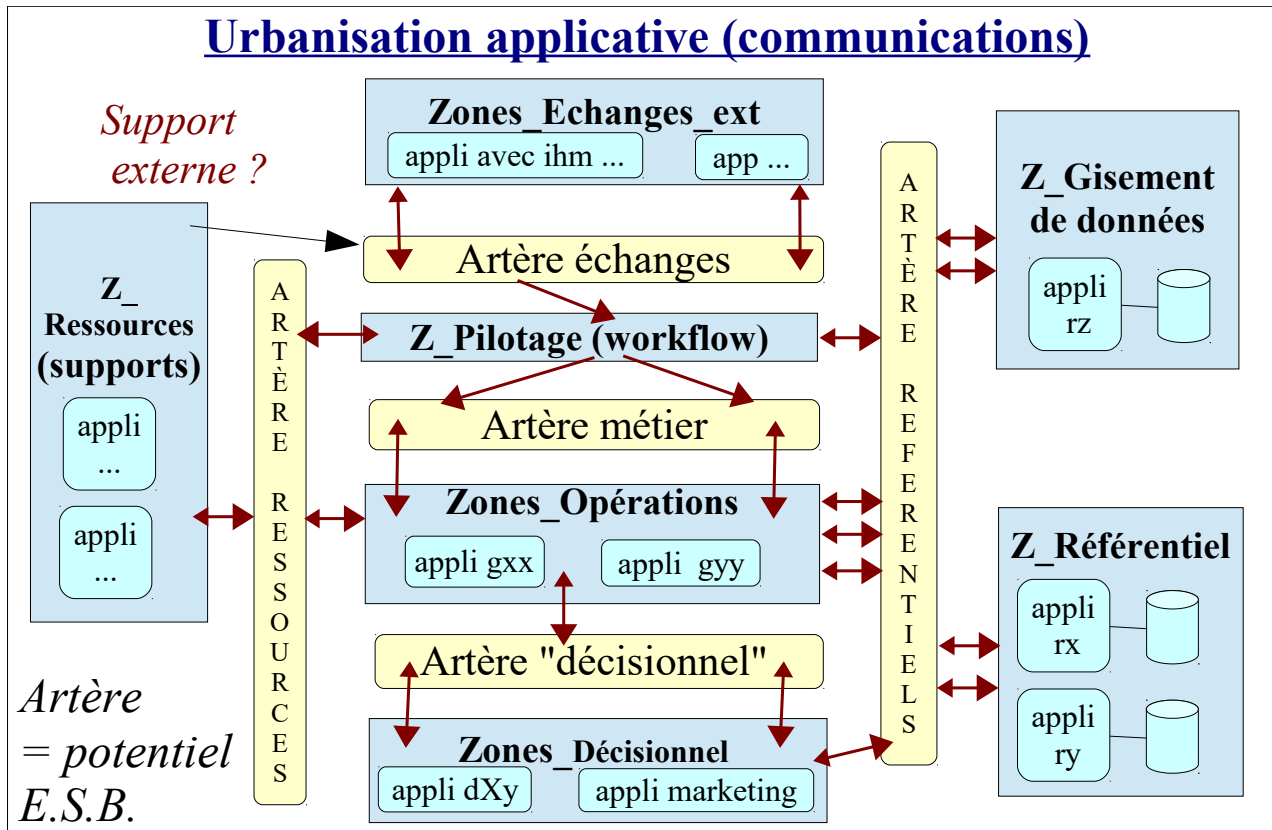
- du cœur de métier (banque, assurance , automobile,)
- de la taille de l'entreprise (TPE , PME, GE)
- de l'existant

- des choix stratégiques
- ...

Si plusieurs "cœurs de métier" coexistent, on aura alors plusieurs "zones opérationnelles".
Certaines zones sont censées être uniques (ex : référentiels, échanges_extérieurs, décisionnel, ...)

12. Urbanisation applicative

Un exemple (discutable) parmi plein d'autres variantes / alternatives :



La Zone de pilotage (plutôt organisationnelle et en partie technique) a pour rôle de s'occuper

- des orchestrations de services métiers
- des workflows à base de communications asynchrones
- ...

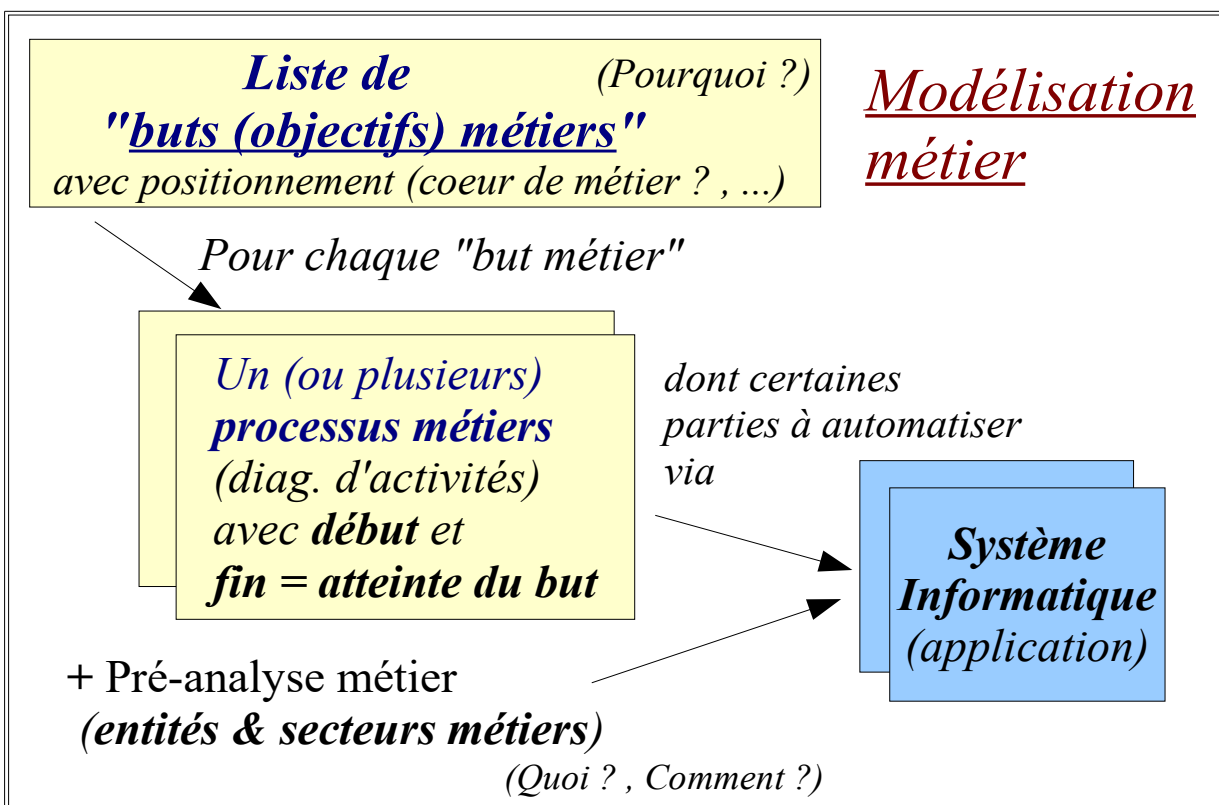
Ce diagramme (à grande échelle) ne montre que les principales artères de communication. Il peut aussi exister des canaux d'échanges entre les quartiers (au cas par cas).

13. Business Modeling et "Business Uses Cases"

13.1. (objectifs métiers)

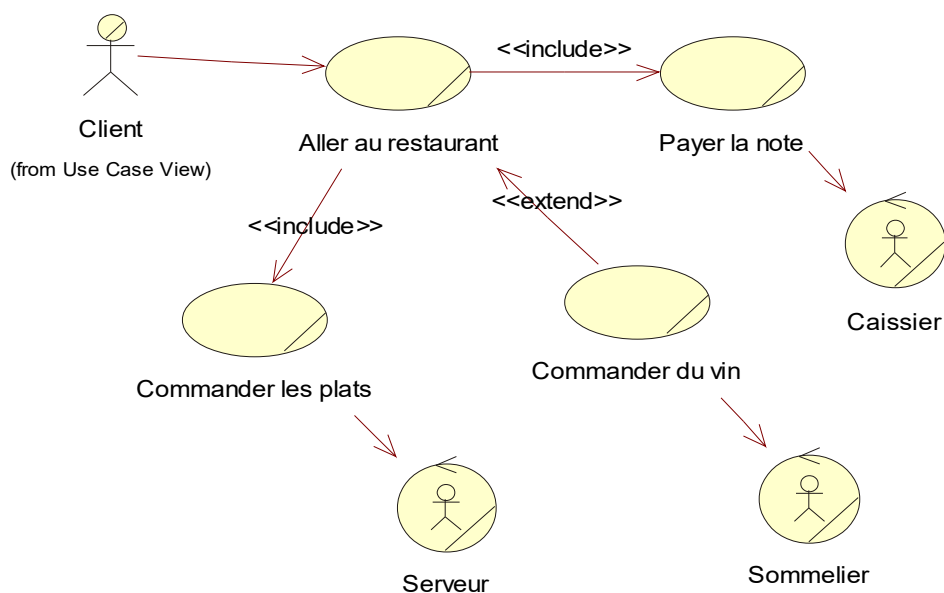
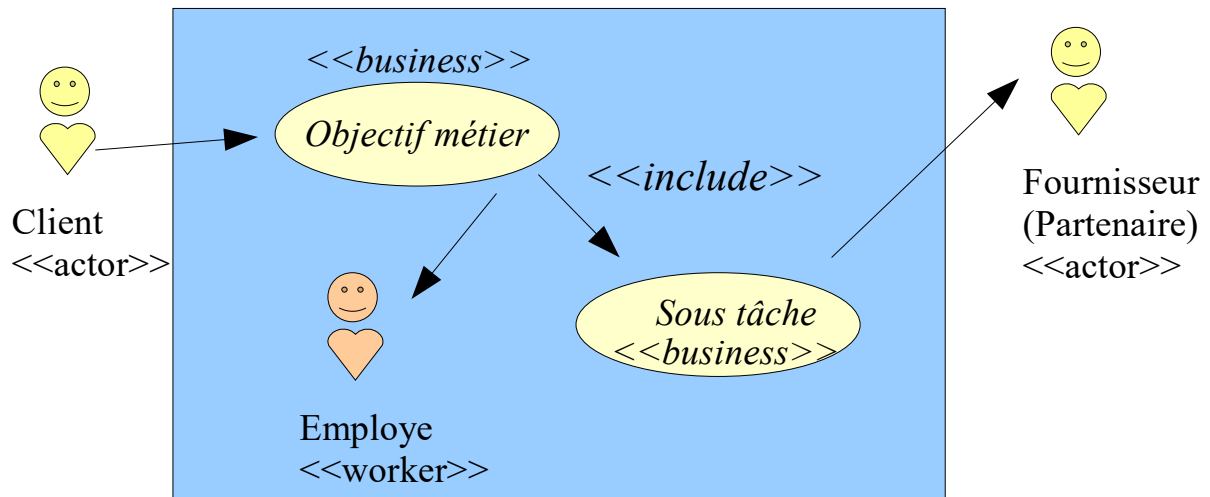
Modélisation métier (*business modeling*)

- Expression du "**pourquoi ?**" (*quelle(s) utilité(s) ? , quels objectifs ? , ...*)
- **Contexte très large** (entreprise + partenaires , ...) *dépassant les frontières d'un seul système informatique*
- Segmentation (*découpage/regroupement*)
--> **secteurs métiers** , packages métiers , ...
- **Processus métiers** (*diagrammes d'activités , ...*)
avec activités informatisées ou non .



13.2. Business Uses Cases

Portée définie par le diagramme des "**business use case**"



Un diagramme de "*business uses cases*" ou "*cas d'utilisations métiers*" est une extension pour UML (provenant de RUP) et qui vise à **montrer les fonctionnalités d'un service ou département d'une l'entreprise** plutôt que les fonctionnalités d'un système informatique précis.

En plus de la notion d'acteur UML (implicitement externe), le stéréotype **<<worker>>** (ici associé aux caissier, sommelier et serveur) désigne **une personne interne (ex: employé)**.

14. Cas d'utilisation d'une partie du SI (soa)

Dans le cadre d'une modélisation UML, les cas d'utilisations ("métiers" ou "applicatifs") sont intelligibles dans le cadre d'un contexte bien précis (SI, sous système, application xy).

Les "processus métiers" (diagrammes d'activités UML ou bien diagramme BPMN) pourront ensuite être considérés comme :

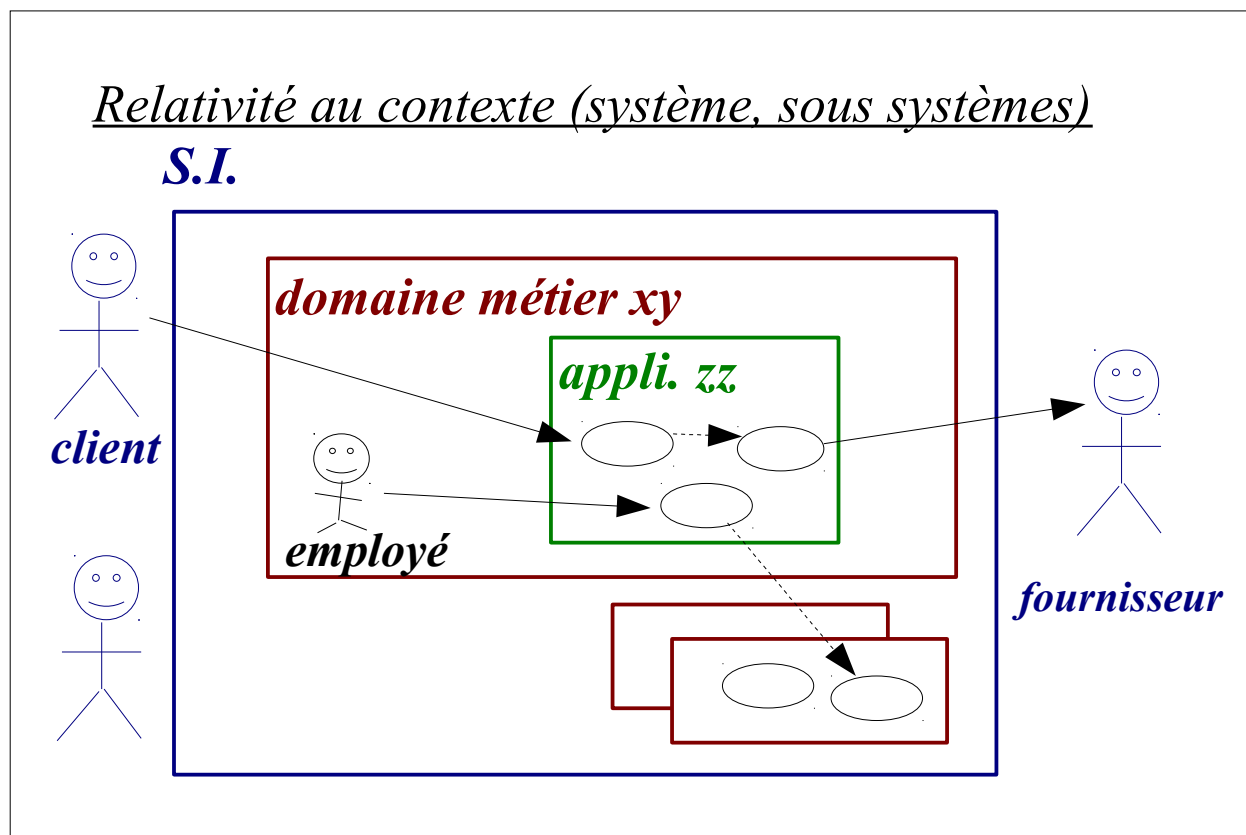
- des illustrations des scénarios des uses cases
- des macro-procédés fonctionnels permettant d'atteindre l'objectif rattaché au cas d'utilisation
- ...

Selon le cadre et la portée exacte de la modélisation SOA, on pourra être amené à :

- modéliser UC et processus au niveau macroscopique "S.I. Complet"
- modéliser UC et processus au niveau d'un domaine/secteur métier précis
- modéliser UC et processus au niveau d'une application précise

Si on se place dans un cycle itératif et incrémental, on a généralement à la fois besoin de :

- parfaire la modélisation macroscopique de l'ensemble du S.I.
- modéliser précisément (ou moins) un secteur métier du S.I.
- étudier les fonctionnalités et/ou l'intégration d'une application



Dans le schéma ci-dessus :

- l'**employé** est un **acteur extérieur** vis à vis de l'application "zz" et est un **"worker" interne** par rapport au SI et par rapport au domaine métier "xy" .
- Les "Client" et "Fournisseurs" sont des **acteurs extérieurs** pour tous les niveaux.

L'incidence du contexte par rapport à la modélisation est à évaluer au cas par cas .

Il est conseillé de rappeler le contexte courant dans les diagrammes pour éviter toute ambiguïté.

15. Modélisation des processus métiers

D'un point de vue méthodologie de modélisation, les processus métiers (diagrammes d'activités) peuvent être vus comme des illustrations des scénarios des cas d'utilisation du SI.

D'un point de vue fonctionnel, les "**processus métier**" constituent *le cœur dynamique de la modélisation métier/soa*.

Ces "processus métier" encapsulent :

- la *logique métier* principale (*stratégie / procédé*) pour atteindre un objectif métier
- l'*identification des éventuels partenaires* (fournisseurs, services externes, ...)
- une représentation des *données échangées* dans le cadre du "*workflow*".
- la prise en compte des *alternatives* (selon les *événements* susceptibles d'intervenir)
- la prise en compte de certaines *exceptions* (avec *actions de compensation*).
-

Quelques généralités sur les diagrammes d'activités représentant les processus métier :

- il doit y avoir au minimum un **début**, un **fin** et un *chemin passant* au milieu.
- Il ne doit normalement pas y avoir d'impasse bloquante (il faut éventuellement prévoir des alternatives ou des annulations explicites)
- ...

15.1. via diagrammes d'activité UML ou via "BPMN"

Le diagramme d'activités du formalisme UML est en théorie assez approprié pour modéliser des "processus métier". En pratique, sa mise en œuvre est plus ou moins simple selon l'ergonomie de l'outil UML utilisé.

UML offre certains avantages (par rapport aux diagrammes d'activités de BPMN) :

- toute la modélisation (Uses Cases, diag de classes, activités, ...) peut se faire avec un même et seul outil. Il est quelquefois possible d'établir les relations entre les différents diagrammes (pour naviguer de l'un à l'autre).
- au cas par cas selon outil UML

==> Etudier l'**annexe "UML"** pour approfondir la syntaxe.

BPMN offre certains avantages (par rapport aux diagrammes d'activités de UML2) :

- syntaxe plus claire pour les personnes du monde "fonctionnel" (non technique)
- outils/éditeurs (pour créer les diagrammes) plus simples à utiliser
- transposition BPMN (vers BPEL ou bpmn+java) plus aisée
-

16. Pièges à éviter :

16.1. Structures de données échangées

- confondre structure de données internes (entités persistantes) et données échangées (DTO) .
- ne pas séparer "Services en lectures/recherches/consultation" et "Services en écritures/modifications/suppressions" --> paramétrages de sécurité plus difficile
- se dire "peut importe le format des données échangée , on pourra toujours adapter si besoin avec un esb et un service d'aptation" : c'est souvent "reporter la complexité " , "renvoyer la patate chaude" , une réflexion sur les données échangées est souvent primordiale au niveau "SOA" .
- Inversement, ne peut vouloir trop en faire (ex : gérer différents formats en //)

16.2. Format de données pivots et gouvernance des données

Une **bonne approche pragmatique** souvent appelée "**gouvernance des données**" consiste à garantir une bonne agilité du SI en appliquant les dispositions suivantes:

- *applications périphériques complètement indépendantes*
- **éléments collaboratifs principaux (cœur de métier) avec données "pivot" dont la structure est en partie déterminée par gouvernance (prise en compte de multiples besoins).**

Soit appli1

---> tx1(requête_format1)

-----> requête_format_pivot

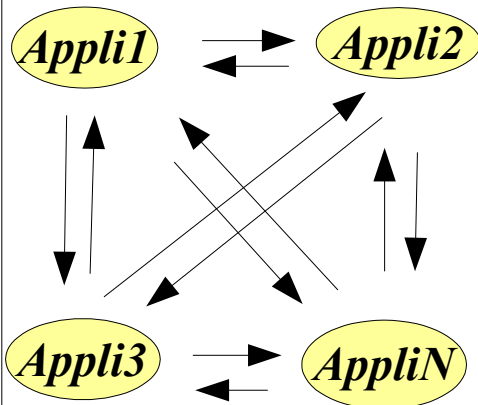
(format normalisé par organisation)

--->tx2(vers_format_appli2)

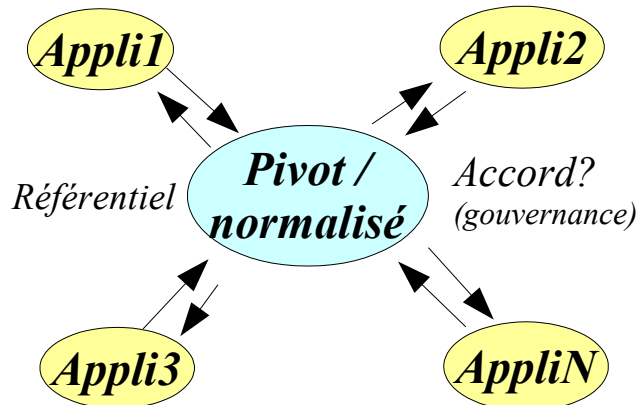
---> requête_format2 reçu par appli2

où tx1(...) et tx2(...) sont deux **transformations** (éventuellement) nécessaires.

SOA : format normalisé / pivot



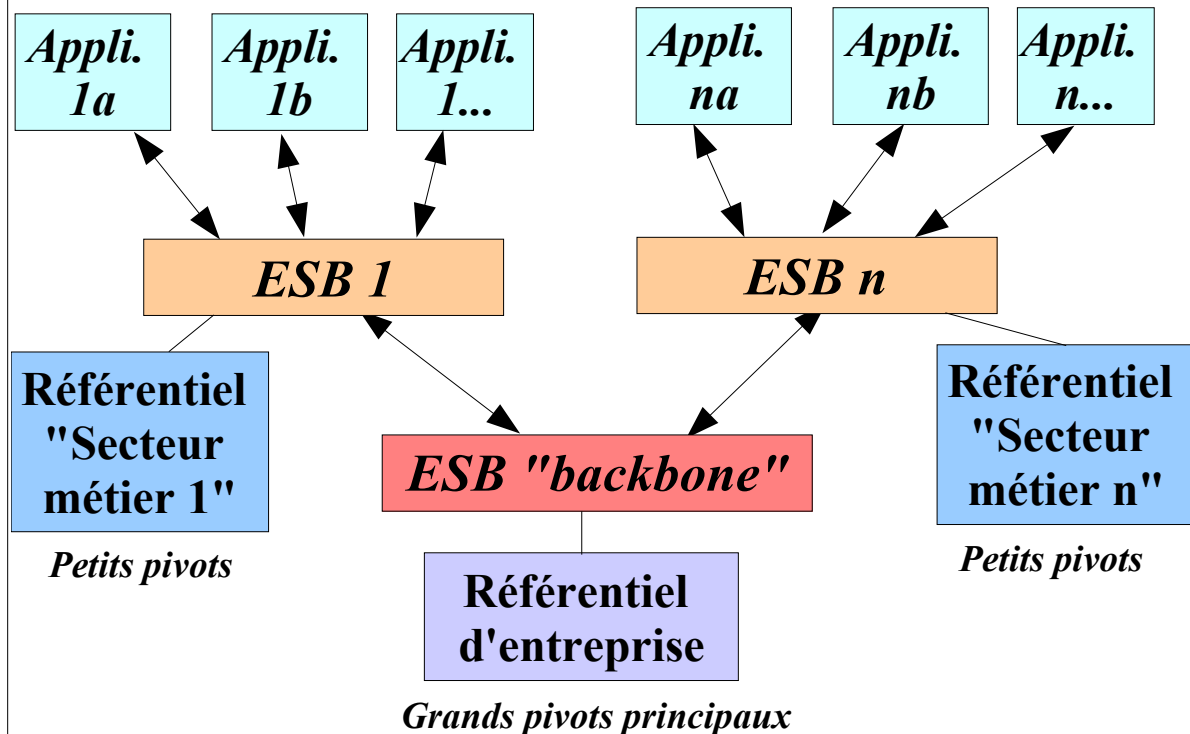
Si transformations
des formats de données
au cas par cas :
--> besoins en $O(n^2)$!



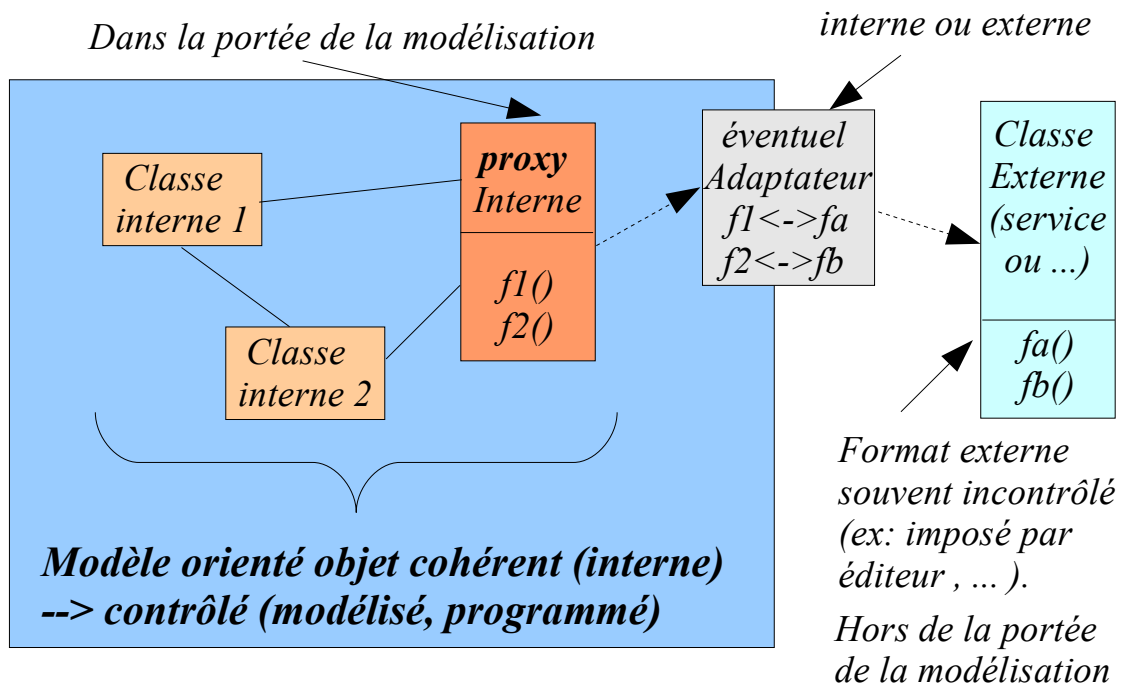
Si transformations avec format
de données intermédiaire
normalisé (pivot) :
--> besoins en $O(n)$!

Formats "pivot" et "sous pivot"
dans architecture idéalement décentralisée

Approche pragmatique/réaliste décentralisée



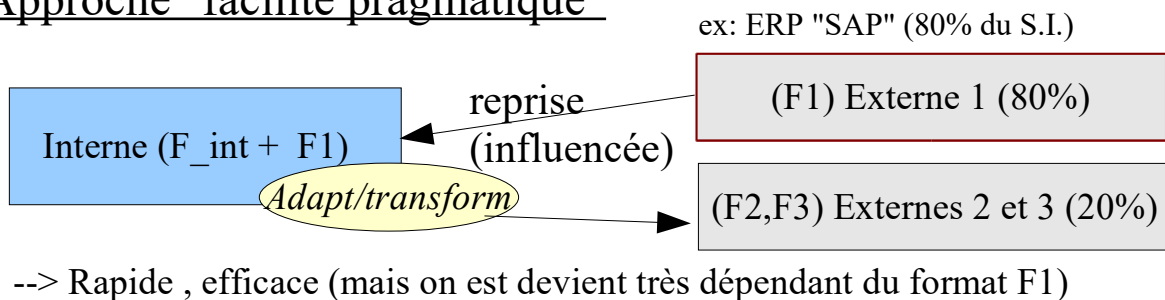
Notion fondamentale de "proxy métier"



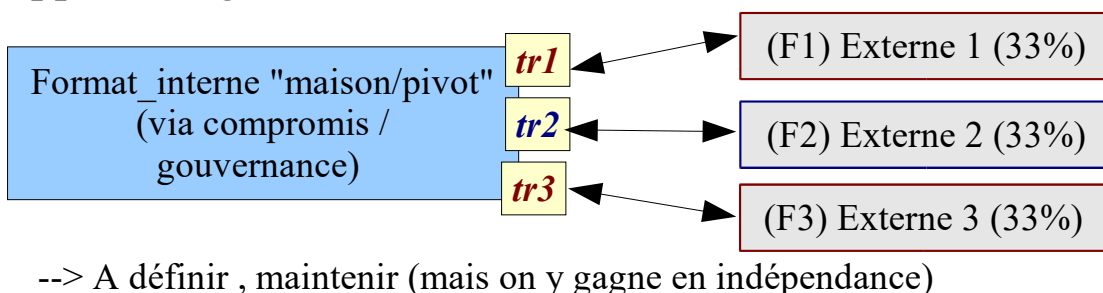
16.3. Approches "conformisme" et "gouvernance"

Choix du format interne (*) quelquefois appelé "format pivot" en SOA

Approche "facilité pragmatique"



Approche "gouvernance des données"



L'approche "gouvernance des données" (théoriquement conseillée) consiste à **définir certains formats fonctionnels (structure de données) en confrontant les besoins (en communications / échanges) de différentes parties du SI** (applications , sous systèmes , ...) .

Pour cela , des **réunions** entre "parties prenantes" sont souvent nécessaires. Pour éviter les réunions inutiles où personne se met d'accord à la fin, on pourra s'organiser avec une matrice "RACI" .

R : Responsable (**r**esponsable , **r**éalisateur)

A : Accoutable , approuver (**a**pprobateur avec droit de vote)

C : Consulted (**c**onsulté – donne son avis)

I : Informed (**i**nformé)

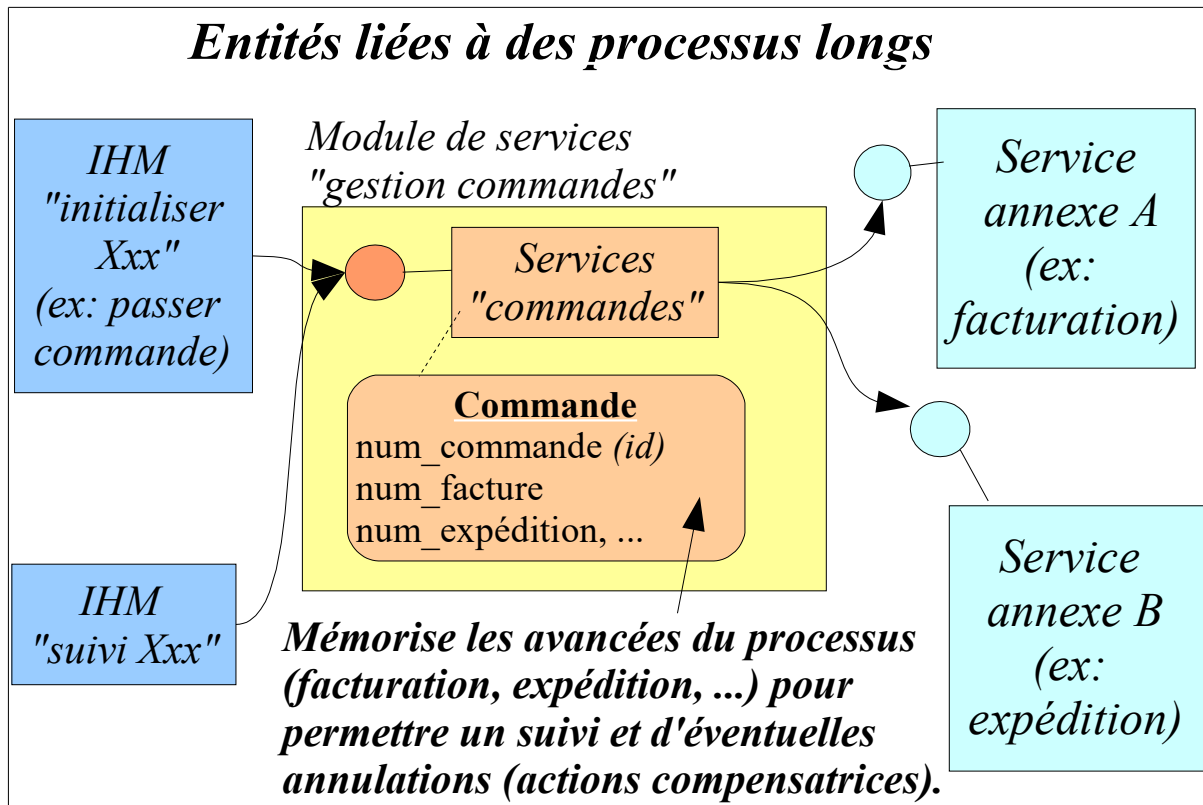
Task Description	Sponsor	Business Owner	Business Program Mgr	Process Manager
Identify missing or incomplete policies		R	A	R
Establish Policies as necessary and ensure adoption globally		A	R	R
Completion of necessary Policies		R	A	R
Document Policies as appropriate		R	R	A

16.4. Transactions longues et compensations

De nombreux échanges SOA sont effectués en mode asynchrone lorsque certains traitements ou sous processus (ex: livraison) sont longs.

Les éventuelles transactions associées, longues elles aussi, ne peuvent pas se permettre d'exiger un verrouillage ou une isolation temporaire des données mises en jeu sur une longue période.

En cas d'échec d'une transaction longue , il faut prévoir des mécanismes de compensation (action inverse annulant l'action d'origine : exemple= remboursement si livraison non effectuée).



Autre exemple classique : entité "dossier" avec num_dossier,

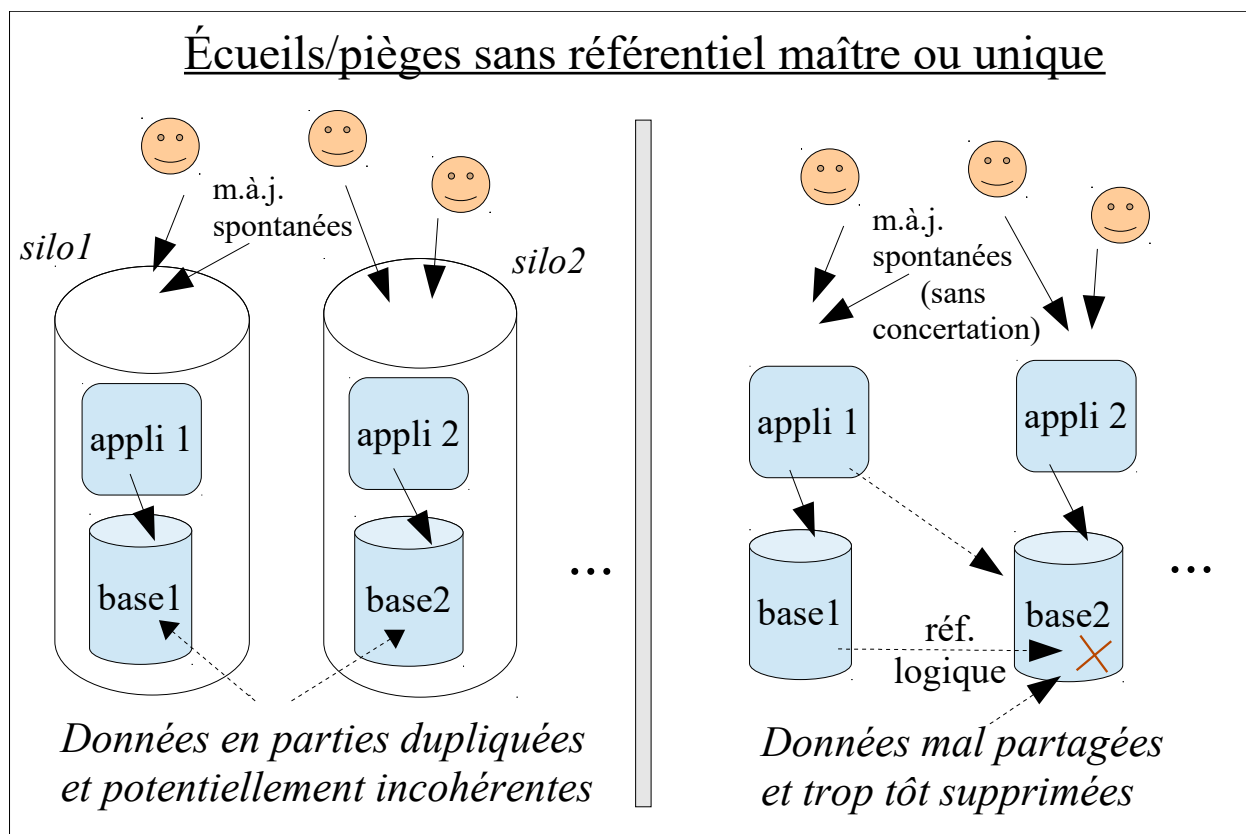
16.5. Stabilités des éléments référencés

Dans un système à contraintes d'intégrités centralisés (telle qu'une base de données unique prise en charge par un SGBDR), il est souvent impossible de supprimer involontairement/accidentellement une entité qui est encore référencée par une autre.

A l'inverse dans un système plutôt décentralisé comme internet ou SOA, une entité peut être supprimée dans un système (application) sans que les autres systèmes soient au courant (ex: URL devenue invalide, service utilisant un "vieux" id pour référencer une entité qui existait et n'existe plus).

==> Les spécifications (cahier des charges, modélisations, ...) devraient idéalement spécifier clairement certains éléments tels que la durée de validité d'une référence (et/ou des régulières vérifications, ...).

sans MDM :



16.6. MDM (Master Data Management)

Pour adresser le problème de l'intégrité référentielle au sein de systèmes décentralisés, on se réfère souvent au concept de "MDM" (*Master Data Management*).

MDM est quelquefois dénommé GDR (*Gestion de Données Référentielles*) en français.

MDM (ou GDR) est surtout utilisé pour gérer les données fondamentales (*de références*) d'une grande entreprise :

- les données « **clients/fournisseurs** »
- les données « **produits** »
- les données « **financières** ».

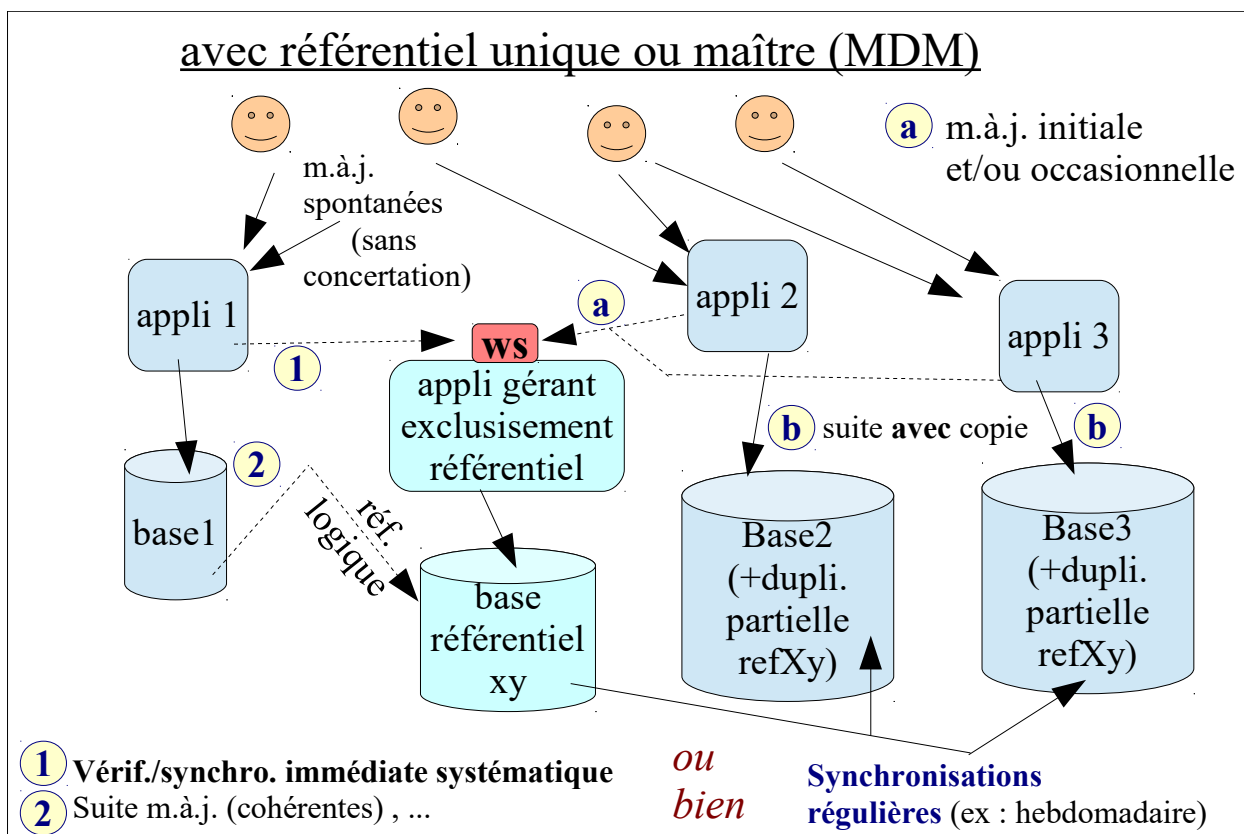
Besoin de distinguer (ne pas confondre):

- données maîtres de référence (partagées, avec différentes vues, stratégies, sous système propriétaire souvent difficile à identifier)
- données procédurales (fruit d'une procédure, d'un processus métier, factuel, date précise, ...) souvent liées directement ou indirectement à la zone d'urbanisation "gisement de données"

et **bien modéliser les cycles de vie** (lorsque c'est possible) des données de référence.

avec MDM/GDR :

- Les données référentielles sont encodées et maintenues en un seul endroit, ce qui diminue le coût opérationnel lié à la maintenance et à l'encodage.
- Le système est le maître des données. Il les contrôle en sélectionnant quelles données il transmet à quel système.
- Le système contient une seule version active (il peut exister plusieurs versions inactives, tant passées que futures ; c'est même recommandé pour une meilleure flexibilité de la solution). Il est donc le garant de la seule version de la vérité et en cas de litige, sa version tient lieu de version officielle.



Un mode "pull" ou bien "push-pull" est envisageable au niveau des synchronisations entre les données de référence du référentiel "maître" et celles des référentiels/bases secondaires.

Fonctionnalités classiques de l'application gérant un référentiel :

- gestion d'un cycle de vie des données de référence (à travers WS de contrôle ou ...)
- ...

16.7. Modéliser les cycles de vie des objets de référence

Pour pouvoir gérer l'intégrité référentielle dans un système décentralisé/éclaté en de multiples applications, il faut absolument définir précisément les cycles de vie des entités fondamentales.

Utilisation adéquate d'un diagrammes d'états UML

Après avoir identifié (et modéliser sous forme de classes UML) les entités de références , il est fortement conseillé d'élaborer quelques **diagrammes d'états UML** .

Dans ce diagramme d'états ,

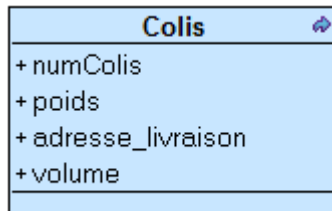
- l'état initial correspondra souvent à la création de la nouvelle entité (ex : nouveau client , nouveau produit)
- l'état final correspondra souvent à la suppression définitive de l'entité en base (delete SQL).
- les états intermédiaires et les transitions associées permettront de bien spécifier l'avancement contrôlé du cycle de vie.

Points importants :

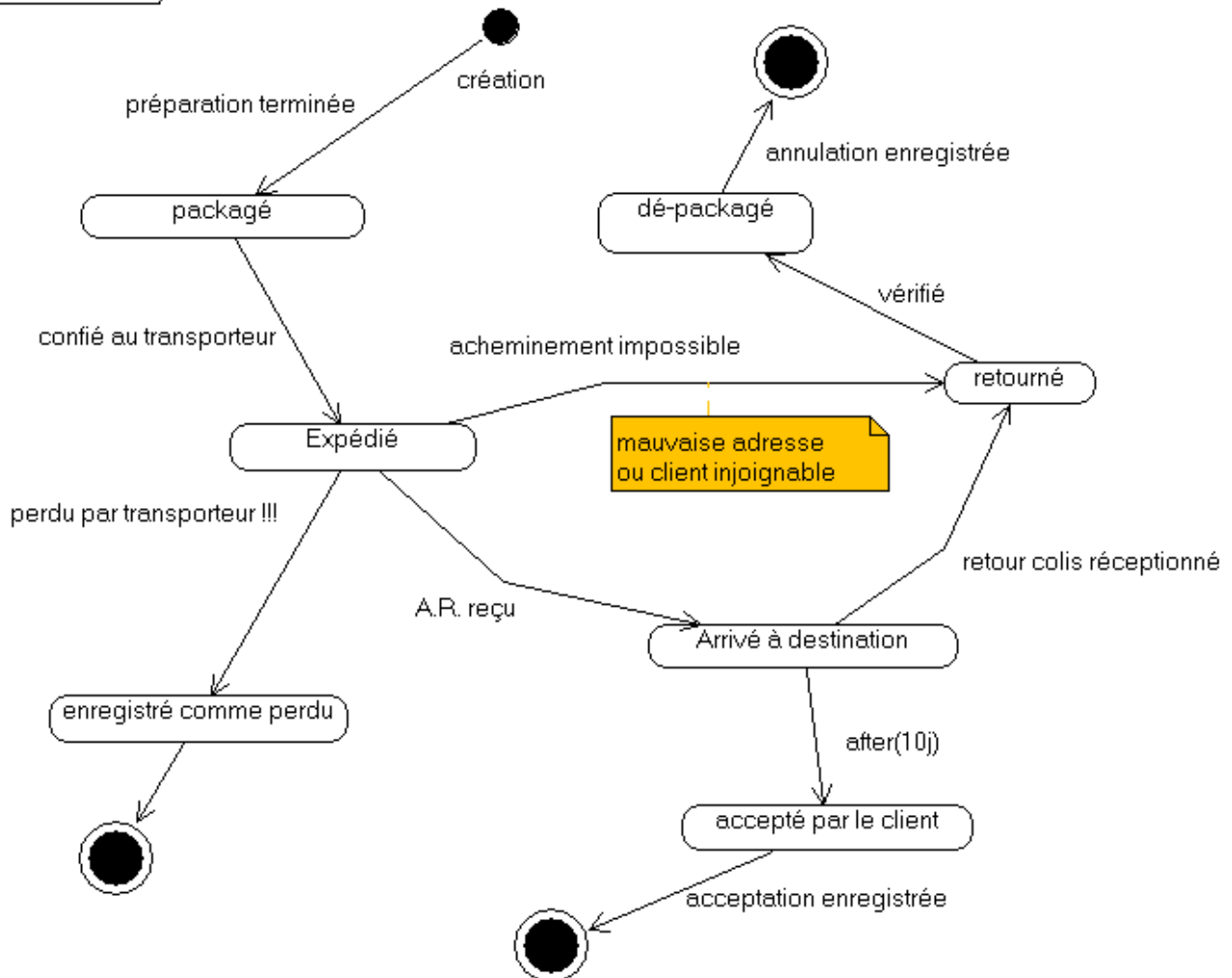
Souvent besoin d'une suppression douce et progressive :

- 1) désactivation (ou suppression) logique en mémorisant la date de désactivation
- 2) attente d'une période écoulée (ex : 2 mois ou 1 an) pour être certain que l'entité ne soit plus référencée par d'autres éléments actifs.
- 3) éventuel archivage
- 4) suppression définitive

Exemple (cycle de vie d'un colis) :

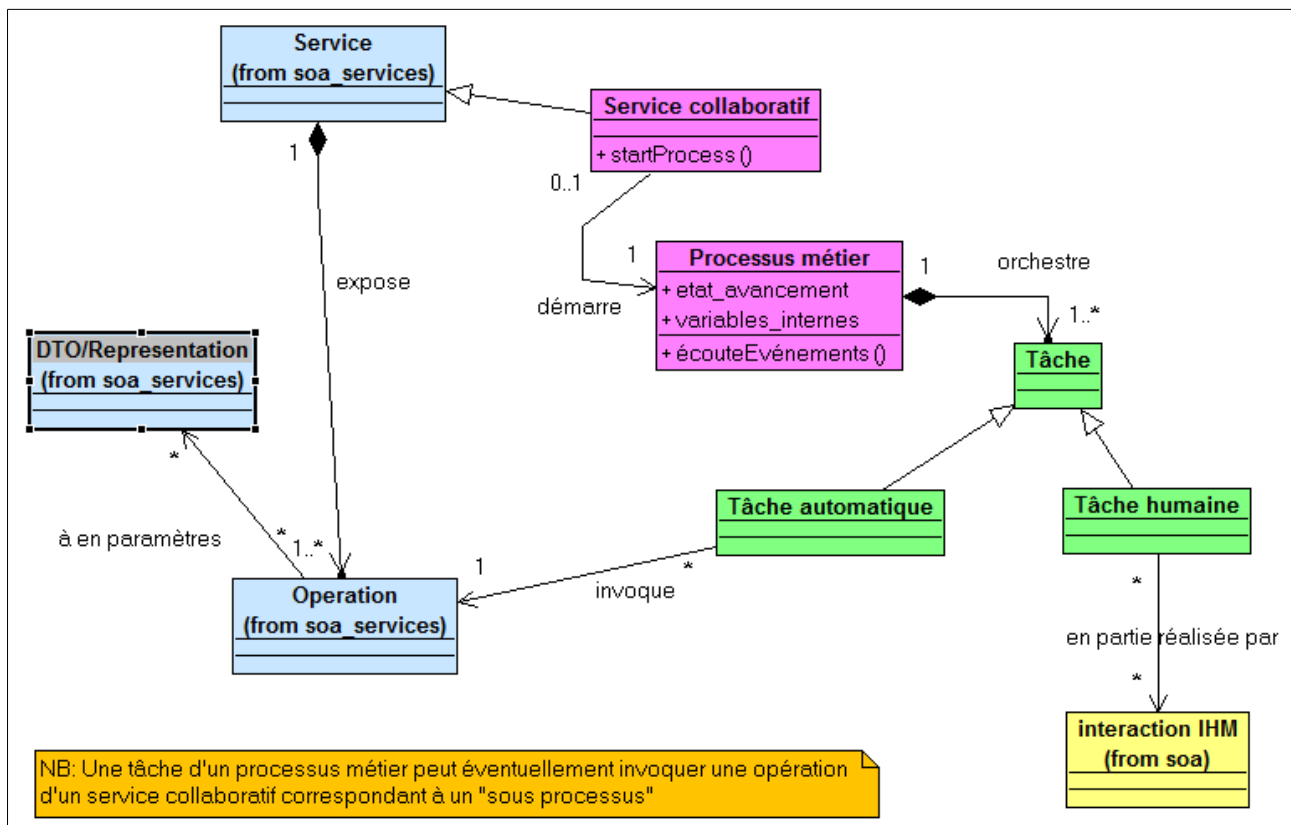


StateMachine



16.8. Dualité "services avec orchestration de sous-services" et "processus"

Méta-modèle pour service collaboratif (de haut niveau) :



NB : Un processus métier peut éventuellement :

1. être modélisé en UML (et/ou BPMN)
2. être implémenté en XML/BPEL (ou bien en Java/jBpm , ...)
3. être vu comme un service collaboratif (orchestrant différents services/tâches élémentaires)
4. être invoquée comme un service web
5. être (en partie) contrôlé par une IHM (web ou ...) invoquant ce service.

16.9. "processus métier vers processus exécutable"

Traçabilité souhaitée

Le principal intérêt des technologies d'orchestration (ex : BPEL , bpmn2 + java , ...) réside dans la traçabilité généralement rencontrée entre les éléments du modèle métier (processus bpmn abstrait) et les éléments du code exécutable .

Autrement dit , si la version N+1 évolue par "suppressions , modifications , ajouts partiels" au sein du modèle métier abstrait , on peut rapidement identifier les parties à mettre à jour au sein du code exécutable .

Sans technologie d'orchestration , on a souvent besoin d'éléments spécifiques éparpillés au 4 coins du code de l'application (colonne d'une table Xy , gestionnaire événementiel Zz , ...) et la traçabilité n'est pas évidente.

Partie exécutable d'un processus

Un processus métier (abstrait) modélisé avec BPMN comporte souvent plusieurs couloirs ("pools") . Par exemple : "client" , "fournisseur" , "logistique" .

Ceci permet de bien montrer certaines collaborations attendues entre partenaires (chorégraphie) .

A l'inverse, **une technologie d'orchestration** (ex : BPEL ou bien bpmn2+java) **ne peut exécuter le code que d'un seul couloir ("pool")** à éventuellement marquer comme **"exécutable"** (selon la technologie) .

En fonction de la technologie qui interprète le processus (bpel , bpmn2 , ...) , les éventuels autres pools seront :

- soit ignorés
- soit utilisés comme base pour la définition des partenariats
- soit "à enlever" pour éviter des erreurs
- ...

Un fichier "bpmn" (décrivant un processus métier abstrait) a donc généralement besoin d'être :

- versionné
- repris (par copie) et adapté à la technologie d'interprétation.

Bonnes pratiques :

- décomposer (dès la modélisation abstraite) un processus en sous-processus (via call-activity)
- environ 60 % des processus modélisés ne resteront que des modèles BPMN (faisant partie des spécifications de chorégraphie à lire , ré-interpréter)
- environ 40 % des processus (ou sous-processus) pourront (partiellement) être rendus exécutables (si la traçabilité est appropriée / souhaitée ou bien si certaines parties à piloter sont asynchrones) .

17. Considérations sur la granularité des services

- plus le découpage est fin, plus c'est modulaire et l'on peut espérer avoir des sous parties réutilisables/partagées
- à l'inverse , un découpage trop fin , avec des délégations vers des traitements trop triviaux (facilement faisables en local) , peut amener à de mauvaises performances (trop d'appels traversant sans arrêt le réseau, trop de sérialisations/dé-sérialisations)
- **Comme repère de granularité , on peut avoir en tête :**
 - **services métiers élémentaires** et services techniques basiques (*niveau 1*)
 - services fonctionnels "métier " (**orchestration simple** , purement métier) ou bien services techniques améliorés (*niveau 2*)
 - **service organisationnel** (spécifique entreprise/organisation) ou bien **service correspondant à un processus métier long** (ex : bpmel , bpmn2, ...) (*niveau 3*)
 - **macro processus métier ou organisationnel (niveau 4)** décomposé lui même en sous processus ..

18. Composants SOA

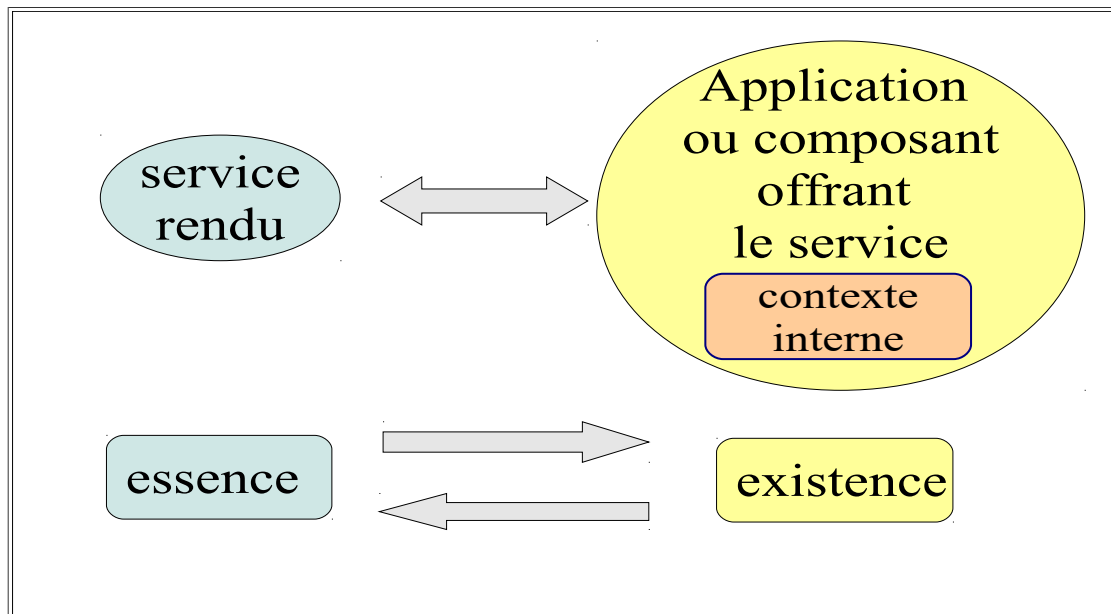
18.1. Bien modéliser (d'une façon ou d'une autre) les contextes

A part quelques cas triviaux (ex: service de calcul élémentaire) , la plupart des services sont intelligibles (sans ambiguïté) que si l'on précise un minimum certains contextes :

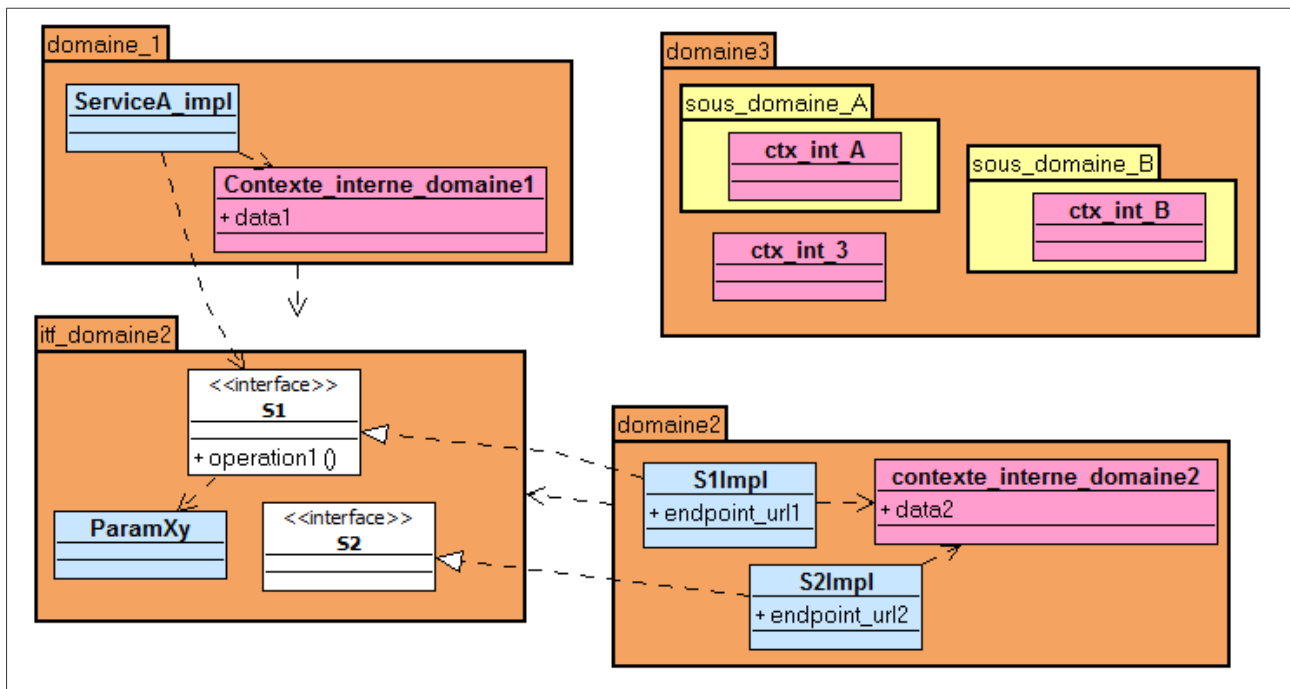
- contexte de persistance (base de données de l'application offrant le service)
- contexte d'exécution (mémorisation de l'état d'avancement d'un processus métier : processus long en partie asynchrone)
- contexte implicite/explicite , privé/public , opaque ,

18.2. Penser à une structuration logique (future implémentation) à base de composants offrant des services

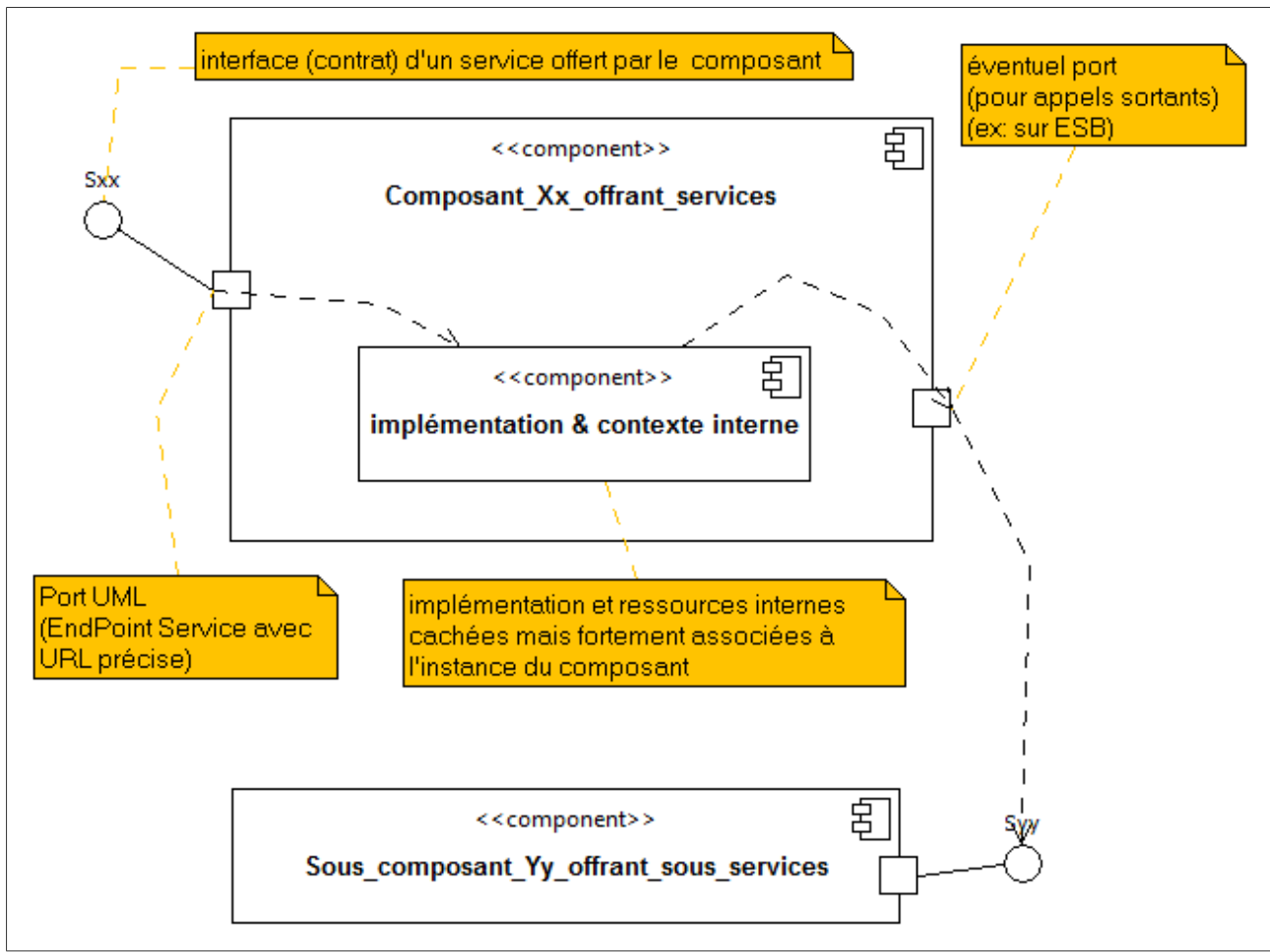
Beaucoup de contextes peuvent être concrètement pris en charge par des composants informatiques (de petites ou grandes tailles) .



En UML, cette notion de contexte peut se modéliser via des diagrammes de classes (avec packages) (avec packages)



ou bien via des diagrammes de composants .../...



19. SOA bien dimensionné (selon moyens et besoins)

19.1. Prix relativement élevé de l'infrastructure SOA

Mettre en œuvre une architecture "SOA" nécessite les différents éléments suivants (dont les coûts s'additionnent) :

- **Analyse / modélisation / organisation (UML , BPMN , urbanisation)** avec gestion des évolutions (versions cohérentes , ...)
- Achat/location d'infrastructures matérielles (ordinateurs , machines virtuelles)
- Achat/location de serveurs logiciels (Serveurs d'applications JEE , **ESB** , **Moteur BPM**, ...)
- Formation ou auto-apprentissage , administration / configurations.
- **Outils de modélisation/développement (avec assistants spécifiques "bpel" ou "bpmn")**
- Développement (java , xml , ...) , tests unitaires
- **Tests d'intégration**
- Maintenance
-

Par rapport à une architecture plus simple , le surcoût de l'infrastructure SOA se situe essentiellement au niveau des points mentionnés ci-dessus en caractères gras.

Certains "ESB commerciaux" coûtent environ 50000 euros !

Attention :

Une technologie qui automatise beaucoup (ex : ESB , moteur BPM , framework complexe,) permet de gagner en "nombre de lignes à coder" mais nécessite plus de "tests" pour être bien fiabilisée. D'autre part , la lente compréhension de certains mécanismes complexes peut rendre les paramétrages délicats à optimiser/retoucher.

19.2. R.O.I. selon bénéfices métiers/fonctionnels

L'infrastructure SOA (potentiellement lourde) trouvera un retour sur investissement qu'en fonction :

- des bénéfices métiers/fonctionnels (éviter les re-saisies , meilleur agilité / réactivité , ...)
- d'un bon choix et dimensionnement des éléments de l'architecture (ESB commerciaux ou bien "open source" ,)
- ...

Comment souvent, au fil des années,

- les technologies lourdes (trop complexes) finissent par être abandonnées (ex : JBI)
- de nouvelles technologies (plus simples , mieux intégrées, ...) font baisser les coûts.

Avec un petit budget (à maîtriser) :

- Web services "SOAP" et "REST"
- ESB "open source" avec paramétrages restants simples
- Technologie d'orchestration légère (ex : bpmn2 + activiti)
- Modélisation / urbanisation légère
- Etudier si possible des hébergements "cloud" .

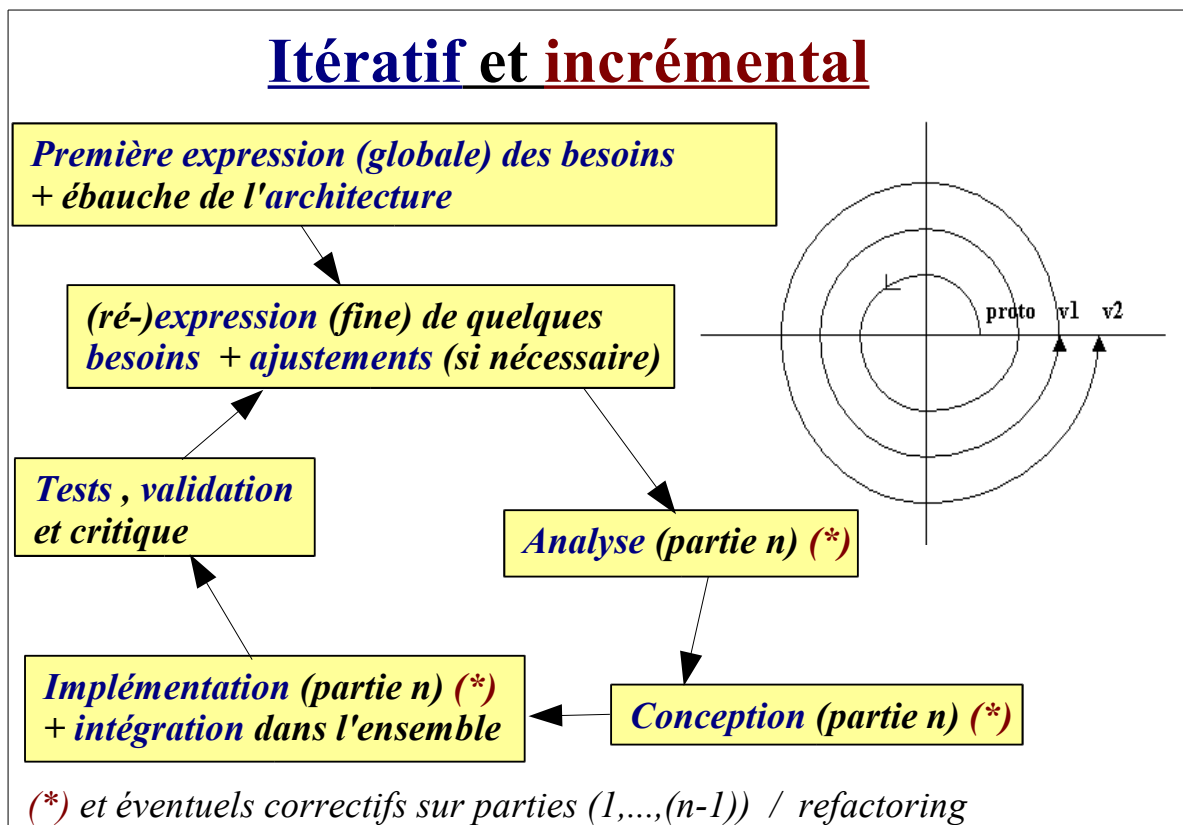
Pour entreprendre de grandes choses (objectifs ambitieux) :

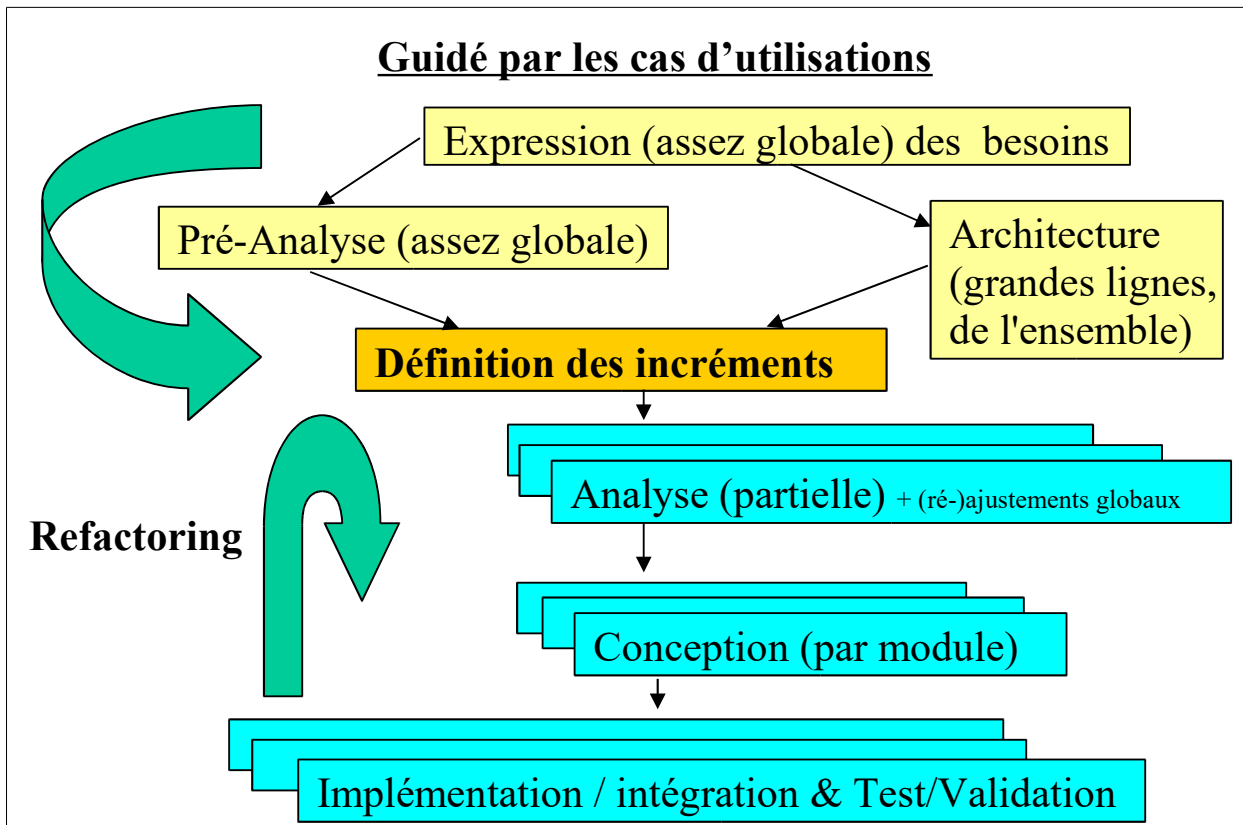
- Technologies évoluées (fiables , pas trop limitées , ...) avec support
- Techniciens qui maîtrisent les technologies
- Budget conséquent (à priori pas pour petite entreprise)
- Cellule de modélisation / urbanisation
- Infrastructure suffisante (bien dimensionnée) pour ne pas rencontrer de problème de performance.
- Bonne organisation (gestion des changements , évolutions convergentes , tests , ...)

Le **cadre de capacité** de "Togaf" met par exemple en avant l'organisation nécessaire pour mener à bien un gros projet d'entreprise .

20. Cycle de vie d'un projet SOA

20.1. Besoins d'un cycle itératif et incrémental





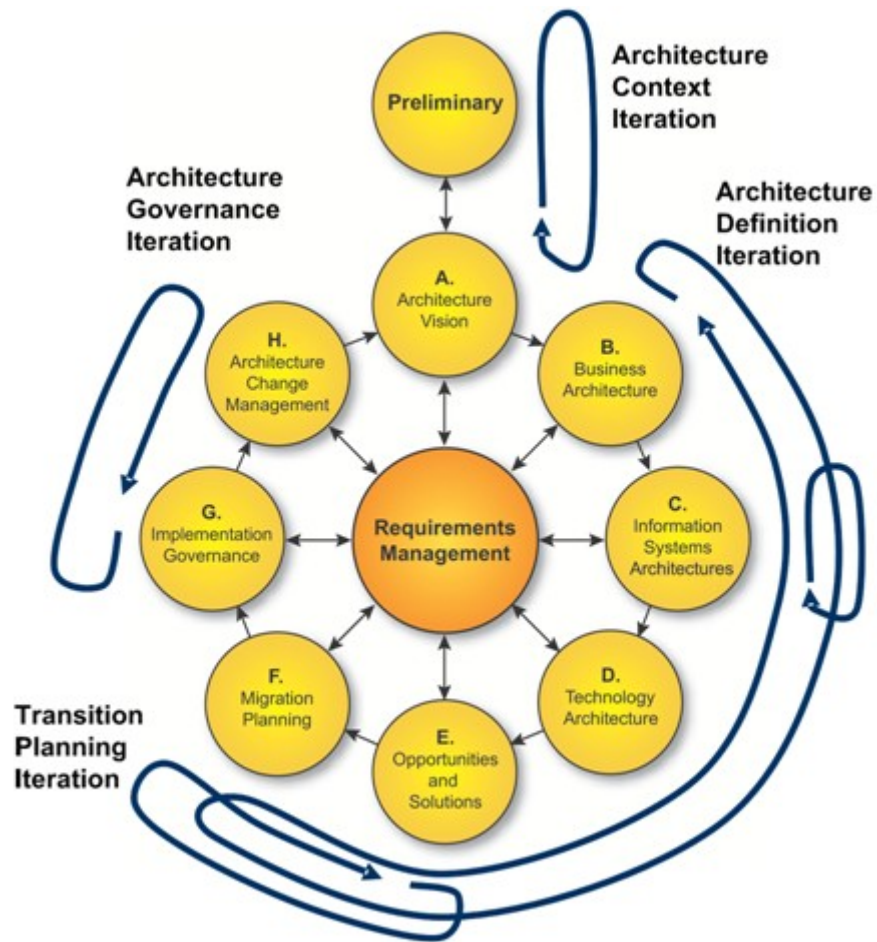
SOA étant lié à une **grande portée** (plusieurs partie du SI , extanet ,) , il est conseillé d'adopter une **démarche** à la fois **pragmatique** (en s'inspirant partiellement des méthodes agiles XP , SCRUM , ...) et à la fois **bien formalisée** (style UP).

Autrement dit des tests et des commentaires dans le code ne suffisent pas.

Il faut absolument définir(et versionner) :

- des **référentiels** de modèles UML et BPMN et de formats internes (XSD/WSDL,...)
- maintenir à jour des **documentations/spécifications précises** .

L'aspect "itératif" (à grande échelle) de SOA est clairement mise en avant dans le cycle "ADM" de "togaf" :



20.2. Besoins de transversalité

Contrairement à un projet applicatif à portée réduite , une bonne modélisation SOA (ayant des ambitions sur le moyen/long terme) doit être **transverse** .

Concrètement, ceci consiste à :

- recueillir les *besoins "soa" précis (échanges extérieurs , structures de données)* de chaque partie concernée du S.I.
- *faire ressortir des structures communes (via gouvernance , ...)*
- *identifier les besoins en "intégrité référentielle" (MDM, ...)*
- ...

20.3. exemple de cycle de vie adapté "soa"

Phase de mise au point (version 1) :

1. Effectuer une *pré-étude SOA (assez globale)*
2. *Identifier une sous partie non critique du SI* et mettre en place un *premier prototype* de l'*architecture SOA* (modèles , référentiels , ESB , services ,)
3. Effectuer (si besoin) les *réajustements nécessaires*

Phases d'intégration successives (versions "2,3,..., n") :

1. *Parfaire* si besoin l'*étude SOA globale* (en tenant compte des changements et des ajouts)
2. *Identifier une sous partie "à besoin SOA prioritaire" du SI*
3. *Intégrer* cette partie fonctionnelle dans l'architecture SOA existante en effectuant toutes les *restructurations* à priori nécessaires.
4. Effectuer (si besoin) les *réajustements* à posteriori nécessaires

21. Activités de modélisation SOA (enchaînement)

- 1) Faire (ou parfaire) un *"recueil textuel des besoins (transverses)"* et une *analyse sommaire de l'existant*.
- 2) Faire (ou parfaire) une **étude d'urbanisation** et éventuellement une *analyse sémantique* de façon à obtenir un **découpage en packages/modules (zones/secteurs/...) significatifs**.
- 3) *Modéliser et/ou ajuster* les **structures de données partagées "de référence"** (structure , cycle de vie)
- 4) Retranscrire/reformuler certaines "expression de besoins" et "objectifs métiers" en **"(business) Uses Cases"** dont les *scénarios* pourront être *illustrés* par des *diagrammes d'activités* (processus métier).
Modéliser (en UML ou BPMN) les principaux "processus métier" (et repérer les besoins en services)
- 5) Modéliser les **services métiers** (opérations, requêtes/réponses/exceptions, ...) et identifier/étudier les besoins en **"adaptateurs"** pour les **intégrations**.
- 6) Dériver les éléments de la modélisation en **"structures et contrats SOA" (XSD , WSDL, ...)** et alimenter les référentiels SOA (... ? , Annuaire interne ? , ...).
- 7) **Programmer** toutes les parties nécessaires (*java* , **web services** , **jbpm** ou **bpel** , **adaptateurs**, *intercepteurs (log, sécurité,...)* ,)
- 8) Tester et ajuster le tout , parfaire la documentation ,

III - Architecture technique SOA (SOAP,ESB, BPM)

1. Deux grands types de WS (REST et SOAP)

2 grands types de services WEB: **SOAP/XML** et **REST/HTTP**

WS-* (SOAP / XML)

- "Payload" systématiquement en **XML** (*sauf pièces attachées / HTTP*)
- Enveloppe SOAP en XML (*header facultatif pour extensions*)
- Protocole de transport au choix (HTTP, JMS, ...)
- Sémantique quelconque (*appels méthodes*) , description WSDL
- Plutôt orienté Middleware SOA (*arrière plan*)

REST (HTTP)

- "Payload" au choix (XML , HTML , **JSON**, ...)
- Pas d'enveloppe imposée
- Protocole de transport = toujours **HTTP**.
- Sémantique "CRUD" (*modes http PUT,GET,POST,DELETE*)
- Plutôt orienté IHM Web/Web2 (*avant plan*)

Points clefs des Web services "SOAP"

Le format "xml rigoureux" des requêtes/réponses (définis par ".xsd" , ".wsdl") permet de retraiter sans aucune ambiguïté les messages "soap" au niveau certains services intermédiaires (dans ESB ou ...). Certains automatismes génériques sont applicables .

Fortement typés (xsd:string , xsd:double) les web-services "soap" conviennent très bien à des appels et implémentations au sein de **langages fortement typés** (ex : "c++" , "c#" , "java" , "...").

A l'inverse , des **langages faiblement typés** tels que "php" ou "js" sont **moins appropriés pour soap** (appels cependant faisables)

La **relative complexité et la verbosité "xml" des messages "soap"** engendrent des **appels moins immédiats** (mode http "post" avec enveloppe à préparer") et des **performances moyennes**.

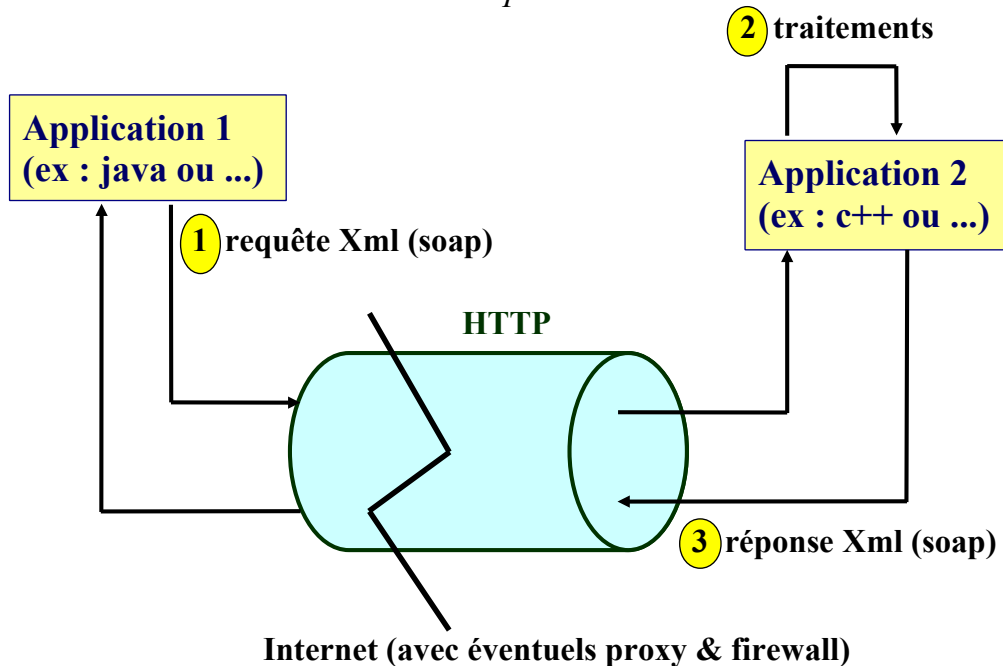
Soap peut être utilisé en mode "envoi de document" mais c'est rare.

Les messages "soap" peuvent être véhiculés par "jms" mais c'est rare.

2. Protocole SOAP

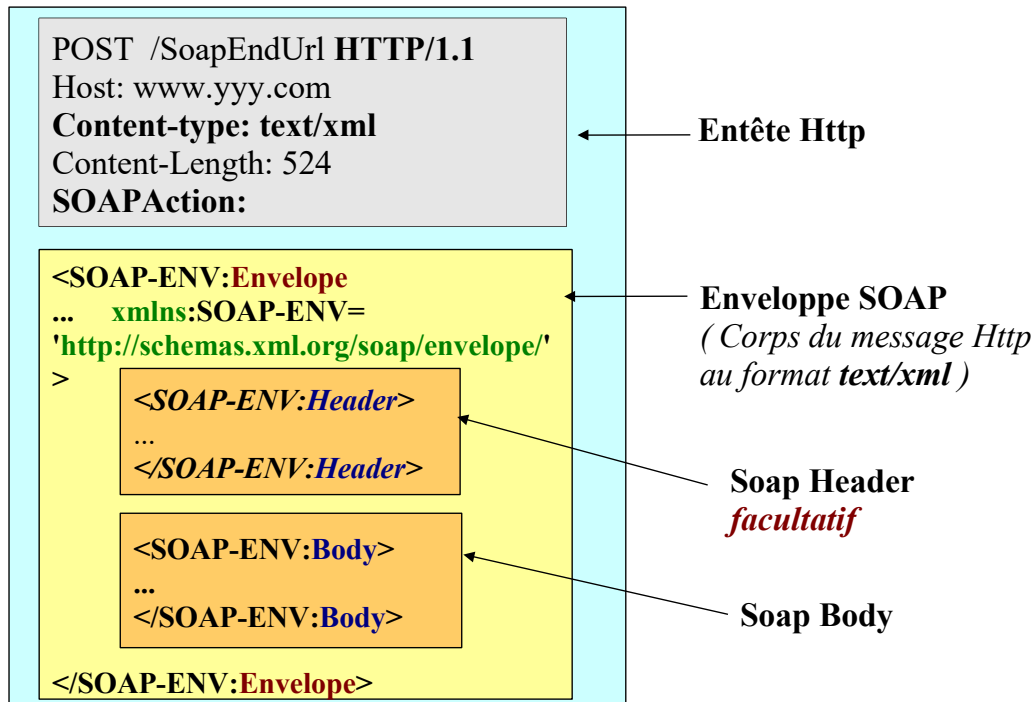
SOAP (Simple Object Access Protocol)

ici en version "SOAP-over-Http"



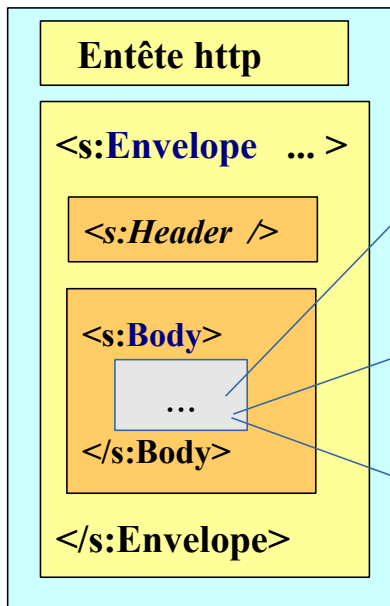
Enveloppe SOAP (ici véhiculée via HTTP)

message http (requête / réponse)



Contenu du corps de l'enveloppe "SOAP"

message http
(avec contenu SOAP)



requête

```
<methodeXy>
  <p1>val1</p1>
  <p2>val2</p2>
</methodeXy>
```

réponse

```
<methodeXyResponse>
  <return>val_resultat</return>
</methodeXyResponse>
```

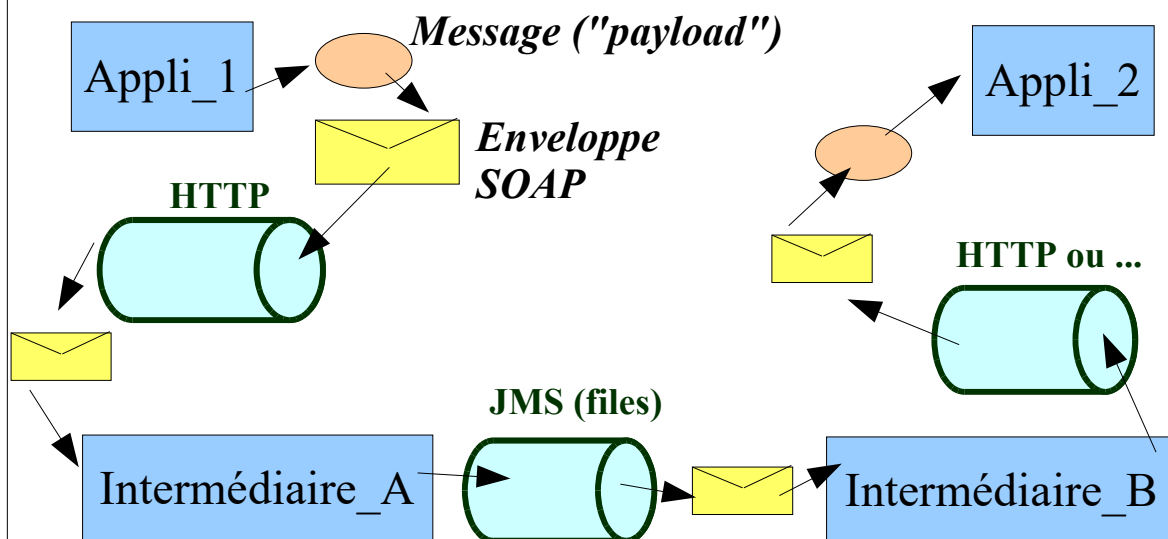
Ou bien

Ou bien

Exception (fault)

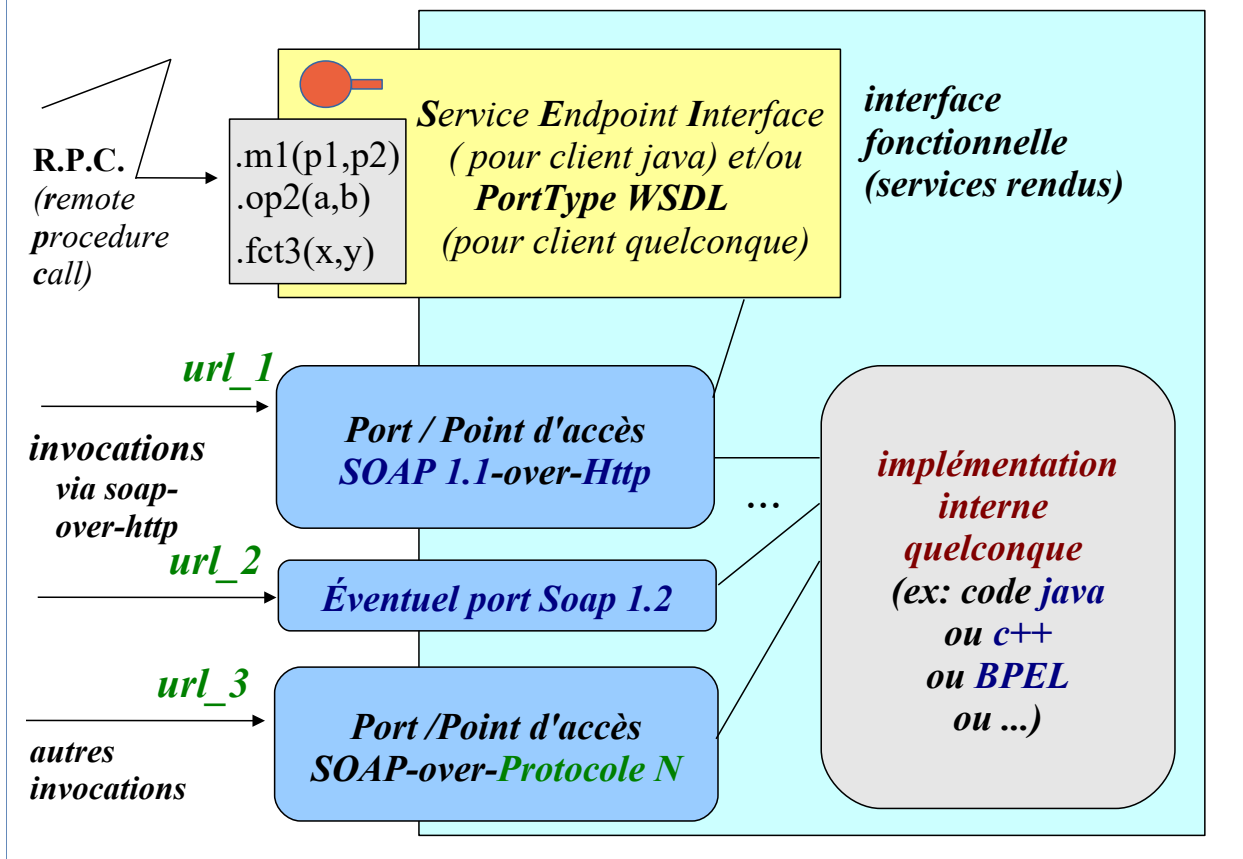
```
<s:fault>
  <faultcode>soap:Server</faultcode>
  <faultstring>msg_error</faultstring>
</s:fault>
```

SOAP: "Payload", enveloppe et transports

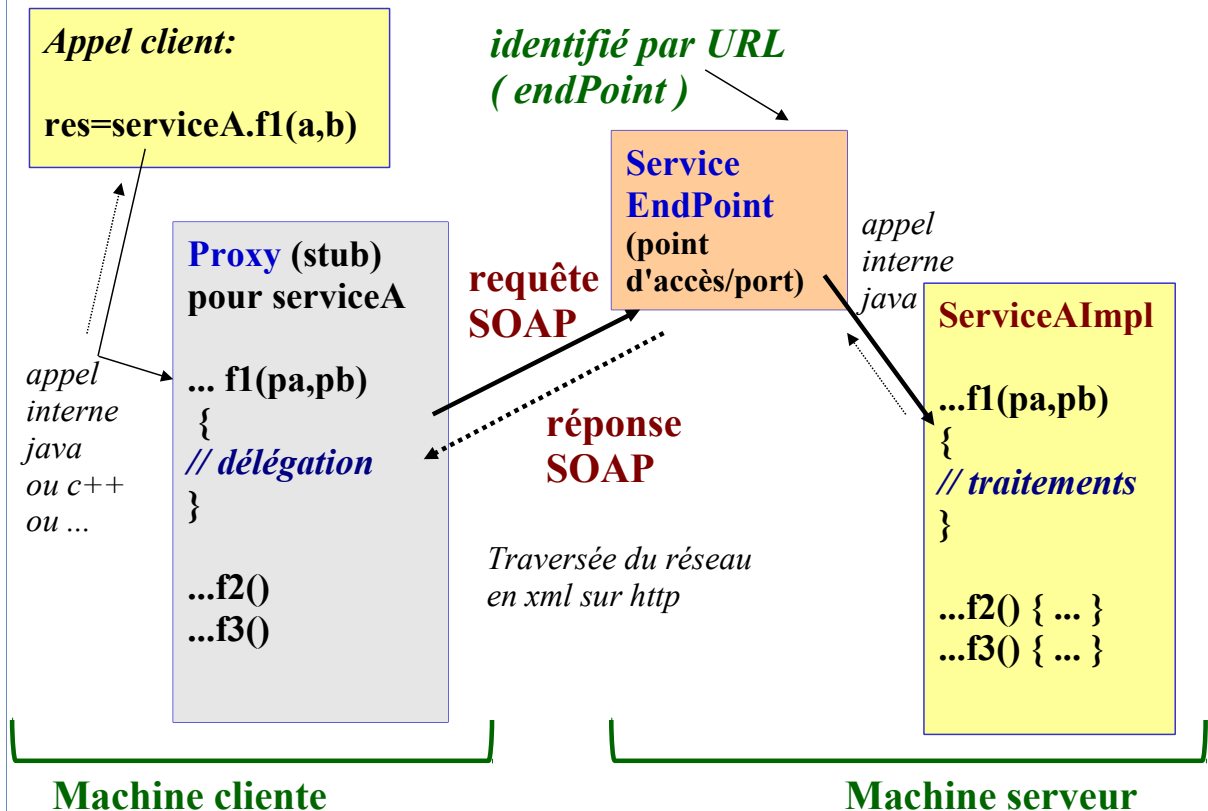


Analogies: Lettre , enveloppe , acheminement (voiture + train + avion + ...)
Charge utile , conteneur , transport (train + bateau + camion + ...)

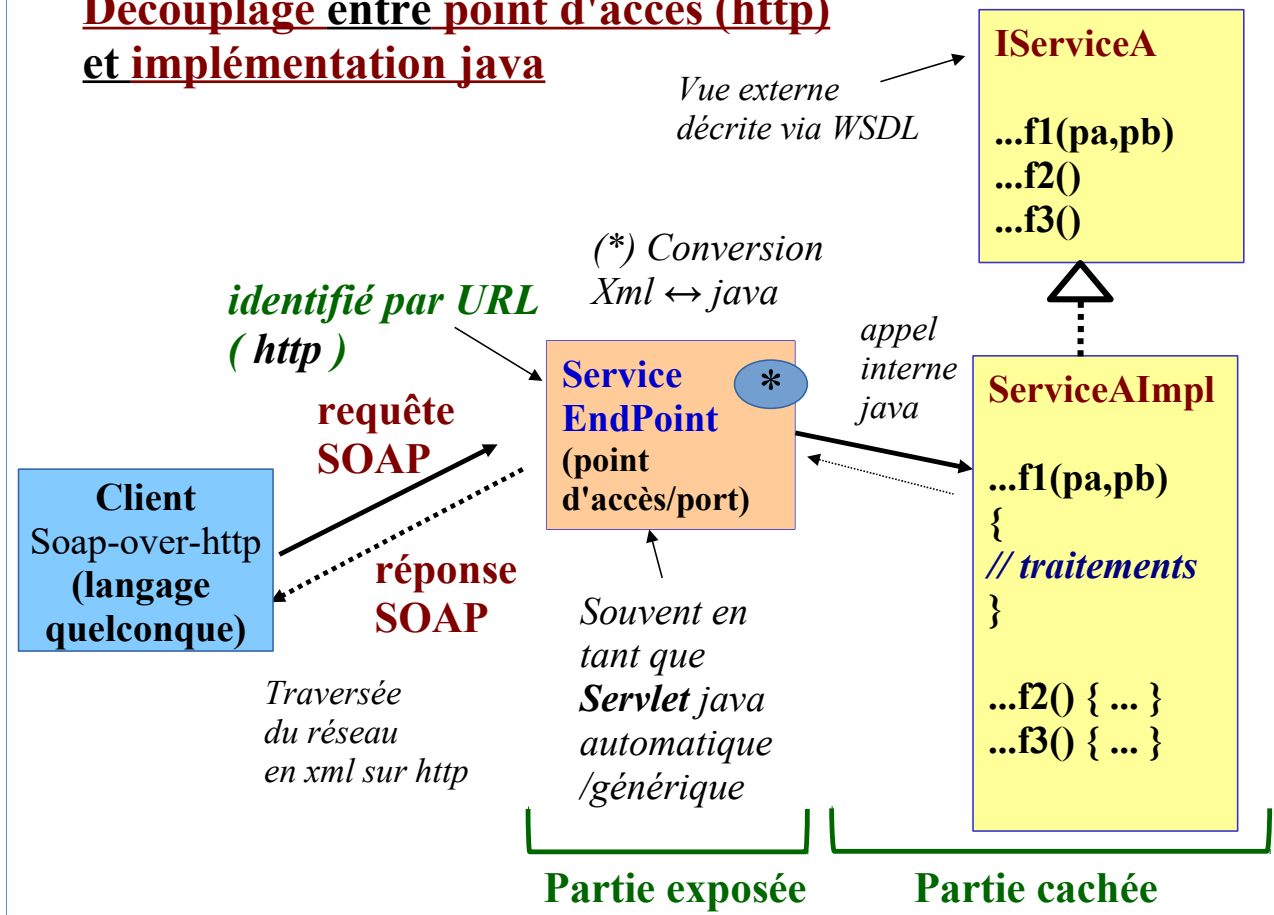
Anatomie d'un Service Web "SOAP"



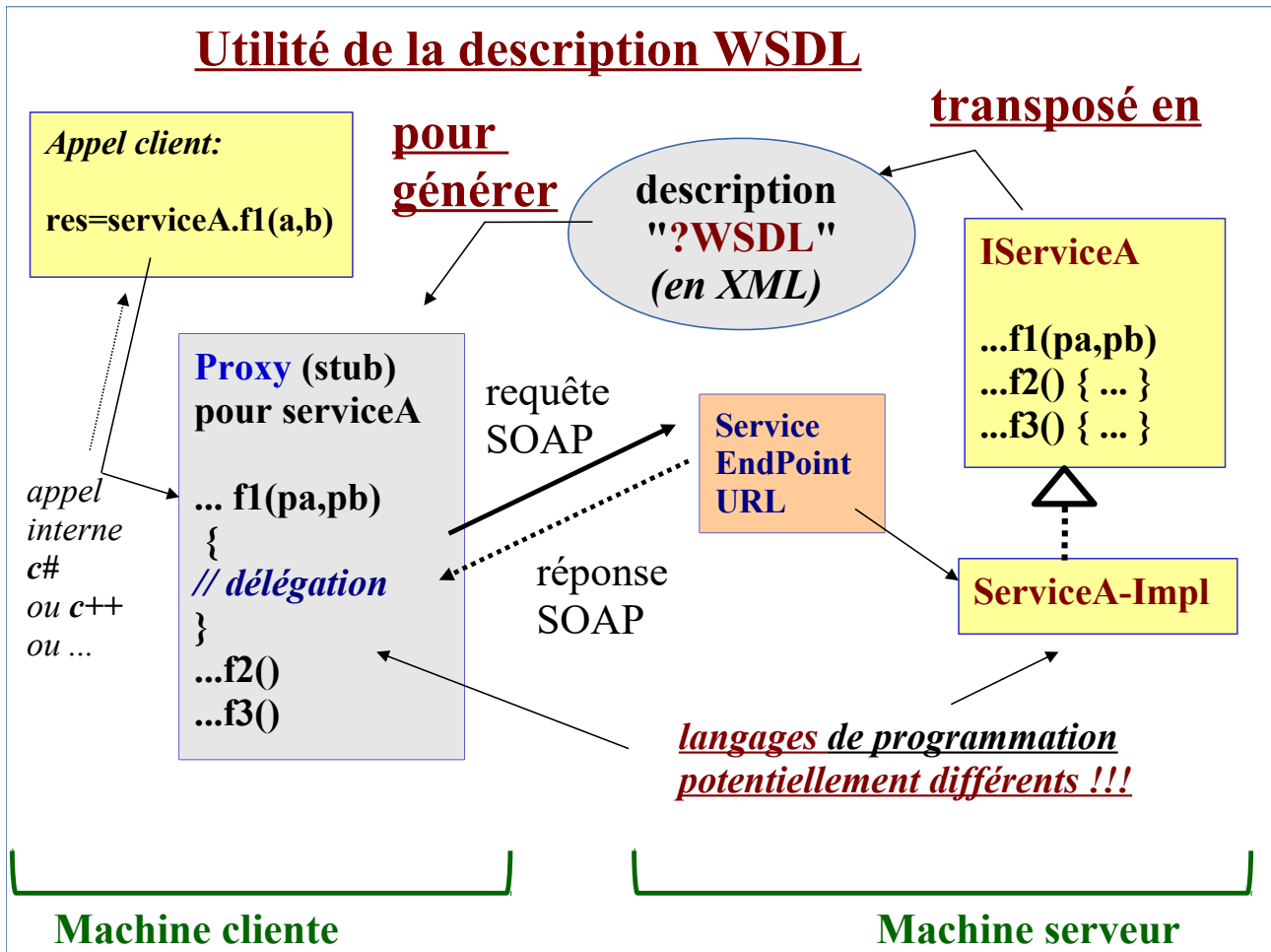
Proxy et Point d'accès / Port avec SOAP



Découplage entre point d'accès (http) et implémentation java



3. WSDL (Web Service Description Language)



Exemple :

Code coté serveur du *service web* en C++ (gSoap)

- génération du fichier WSDL via soapcpp2

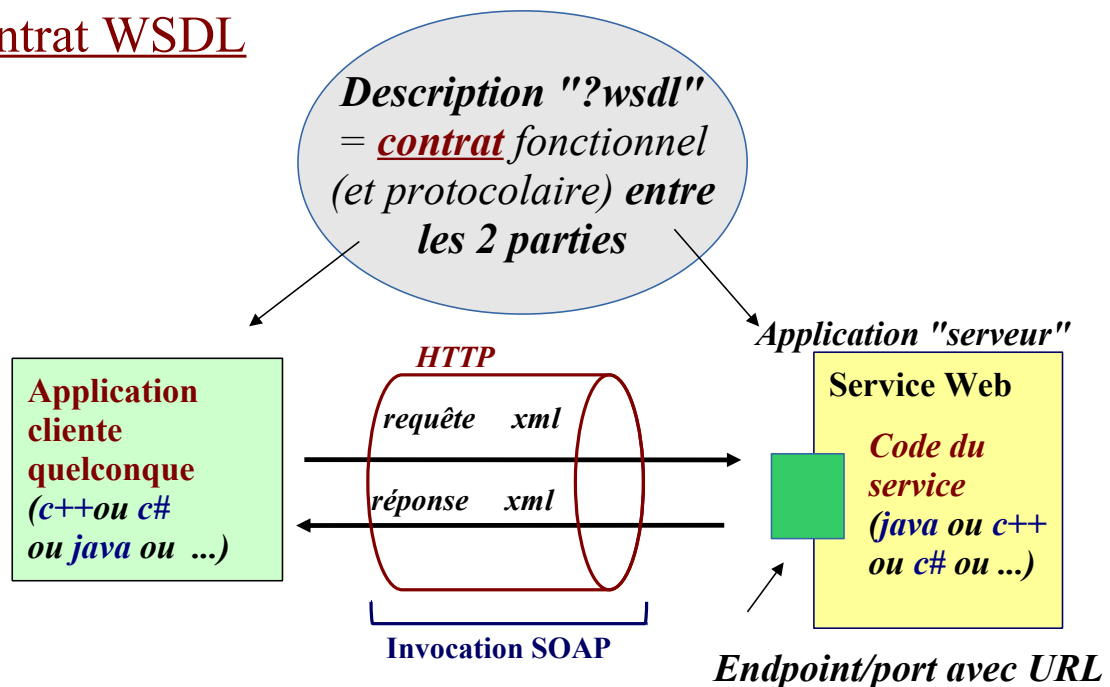
WSDL

- génération via **WSDL2Java** (ou **wsimport**)

Description WSDL (pour WS Soap)

- **WSDL = Web Service Description Language**
- C'est une **description XML de la structure d'un service** soap accessible qui est volontairement **indépendante des langages de programmation** (c++ ou java ou c# ou ...).
- L'essentiel d'une description WSDL peut être vu comme une transposition xml d'une interface java (liste d'opérations avec des types précis de paramètres en entrée et en retour).
- Un Fichier WSDL est souvent **généré automatiquement coté serveur**, publié au bout d'une URL, téléchargé, puis analysé par un programme du genre **wsdl2Java** ou **wsimport** pour générer un **"proxy"** coté client

Contrat WSDL



Par convention, si l' url d'invocation soap d'un service publié est
<http://hostName/xxx/yyy>
 alors l'URL menant à la description WSDL est
<http://hostName/xxx/yyy?wsdl> ← **?wsdl en plus**

Description WSDL (approches)

- Approche "à partir d'une implémentation" :
la description WSDL est automatiquement générée à partir de la structure orientée objet du code d'implémentation (ex : interface et classe java ou c++)
- Approche "à partir du contrat WSDL / xml" :
Le code d'implémentation du service web est construit (généré / codé / paramétré) en fonction d'un fichier WSDL existant (par exemple le fruit d'une norme ou d'une modélisation UML).

Dans tous les cas , le code des invocations "coté client" doit se conformer à un fichier WSDL.

Ce code peut être complètement dynamique (ex : php)
ou bien sous forme de proxy généré/compilé (ex : java)

3.1. Versions et évolutions de SOAP/WSDL

Différentes versions (et évolution)

... , **SOAP 1.1** (xmlns="<http://schemas.xml.org/soap/envelope/>")
SOAP 1.2 (xmlns="<http://www.w3.org/2003/05/soap-envelope>")

Attention: SOAP 1.2 n'est pas pris en charge par toutes les technologies "WebService" et la version 1.1 suffit en général.

... , **WSDL 1.1** (xmlns="<http://schemas.xmlsoap.org/wsdl/>")
WSDL 2 (xmlns="<http://www.w3.org/ns/wsdl>")

Attention: WSDL 2 n'est pas pris en charge par toutes les technologies "WebService" et la version 1.1 suffit en général. Cependant, certaines extensions WSDL2 (MEP) sont utilisées par certains produits (ex: ESB)

Pour SOAP, style/use = *rpc/encoded* (complexe et "has been")
puis *rpc/literal* (plus performant, plus simple)
puis *document/literal* (avec schéma "xsd" pour valider)

4. Détails sur WSDL

4.1. Structure générale d'un fichier WSDL

Structure détaillée WSDL

definitions (avec *targetNamespace*)

types

schema

(*xsd*) avec *element* & *complexType* ,

message * (*message* de requête ou de réponse / paquet de paramètres)

part * (paramètre in/out ou bien return)

portType (interface , liste d'opérations abstraites)

operation *

input *mxxxMessageRequest*

output *mxxxMessageResponse*

binding *

associations (*portType* ,
protocoleTransport [ex: HTTP] ,
style/use [ex: document/literal])

service (nom)

port * (point d'accès)

<**address** *location*="url_du_service" />

NB: Le contenu de la partie <xsd:schema> (sous la branche "wsdl:types") peut éventuellement correspondre à un sous fichier annexe "*xxx.xsd*" relié au fichier "*xxx.wsdl*" via un "*xsd:import*" .

4.2. Exemple de fichier WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://calcul"
xmlns:apachesoap="http://xml.apache.org/xml-soap" xmlns:impl="http://calcul"
xmlns:intf="http://calcul" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.4 Built on Apr 22, 2006 (06:55:48 PDT)-->

<wsdl:types>
  <schema elementFormDefault="qualified" targetNamespace="http://calcul"
xmlns="http://www.w3.org/2001/XMLSchema">
    <element name="addition">
```

```

<complexType>
  <sequence>
    <element name="a" type="xsd:int"/>    <element name="b" type="xsd:int"/>
  </sequence>
</complexType>
</element>
<element name="additionResponse">
  <complexType>
    <sequence>
      <element name="additionReturn" type="xsd:int"/>
    </sequence>
  </complexType>
</element>
</schema>
</wsdl:types>

<wsdl:message name="additionResponse"
  <wsdl:part element="impl:additionResponse" name="parameters"/>
</wsdl:message>

<wsdl:message name="additionRequest">
  <wsdl:part element="impl:addition" name="parameters"/>
</wsdl:message>

<wsdl:portType name="Calculator">
  <wsdl:operation name="addition">
    <wsdl:input message="impl:additionRequest" name="additionRequest"/>
    <wsdl:output message="impl:additionResponse" name="additionResponse"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="CalculatorSoapBinding" type="impl:Calculator">
  <wsdlsoap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="addition">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="additionRequest"> <wsdlsoap:body use="literal"/> </wsdl:input>
    <wsdl:output name="additionResponse"> <wsdlsoap:body use="literal"/> </wsdl:output>
  </wsdl:operation>
</wsdl:binding>

<wsdl:service name="CalculatorService">
  <wsdl:port binding="impl:CalculatorSoapBinding" name="Calculator">
    <wsdlsoap:address
      location="http://localhost:8080/axis2-webapp/services/Calculator"/>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>

```

4.3. WSDL abstraits et concrets

Structure **WSDL** (*abstrait* ou *concret*)

WSDL abstrait (*interface*)

```

<definitions ...>
  <types>
    <schema>
      <complexType .../> ...
      <element .../> ...
    </schema>
  </types>
  <message>...</message>
  <message>...</message>
  <portType>
    <operation ...>
      <input .../>
      <output .../>
    </operation>
    <operation>...</operation>
  </portType>
</definitions>

```

WSDL concret/complet (*impl*)

```

<definitions>
  <types>
    <schema ... />
  </types>
  <message>...</message> ...
  <portType>
    <operation .../> ...
  </portType>
  <binding>
    ... HTTP/SOAP style="document"
    ... use="literal" ...
  </binding>
  <service>
    <port><address location="...url..." />
  </port> </service>
</definitions>

```

Partie abstraite

Structure et sémantique WSDL (partie abstraite)

- Un **même "WSDL abstrait"** (idéalement issu d'une norme) pourra être physiquement implémenté par **plusieurs services distincts** (avec des **points d'accès (et URL) différents**).
- Un **portType** représente (en XML) l'**interface fonctionnelle** qui sera accessible depuis un futur "port" (*alias "endPoint"*)
- Chaque opération d'un "portType" est associée à des **messages** (structures des **requêtes[input]** et **réponses[output]**).
- En mode "*document/literal*", ces **messages** sont définis comme des **références** vers des "**element**"s (**xml/xsd**) qui ont la structure XML précise définie dans des "**complexType**".

Sémantique de la partie abstraite WSDL

WSDL abstrait (*interface*)

```

<definitions ...>
  <types>
    <schema>
      <complexType .../> ...
      <element .../> ...
    </schema>
  </types>
  <message>...</message>
  <message>...</message>
  <portType name="IServiceA">
    <operation name="op1">
      <input .../>
      <output .../>
    </operation>
    <operation>...</operation>
  </portType>
</definitions>

```

Pour chaque opération opXy :

element="**opXy**" et "**opXyResponse**"
Avec **complexType** associés comportant
la liste des **paramètres** (**noms** et **types**)

"pa" et "pb"

"xsd:int" et "xsd:string"

Simple renvois (pour compatibilité ancienne version)

```

<<interface>>
IServiceA


---


.op1(pa:int,pb:string)
.op2():int

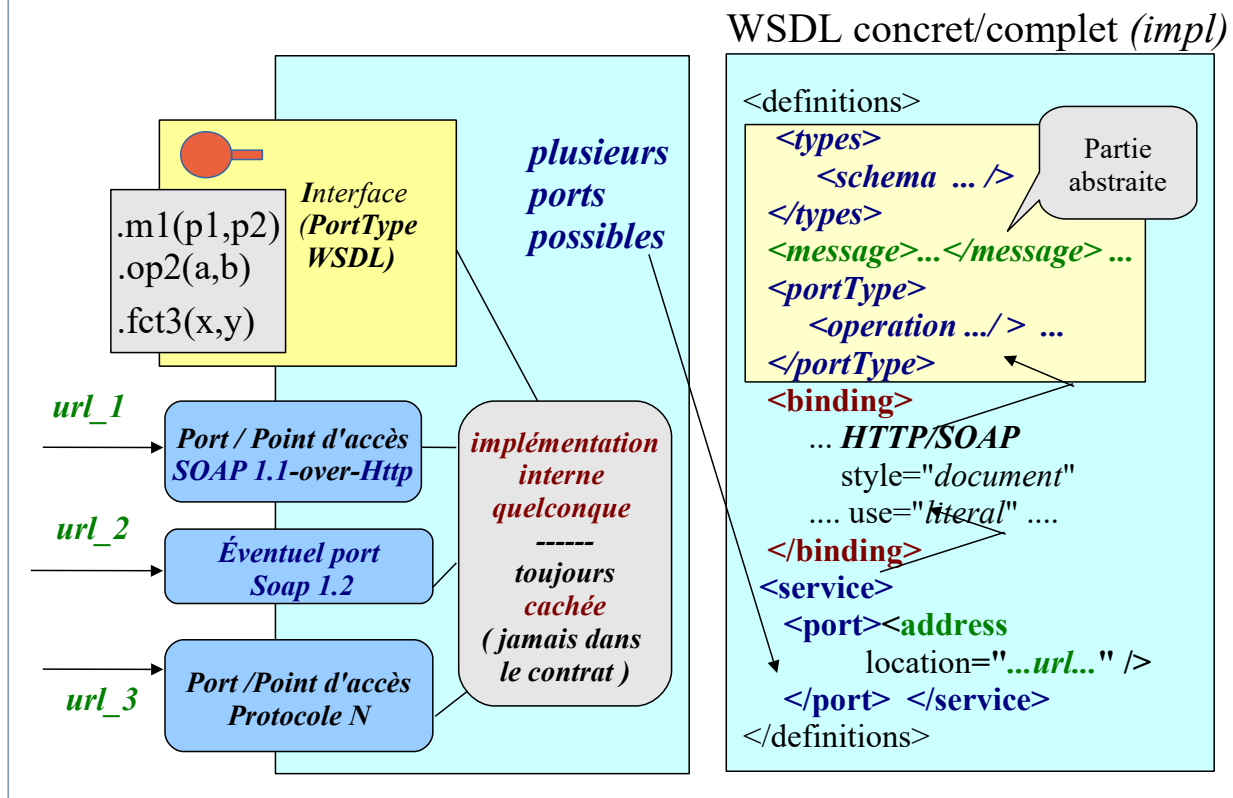
```

PortType WSDL = interface

Structure et sémantique WSDL (partie concrète)

- Un fichier "**WSDL concret**" comporte toute la partie abstraite précédente plus tous les paramétrages nécessaires pour pouvoir invoquer **un point d'accès précis sur le réseau**.
- Un **port** (alias "*endPoint*") permet d'atteindre une **implémentation** physique d'un service publié sur un serveur. La principale information rattachée est l'**URL** permettant d'invoquer le service via SOAP.
- La partie "**binding**" (référéncée par un "port") permet d'**associer/relier** entre eux les différents éléments suivants:
 - * **interface abstraite (portType)** et ses opérations
 - * **protocole de transport** (HTTP + détails ou ...)
 - * **point d'accès (port)**

Sémantique de la partie concrète WSDL



Importance des namespaces XML d'un WSDL

- * La plupart des éléments décrits dans un fichier WSDL ont des noms qualifiés (par le **targetNamespace** de la balise <definition>)
- * Le nom complet d'une **classe java** est "**packageName.className**"
- * Le nom complet d'un **service** ou **portType** est "**tns:XyName**" avec **xmlns:tns="http://packageName_a_l_envers/"** (coïncidant avec la valeur du **targetNamespace** du fichier WSDL).

Exemple :

com.xy.Customer en tant que POJO de donnée en retour d'un WS devient en WSDL **xmlns:ns="http://xy.com/"** et **ns:Customer** et est re-traduit en **com.xy.Customer** au sein des applications clientes.

Quelquefois "package java" --> namespace xml --> namespace c++

5. Tests d'un service web "soap" (soap-ui)

Test d'un web-service "soap"

- 1) **Coder** le service web et **démarrer** l'application ou le serveur qui l'héberge.
- 2) Utiliser un **navigateur internet** pour **visualiser la description WSDL** qui devrait être accessible si tout va bien :

<http://localhost:8080/myApp/xx/yy/serviceA?wsdl>

```
<?xml version="1.0" encoding="UTF-8"?>
  <wsdl:definitions>
    ....
  </wsdl:definitions>
```

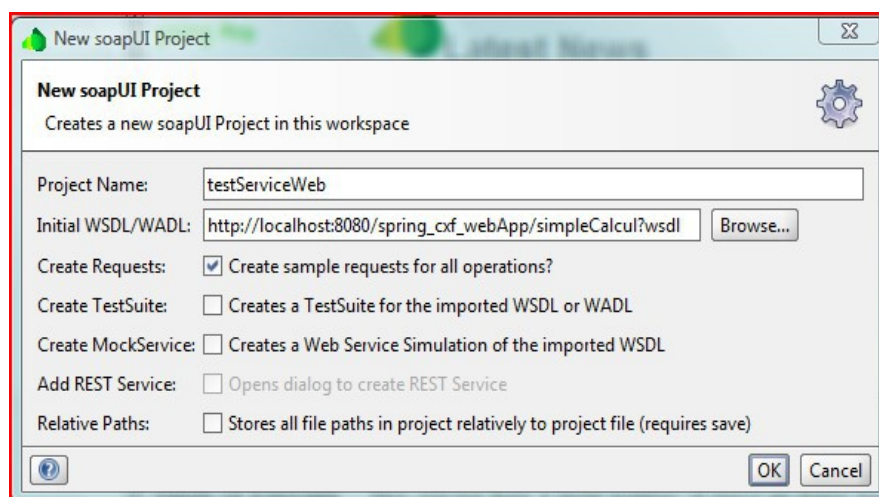
ne pas
oublier
?wsdl

- 3) Utiliser l'application de test "soap-ui" (ou un équivalent) pour analyser le fichier WSDL et **lancer des requêtes "soap"**.

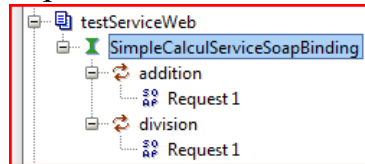
Une version gratuite de **soap-ui** peut se télécharger et s'installer facilement.

Après avoir démarré l'application soap-ui, il faut **créer un nouveau projet de test (à nommer)**.

Paramètre à fixer : l'**url** complète de la description **wSDL**.



Soap-ui analyse la description wsdl et reconnaît les opérations existantes que l'on peut tester :



Un (double-)clic sur "request1" d'une méthode à tester fait apparaître une structure de requête soap/xml où il faut simplement spécifier quelques valeurs .

déclenchement

requête soap

réponse soap (ou erreur)

```

<?xml version='1.0'?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header/>
  <soapenv:Body>
    <myws:addition>
      <x>2</x>
      <y>3</y>
    </myws:addition>
  </soapenv:Body>
</soapenv:Envelope>
  
```

```

<?xml version='1.0'?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns1:additionResponse xmlns:ns1="http://myws">
      <return>5.0</return>
    </ns1:additionResponse>
  </soap:Body>
</soap:Envelope>
  
```

response time: 711ms (199 bytes) 1:1

6. Quelques API et technologies "web services"

Principales API associées au web-services

Langage **C++** ---> **gSoap** , ...

Langage **C#** ---> infrastructure ".net" et "wcf"

Langage **Java** --->

Ancienne API "JAX-RPC" (pour jdk <= 1.4)

Nouvelles API "**JAX-WS**" et "**JAX-RS**" (pour jdk >= 1.6)

L'api "**JAX-WS**" (Java Api for Xml Web-Services) se paramètre avec des annotations java (**@WebService** , **@WebParam** ,) et utilise en interne JAXB pour gérer les correspondances java <---> xml .

L'api "**JAX-RS**" (pour les Web Service **REST**) est assez récente (depuis JEE 6) et se paramètre via des annotations (**@Produces** , **@GET** , **@POST** ,) qui permettent de préciser les formats des données (XML , JSON ,) et les paramétrages HTTP (fin d'URL ,) .

Autres API "web services" secondaires

WSDL4J permet (si besoin) de manipuler directement un fichier "WSDL" depuis du code java . En couplant "WSDL4J" avec un peu d'introspection java on peut éventuellement déclencher des appels entièrement dynamiques (en groovy par exemple).

JAX-R (Java Api for Xml Registry) est une api officielle java permettant de se connecter à des annuaires de services (ex: UDDI , ebXml ,) .
UDDI4J (non officiel) existe aussi et est plus étroitement associé à UDDI.

7. API WS SOAP JAVA : JAX-WS

Présentation de l'api JAX-WS

JAX-WS signifie *Java Api for Xml Web Services*

JAX-WS est déjà intégré dans le jdk >=1.6

Principales caractéristiques :

- paramétrage par annotations (ex: *@WebService* , *@WebParam*)
- utilisation interne de *JAXB2* (et ses annotations *@Xml...*)

JAX-WS est dédié au services "SOAP" avec description WSDL.

Une partie de JAX-WS est dédiée à WS-S (WS-Security).

La technologie "*CXF*" complète le jdk sur certains points (sécurité , intercepteurs , intégration webApp , intégration spring, ...)

7.1. Mise en oeuvre de JAX-WS (coté serveur)

Paramétrage d'une **interface** de service web

```
//@WebService(targetNamespace="http://services.myapp.xy.com/")
@WebService
public interface GestionComptes {

    public Compte getCompteByNum(
        @WebParam(name="numCpt") long numCpt)
        throws MyServiceException;

    public List<Stat> getStats(
        @WebParam(name="annee") int annee);

    public void transferer(
        @WebParam(name="montant") double montant,
        @WebParam(name="numCptDeb") long numCptDeb,
        @WebParam(name="numCptCred") long numCptCred)
        throws MyServiceException;
}
```

Paramétrage d'une **classe d'implémentation**

```
/*@WebService(targetNamespace="http://impl.services.myapp.xy.com/",
endpointInterface="com.xy.myapp.services.impl.GestionComptes")*/

@WebService(endpointInterface=
    "com.xy.myapp.services.impl.GestionComptes")
public class GestionComptesImpl implements GestionComptes {
    ... // code d'implémentation java habituel (R.A.S.)
}
```

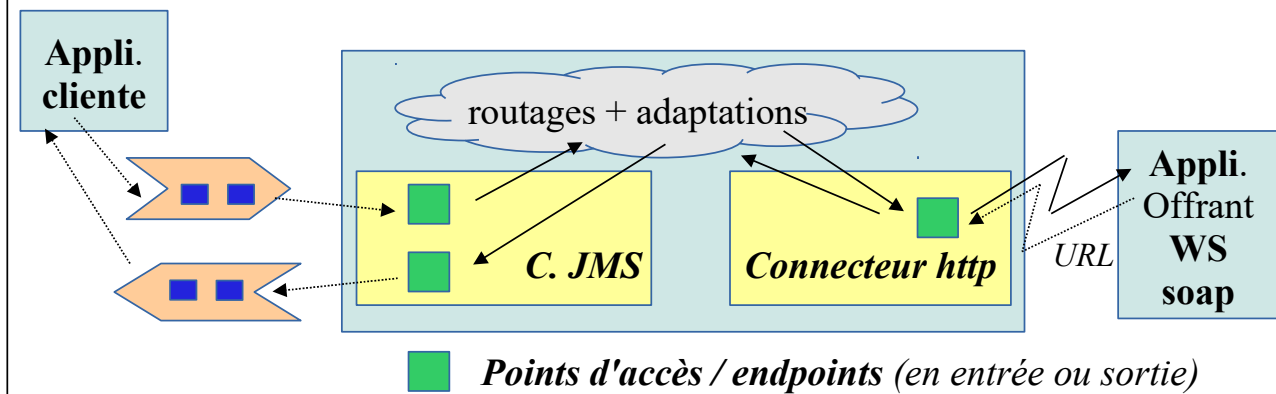
Paramétrage d'une **classe de données** (en entrée ou sortie)

```
@XmlType(namespace="http://data.myapp.xy.com/")
@XmlRootElement(name="stat")
public class Stat {
    private int num_mois; //de 1 a 12
    private double ventes; //+get/set
    ... }
}
```

8. EAI et ESB

E.S.B. = Enterprise Service Bus :

- Un **ESB** est essentiellement à voir comme un serveur d'accès à des services distants .
- Une application cliente peut se connecter à l'ESB selon un **grand choix de protocoles** (avec **points d'accès** gérés par connecteurs).
- Les services fonctionnels sont souvent rendus par d'autres applications en arrière plan et combinés ou enrichis par l'ESB.



Principales fonctionnalités d'un ESB

- **Routage de messages** (*selon URL, éventuellement conditionné*)

Changement de protocoles

(*http:// , file:// , jms , smtp , rmi , ...*)

- **Transformation de format de message**
(*xml , json , ...*)

- **Adaptations fonctionnelles** (*traductions, mapping entre Api , xslt , ...*)

- **Enrichissements techniques**
(*logs, sécurité, monitoring, ...*)

- **Combinaison/orchestration de services**

- **Éventuels services internes** (*java , ...*)

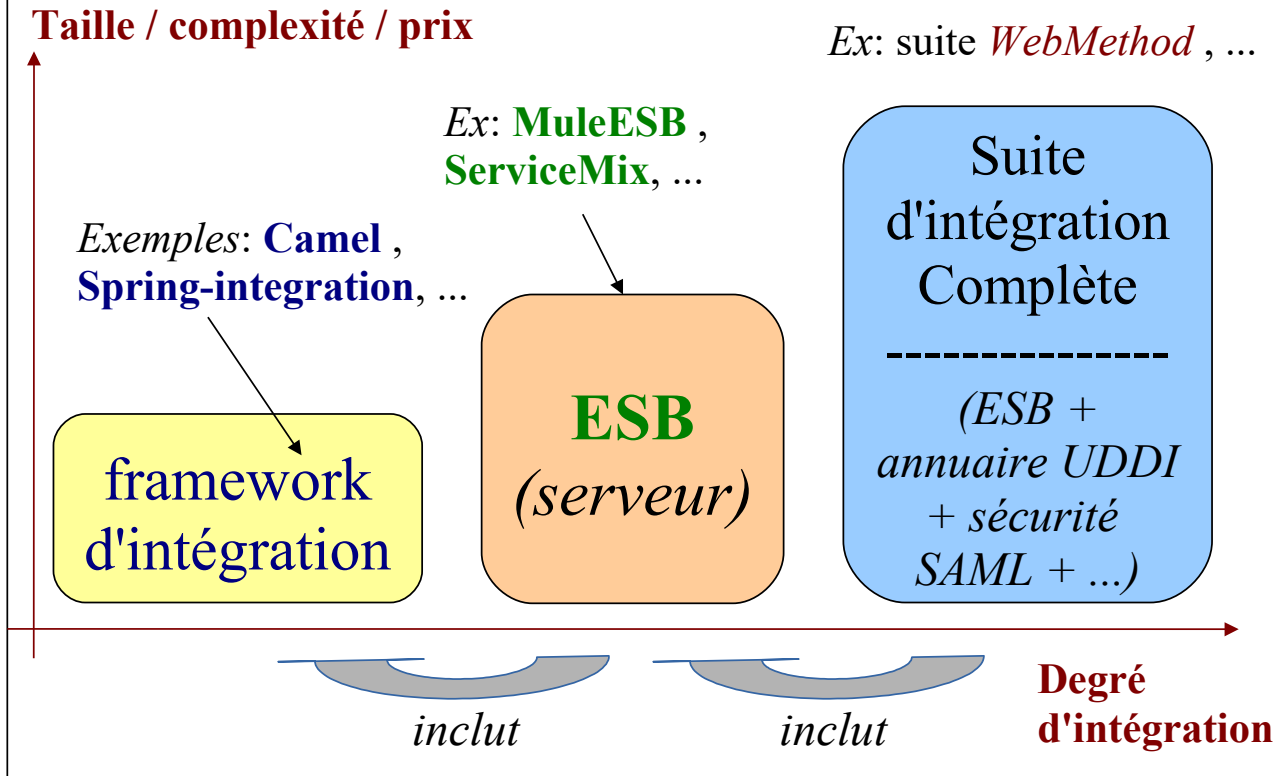
- **Supervision** (*PKI, SAM, BAM*)

essentiel

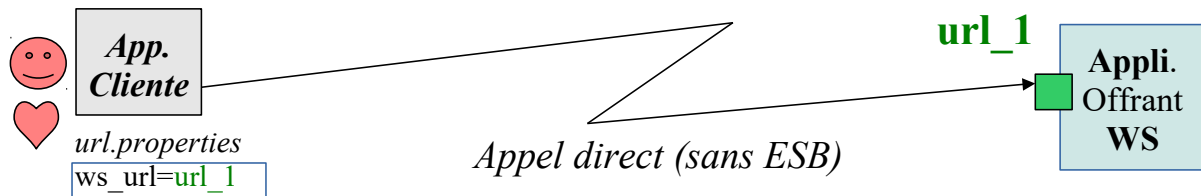
*éventuelle
intégration
de bpel ou bpmn2*

*potentiellement
serveur d'applications
(comme jee)*

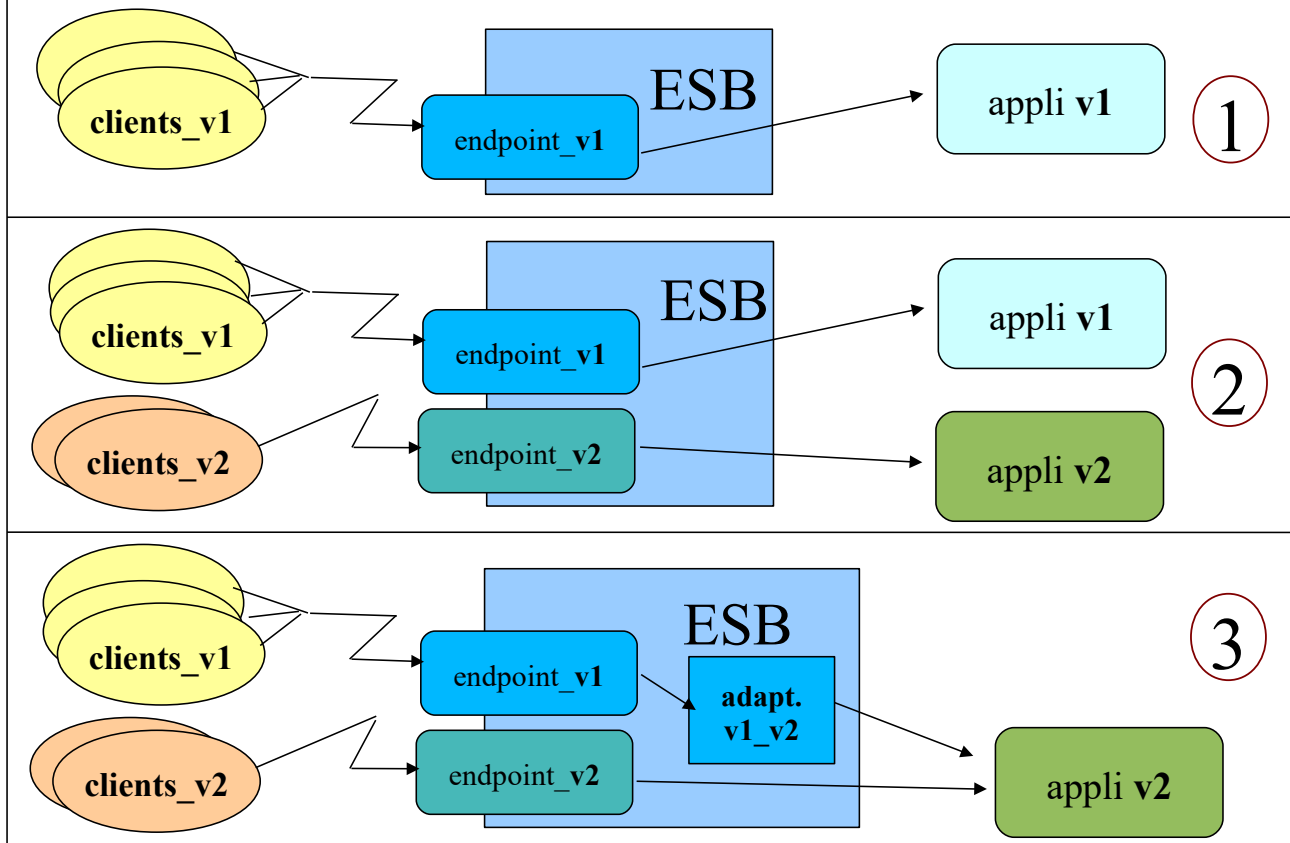
ESB et framework d'intégration



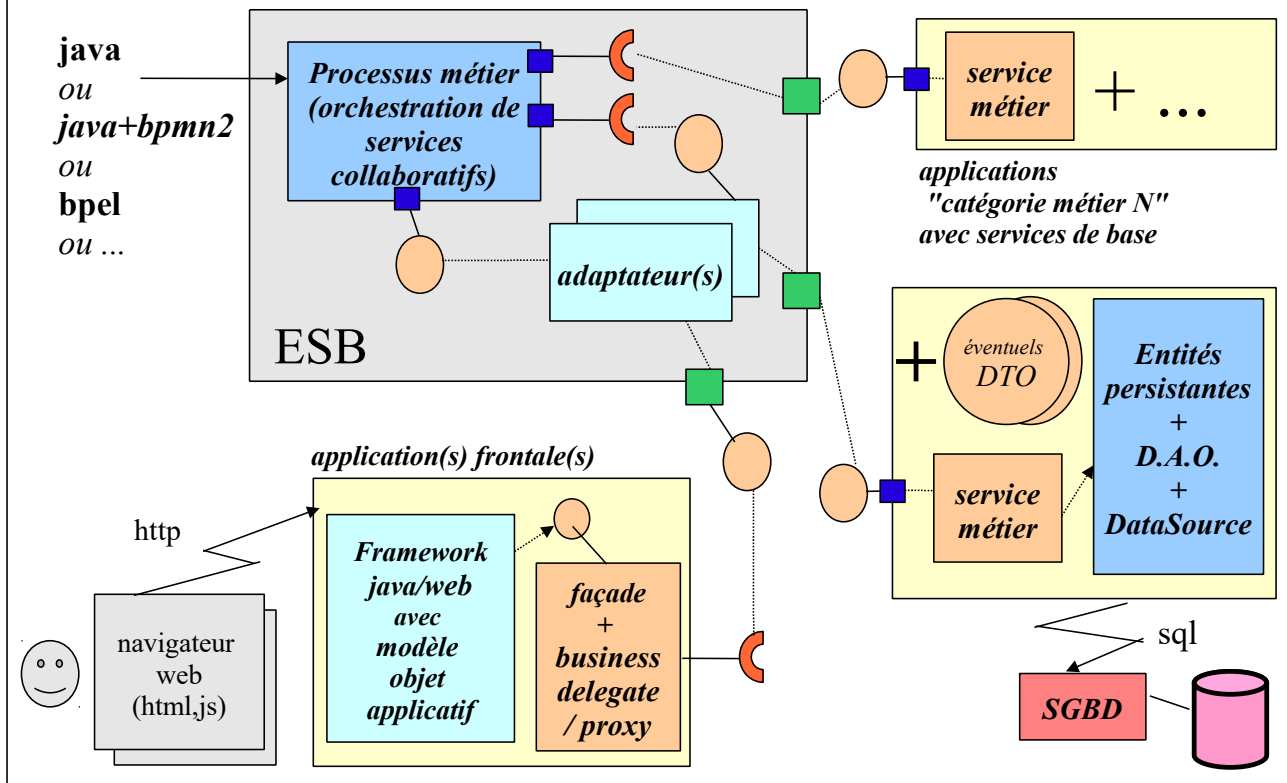
Trois variantes fonctionnellement identiques :



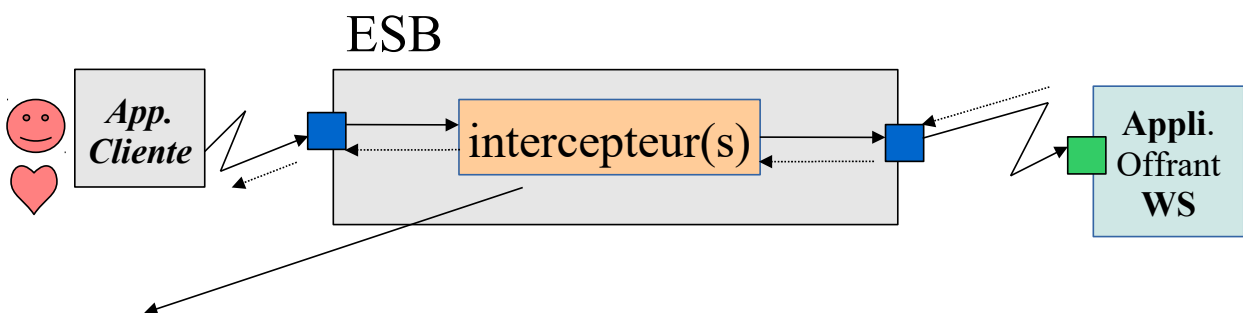
Exemple d'évolution de versions en douceur



ESB hébergeant éventuellement des services d'orchestration au sein d'une architecture SOA :



Intercepteurs, PKI, SAM et BAM



Production potentielle de **mesures/statistiques** selon paramétrages
(**KPI**= **K**ey **P**erformance **I**ndicator)

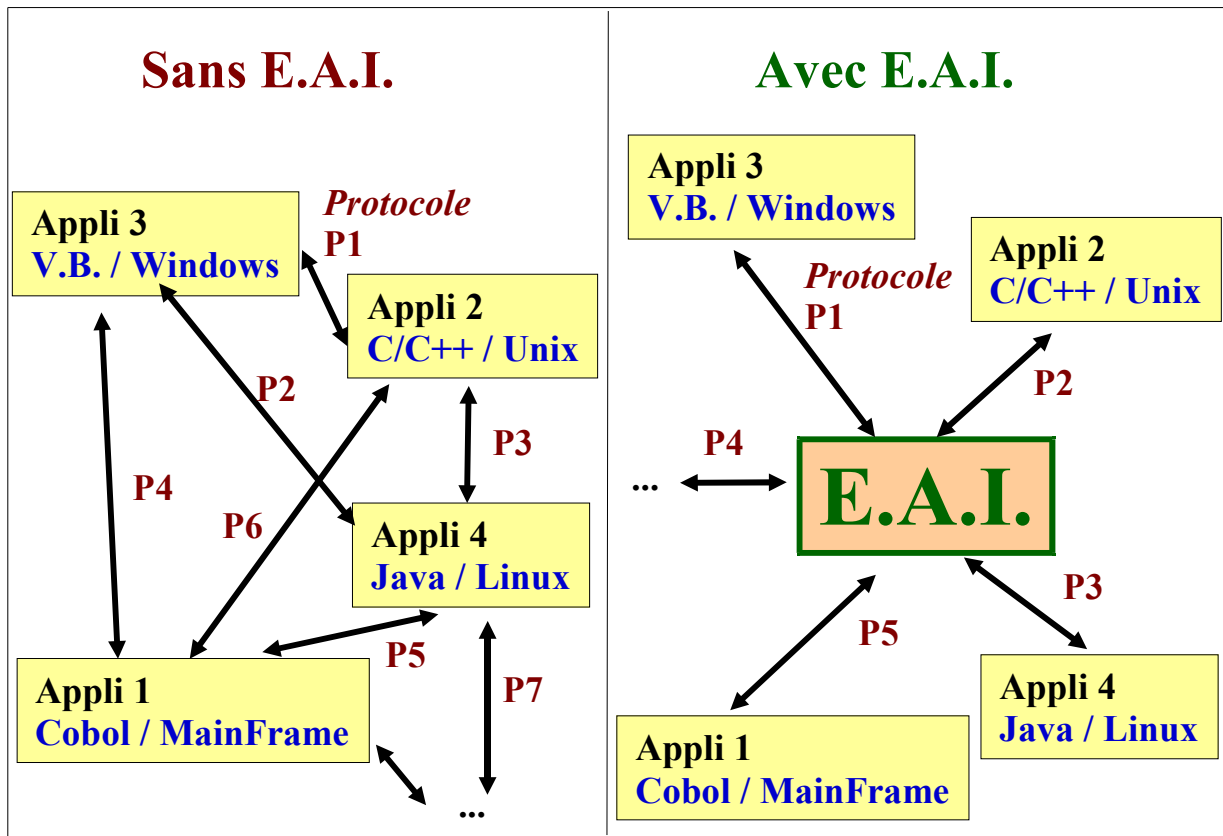
SAM = **S**ystem **A**ctivity **M**onitoring
(supervision technique/opérationnelle)

BAM = **B**usiness **A**ctivity **M**onitoring
(supervision fonctionnelle/métier)

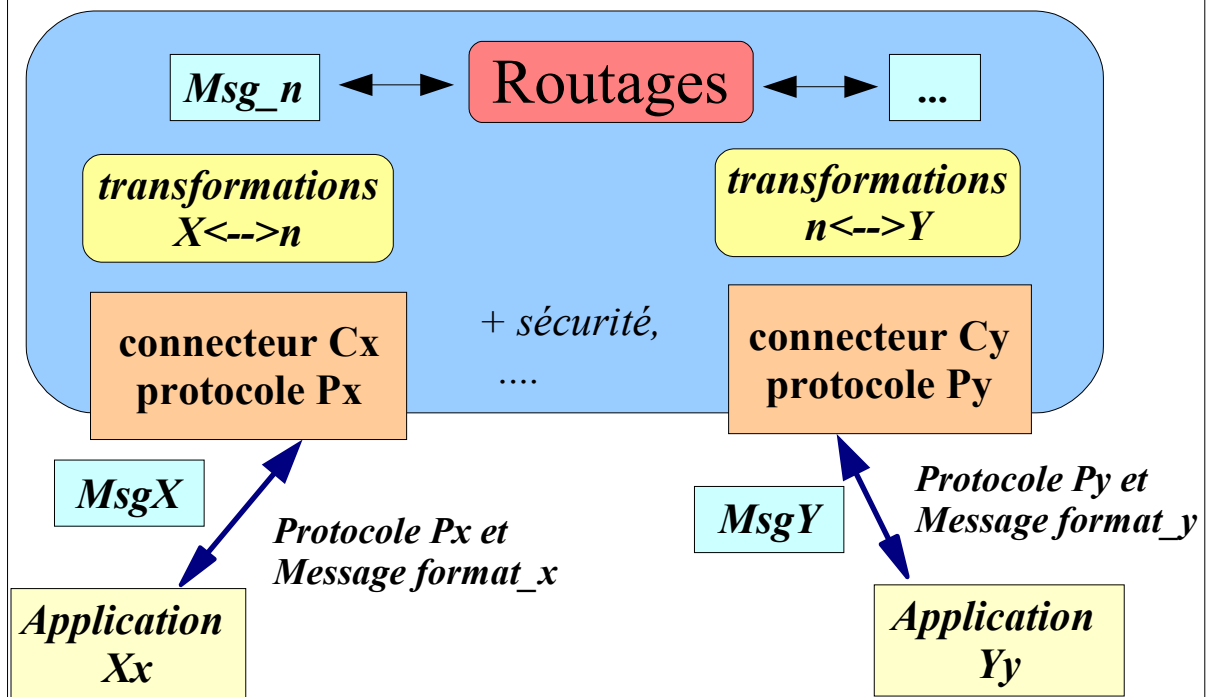
À bien doser !

(trop de mesures/stats peuvent sensiblement ralentir les flux dans l'ESB) .

9. EAI (Enterprise Application Integration)



Fonctionnalités d'un EAI:

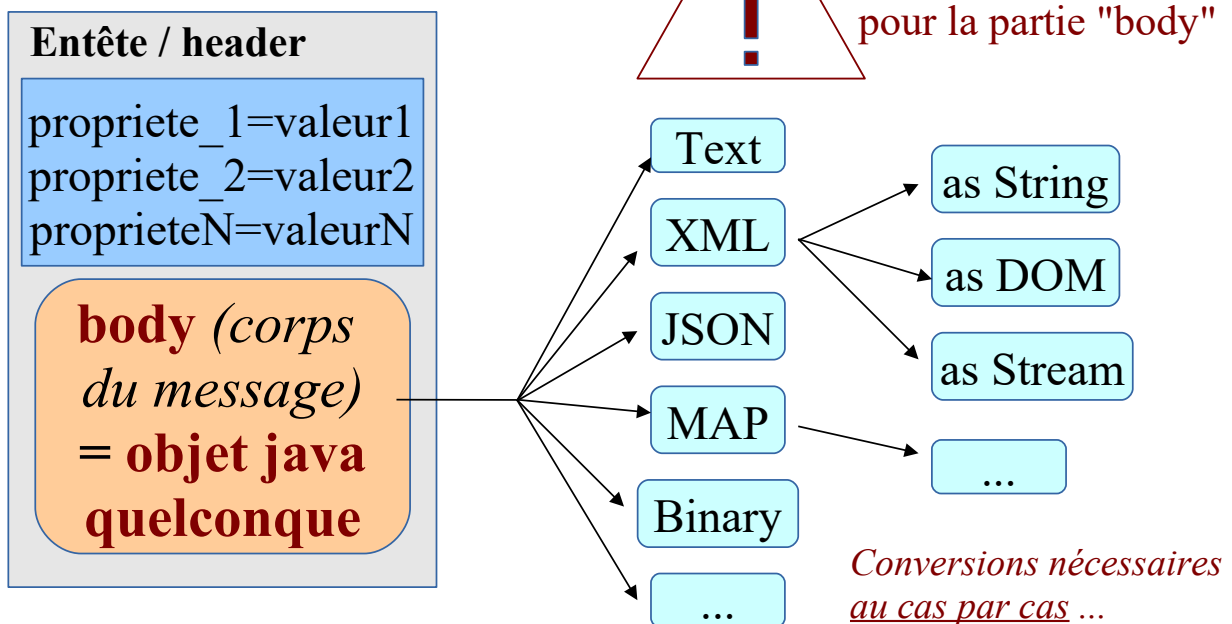


Transformations techniques internes à l'EAI/ESB

- * Si au cas par cas : potentiellement **$O(n^2)$**
transformations directes
[complexe , performant]
- * Si format interne "*neutre*" ou "*normalisé*"
 $(x \leftrightarrow n \leftrightarrow y)$
 $\rightarrow 2*n$ doubles transformations
[simple , moins performant]

Cas fréquent des ESB basés sur java :

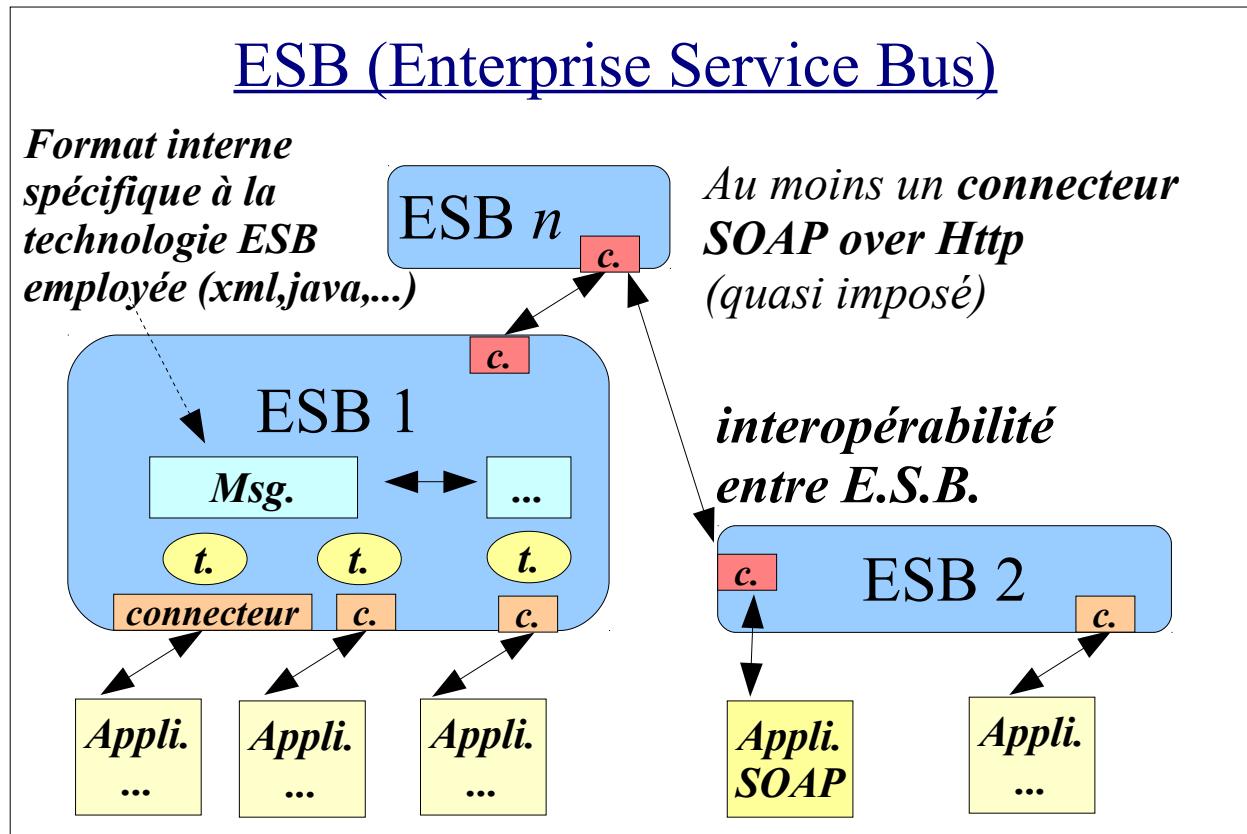
Structure d'un **message** interne géré par l'ESB :



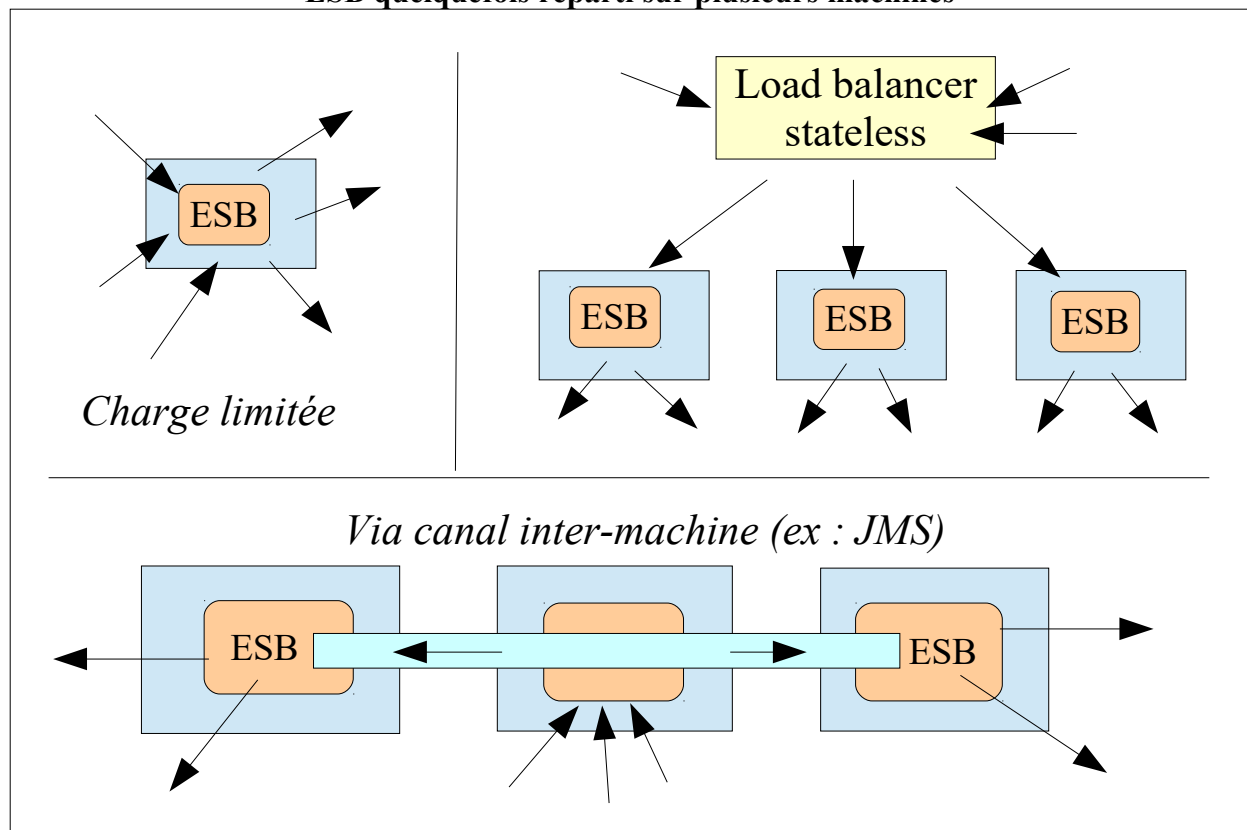
Grands traits des premiers EAI:

- Xml quelquefois utilisé pour le format des messages intermédiaires (re-transformations pratiques).
- EAI = concept
==> différentes implémentations
assez propriétaires (ex: **Tibco** , **WebMethod**, ...) et pas directement interopérables.
- Attention aux performances (beaucoup de trafic de messages + traitements CPU liés aux transformations , ...).

10. ESB (Enterprise Service Bus)



ESB quelquefois réparti sur plusieurs machines



Configuration requise au sein d'un ESB

Un ESB offre généralement une infrastructure à base de connecteurs et transformateurs paramétrables .

On a généralement besoin de configurer :

- * des *points d'entrées et de sorties (endpoints)* avec des *URLs* et protocoles particuliers.
- * des *transformations* et *routages internes* au cas par cas (selon les besoins)
- * *des définitions logiques des services rendus ou accessibles* (ex : fichier **wSDL** , code java annoté , ...)

ESB: éléments libres & imposés (propriétaires & standards)

Tout ESB se doit de:

- * Offrir un accès interne dans un format local unifié (java, xml/soap/wSDL , ou ...) aux services (internes et externes) enregistrés.
- * Permettre à une application externe de se connecter (au moins via (SOAP, WSDL)) à un service pris en charge par l'ESB (indirectement via un connecteur)

Chaque ESB est libre de:

- * se configurer à sa façon (scripts, console , fichiers de configuration, ...).
- * de gérer à sa guise les enregistrements / déploiements de services.

10.1. Principaux ESB

Liste (non exhaustive) de quelques ESB :

ESB des grandes marques (avec prix quelquefois élevé mais un support généralement sérieux) :

<i>ESB</i>	<i>Editeur</i>	<i>Caractéristiques</i>
Tibco ESB	Tibco	Un des pionniers (anciennement EAI)
WebMethod	Software AG	Autre pionnier (anciennement EAI). Les versions récentes ont été beaucoup améliorées.
OSB (<i>Oracle Service Bus</i>)	Oracle	ESB très complet (<i>attention : pas mal de différences entre certaines versions</i>). <i>Architecture très propriétaire.</i>
WebSphere ESB	IBM	ESB très complet (utilisant norme SCA)
BizTalk	Microsoft	Lien avec norme WCF de .net
Talend ESB	Talend (éditeur ETL)	Suite SOA complète (en interne basé sur composants "open-source")

ESB "Open source" ou mixte (version "community" et version "payante") :

<i>ESB</i>	<i>Editeur</i>	<i>Caractéristiques</i>
<i>OpenESB "has been" ServiceMix [3.x , 4.3] Petals</i>	SUN, Apache, OW2 (éditeur de Jonas)	<i>JB1</i> et "has been"
ServiceMix >= 4.4	Apache	OSGi (quasiment plus JBI) , plus d'intégration possible de ODE/BPEL mais intégration possible de <i>activiti5</i>
FuseESB	FuseSource puis reprise par Jboss/Red-Hat	Version améliorée de ServiceMix (avec support , documentation plus précise,)
Mule ESB	MuleSoft	Bon ESB assez populaire relativement léger. Il existe une version payante/améliorée avec plus de connecteurs. Accompagné de l'IDE "MuleStudio" .
WSO2	WSO2	Basé sur OSGi (comme ServiceMix et FuseESB). Suite SOA assez complète
<i>Spring Intégration</i> et <i>Camel</i>	Spring , Apache	Framework d'intégration (API) pour intégration légère.

NB : L'ancienne norme JBI (de SUN) est tombée à l'eau car trop complexe.

11. Notion de "moteur de services"

Utilités des "moteurs de services (java/bpel,...)"

* **transformations des formats des messages fonctionnels**

(ex : *addition(a,b)* <---> *add(x,y)* ,

chaîne_adresse <----> *adresse xml(rue + codePostal + ville)*)

techniquement , ce type de transformation peut être effectué via **XSLT** (via "interpréteur xslt") ou via **java** (avec dozer ou ...).

* **orchestration de services**

Les technologies "**BPEL**" et "**java/jBpm**" nécessitent des moteurs d'exécutions (interpréteurs) spécifiques pour faire fonctionner des services collaboratifs (*invokant des services élémentaires selon un algorithme/processus convenu*)

Moteurs de services

(exécution, intégration)

Moteur de Services BPEL

*Service n
(script BPEL
interprété)*

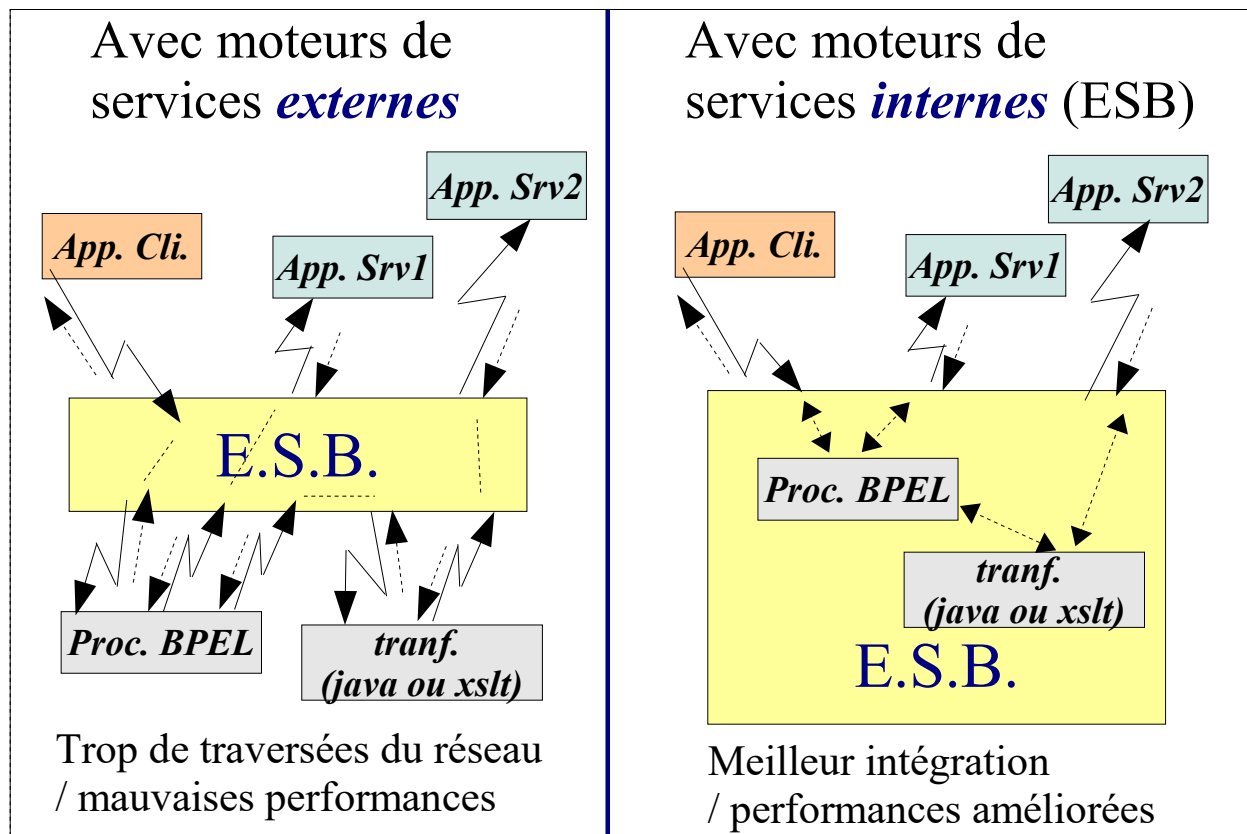
Moteur de Services Java (Serveur d'applications)

*Service Yy
(annotations jax-ws
interprétées)*

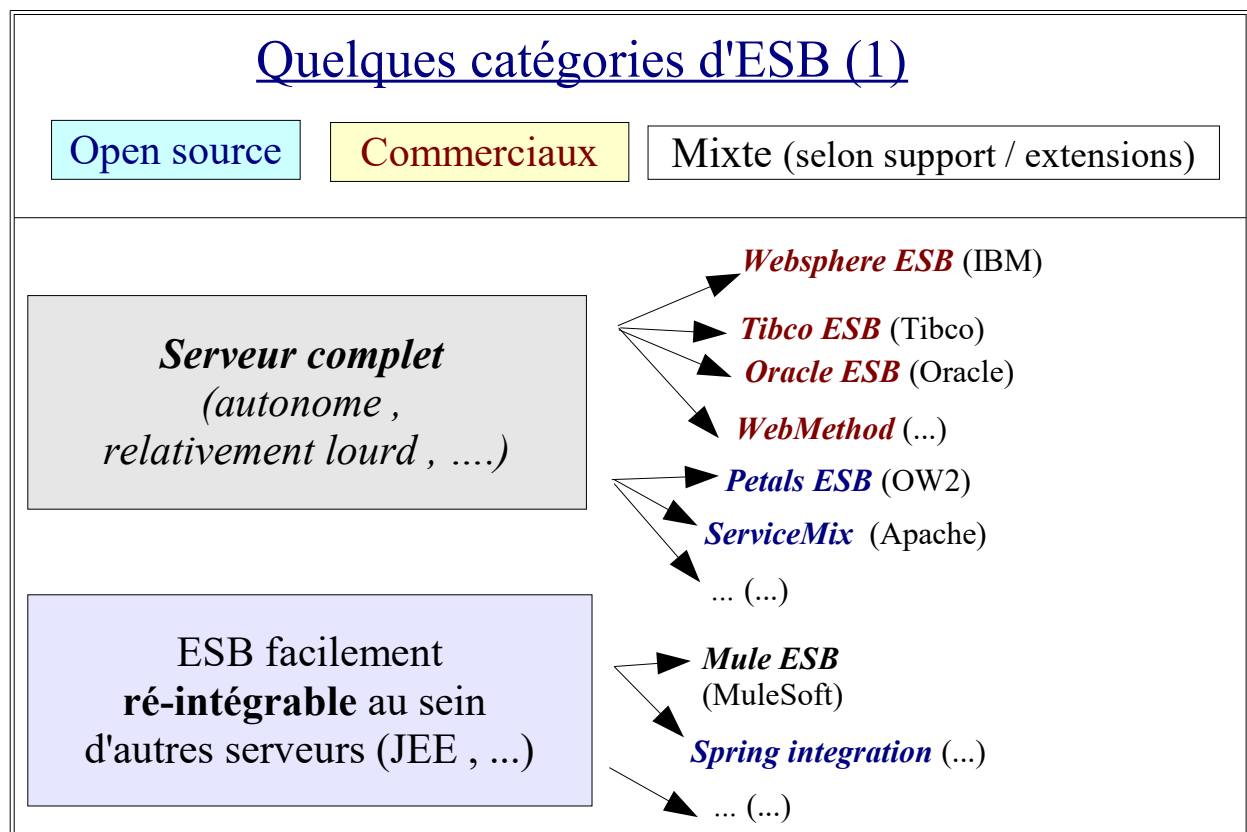
* **Définition logique WSDL indépendante du déploiement.**

* ***l'URL directe du point d'accès au service est liée au moteur de services dans lequel le service est déployé et interprété.***

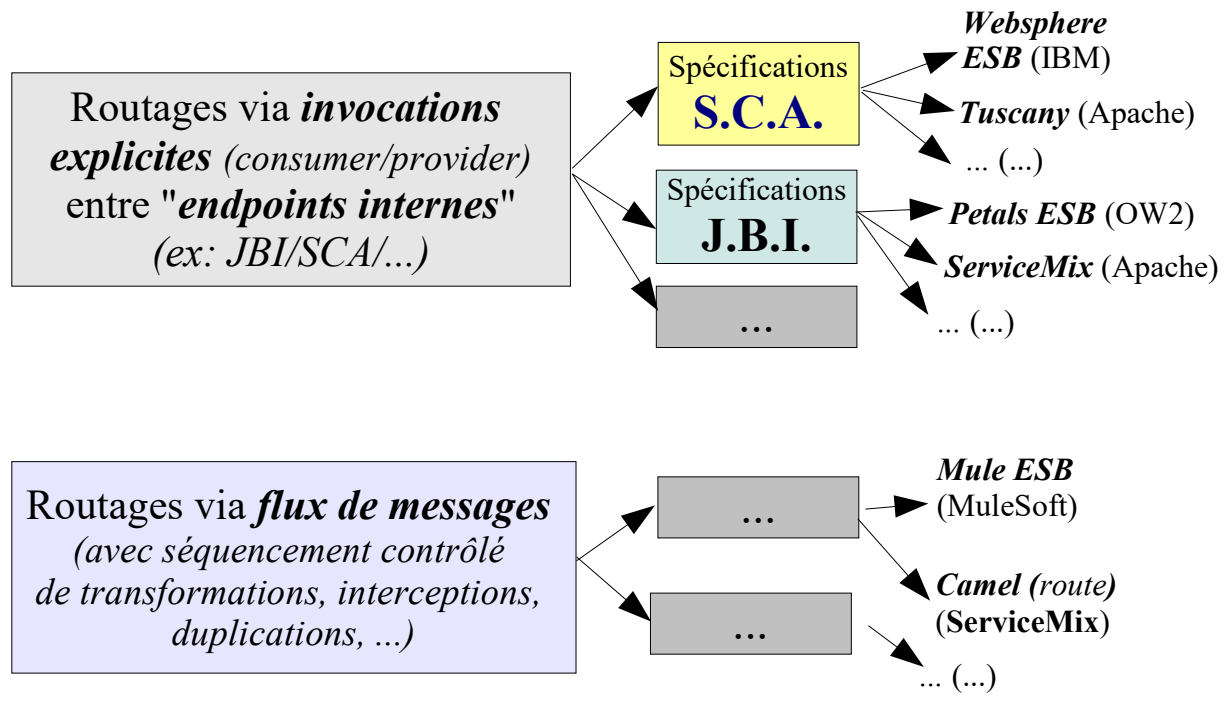
* ***Application "Moteur de services" liée à l'ESB ? (interne , externe ou ... ?)***



12. Catégories d'ESB

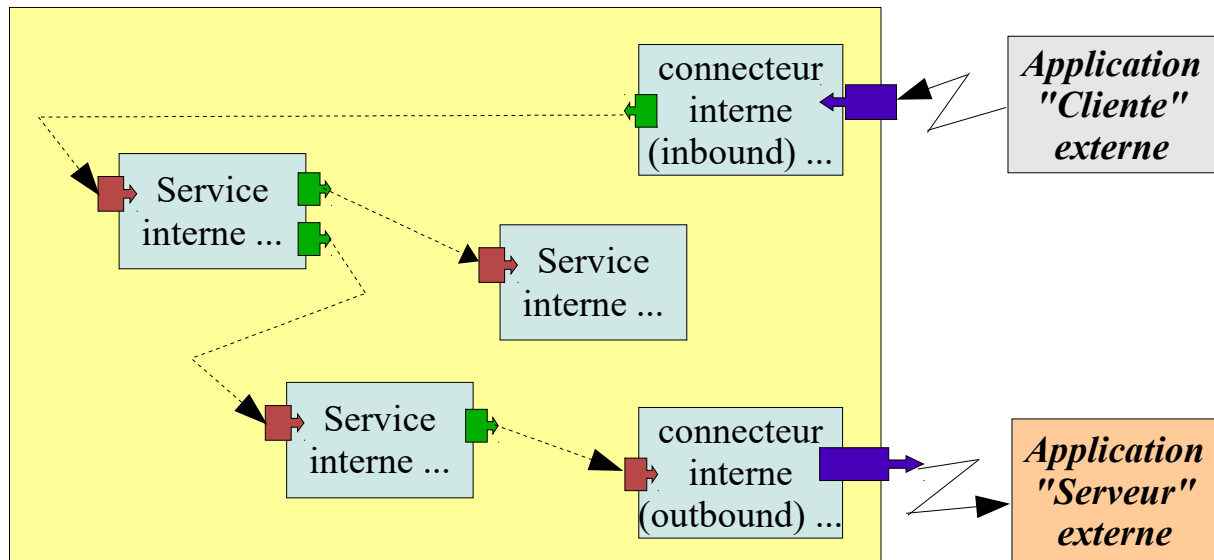


Quelques catégories d'ESB (2)



"Endpoints" internes à un ESB et connexions

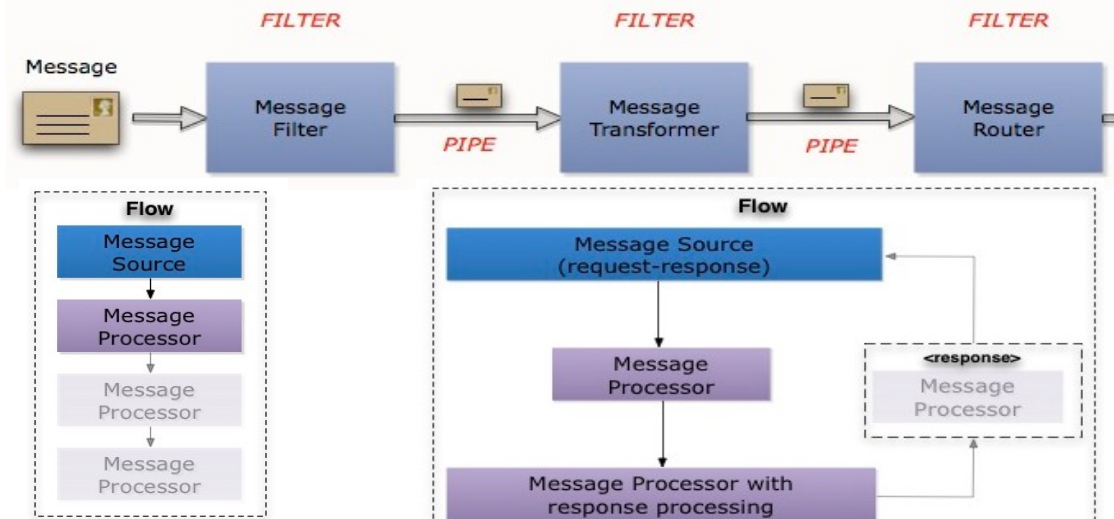
➡ = point d'accès (endpoint) interne à l'esb



ESB (de type SCA , JBI ou ...)

ESB configuré(s) via des flux/flots de messages

Au lieu d'expliciter des connexions entre "endPoints internes", *certaines ESB (ex : MuleESB) se configurent en paramétrant une **suite (séquentielle) de traitements à appliquer sur des flux de messages** qui entrent dans l'ESB au niveau d'un point d'accès précis (url , ...)* .



ESB = microcosme , écosystème

*Chaque type d'ESB (SCA, JBI , Mule, ...) peut être vu comme une sorte de **microcosme (ou écosystème)** à l'intérieur duquel seront pris en charge des assemblages de services internes.*

La configuration exacte d'un service interne dépend énormément de l'ESB hôte .

D'un ESB à un autre, des fonctionnalités identiques peuvent se configurer de manières très différentes.

Attention : connecteurs quelquefois très limités et paramétrages quelquefois complexes (au sein de certains ESB) !!!!

13. SOA et mode asynchrone

Liens entre "appel de fonction" et document

Une *structure XML* de type

```
<commande>
  <numero>1</numero>
  <adresse>...</adresse>
  ...
</commande>
```

peut aussi bien représenter :

- * une requête vers une opération d'un service web
- * un document transmis à traiter/analyser

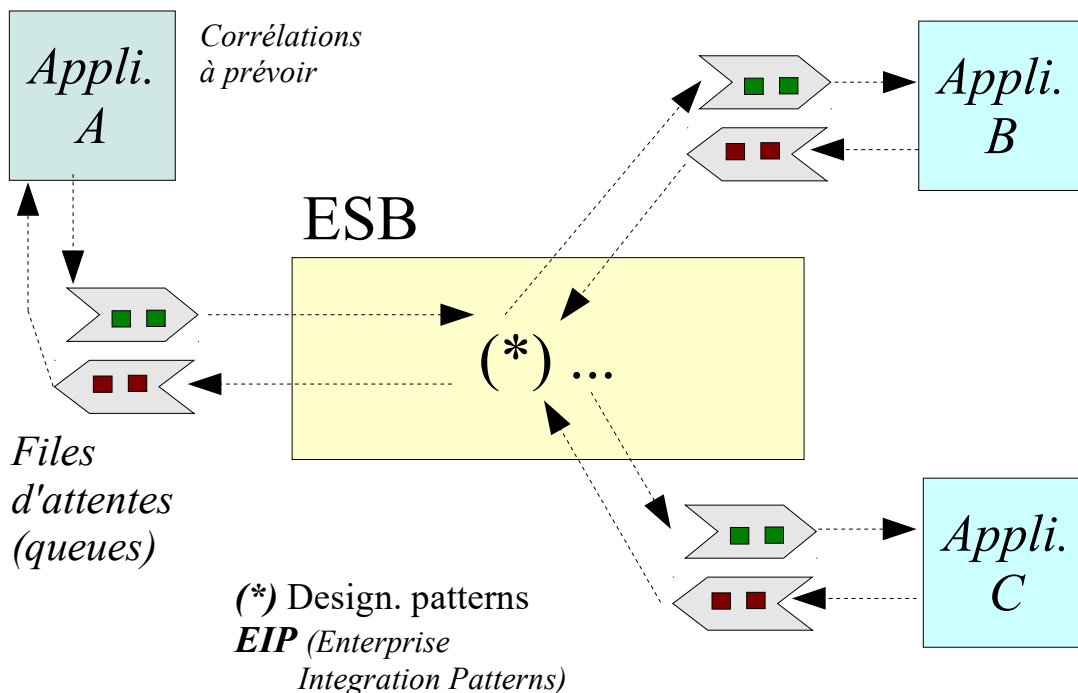
Cette structure peut être véhiculée par :

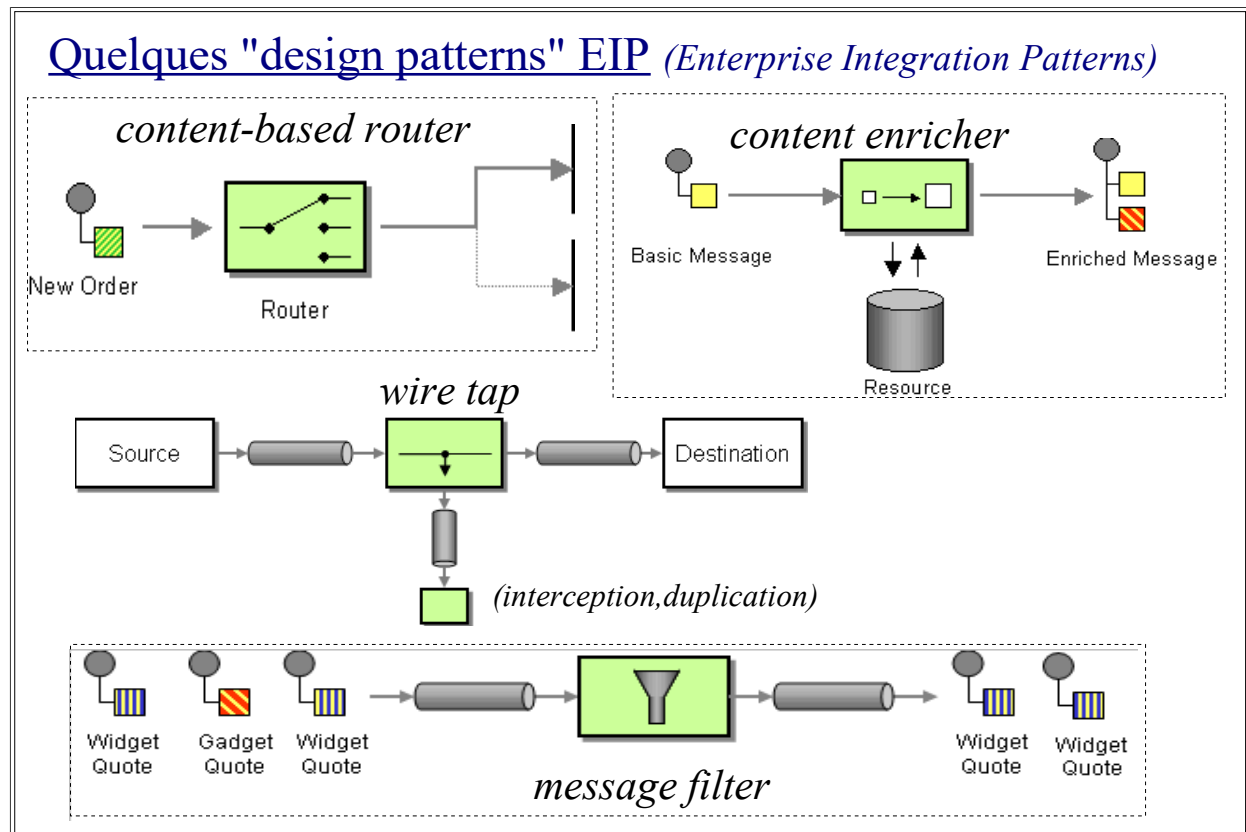
- * une enveloppe SOAP (elle même véhiculée par HTTP ou ...)
- * un message JMS (déposé dans une file d'attente)
- * ...

==> passerelles techniques envisageables entre :

- * synchrones et asynchrones
- * document et RPC (Remote Procedure Call) .

SOA en mode asynchrone





14. BPEL (présentation)

BPEL (*Business Process Execution Language*)

BPEL (*BPEL4WS* renommé *WS-BPEL*) a été créé par le consortium OASIS et vise à encoder en Xml des services Web de haut niveau qui orchestrent ou pilotent des services Web de plus bas niveaux.

On parle généralement en terme de "*processus collaboratif*" pour désigner le type de services produit via BPEL .

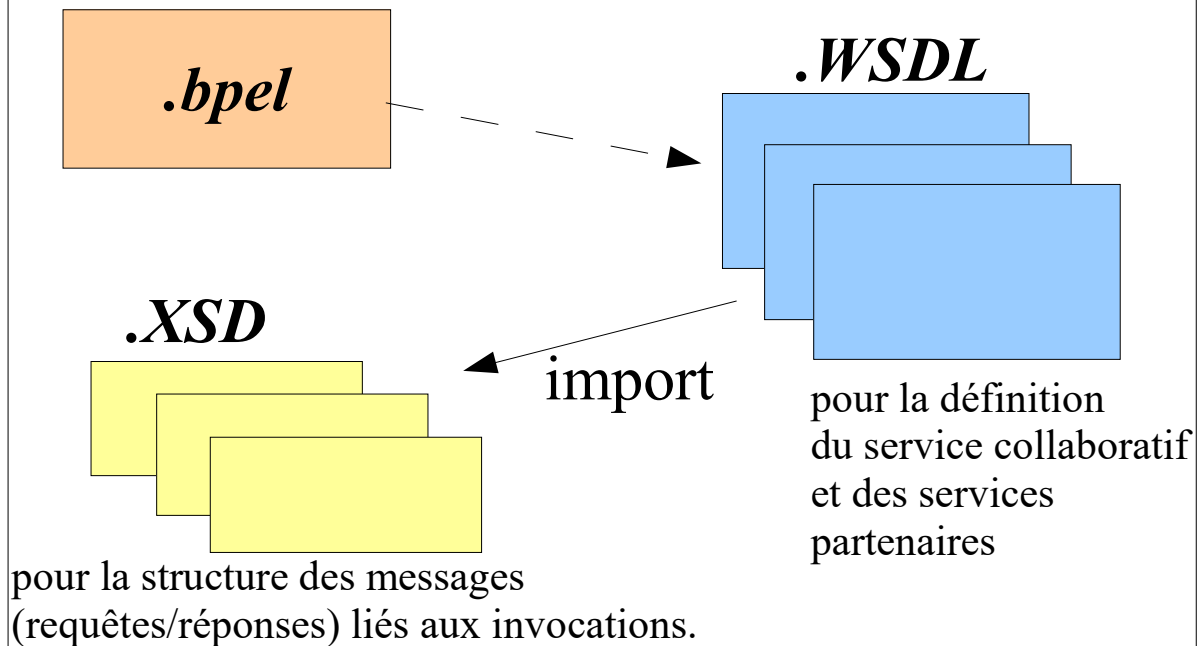
NB: Deux grandes versions :

- * BPEL 1.1 (2003) et BPEL 2.0 (2007)
- * En pratique quelques variantes d'interprétations selon le moteur d'exécution (ODE, Orchestra , ...)

Principales fonctionnalités de BPEL:

- **invoker** des services externes (partenaires)
- gérer finement le *contenu des messages de requêtes et de réponses* avant et/ou après les différentes invocations.
- ajouter une logique métier de haut niveau (tests , vérifications, *copies et calculs de valeurs à retransmettre*).
- **orchestrer / séquencer** de façon adéquate les invocations(*synchrones ou asynchrones*) de services partenaires.

Technologies XML utilisées par BPEL:



Quelques implémentations : moteurs BPEL

- **ODE** (open source , apache)
à intégrer dans *Tomcat* ou *ServiceMix*
- **BPEL-SE** de OpenESB et GlassFish V2 (*SUN*)
- **Orchestra** (Open source, *OW2*)
- **BizTalk Server** (*Microsoft*)
- **WebSphere Process Server** (*IBM*)
- **Oracle BPEL Process Manager**
- **SAP Exchange Infrastructure**
- **ActiveVOS**
- ...

14.1. Structure BPEL

Structure générale d'un processus BPEL:

```
<?xml ... ?>
<process ...>
  <import ...="...wsdl"/>
  <import ...="...wsdl" />
  <partnerLinks>
    <partnerLink .../>
  </partnerLinks>
  <variables>
    <variable .../>
  </variables>
  ... instructions ...
</process>
```

*déclarations logiques
des services partenaires*
(types précis définis dans
wsdl , partenaire spécial =
le process bpel lui même)

déclarations de variables
(messages transmis , parties
recopiées ou calculées)
(types précis définis dans
schémas ".xsd" des ".wsdl")

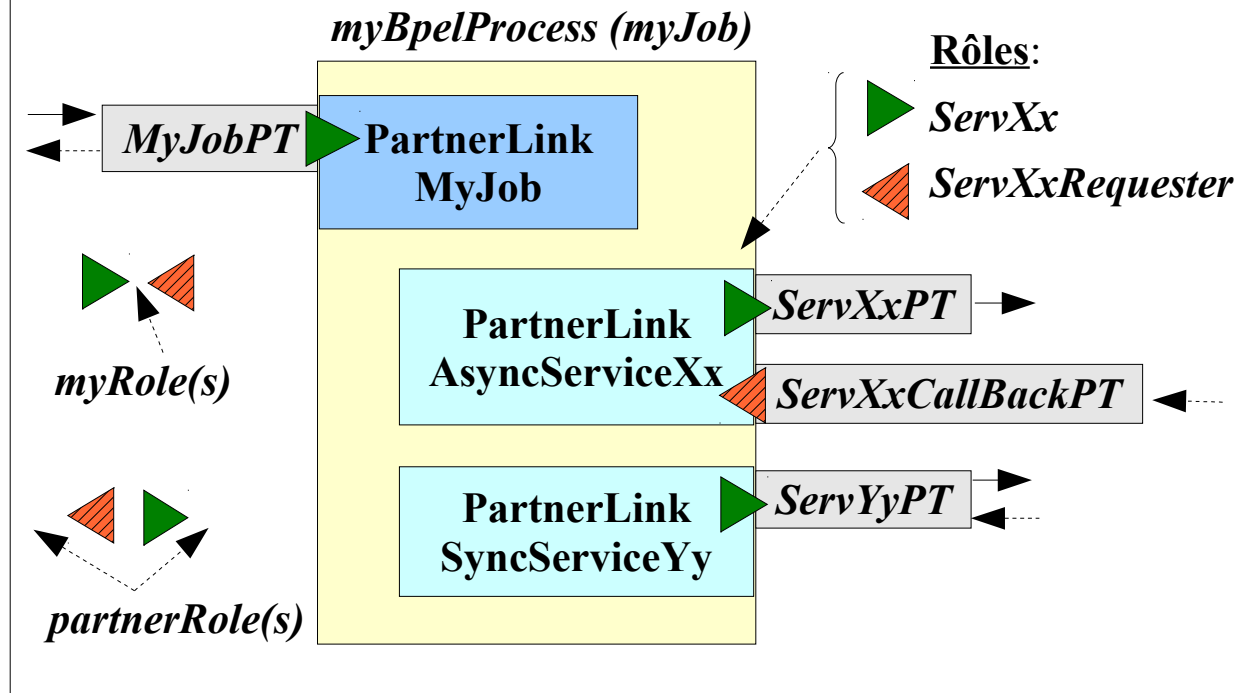
Séquence type d'un processus BPEL:

```
<process ...> ....
  <sequence>
    <receive partnerLink=".."
      operation="..." variable="..." />
    <assign>
      <copy> from ... to ... </copy>
    </assign>
    <invoke partnerLink=".."
      operation=".." inputVariable=
      "..." outputVariable="..." /> ...
    <reply partnerLink=".."
      operation="..." variable="..." />
  </process>
```

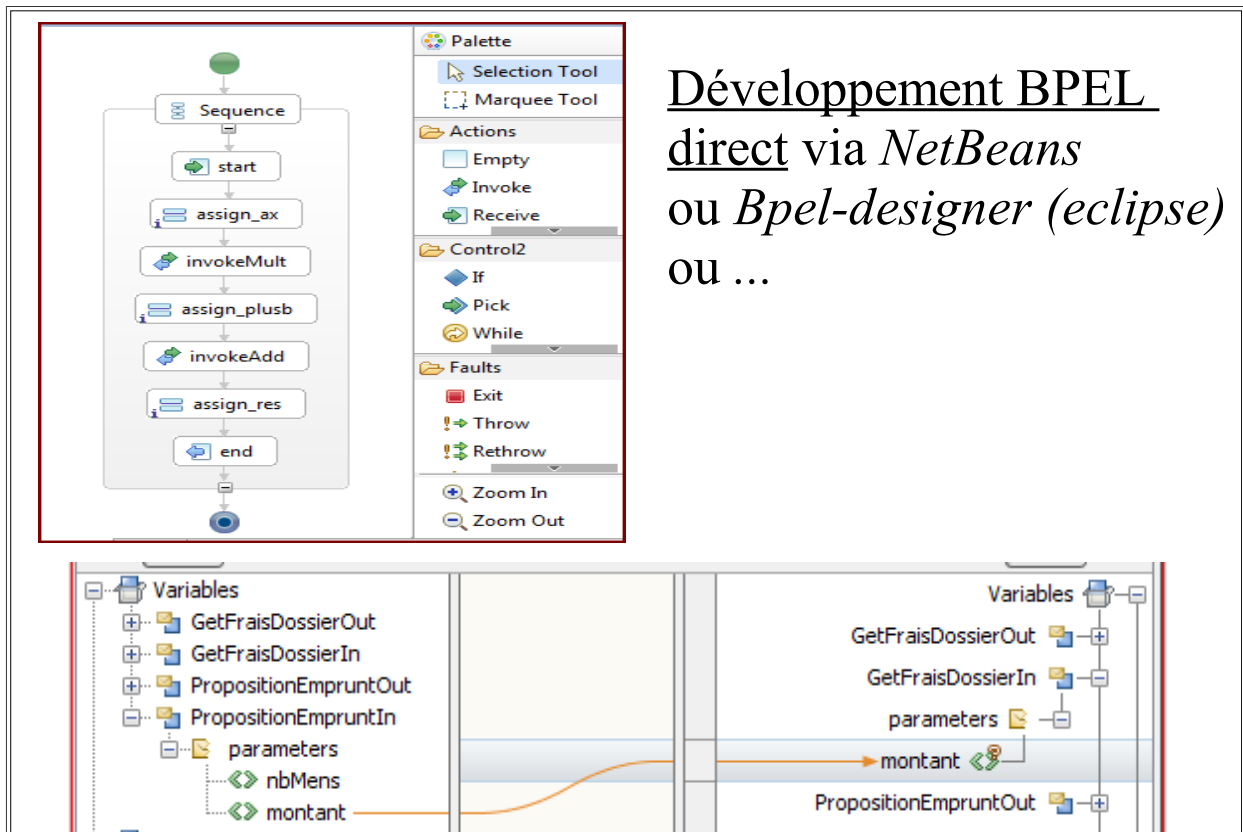
*réception d'une
requête et affectation
du message
dans une variable.*

*invocation d'une
opération
sur un service Web
partenaire*

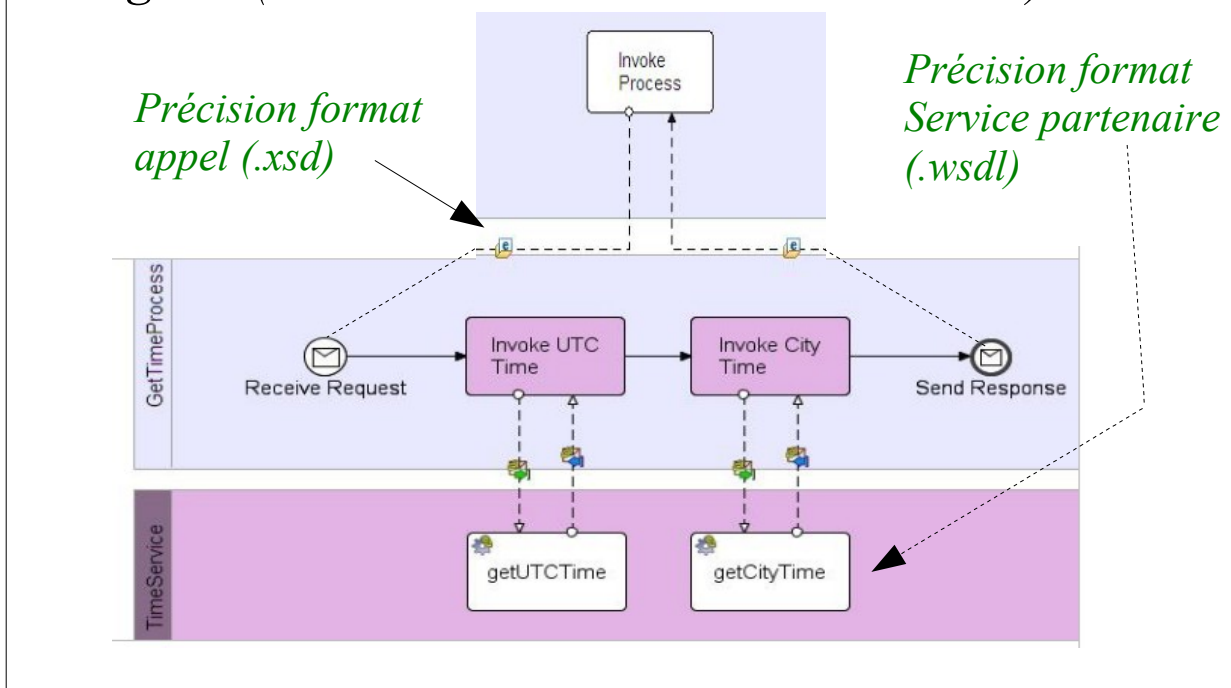
BPEL – PartnerLinks & PortType:



14.2. Vues sur le développement de processus BPEL

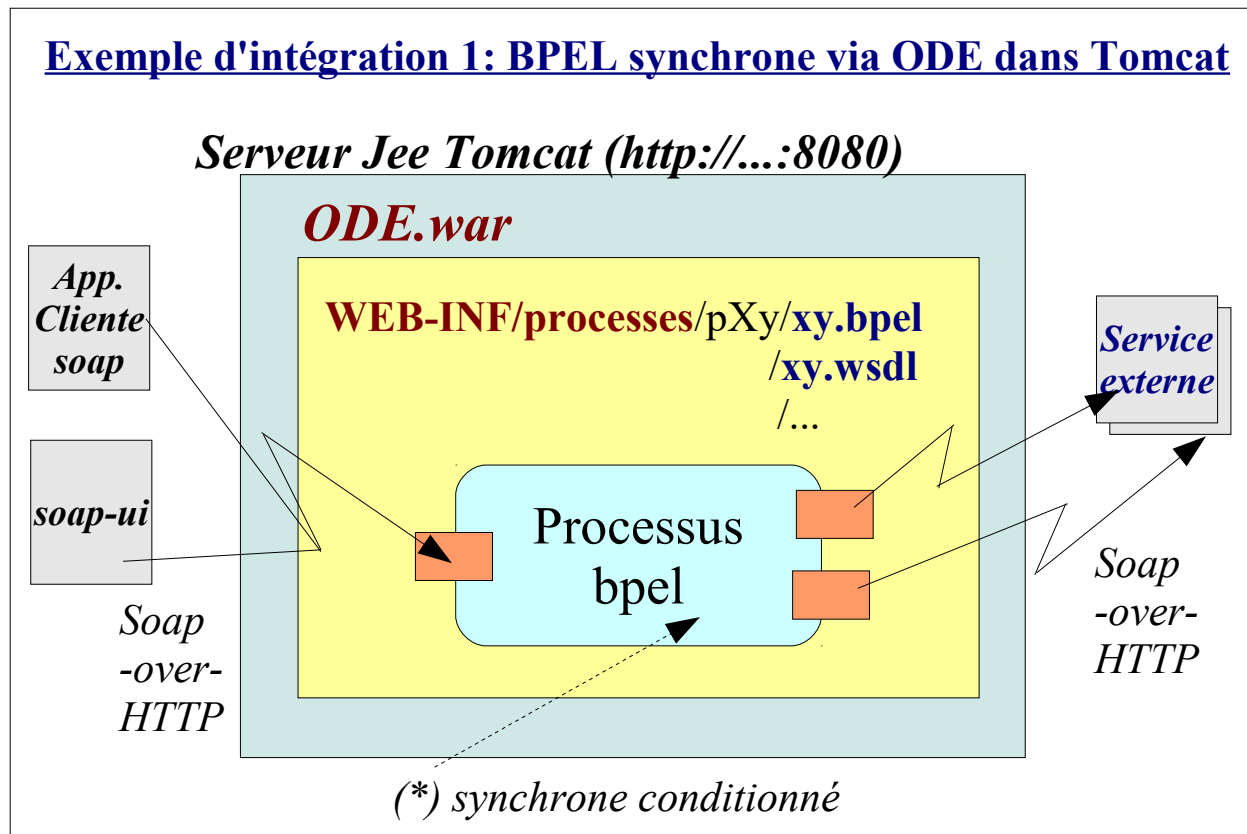


Développement BPEL indirect depuis *Intalio BPMNs Designer* (*modélisation BPMN → code BPEL*)



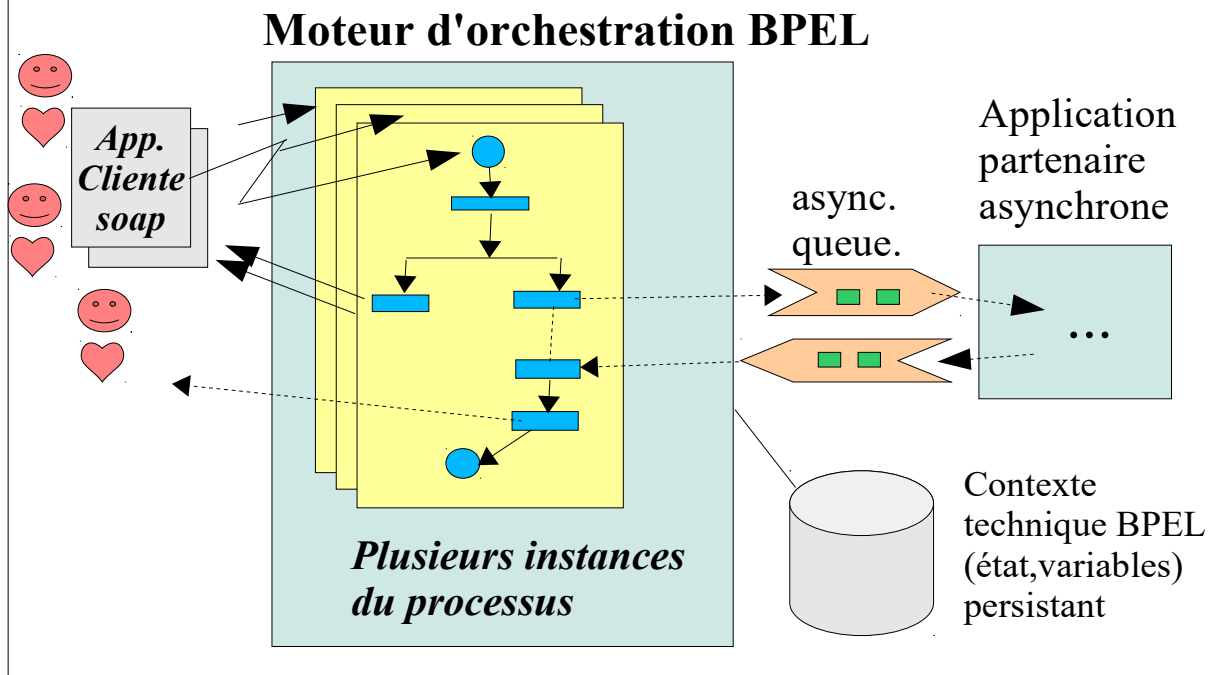
14.3. BPEL en mode synchrone et exemple d'intégration

Exemple d'intégration 1: BPEL synchrone via ODE dans Tomcat



15. BPEL en mode asynchrone

BPEL (dans tout son potentiel) en mode asynchrone



BPEL - mode asynchrone et corrélations

```
<invoke operation="xxx" ...>
  <correlations>
    <correlation set="xxxCorSet"
      initiate="yes" />
  </correlations> ... </invoke>
```

one-way

request(s)

```
<pick> <onMessage
  operation="xxxRespCallBack">
  <correlations>
    <correlation set="xxxCorSet"
      initiate="no" />
  </correlations> ... </onMessage>
```

one-way

response(s)

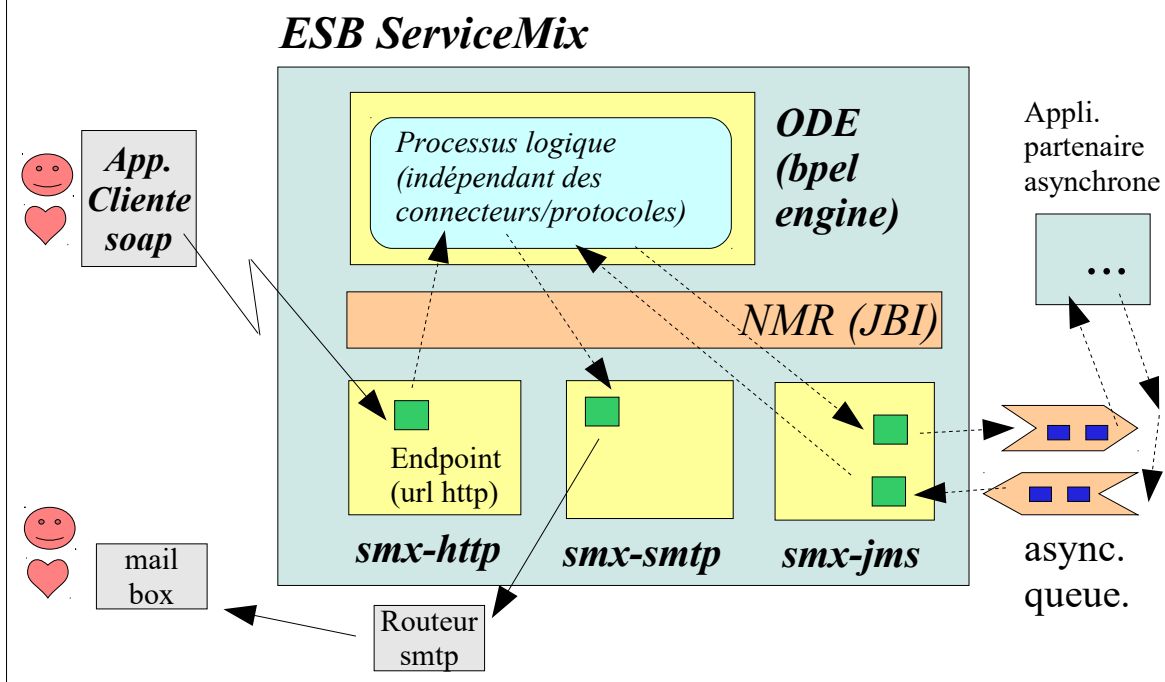
BPEL – *pick* (receive (if correlation) ... or)

```

<pick>
  <onMessage operation="xxxResponse"
    ....> <correlations> ...</correlations>
    <!-- gérer la réponse positive (acceptation) -->
  </onMessage>
  <onMessage operation="xxxReject"
    ....> <correlations> ...</...>
    <!-- gérer le refus -->
  </onMessage>
  ...
</pick>
    
```

callback

Ex. d'intégration 2: BPEL asynchrone via ODE dans ServiceMix

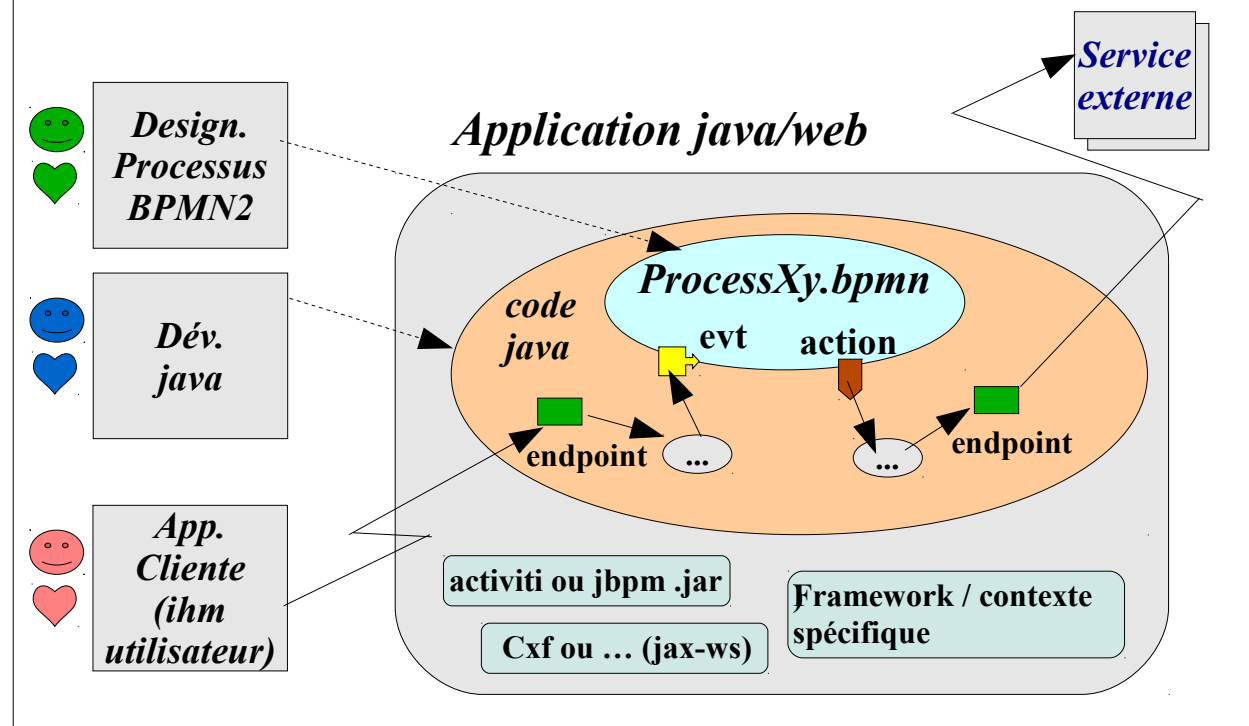


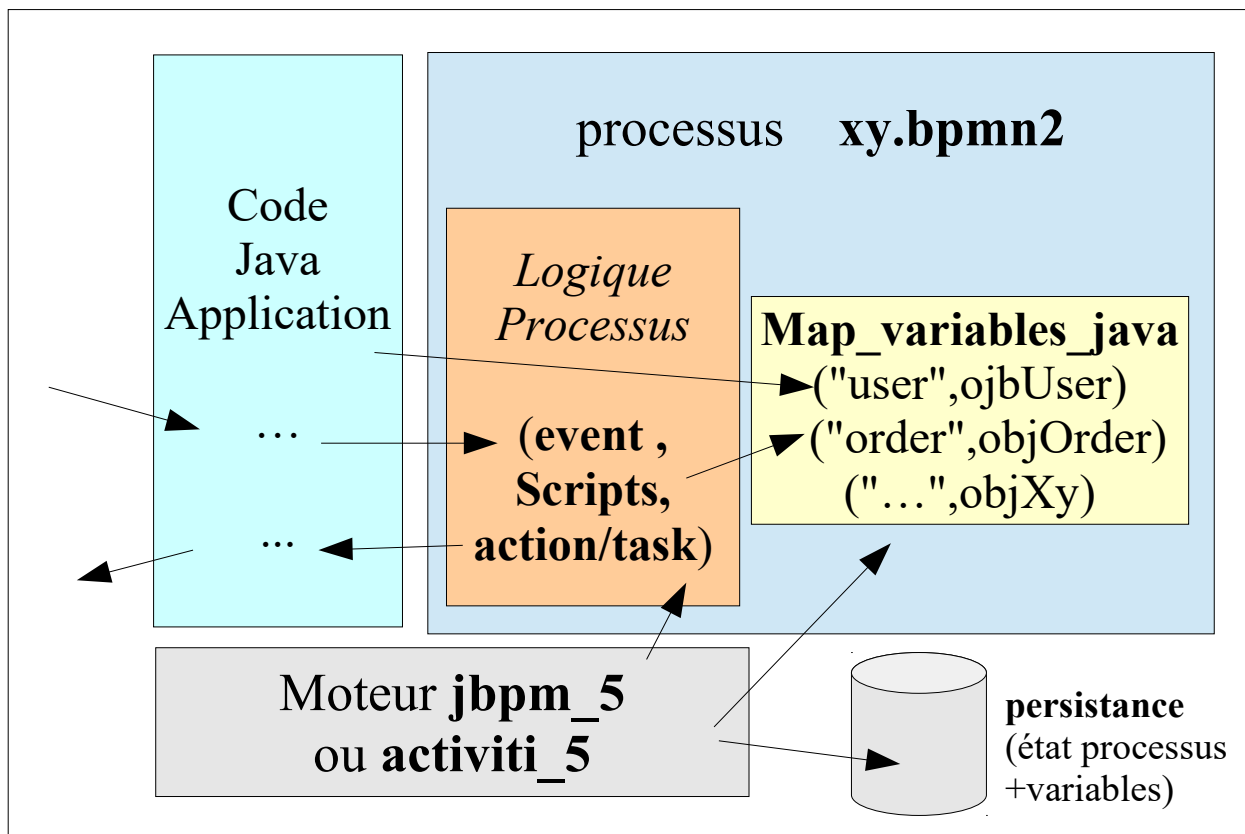
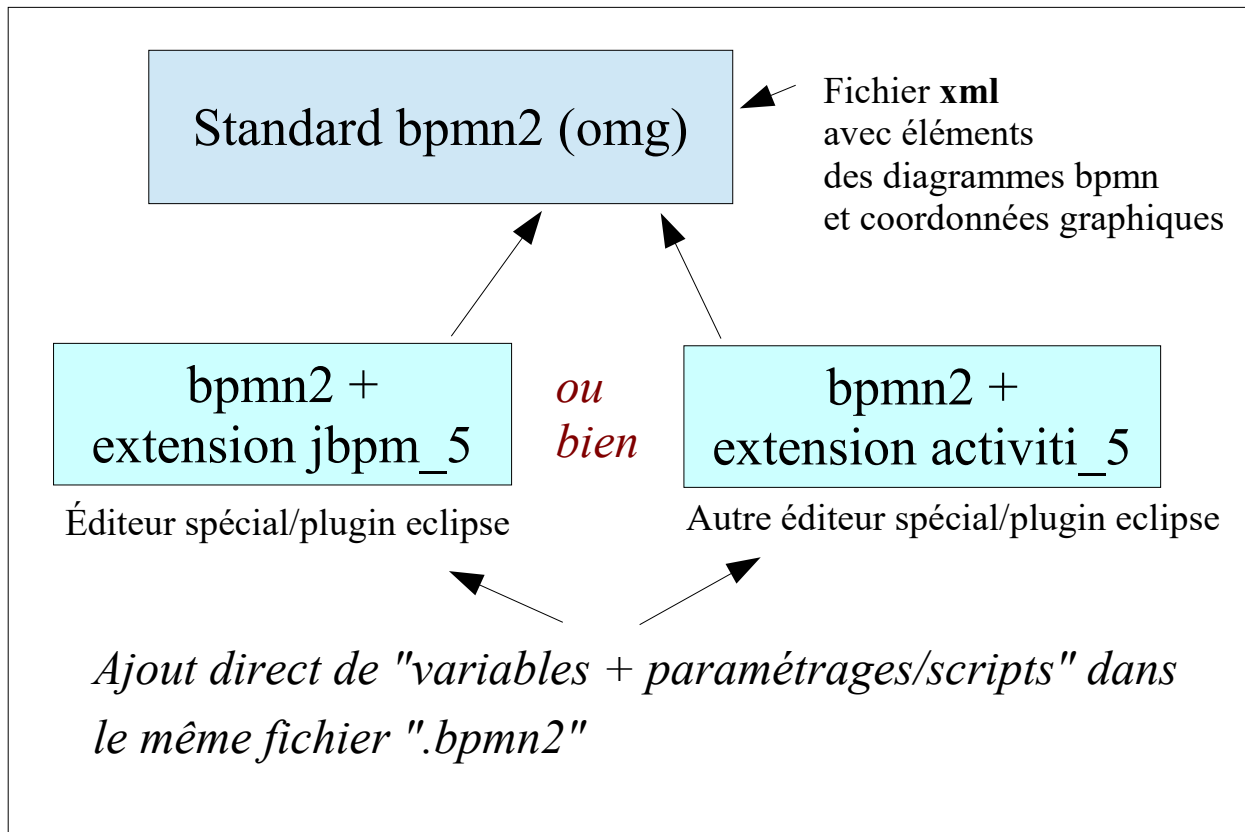
16. jBPM et activiti (présentation)

Eventuelle alternative (java/BPM) à BPEL (xml)

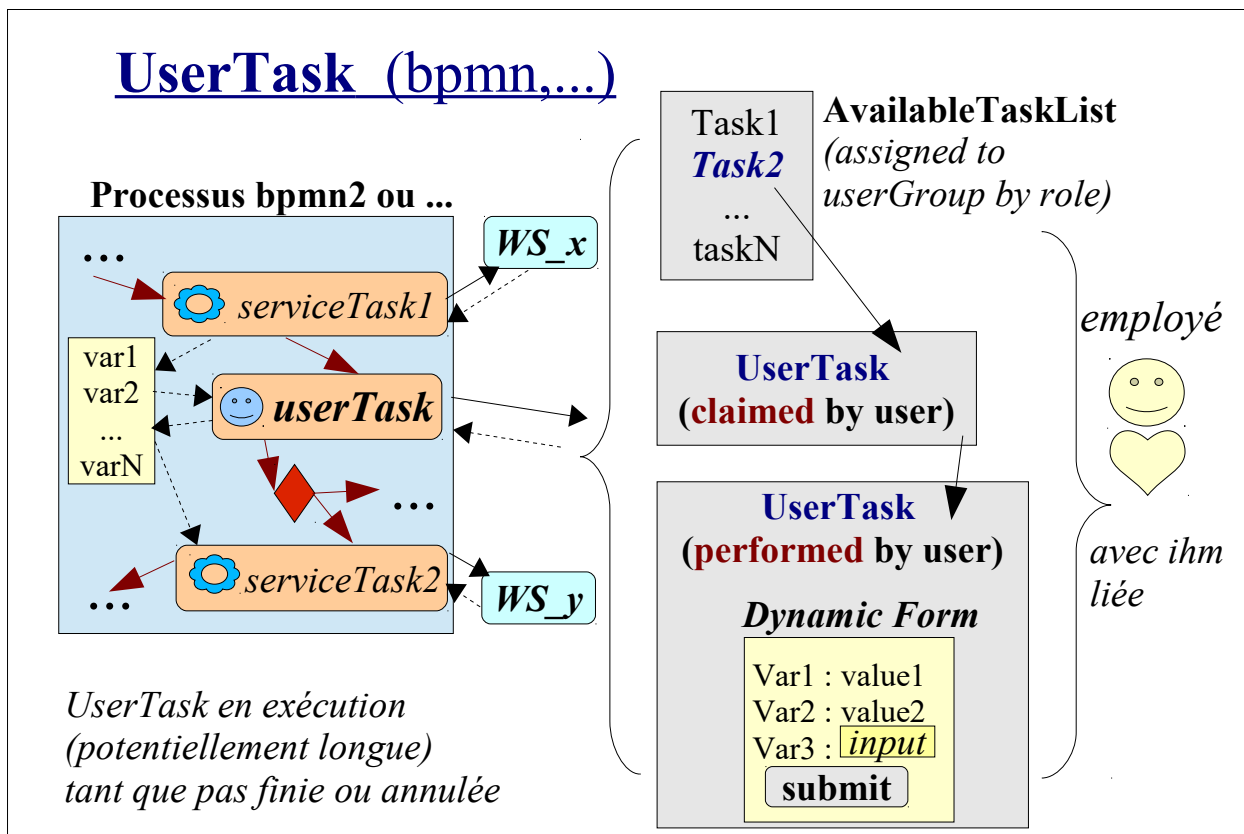
- * **jBPM_5** et **activiti_5** sont des **moteurs de workflow** à intégrer des applications **java** (sous forme de ".jar").
- * *Ces deux technologies (très proches) s'appuient directement sur le standard **BPMN2**.*
- * **jBPM_5** est en fait un des constituants de **Drools 5** et s'intègre facilement dans **Jboss 7.1** ou **Jboss EAP 6**
- * **activiti_5** est plus léger et peut facilement s'intégrer dans le framework **Spring** (et donc dans **tomcat 7**)
- * Bien que nécessitant pas mal de code java (variables, invocations,...), jBPM5 et activiti peuvent être utilisés pour effectuer de l'orchestration de services.

Principe de fonctionnement de (java/BPM)





17. UserTask



Appelée "UserTask" ou "HumanTask", ce type de tâche correspond à :

- Une tâche réalisée par une personne humaine (ex : employé) à l'aide d'un terminal informatique.
- Une tâche qui ne sera considérée comme "terminée et réussie" que lorsqu'un des employés aura effectué la tâche (ce qui peut prendre plusieurs minutes/heures/jours)
- Une tâche dont les entrées/sorties du formulaire dynamique sont liées (par paramétrages) à certaines variables du processus bpmn2 ou bpel .

18. Considérations diverses sur l'architecture SOA :

18.1. Standards de sécurité pour le "SOA XML/SOAP"

- "Basic Http Auth" + HTTPS/SSL : pour **authentification basique**
- **WS-Security** (normes liée aux WS SOAP) : pour **authentification forte**
- **SAML** (Security assertion markup language) : SSO basé sur XML (pour WS SOAP)

NB : Pour le SOA "HTTP/REST" --> jetons JWT et OAuth2 .

18.2. IHM web "riches" : RIA/RDA

RIA = "**R**ich **I**nternet **A**pplication" = Application WEB/WEB2/WEB3/WEB4 avec fonctionnalités graphiques évoluées (javascript , HTML5 + Ajax + ... + CSS3 , ...).

souvent opposée/comparée à un "Client lourd" = "Application complète non web" à télécharger et exécuter sur le poste client (ex : traitement de texte , Editeur de code , ...). Ce type d'application est généralement constitué d'une fenêtre principale , de menu , toolbar , sous fenêtres et peut éventuellement communiquer (via RPC/Web services) avec d'autres applications . Lourdeur moyenne du code = 100 Mo .

NB : Un client lourd peut éventuellement être basé sur "Eclipse RCP" ou un équivalent.

Variante RDA : **R**ich **D**esktop **A**pplication (un peu comme "RIA" mais *s'exécutant au dehors du navigateur*) .

Avantages RIA :

- rien à télécharger , mise à jour transparente pour le client
- graphiquement quasi aussi évolué qu'un client lourd (avec CSS3, HTML5 , ...)
- **bien adapté au mode "Saas" (Software as a service) / application en location**

Inconvénients RIA :

- temps de réaction pouvant dépendre de l'état du réseau internet (bande passante, ...) et de la charge du serveur.

Technos pour appli "RDA" : "**Java Web Start**" , "**Adode AIR** : runtime javascript et flex", ...)

Anciennes technos pour RIA : **GWT , RichFaces/Primefaces , Dojo , ... , JavaFx**

(java + javascript ou autres)

Nouvelles technos pour RIA : **Angular , React, VueJs,** (javascript seulement)

18.3. Mashups

Un WS REST (ou un WS-SOAP adapté en REST) peut très bien renvoyer une réponse "HTML+CSS" (et pas obligatoirement XML ou JSON) .

Et dans ce cas un site web peut très bien **incorporer ce web service directement dans une zone d'une page HTML** .

Exemples concrets : "**Google Calendar / agenda**" , "**Google Maps/Geolocation**" ,

Notion de "Mashup web" : Une **application composite** (ou **mashup** ou encore **mash-up**) est une application qui combine du contenu ou du service provenant de plusieurs applications plus ou moins hétérogènes.

18.4. Orchestration/intégration des applications "SaaS" ?

Etant donné qu'une application SaaS est avant tout prévue pour un "utilisateur final" (potentiellement un employé de l'entreprise cliente), la saisie / consultation des données s'effectuera généralement sous forme d'interactions web (au sein d'un navigateur).

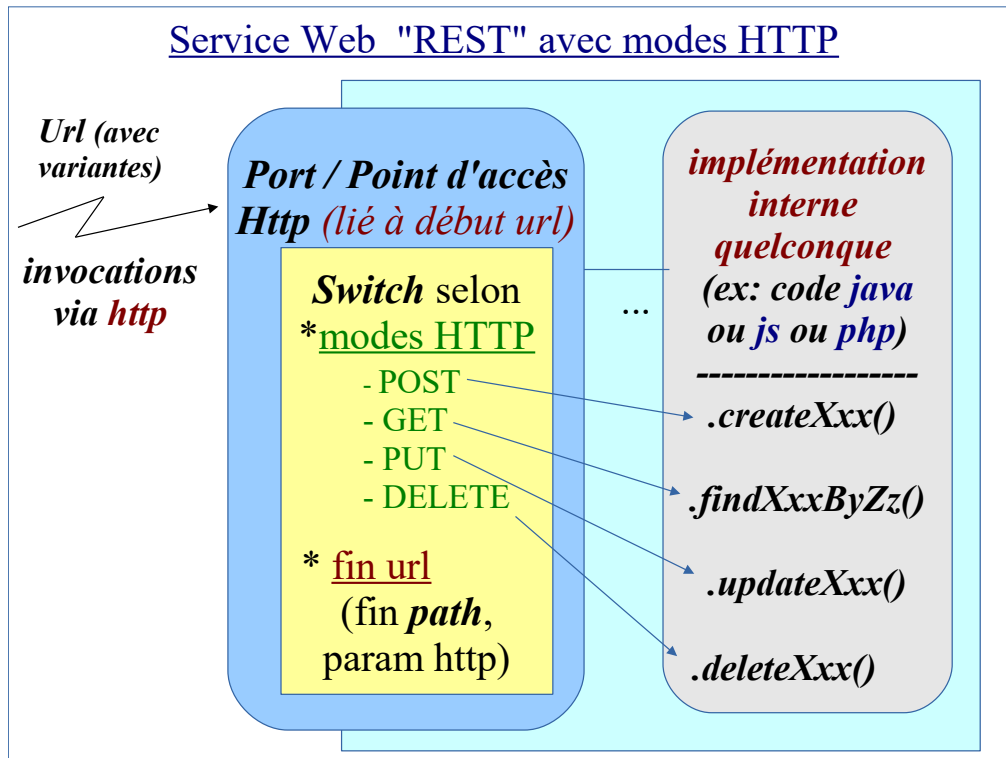
Pour ne pas obliger à effectuer des "re-saisies" inutiles, il faudra idéalement choisir une solution "SaaS" qui offre un minimum de fonctionnalités annexes tel que :

- **import/export de données** (idéalement filtrées/filtrables) **dans un format ouvert (XML, JSON, CSV, ...)**
- **éventuelle API "SOAP" ou "REST" permettant d'extraire (ou alimenter) simplement certaines données de l'application louée.**

Sans cela, l'intégration ou l'orchestration d'une application SaaS serait très laborieuse (quasi impossible).

IV - L'approche REST

1. Caractéristiques clefs web-services "REST" / "HTTP"



Points clefs des Web services "REST"

Retournant des données dans un format quelconque ("**XML**" , "**JSON**" et éventuellement "**txt**" ou "**html**") les web-services "**REST**" offrent des **résultats qui nécessitent généralement peu de re-traitements** pour être mis en forme au sein d'une IHM web.

Le format "**au cas par cas**" des données retournées par les services REST permet peu d'automatisme(s) sur les niveaux intermédiaires.

Souvent associés au format "**JSON**" les web-services "**REST**" **conviennent parfaitement** à des appels (ou implémentations) au sein du **langage javascript** .

La **relative simplicité des URLs d'invocation des services "REST"** permet des **appels plus immédiats** (un simple *href="..."* suffit en mode **GET** pour les recherches de données) .

La **compacité/simplicité des messages "JSON"** souvent associés à "**REST**" permet d'obtenir d'assez bonnes performances .

2. Web Services "R.E.S.T."

REST = style d'architecture (conventions)

REST est l'acronyme de **R**epresentational **S**tate **T**ransfert.

C'est un **style d'architecture** qui a été décrit par *Roy Thomas Fielding* dans sa thèse «*Architectural Styles and the Design of Network-based Software Architectures*».

L'information de base, dans une architecture REST, est appelée **ressource**.

Toute information (à sémantique stable) qui peut être nommée est une ressource: un article, une photo, une personne, un service ou n'importe quel concept.

Une ressource est identifiée par un **identificateur de ressource**. Sur le web ces identificateurs sont les **URI** (Uniform Resource Identifier).

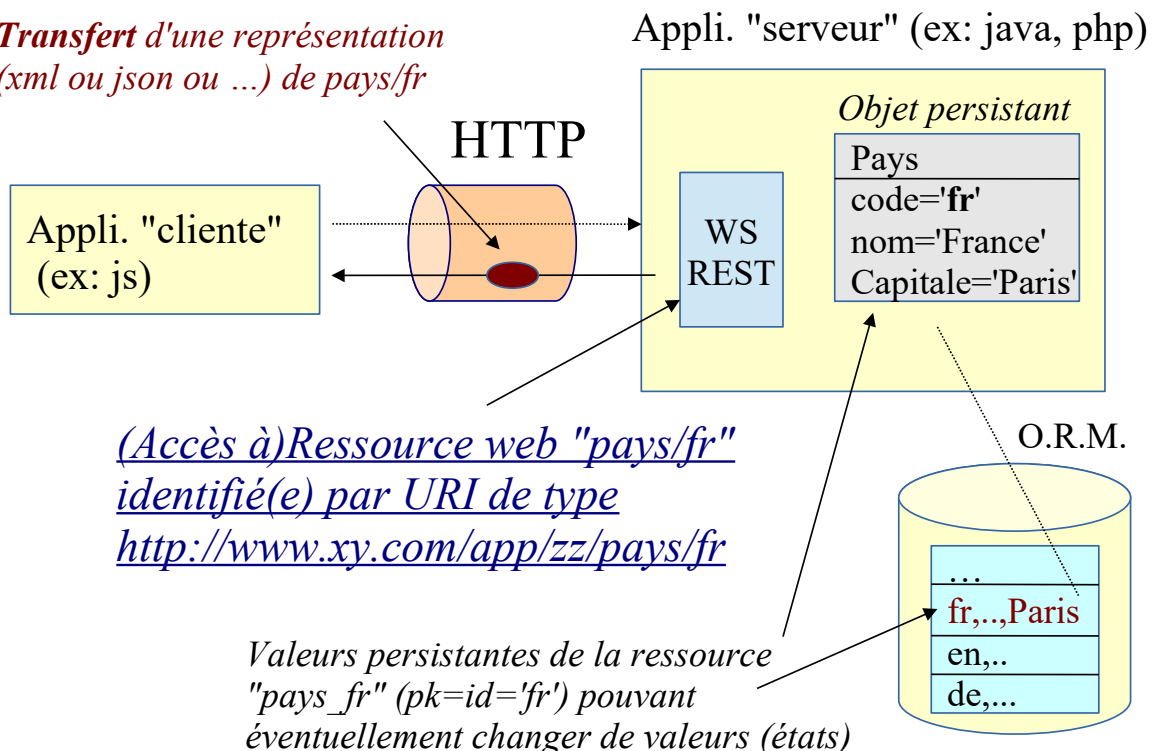
NB: dans la plupart des cas, une ressource REST correspond indirectement à un enregistrement en base (avec la *clef primaire* comme partie finale de l'uri "identifiant").

Les composants de l'architecture REST manipulent ces ressources en **transférant à travers le réseau** (via HTTP) des **représentations de ces ressources**.

Sur le web, on trouve aujourd'hui le plus souvent des représentations au format **HTML, XML ou JSON**.

REST : transferts de représentations de ressources

*Transfert d'une représentation
(xml ou json ou ...) de pays/fr*



REST et principaux formats (xml,json)

Une invocation d'URL de service REST peut être accompagnée de données (en entrée ou en sortie) pouvant prendre des formats quelconques :

text/plain , text/html , application/xml , application/json , ...

Dans le cas d'une lecture/recherche d'informations , le format du résultat retourné pourra (selon les cas) être :

- **imposé (en dur) par le code du service REST .**
- **au choix (xml , json) et précisé par une partie de l'url**
- **au choix (xml , json) et précisé par le champ "Accept :" de l'entête HTTP de la requête. (exemple: Accept: application/json) .**

Dans tous les cas, la réponse HTTP devra avoir son format précisé via le champ habituel **Content-Type: application/json** de l'entête.

Format JSON (JSON = *JavaScript Object Notation*)

Les 2 principales caractéristiques de JSON sont :

- Le principe de clé / valeur (map)
- L'organisation des données sous forme de tableau

Les types de données valables sont :

- tableau
- objet
- chaîne de caractères
- valeur numérique (entier, double)
- booléen (true/false)
- null

```
[
  {
    "nom": "article a",
    "prix": 3.05,
    "disponible": false,
    "descriptif": "article1"
  },
  {
    "nom": "article b",
    "prix": 13.05,
    "disponible": true,
    "descriptif": null
  }
]
```

une liste d'articles

une personne

```
{
  "nom": "xxxx",
  "prenom": "yyyy",
  "age": 25
}
```

REST et méthodes HTTP (verbes)

Les **méthodes HTTP** sont utilisées pour indiquer la **sémantique des actions demandées** :

- **GET** : **lecture/recherche** d'information
- **POST** : **envoi** d'information
- **PUT** : **mise à jour** d'information
- **DELETE** : **suppression** d'information

Par exemple, pour récupérer la liste des adhérents d'un club, on peut effectuer une requête de type **GET** vers la ressource <http://monsite.com/adherents>

Pour obtenir que les adhérents ayant plus de 20 ans, la requête devient <http://monsite.com/adherents?ageMinimum=20>

Pour supprimer numéro 4, on peut employer une requête de type **DELETE** telle que <http://monsite.com/adherents/4>

Pour envoyer des informations, on utilise **POST** ou **PUT** en passant les informations dans le corps (invisible) du message HTTP avec comme URL celle de la ressource web que l'on veut créer ou mettre à jour.

Exemple concret de service REST : "Elevation API"

L'entreprise "**Google**" fournit gratuitement certains services WEB de type REST. "**Elevation API**" est un service REST de Google qui renvoie l'altitude d'un point de la planète selon ses coordonnées (latitude, longitude) .

La documentation complète se trouve au bout de l'URL suivante :

<https://developers.google.com/maps/documentation/elevation/?hl=fr>

Sachant que les coordonnées du Mont blanc sont :

Lat/Lon : 45.8325 N / 6.86417 E (GPS : 32T 334120 5077656)

Les invocations suivantes (du service web rest "api/elevation")

<http://maps.googleapis.com/maps/api/elevation/json?locations=45.8325,6.86417>

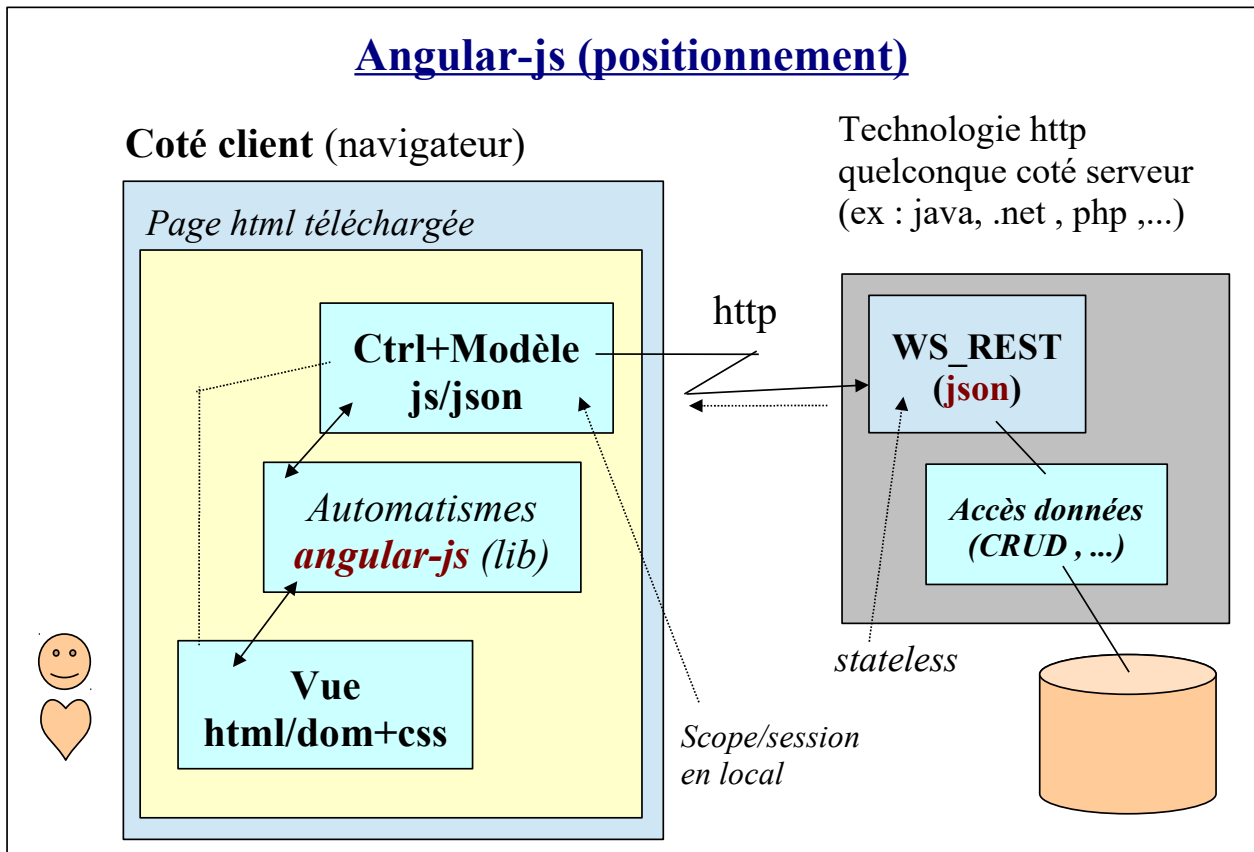
<http://maps.googleapis.com/maps/api/elevation/xml?locations=45.8325,6.86417>

donne les résultats suivants "json" ou "xml":

```
{ "results" : [
  {
    "elevation" : 4766.466796875,
    "location" : {
      "lat" : 45.8325,
      "lng" : 6.86417
    },
    "resolution" : 152.7032318115234
  }
], "status" : "OK"
}
```

```
?xml version="1.0" encoding="UTF-8"?>
<ElevationResponse>
  <status>OK</status>
  <result>
    <location>
      <lat>45.8325000</lat>
      <lng>6.8641700</lng>
    </location>
    <elevation>4766.4667969</elevation>
    <resolution>152.7032318</resolution>
  </result>
</ElevationResponse>
```

Angular-js (positionnement)



Conventions "angular-js" sur URL des ressources REST

Type requêtes	HTTP Method	URL ressource(s) distante(s)	Request body	Réponse JSON
Recherche multiple	GET	.../products .../products? crit1=v1&crit2=v2	vide	Liste/tableau d'objets
Recherche par id	GET	.../products/ idRes (avec idRes=1,...)	vide	Objet JSON
Ajout (seul)	POST		Objet JSON	Objet JSON avec id quelquefois calculé (incr)
Mise à jour (seule)	PUT	.../products/ idRes	Objet JSON	Statut ou ...
SaveOr Update	POST	.../products/ idRes	Objet JSON	Objet JSON modifié (id)
suppression	DELETE	.../products/ idRes	vide	Statut ou ...
Autres/product-action/opXy/....

3. Quelques API et technologies "web services REST"

3.1. Vue d'ensemble sur technologies "REST" :

Principales API associées au web-services REST

Langage **Javascript** ---> **nodeJs** + routes **express** , ...

Langage **PHP** --> routes \$app->get("...",...) , \$app->post("...",...) avec "Silex" ou autre et json_encode(...)

Langage **C#** ---> ASP.NET , classe de type "ApiController" , ...

Langage **Java** --->

- API officielle(standard) "**JAX-RS**" (v1 : JEE6 , v2 : JEE7)
(@Produces , @GET, @POST, ... , @Path)
ou bien
- API efficace "**Spring-mvc**" (pour eco-système "Spring")
(@RestController , @RequestMapping() ,)

Autres Langages --> n'importe quelle combinaison de technologies permettant de générer une réponse HTTP au format prédominant JSON .

Compléments intéressants pour **web-services REST**

Pour la sécurité :

- * preuve d'authentification via jeton "**JWT**" (JSON Web Token)
- * **HTTPS** à utiliser et configurer en prod.
- * **OAuth** (Open Authentication : délégation d'authentification vers serveur (ou partie de serveur) spécialisé .

Pour la description/documentation et les tests:

- * **Swagger2** ou RAML ou
- * éventuel diagramme UML , ...
- * **Postman** , Tests unitaires automatisés (js, java)

3.2. W.S. REST en java avec JAX-RS ou Spring-MVC

Exemple de WS-REST codé avec "JAX-RS"

```

@Path("devises") @Produces("application/json")
public class DeviseListCtrl {
    @Inject
    private GestionDevises serviceDevises; //internal (EJB3 ou ...) local service

    @Path(value="/") @GET
    List<Devise> getAllDevises() {
        return serviceDevises.getListeDevises();
    }

    @Path("/{name}") @GET
    Devise getDeviseByName(@PathParam("name") String deviseName) {
        return serviceDevises.getDeviseByName( deviseName);
    }
}

```

→ à intégrer dans une appli web (.war)

→ nécessite une très légère configuration annexe :

```

//pour url en http://localhost:8080/myWebApp/services/rest + @Path() java
@ApplicationPath("/services/rest")
public class MyRestApplicationConfig extends Application { ... }

```

Alternative "Spring-MVC" (vis à vis de JAX-RS)

Dans le monde **java** , il est possible de programmer un web service "REST" avec **Spring-MVC** à la place de **JAX-RS**.

Inconvénients de Spring-MVC (pour WS REST):

- * ***ce n'est pas le standard officiel** , c'est du **spécifique "Spring"***
... sachant que le standard s'est maintenant amélioré en version 2 ...
- * la technologie concurrente JAX-RS s'intègre pas trop mal dans Spring (via jersey ou cxf).

Avantages de Spring-MVC (pour WS REST):

- * **l'intégration au sein d'un projet "Spring" est plus aisée (configuration plus simple , moins de librairies ".jar" nécessaires , ...)**
- * **facilement combinable avec d'autres extensions de Spring (spring-security , spring-boot , ...)**
- * logique "MVC" pouvant être utile si l'on retourne des portions d'HTML.

Au final, si projet JEE/CDI → standard JAX-RS
si projet Spring → Spring-MVC ou bien JAX-RS

Exemple de WS-REST codé avec "Spring-MVC"

```

@RestController
@RequestMapping(value="/devises" , headers="Accept=application/json")
public class DeviseListCtrl {
    @Autowired
    private GestionDevises serviceDevises; //internal Spring local service

    @RequestMapping(value="/" , method=RequestMethod.GET)
    @ResponseBody
    List<Devise> getAllDevises() {
        return serviceDevises.getListeDevises();
    }

    @RequestMapping(value="/{name}" , method=RequestMethod.GET)
    @ResponseBody
    Devise getDeviseByName(@PathVariable("name") String deviseName) {
        return serviceDevises.getDeviseByName( deviseName);
    }
}

```

- même logique de paramétrage que JAX-RS (HTTP method , path , ...)
- annotations différentes de celles de JAX-RS (avec davantage d'attributs)
- quasiment aucune configuration annexe nécessaire (avec java-config , spring-boot)

3.3. W.S. REST en javascript avec nodeJs+Express

....

4. Appels de WS REST (HTTP) depuis js/ajax

Avec ajax , ça va briller!

4.1. Cadre des appels

Lorsqu'une requête http est initiée depuis du code javascript s'exécutant dans le contexte d'une page html , on dit que l'on effectue un appel "ajax" .

Le sigle Ajax correspondant à peu près à "asynchronous javascript activation framework" indique :

- le déclenchement non bloquant d'une requête http
- l'enregistrement d'une fonction "callback" qui sera automatiquement appelée en différé lorsque la réponse reviendra

NB : l'appel non bloquant peut être considéré comme asynchrone mais le protocole HTTP est un protocole de transport synchrone (avec timeout si la réponse ne revient pas dans un délai raisonnable).

Techniquement, un appel ajax s'effectue en s'appuyant sur un objet technique "XMLHttpRequest" fourni par tous les navigateurs pas trop anciens .

La partie Xml de XMLHttpRequest tient au fait qu'historiquement les premiers webServices normalisés (SOAP) étaient au format Xml. Bien que le terme "XMLHttpRequest" n'ait pas été changé pour des raisons de compatibilité ascendante du code javascript, il est possible de déclencher n'importe quelle requête HTTP depuis XHR , y compris des requêtes au format "JSON" .

Pour simplifier la syntaxe d'un appel ajax on peut éventuellement s'appuyer sur des librairies "javascript" complémentaires ("jquery" , "fetch" , "RxJs" , ...) .

Le principe des appels "ajax" sert essentiellement à déclencher une requête HTTP suite à un événement utilisateur en vue d'obtenir des données permettant de réactualiser une partie de la page HTML courante . La page courante n'est pas entièrement remplacée par une autre. Seule une sous partie de celle ci est ajustée par code js/DOM (Document Object Model).

Certaines applications , dites "SPA: Single Page Application" sont entièrement bâties sur ce principe . Au lieu de switcher de pages html on switch de sous pages (<div ...>....</div>) .

4.2. XHR (XmlHttpRequest) dans tout navigateur récent

Exemple :

```
function makeAjaxRequest(callback) {
    var xhr = new XMLHttpRequest();

    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4 && (xhr.status == 200 || xhr.status == 0)) {
            callback(xhr.responseText);
        }
    };

    xhr.open("GET", "handlingData.php", true);
    xhr.send(null);
}

function readData(sData) {
    if (sData!=null) {
        alert("good response");
        // + display data with DOM
    } else {
        alert("empty response");
    }
}

makeAjaxRequest(readData);
```

variations en mode "POST" :

```
xhr.open("POST", "handlingData.php", true);
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");

xhr.send("param1=xx&param2=yy");
```

```
xhr.open("POST", "/json-handler");
xhr.setRequestHeader("Content-Type", "application/json");
xhr.send(JSON.stringify({prenom:"Jean", nom:"Bon"}));
```


4.3. Appel ajax via jQuery

La syntaxe des appels "ajax" est un peu plus explicite en s'appuyant sur la bibliothèque "jquery".

Exemple :

```
<html>
<head>
  <meta charset="ISO-8859-1">
  <title>browse-spectacles</title>
  <script src="lib/jquery-3.3.1.min.js"></script>
  <script src="js/my-jq-ajax-util.js"></script>
</script>
$(function() {

  //appel ajax pour récupérer la liste des catégories et remplir le <select>
  $.ajax({
    type: "GET",
    url: "spectacle-api/public/spectacle/allCategories",
    contentType : "application/json",
    success: function (data,status,xhr) {
      if (data) {
        var categoryList = data;
        for(categoryIndex in categoryList){
          var category=categoryList[categoryIndex];
          $('#selectCategory').append('<option value="'+ category.id +"'>'+
            category.id + ' (' + category.title + ')</option>');
        }
        //$("#spanMsg").html(JSON.stringify(data));
      }
    },
    error: function( jqXHR, textStatus, errorThrown ){
      $("#spanMsg").html( xhrStatusToErrorMessage(jqXHR) );
    }
  }); //end $.ajax
});

</script>
</head>
<body>
  <h3> BROWSE Spectacles </h3>
  categorie : <select id="selectCategory"> </select><br/>
  ... <span id="spanMsg"></span> <br/>...
</body>
</html>
```

fonctions utilitaires dans *js/my-jq-ajax-util.js*

```
function setSecurityTokenForAjax(){

  var authToken = sessionStorage.getItem("authToken");
  //localStorage.getItem("authToken");

  $(document).ajaxSend(function(e, xhr, options) {
```

```

        //retransmission du jeton d'authentification dans l'entête http de la requete ajax
        xhr.setRequestHeader('Authorization','Bearer '+ authToken);
    });
}

function xhrStatusToErrorMessage(jqXHR){
    var errMsg = "ajax error";//by default
    var detailsMsg=""; //by default
    console.log("jqXHR.status="+jqXHR.status);
    switch(jqXHR.status){
        case 400 :
            errMsg = "Server understood the request, but request content was invalid.";
            if(jqXHR.responseText!=null)
                detailsMsg = jqXHR.responseText;
            break;
        case 401 :
            errMsg = "Unauthorized access (401)"; break;
        case 403 :
            errMsg = "Forbidden resource can't be accessed (403)"; break;
        case 404 :
            errMsg = "resource not found (404)"; break;
        case 500 :
            errMsg = "Internal server error (500)"; break;
        case 503 :
            errMsg = "Service unavailable (503)"; break;
    }
    return errMsg+" "+detailsMsg;
}

```

Variation en mode "POST" et "[application/x-www-form-urlencoded](#)"

```

var dataJsObject = { prenom : "jean", nom : "Bon", taille: 175 } ;
$.ajax({
    type: "POST",
    url: "./my-api/person",
    contentType : "application/x-www-form-urlencoded; charset=utf-8",
    data: $.param(dataJsObject),
    dataType : 'json_or_text_or_...',
    beforeSend: function (xhr) {
        xhr.setRequestHeader ("Authorization",
                                "Basic " + btoa("usernameXy" + ":" + "passwordXy"));
    },
    success: ... , error: ....
}); //end $.ajax

```

Variation en mode "POST" et "JSON" in/out :

```
//setSecurityTokenForAjax();//js/my-jq-ajax-util.js
$.ajax({
  type: "POST",
  url: "spectacle-api/spectacle",
  data : JSON.stringify(spectacleAdditionJsObject),
  dataType : "json",
  contentType : "application/json",
  success: function (data,status,xhr) {
    if (data) {
      $("#spanMsg").html(JSON.stringify(data));
    }
  },
  error: function( jqXHR, textStatus, errorThrown ){
    $("#spanMsg").html( xhrStatusToErrorMessage(jqXHR) );
  }
});//end $.ajax
```

4.4. Api fetch

Api récente (syntaxe concise basé sur enchaînement asynchrone et "**Promise**") mais pas encore supporté par tous les navigateurs.

Exemple :

```
fetch('./api/some.json')
  .then(
    function(response) {
      if (response.status !== 200) {
        console.log('Problem. Status Code: ' + response.status);
        return;
      }
      // Examine the text in the response :
      response.json().then(function(data) {
        console.log(data);
      });
    }
  )
  .catch(function(err) {
    console.log('Fetch Error :-S', err);
  });
```

4.5. Appel ajax via RxJs (api réactive)

Le framework "RxJs" lié au concept de "programmation asynchrone réactive" est assez sophistiqué et permet de déclencher une série de traitements d'une façon assez indépendante de la source de données (ex : données statiques , réponse ajax , push web-socket , ...) .

RxJs peut soit être directement utilisé en tant que bibliothèque javascript dans une page HTML , soit être utilisé via "typescript et le framework Angular 2,4,5,6 ou autre) .

Attention à la version utilisée (différences significatives dans la version récente de RxJs accompagnant Angular 6) .

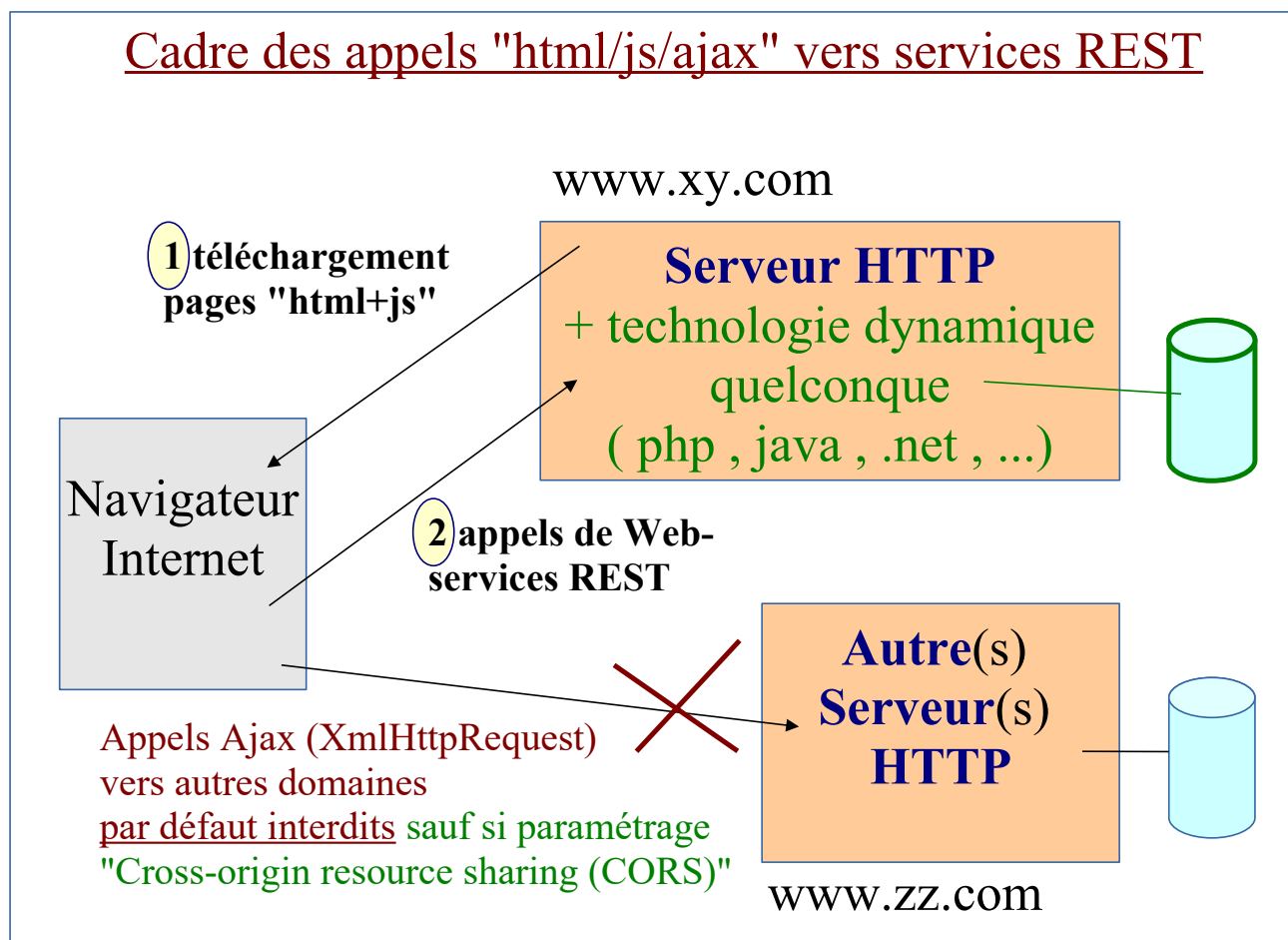
--> une bonne présentation de RxJS est accessible au bout de l'URL suivante

<https://www.julienpradet.fr/tutoriels/introduction-a-rxjs/>

--> exemples d'appel ajax via RxJs et de config proxy dans le support de cours "Angular 4,5,6"

5. Limitations Ajax sans CORS

Cadre des appels "html/js/ajax" vers services REST



6. CORS (Cross Origin Resource Sharing)

CORS=Cross Origin Resource Sharing

CORS est une **norme du W3C** qui précise certains **champs** à placer dans une **entête HTTP** qui serviront à échanger entre le navigateur et le serveur des informations qui serviront à décider si une requête sera ou pas acceptée.

(utile si domaines différents) , dans requête simple ou bien dans pré-échange préliminaire quelquefois déclenché en plus :

Au sein d'une requête "demande autorisation" envoyée du client vers le serveur :

Origin: <http://www.xy.com>

Dans la "réponse à demande d'autorisation" renvoyée par le serveur :

Access-Control-Allow-Origin: <http://www.xy.com>

Ou bien

Access-Control-Allow-Origin: * *(si public)*

→ requête acceptée

*Si absence de "Access-Control-Allow-Origin :" ou bien valeur différente
---> requête refusée*

CORS=Cross Origin Resource Sharing (2)

NB1: toute requête "CORS" valide doit absolument comporter le champ "**Origin** :" dans l'entête http. Ce champ est toujours construit automatiquement par le navigateur et jamais renseigné par programmation javascript.

Ceci ne protège que partiellement l'accès à certains serveurs car un "méchant hacker" utilise un "navigateur trafiqué".

Les mécanismes "CORS" protège un peu le client ordinaire (utilisant un vrai navigateur) que dans la mesure où la page d'origine n'a pas été interceptée ni trafiquée (l'utilisation conjointe de "https" est primordiale) .

NB2 : Dans le cas (très classique/fréquent) , où la requête comporte "**Content-Type: application/json**" (ou **application/xml** ou ...) , la norme "CORS" (considérant la requête comme étant "pas si simple") impose un pré-échange préliminaire appelé "**Preflighted request/response**" .

Paramétrages CORS à effectuer coté serveur

L'application qui coté serveur, fourni quelques Web Services REST , peut (et généralement doit) autoriser les requêtes "Ajax / CORS" issues d'autres domaines ("*" ou "www.xy.com").

Attention: ce n'est pas une "sécurité coté serveur" mais juste **un paramétrage autorisant ou pas à rendre service à d'autres domaines et en devant gérer la charge induite** (*taille du cluster, consommation électrique, ...*) .

// Exemple : CORS enabled with express/node-js :

```
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*"); // "*" ou "xy.com , ..."
  res.header("Access-Control-Allow-Methods",
    "POST, GET, PUT, DELETE, OPTIONS"); //default: GET, ...
  res.header("Access-Control-Allow-Headers",
    "Origin, X-Requested-With, Content-Type, Accept , Authorization");
  next();
});
```

Paramétrages CORS avec CXF et JAX-RS

```
<bean id="corsFilter" class="org.apache.cxf.rs.security.cors.
    CrossOriginResourceSharingFilter">
    <!-- <property name="allowCredentials" value="true"/> -->
</bean>
...
<jaxrs:server id="myRestServices" address="/rest">
    <jaxrs:providers>
        <ref bean='jacksonJsonProvider' />
        <ref bean='corsFilter' />
    </jaxrs:providers>
    <jaxrs:serviceBeans> ...
        <ref bean="serviceClientsRest" />
    </jaxrs:serviceBeans> ...
```

config
spring/cxf

```
@Path("/json/gestionclients")
@Produces("application/json")
@Consumes("application/json")
@CrossOrigin(allowAllOrigins = true)
// ou bien autorisations plus fines
public class ClientRestJsonService {
    ...}
```

code java

Paramétrage "CORS" avec Spring-mvc

```
import org.springframework.web.bind.annotation.CrossOrigin;
...
@RestController
@CrossOrigin(origins = "*")
//@CrossOrigin(origins = { "http://localhost:4200" ,
//                        "http://www.partenaire-particulier.com" })
@RequestMapping(value="/rest/products" , headers="Accept=application/json")
public class ProductCtrl {
    ...
}
```

et/ou implémentation de *WebMvcConfigurerAdapter*
comportant une redéfinition de `public void addCorsMappings(CorsRegistry registry) .`

7. Web services "REST" pour application Spring

Pour développer des Web Services "REST" au sein d'une application Spring , il y a deux possibilités distinctes (à choisir) :

- s'appuyer sur l'API standard **JAX-RS** et choisir une de ses implémentations (**CXF3** ou **Jersey** ou ...)
- s'appuyer sur le framework "**Spring web mvc**" et utiliser **@RestController** .

La version "JAX-RS standard" nécessite pas mal de librairies (jax-rs, jersey ou cxf , jackson et tout un tas de dépendances indirectes) .

La version spécifique spring nécessite un peu moins de librairies (spring-web , spring-mvc , jackson) et s'intègre mieux dans un écosystème spring (spring-security ,) :

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>4.3.2.RELEASE</version>
    <scope>compile</scope>
</dependency>

<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.5.4</version> <!-- to produces json -->
</dependency>

...
```

8. WS REST via Spring MVC et @RestController

Exemple :

DeviseJsonRestCtrl.java

```
package tp.app.zz.web.rest;

import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

...

@RestController
@RequestMapping(value="/rest/devises" , headers="Accept=application/json")
public class DeviseJsonRestCtrl {
```



```

@Autowired //ou @Inject
private GestionDevises gestionDevises;

//URL de déclenchement: webappXy/mvc/rest/devises/
//ou webappXy/mvc/rest/devises/?name=euro
@RequestMapping(value="/", method=RequestMethod.GET)
//@ResponseBody par défaut avec @RestController
List<Devise> getDevisesByCriteria(@RequestParam(value="name",required=false)
                                String nomMonnaie) {
    if(nomMonnaie==null)
        return gestionDevises.getListeDevises();
    else{
        List<Devise> listeDev= new ArrayList<Devise>();
        Devise devise = gestionDevises.getDeviseByName(nomMonnaie);
        if(devise!=null)
            listeDev.add(devise);
        return listeDev;
    }
}

//URL de déclenchement: webappXy/mvc/rest/devises/EUR
@RequestMapping(value="/{codeDevise}", method=RequestMethod.GET)
//@ResponseBody par défaut avec @RestController
Devise getDeviseByName(@PathVariable("codeDevise") String codeDevise) {
    return gestionDevises.getDeviseByPk(codeDevise);
}
}

```

Techniquement possible mais très rare : retour direct d'une simple "String" (text/plain) :

```

//URL : webappXy/mvc/rest/devises/convert?amount=50&src=EUR&target=USD
@RequestMapping(value="/convert", method=RequestMethod.GET ,
                headers="Accept=text/plain")
//@ResponseBody par défaut avec @RestController
String convert(@RequestParam("amount") double amount,
               @RequestParam("src") String src ,
               @RequestParam("target") String target) {
    double sommeConvertie=gestionDevises.convertir(amount, src, target);
    System.out.println("sommeConvertie="+sommeConvertie);
    return String.valueOf(sommeConvertie);
    //ou bien résultat au format ClasseSpecifiqueJava convertie au format json
}

```

Prise en charge des modes "PUT", "POST", "DELETE" :v

NB : il est techniquement possible de convertir explicitement une "Json String" en objet java via l'api "jackson" comme le montre l'exemple suivant :

```

...
import org.springframework.web.bind.annotation.RequestBody;

```

```

import org.springframework.web.bind.annotation.RequestMethod;
import com.fasterxml.jackson.databind.DeserializationFeature;
import com.fasterxml.jackson.databind.ObjectMapper;

@RestController
@RequestMapping(value="/rest/devises" , headers="Accept=application/json")
public class DeviseJsonRestController {
    ...
    @RequestMapping(value="/" , method=RequestMethod.PUT )
    @ResponseBody // ou @ResponseStatus(value = HttpStatus.OK)
    Devise updateDevise(@RequestBody String deviseAsString) {
        Devise devise=null;
        try {
            ObjectMapper jacksonMapper = new ObjectMapper();
            jacksonMapper.configure(
                DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
            devise = jacksonMapper.readValue(deviseAsString,Devise.class);
            System.out.println("devise to update:" + devise);
            gestionDevises.updateDevise(devise);
            return devise;
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
}

```

Ceci dit , Spring-Mvc est capable d'effectuer de lui même automatiquement cette conversion.
L'écriture suivante (plus simple) assure les mêmes fonctionnalités :

```

@RestController
@RequestMapping(value="/rest/devises" , headers="Accept=application/json")
public class DeviseJsonRestController {
    ...
    @RequestMapping(value="/" , method=RequestMethod.PUT )
    Devise updateDevise(@RequestBody Devise devise) {
        System.out.println("devise to update:" + devise);
        gestionDevises.updateDevise(devise);
        return devise;
    }
}

```

NB : dans tous les cas , il sera souvent nécessaire de contrôler le comportement des "sérialisations/dé-sérialisations java <--> json" en incorporant certaines annotations de "jackson" au sein des classes de données (dto / payload) à véhiculer.

A ce sujet , l'annotation **@JsonIgnore** (sémantiquement équivalent à **@XmlTransient**) est assez souvent utile pour limiter la profondeur des données échangées .

Apport important de la version 4 : **ResponseEntity<T>**

Depuis "Spring4" , une méthode d'un web-service REST a plutôt intérêt à retourner une réponse de Type **ResponseEntity<T>** ce qui permet de retourner d'un seul coup:

- un statut (OK , NOT_FOUND , ...)
- le corps de la réponse : objet (ou liste) T convertie en json
- un éventuel "header" (ex: url avec id si auto_incr lors d'un POST)

Exemple:

```
@RequestMapping(value="/{codeDev}" , method=RequestMethod.GET)
ResponseEntity<Devise> getDeviseByName(@PathVariable("codeDev") String codeDevise) {
    Devise dev = gestionDevises.getDeviseByPk(codeDevise);
    return new ResponseEntity<Devise>(dev, HttpStatus.OK);
}
```

Autre exemple :

```
//url : http://localhost:8080/serverSpringMvc/ws/rest/devise/EUR
@RequestMapping(value="/{codeDev}",method=RequestMethod.DELETE)
public ResponseEntity< ?> deleteDeviseByCode(
    @PathVariable("codeDev")String codeDevise){
    try {
        deviseDao.deleteDeviseBycode(codeDevise);
        return new ResponseEntity< ?>(HttpStatus.OK);
    } catch (Exception e) {
        e.printStackTrace(); //ou logger.error(e) ;
        return new ResponseEntity< ?>(HttpStatus.NOT_FOUND);
        //ou HttpStatus.INTERNAL_SERVER_ERROR
    }
}
```

Autres variations :

@GetMapping(...) est équivalent à **@RequestMapping(... , method=RequestMethod.GET)**

@PostMapping(...) est équivalent à **@RequestMapping(... , method=RequestMethod.POST)**

@PutMapping(...) est équivalent à **@RequestMapping(... , method=RequestMethod.PUT)**

@DeleteMapping(...) équivalent à **@RequestMapping(..., method=RequestMethod.DELETE)**

8.1. Réponse et statut http par défaut en cas d'exception

Si une méthode d'un contrôleur REST remonte une exception java qui n'est pas rattrapée par un try/catch, la technologie Spring-Mvc retourne alors une réponse et un statut HTTP par défaut :

```
{ "timestamp" : 152....56,
  "status" : 500 ,
  "error" : "Internal Server Error",
  "exception" : "java.lang.NullPointerException",
  "message" : ".....",
  "path" : "/rest/devise/67573567" }
```

Le statut HTTP retourné par défaut dans l'entête de la réponse en cas d'exception est généralement **500 (INTERNAL_SERVER_ERROR)**.

Dans le cadre d'un échec de validation de la requête avec **@Valid** sur le paramètre d'entrée d'une méthode d'un contrôleur REST et avec des annotations de javax.validation (**@Min**, **@Max**, ...) sur la classe du "DTO" (ex : Devise), le statut HTTP alors automatiquement remonté dans l'entête de la réponse HTTP est **400 (Bad Request)** et le corps de la réponse comporte tous les détails sur les éléments invalides.

```
public ResponseEntity<Void> ajouterDevise(@Valid @RequestBody Devise devise) {
....
}
```

```
public class Devise{
...
@Length(min=3, max=20, message = "Nom trop long ou trop court")
private String nom;
}
```

Dans le cadre d'une remontée d'exception personnalisée il est possible de préciser le statut HTTP qui sera remonté via l'annotation **@ResponseStatus()**

Exemple :

@ResponseStatus(HttpStatus.NOT_FOUND) //404

```
public class MyEntityNotFoundException extends RuntimeException {
    //... constructeurs avec super()
```

}

8.2. Exemple d'appel en js/jquery/ajax

Exemple de page HTML/jquery pour le déclenchement des WS "REST" :

JSON tests for devise app (REST/JSON via spring)

devises avec code=EUR

devises avec name=dollar

toutes les devises

50 euros en dollar

devise (to update) : ▼

code : EUR

monnaie :

change :

updated data (server side):

{"monnaie":"euro","codeDevise":"EUR","dchange":1.1152,"pk":"EUR"}

```
<html >
  <head>
    <script src="jquery-2.2.1.js"></script>
  </head>
  <script>
    var deviseList;
    var deviseIdSelected;//id=.codeDevise
    var deviseSelected;

    function display_selected_devise(){
      $("#spanMsg").html( "selected devise:" + deviseIdSelected) ;
      $('#spanCode').html(deviseSelected.codeDevise);
      $('#txtName').val(deviseSelected.monnaie);
      $('#txtExchangeRate').val(deviseSelected.dchange);
    }

    function local_update_selected_devise(){
      deviseSelected.monnaie = $('#txtName').val();
      deviseSelected.dchange= $('#txtExchangeRate').val();
    }
  </script>
</html>
```

```

$(function() {
    $.ajax({
        type: "GET",
        url: "mvc/rest/devises/",
        success: function (data) {
            if (data) {
                //alert(JSON.stringify(data));
                deviseList = data;
                for(deviseIndex in deviseList){
                    var devise=deviseList[deviseIndex];
                    if(deviseIndex==0)
                        { deviseSelected = devise; deviseIdSelected = devise.codeDevise; }
                    //alert(JSON.stringify(devise));
                    $('#selDevise').append('<option value="'+ devise.codeDevise +'>'+
                        devise.codeDevise + ' (' + devise.monnaie + ')</option>');
                }
                display_selected_devise();
            } else {
                $('#spanMsg').html("Cannot GET devises !");
            }
        }
    });
});

```

```

$('#btnUpdate').on('click',function(){
    // $('#spanMsg').html( "message in the bottle" );
    local_update_selected_devise();
    $.ajax({
        type: "PUT",
        url: "mvc/rest/devises/",
        contentType : "application/json",
        dataType: "json",
        data: JSON.stringify(deviseSelected),
        success: function (updatedData) {
            if (updatedData) {
                $('#spanMsg').html("updated data (server side):" + JSON.stringify(updatedData));
            } else {
                $('#spanMsg').html("Cannot PUT updated data");
            }
        }
    });
});

```

```

$('#selDevise').on('change',function(evt){
    deviseIdSelected = $(evt.target).val();
    for(deviseIndex in deviseList){
        var devise=deviseList[deviseIndex];
    }
});

```

```

        if(devise.codeDevise == deviseIdSelected)
            deviseSelected = devise;
    }
    display_selected_devise();
    });
</script>
</head>
<body>

<h3> JSON tests for devise app (REST/JSON via spring) </h3>
    <a href="mvc/rest/devises/EUR"> devises avec code=EUR </a> <br/>
    <a href="mvc/rest/devises/?name=dollar"> devises avec name=dollar </a> <br/>
    <a href="mvc/rest/devises/"> toutes les devises</a> <br/>
    <a href="mvc/rest/devises/convert?amount=50&src=euro&target=dollar">
        50 euros en dollar</a> <br/>
</hr>
    devise (to update) : <select id="selDevise"> </select>
</hr>
    code : <span id="spanCode" ></span><br/>
    monnaie : <input id="txtName" type='text' /><br/>
    change : <input id="txtExchangeRate" type='text' /><br/>
    <input type='button' value="update devise" id="btnUpdate"/> <br/>
    <span id="spanMsg"></span> <br/>
</body>
</html>

```

8.3. Invocation java de service REST via RestTemplate de Spring

Utile pour une **délégation de service** ou bien pour un **test d'intégration** (automatisable via maven et intégration continue).

```

.....
import org.junit.Assert;
import org.junit.BeforeClass;
import org.junit.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.client.RestTemplate;

/* cette classe à un nom qui commence ou se termine par IT (et par par Test)
 * car c'est un Test d'Integration qui ne fonctionne que lorsque toute l'application
 * est entièrement démarrée (avec EmbeddedTomcat ou équivalent) . */

```

```

public class PersonWsRestIT {

    private static Logger logger = LoggerFactory.getLogger(PersonWsRestIT.class);

    private static RestTemplate restTemplate; //objet technique de Spring pour test WS REST

    //pas de @Autowired ni de @RunWith
    //car ce test EXTERNE est censé tester le WebService sans connaître sa structure interne
    // (test BOITE_NOIRE)
    @BeforeClass
    public static void init(){
        restTemplate = new RestTemplate();
    }

    @Test
    public void testGetSpectacleById(){
        final String BASE_URL =
            "http://localhost:8888/spring-boot-spectacle-ws/spectacle-api/public";
        final String uri = BASE_URL + "/spectacle/1";
        String resultAsString = restTemplate.getForObject(uri, String.class);
        logger.info("json string of spectacle 1 via rest: " + resultAsString);
        Spectacle s1 = restTemplate.getForObject(uri, Spectacle.class);
        logger.info("spectacle 1 via rest: " + s1);
        Assert.assertTrue(s1.getId()==1L);
    }

    @Test
    public void testListeComptesDuClient(){
        final String villeDepart = "Paris";
        final String dateDepart = "2018-09-20";
        final String uri = "http://localhost:8080/flight_web/mvc/rest/vols/byCriteria"
            + "?villeDepart=" + villeDepart + "&dateDepart=" + dateDepart;
        String resultAsString = restTemplate.getForObject(uri, String.class);
        logger.info("json listeVols via rest: " + resultAsString);
        Vol[] tabVols = restTemplate.getForObject(uri, Vol[].class);
        logger.info("java listeComptes via rest: " + tabVols.toString());
        Assert.assertNotNull(tabVols); Assert.assertTrue(tabVols.length>=0);
    }
}

```



```

        for(Vol cpt : tabVols){
            System.out.println("\t" + cpt.toString());
        }
    }

    @Test
    public void testVirement(){
        final String uri =
            "http://localhost:8080/tpSpringWeb/mvc/rest/compte/virement";
        //post/envoi:
        OrdreVirement ordreVirement = new OrdreVirement();
        ordreVirement.setMontant(50.0);
        ordreVirement.setNumCptDeb(1L);
        ordreVirement.setNumCptCred(2L);
        OrdreVirement savedOrdreVirement =
            restTemplate.postForObject(uri, ordreVirement, OrdreVirement.class);
        logger.info("savedOrdreVirement via rest: " + savedOrdreVirement.toString());
        Assert.assertTrue(savedOrdreVirement.getOk().equals(true));
    }
}

```

Exemple 2 (délégation de service) :

```

...
import java.nio.charset.Charset;
import java.util.Base64;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.util.LinkedMultiValueMap;
import org.springframework.util.MultiValueMap;

```

```

import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@RestController
@RequestMapping(value="/myapi/auth" , headers="Accept=application/json")
public class LoginDelegateCtrl {

    private static Logger logger = LoggerFactory.getLogger(LoginDelegateCtrl.class);

    private static final String ACCESS_TOKEN_URL =
        "http://localhost:8081/basic-oauth-server/oauth/token";

    private static RestTemplate restTemplate = new RestTemplate();

    HttpHeaders createBasicHttpAuthHeaders(String username, String password){
        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
        String auth = username + ":" + password;
        byte[] encodedAuth = Base64.getEncoder().encode(
            auth.getBytes(Charset.forName("US-ASCII")) );
        String authHeader = "Basic " + new String( encodedAuth );
        headers.add("Authorization", authHeader);
        return headers;
    }

    @PostMapping("/login")
    public ResponseEntity<?> authenticateUser(@RequestBody AuthRequest loginRequest) {
        logger.debug("/login , loginRequest:"+loginRequest);
        String authResponse="{ }";
        try{
            MultiValueMap<String, String> params= new LinkedMultiValueMap<String,
String>();
            params.add("username", loginRequest.getUsername());
            params.add("password", loginRequest.getPassword());

```

```

params.add("grant_type", "password");
//ResponseEntity<String> tokenResponse =
//      restTemplate.postForEntity(ACCESS_TOKEN_URL,params, String.class);
// si pas besoin de spécifier headers spécifique .

HttpHeaders headers = createBasicHttpAuthHeaders("fooClientIdPassword","secret");
HttpEntity<MultiValueMap<String, String>> entityReq =
    new HttpEntity<MultiValueMap<String, String>>(params, headers);

ResponseEntity<String> tokenResponse=
    restTemplate.exchange(ACCESS_TOKEN_URL,
                          HttpMethod.POST,
                          entityReq,
                          String.class);

authResponse=tokenResponse.getBody();
logger.debug("/login authResponse:" + authResponse.toString());
return ResponseEntity.ok(authResponse);
}
catch (Exception e) {
    logger.debug("echec authentication:" + e.getMessage()); //for log
    return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
                          .body(authResponse);
}
}
}

```

8.4. Test d'un "RestController" via @WebMvcTest et MockMvc

Pour tester le comportement d'un composant "RestController" de Spring-Mvc sans avoir à démarrer un serveur complet tel que Tomcat (ou un équivalent) , on peut utiliser la classe **MockMvc** et l'annotation **@WebMvcTest** qui sont spécialement prévues pour faire fonctionner le code d'un web service rest de spring-mvc en recréant un contexte local ayant à peu près de même comportement que celui d'un conteneur web mais sans accès réseau/http .

```

@RunWith(SpringRunner.class)
@WebMvcTest(DeviseJsonRestCtrl.class)
public class DeviseJsonRestCtrlIntegrationTest {

    @Autowired
    private MockMvc mvc;

```

```

@Test
public void testXyz(){
    mvc.perform(get("/rest/devises/?name=euro")
        .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$", hasSize(1) ))
        .andExpect(jsonPath("$[0].name", is("euro") ));
}
}

```

NB : Spring5 propose une variante `@WebFluxTest` et `WebTestClient` pour `WebFlux` .

...

9. Test de W.S. REST via Postman

9.1. paramétrages "postman" pour une requête en mode "post"

The screenshot shows the Postman interface for a POST request. The URL is `http://localhost:8080/serverSoap/ws/rest/devise`. The request body is set to raw JSON with the following content:

```

{
  "codeDevise" : "MS2",
  "tauxChange" : 1.4568
}

```

The response status is 200 OK, with a time of 82 ms and a size of 156 B.

10. Test de W.S. REST via curl

curl (*command line url*) est un programme utilitaire (d'origine linux) permettant de déclencher des requêtes HTTP via une simple ligne de commande.

Via certaines options, curl peut effectuer des appels en mode "GET", "POST", "DELETE" ou "PUT".

Ceci peut être très pratique pour tester rapidement un web service REST via quelques lignes de commandes placées dans un script réutilisable (.bat, .sh,) .

lancer_curl.bat

```
cd /d "%~dp0"
REM instructions qui vont bien
set URL=http://localhost:8081/my-api/info/1
curl %URL%
pause
```

curl fonctionne en **mode GET par défaut** si pas de -d (*pas de data*)

```
curl %URL%
REM "verbose" (-v) très pratique pour connaître les détails de la communication réseau
curl -v %URL%
```

```
curl -o out.json %URL%
```

pour stocker la réponse dans un fichier texte (ici out.json)

curl fonctionne en mode **POST** par défaut avec data (-d ...)

curl fonctionne en mode PUT si **-X PUT** et mode DELETE si **-X DELETE**

appel au format par défaut (application/x-www-form-urlencoded)

si pas d'option -H "Content-Type: ..." au niveau de la requête
alors par défaut logique champ/paramètre de formulaire en mode POST avec

-d paramName1=valeur1 -d paramName2=valeur2 ...

Exemple :

```
set URL=clientIdPassword:secret:@localhost:8081/basic-oauth-server/oauth/token
set PWD=d8dfc382-e012-491a-8d03-ca6ad9d81083
```

```
curl %URL% -d grant_type=password -d username=user -d password=%PWD%
```

Requête au format "application/json" :

NB : en version windows , curl ne gère pas bien les simples quotes et il faut préfixer les " internes par des \

```
curl %LOGIN_URL% -H "Content-Type: application/json"
-d '{"username\":\"member1\", \"password\": \"pwd1\"}'
```

il vaut mieux donc utiliser un fichier pour les données en entrée :

```
curl %LOGIN_URL% -H "Content-Type: application/json" -d @member1-login-request.json
```

avec

member1-login-request.json

```
{
  "username": "member1",
  "password": "pwd1"
}
```

Authentification avec curl :

```
curl --user myUsername:myPassword ... permet une "BASIC HTTP AUTHENTICATION"
```

ou bien

```
curl -H "Authorization: Bearer b1094abc.._ou_autre_jeton" permet une demande d'autorisation
en mode "Bearer / au porteur de jeton" (jeton à préalablement récupérer via login ou autre )
```

11. Notion d'Api REST et description

11.1. Description détaillée d'Api REST (Swagger, RAML, ...)

Notion d' API REST

Dès le début , la structure d'un web-service "*SOAP*" a été décrite de façon standardisée via la norme *WSDL* (standard officiel du W3C).

A l'inverse les Web-services REST (qui sont basés sur de simples recommandations autour de l'usage d'HTTP) ne sont toujours pas associés à un type de description unique et standardisé.

Un **document qui décrit la structure d'un web-service REST** est généralement appelé "**API REST**" et peut être écrit en XML , en JSON ou en YAML.

Les principaux formalismes existants pour décrire une API REST sont :

- **WADL** (existant depuis longtemps en **XML** mais en perte de vitesse)
- **Swagger** (version 1.x basée sur **YAML** , v2 basée sur **JSON** que l'on peut convertir en **YAML**). Bien outillé et existant depuis plusieurs années, swagger a pour l'instant une petite longueur d'avance.
- **RAML** (pour l'instant basé sur une syntaxe **YAML** volontairement simple , pris en charge par quelques marques telles que "MuleSoft" , ...)
- **Blueprint API** : basé également sur **YAML**

Que choisir entre Swagger , RAML , Blueprint-API , ... ?

- ??? (l'avenir le dira)
- à court terme : tester un peu tout et garder ce qui semble le plus pratique .

Faut-il attendre un standard officiel et unique ?

- **Non** (car on peut attendre encore longtemps un standard qui ne viendra peut être pas ou qui viendra tardivement) .
- **Des passerelles sont déjà (et seront encore plus) disponibles pour passer d'un format à un autre .**

11.2. Fragile format YAML

Format YAML

YAML (*YAML Ain't Markup Language*) n'est d'après son nom , pas un langage à balise mais se veut être un équivalent d'un point de vue fonctionnalité (à soir sérialisation/dé-sérialisation de documents informatiques arborescents).

books.yaml (exemple)

YAML est en fait **structuré via des indentations** et se veut être **facile à lire (ou à écrire) par une personne humaine** .

*De la rigueur
dans les indentations
S'impose !!!*

(fragile comme CSV)

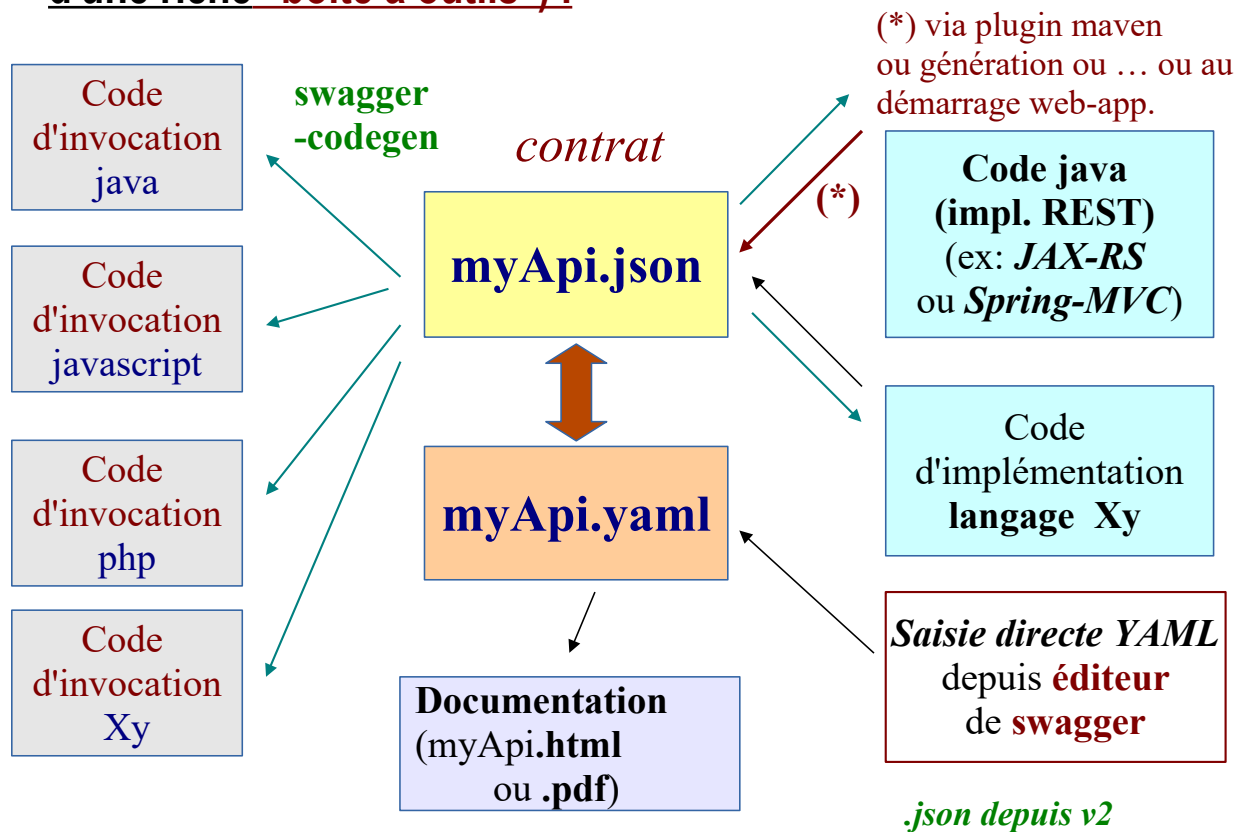
```
/books:
  /{bookTitle}
  get:
    queryParameters:
      author:
        displayName: Author
        type: string
        description: An author's full name
        example: Mary Roach
        required: false
      publicationYear:
        displayName: Pub Year
        type: number
        description: The year released for the first time
    ...
```

YAML semble très fragile .

Heureusement **Swagger 2** s'appuie maintenant principalement sur une description au format JSON.

12. Swagger

Swagger (spécifications pour **API REST** accompagnées d'une riche **"boîte à outils"**).



13. Config swagger2 / swagger-ui pour spring-mvc

à ajouter dans pom.xml

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.7.0</version>
</dependency>

<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.7.0</version>
</dependency>
```

dans beans.xml (ou via un @Import équivalent en java-config) :

```
...
<bean class="fr.xyz.dja.swagger.config.MySwaggerConfig" /> ...
```

MySwaggerConfig

```
package fr.xyz.dja.swagger.config;
```

```

import java.util.Collections;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.service.ApiInfo;
import springfox.documentation.service.Contact;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration
@EnableSwagger2
@Import(MySwaggerUiConfig.class) //config des ressources nécessaires à swagger-ui
public class MySwaggerConfig {

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.any())
            //apis(RequestHandlerSelectors.basePackage("fr.xyz.dja.rest"))
            .paths(PathSelectors.any())
            //paths(PathSelectors.ant("/rest/*"))
            .build()
            .apiInfo(apiInfo());
    }

    private ApiInfo apiInfo() {
        return new ApiInfo(
            "My REST API (serverSpring MVC)",
            "Api pour Devise",
            "API TOS",
            "Terms of service",
            new Contact("DJA1", "www.xyz.fr", "toto@worldcompany.com"),
            "License of API", "API license URL", Collections.emptyList());
    }
}

```

MySwaggerUiConfig

```

package fr.xyz.dja.swagger.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@Configuration
@EnableWebMvc
public class MySwaggerUiConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        //System.out.println("##### MySwaggerUiConfig #####");
        //configuration nécessaire pour que swagger-ui.html puisse accéder
        //à ses ressources (.html, .css, .js) dans une partie "META-INF/resources"
        //du fichier springfox-swagger-ui.jar téléchargé via maven
        //NB: l'url déclenchante doit invoquer swagger-ui à côté de v2/api-docs
        //systématiquement recherché en relatif
        registry.addResourceHandler("*/swagger-ui.html")
            .addResourceLocations("classpath:/META-INF/resources/");
        registry.addResourceHandler("*/**").addResourceLocations("classpath:/META-INF/resources/");
    }
}

```

}

dans **index.html** (ou ailleurs) :

```
<a href="/ws/v2/api-docs">description swagger2(json) de l'api REST</a> <br/>
...
<a href="/ws/swagger-ui.html">documentation Api REST générée dynamiquement par swagger2 et swagger-ui</a>
```

où "**ws**" (ou bien "**mvc**" ou bien "**services**" ou "...") est la valeur choisie de l'url-pattern de `org.springframework.web.servlet.DispatcherServlet` (de Spring-Web-Mvc)

(exemple (dans web.xml) : `<url-pattern>/ws/*</url-pattern>`)

The screenshot shows the Swagger UI interface. At the top, there's a green header with the Swagger logo, a dropdown menu set to 'default (/v2/api-docs)', and an 'Explore' button. Below the header, the title 'My REST API (serverSpring MVC)' is displayed, followed by 'Api pour Devise'. It also mentions 'Created by DJA1' and provides links to 'www.afcepf.fr', 'Contact the developer', and 'License of API'.

The main section is titled 'rest-devise-service : Rest Devise Service'. It lists four API endpoints:

- GET** `/rest/devise` with operation `devisesByCriteria`
- POST** `/rest/devise` with operation `saveOrUpdateDevise`
- DELETE** `/rest/devise/{codeDev}` with operation `deleteDeviseByCode`
- GET** `/rest/devise/{codeDev}` with operation `deviseByCode`

Below this, there are two more sections: 'ws-auth : Ws Auth' and 'ws-confidentiel : Ws Confidentiel', each with 'Show/Hide', 'List Operations', and 'Expand Operations' links.

At the bottom, it shows the base URL and API version: '[BASE URL: /serverSpringMvc/ws , API VERSION: API TOS]'.

14. HATEOAS

14.1. Principes HATEOAS

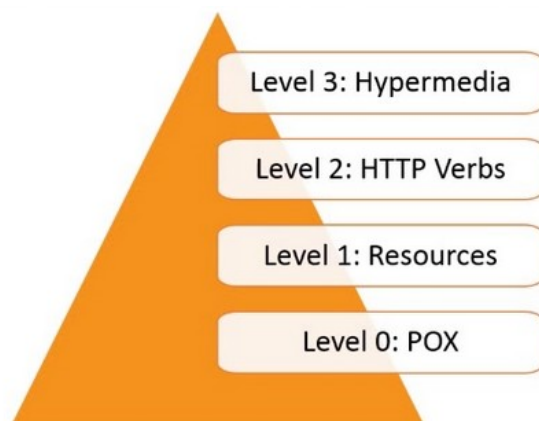
HATEOAS est l'abréviation d'**H**ypermedia **A**s **T**he **E**ngine of **A**pplication **S**tate (Hypermédia en tant que moteur de l'état d'application).

Hypermédia est ici à comprendre au sens *extension de hypertexte (avec liens possibles)*

Concrètement, suite à un premier appel d'un WS REST lié à une ressource précise (ex : `.../accounts/12345`), on récupère une réponse (souvent JSON ou XML) comportant non seulement des données mais aussi des liens (urls) vers des traitements possibles sur la ressource dont on a récupéré une représentation :

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: ...
<?xml version="1.0"?>
<account>
  <account_number>12345</account_number>
  <balance currency="usd">100.00</balance>
  <link rel="deposit" href="/accounts/12345/deposit" />
  <link rel="withdraw" href="/accounts/12345/withdraw" />
  <link rel="transfer" href="https://bank.example.com/accounts/12345/transfer" />
  <link rel="close" href="https://bank.example.com/accounts/12345/close" />
</account>
```

14.2. Le modèle de maturité de Richardson pour "REST"



Le niveau "0" (jamais utilisé en REST, proche des "RPC SOAP") serait : une seule URL pour tout un paquet de Web Services (sans fin d'URL identifiant une ressource, sans verbe HTTP "get/post/delete/put" pour indiquer l'action).

Le niveau "2" est le plus couramment utilisé pour l'instant ,
Le niveau "3" correspond à HATEOAS .

Dans le contexte théorique "HATEOAS" , certaines contraintes (au niveau de l'interaction client/serveur) sont formulées mais il ne s'agit là que d'une vision "puriste" (avec des aménagements possibles) .

14.3. HATEOAS concrètement à court terme :

Dans l'exemple précédent, en suivant ces liens (véhiculés dans la réponse) , on peut alors déclencher des dépôts , retraits , virements , fermeture sur le compte bancaire récupéré .

L'extension "**Spring HATEOAS**" (pour le framework spring) permet de mettre en oeuvre des services "HATEOAS" .

Autre implémentation possible : l'extension *express-hateoas-links* pour nodeJs/express .

Attention, Attention :

- On en est pour l'instant au stade "expérimentation d'idée récente"
- couplage données/traitements annexes à gérer/assumer
- le client (ex : IHM HTML/CSS/js) doit idéalement pouvoir proposer des liens hypertextes d'une manière dynamique
- les habilitations doivent être prises en compte (droits ou pas aux traitements liés)

HATEOAS peut être vu comme une extension facultative (potentiellement ignorable à court terme par les clients qui ne comprennent pas l'extension) des WS-REST .

Son utilisation peut être sérieusement envisagée sur des éléments où la découverte d'éléments au départ inconnus peut apporter un plus (ex : exploration d'une base de connaissances) .

V - L'approche micro-services

1. Architecture "micro-services" (essentiel)

1.1. Présentation architecture micro-services

Architecture "micro-service" (MSA)

L'architecture "micro-services" peut être vue comme une évolution de l'architecture "SOA/orientée services" avec :

- ***plus de légèreté dans l'infrastructure logicielle*** (moins ou pas de ESB, Orchestrateurs "BPM" et autres intermédiaires sophistiqués).
- **alignement sur les technologies "cloud" (IaaS, PaaS) et "micro-conteneurs"** (infrastructure "scalable"/élastique (utilisation fréquente de *Docker* , *Kubernetes* , ...))
- ***tolérance vis à vis des services inaccessibles*** (pas de "point de panne simple" , cluster avec prise en compte de l'état des serveurs)
- alignement avec "WOA" (xml/soap --> ***json/REST/HTTP***)
- ***très peu d'inter-dépendances entre les micro-services*** (idéalement "stateless" et ***programmables avec technologies très variables*** : Java/SpringBoot , NodeJs/Express , python,)

1.2. illustration (exemples typiques)

Micro-services typiques en n-tiers

micro-service
IHM/GUI web

ex: serveur http "***nginx***"
pour télécharger front-end SPA
(Angular ou React ou ...) avec config
"reverse-proxy" vers api rest

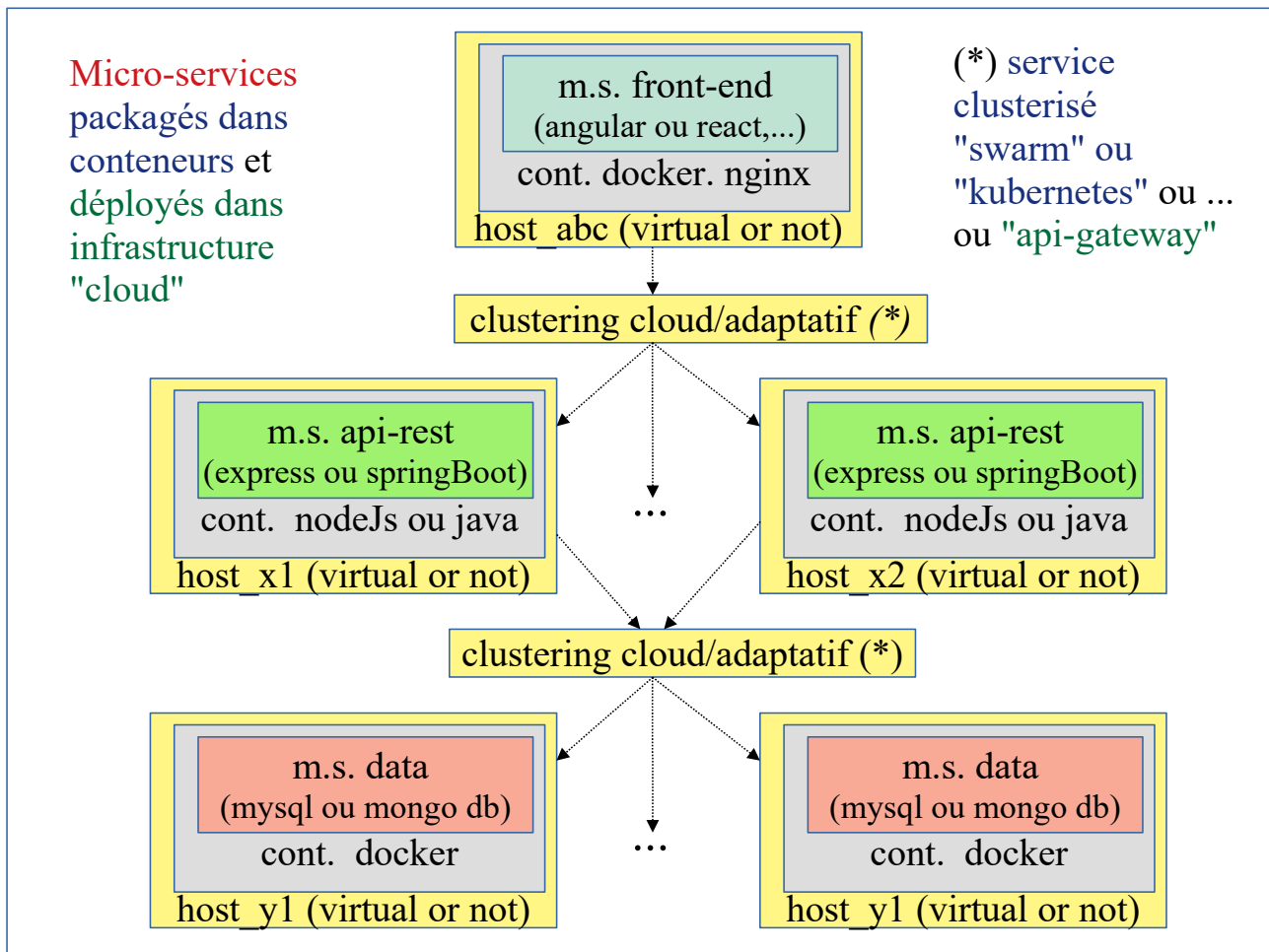
micro-service
API-REST
(stateless)

ex: ***Java/SpringBoot***
ou ***NodeJs/express***

micro-service
"données"

ex: ***mysql*** ou ***mongo-db***
clusterisable

1.3. illustration (packaging / fonctionnement "cloud")



1.4. Caractéristiques clefs des micro-services

Caractéristiques structurelles d'un micro-service

- ***forte cohésion interne , périmètre fonctionnel très réduit***
(faire une seule chose et le faire bien/efficacement)
- ***idéale indépendance*** (conçus et développés
indépendamment , testés et déployés ***indépendamment*** des
autres (dans micro-conteneurs)
--> agilité / souplesse technologique
- ***beaucoup de délégation*** (très grande sollicitation des
réseaux informatiques)
- ***"brique de code simple"*** packagée et déployée dans
topologie "compatible cloud" sophistiquée (ex : réseaux
superposés virtuels sur réels)

1.5. Souplesse dans le choix des technologies

Agilité/souplesse technologique

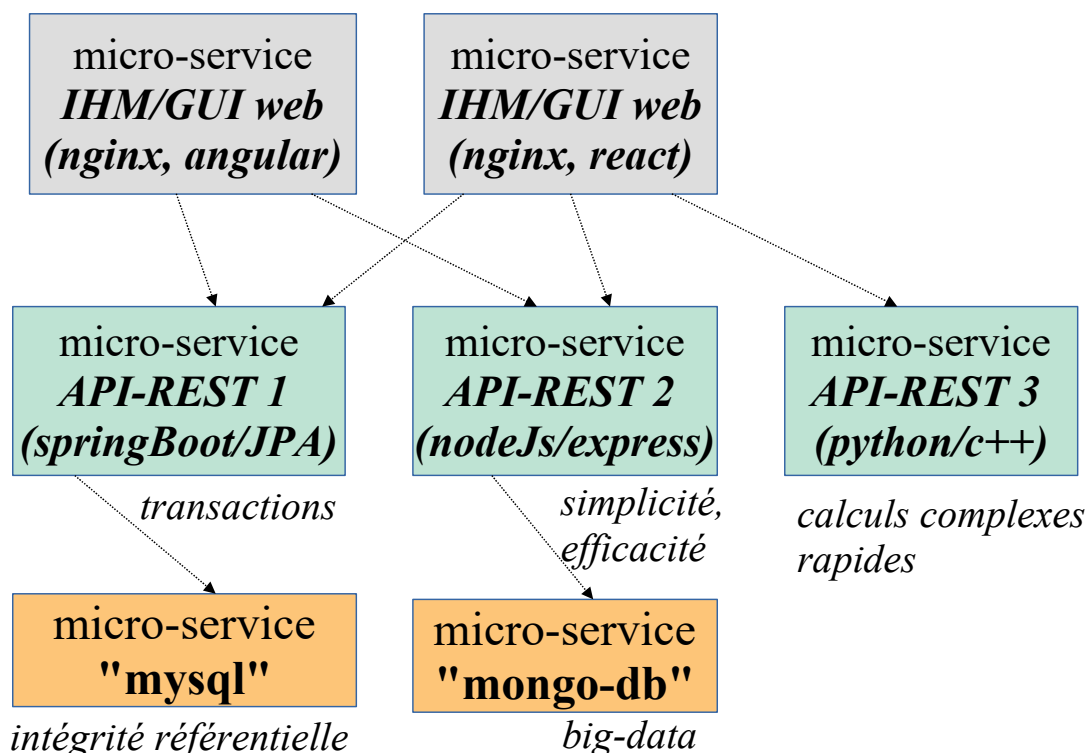
- Mis à part quelques petits éléments protocolaires généralement imposés (tcp/ip , ...) , **un micro-service peut être programmé dans une technologie quelconque** (java , javascript, python, ...) car toutes les librairies et configurations spécifiques seront cachées/isolées au sein de micro-conteneurs opaques .

- *chaque équipe de développement peut donc choisir librement la technologie la plus appropriée pour développer son micro-service*

(ex : java/jee/spring-boot pour transactions courtes
 nodeJs/express pour simplicité/efficacité,
 base relationnelle (mysql, ...) pour données référentielles
 mongoDB/... pour données en grand volume (big data)
 elastic-search pour gestion historique et recherches
 python et c/c++ pour service scientifique (calculs , ...)

...

Variétés technologiques des micro-services



1.6. Impacts de l'architecture micro-services

Quelques impacts de l'architecture micro-service

- "périmètre réduit" + "liberté dans technologie de mise en oeuvre" font que :
 - **les risques sont plus limités/circoncis** .
(moins de problème d'incompatibilité entre technologies, si un sous projet échoue on change de technologie et on recode mieux le micro-service à problème . ce n'est pas l'ensemble qui est compromis)
 - beaucoup de liberté dans le choix des frameworks , langages et serveurs :
 - > **marché des technos très ouvert** (ça bouillonne , ça part dans tous les sens , orientation "open source", les grands éditeurs "Microsoft, IBM, ..." nous imposent moins leurs "solutions coûteuses").
 - > **développeurs devant de plus en plus être polyglottes !!!**
(plein de langages/technos à apprendre/maîtriser !!!).

Autres impacts de l'architecture micro-service

- **Tests d'intégration facilités** :
 - certains micro-services sont dépendants d'autres micro-services en arrière plan (ex : api-rest nécessitant accès database)
 - une fois packagé sous forme de conteneur "docker" , un micro-service d'arrière plan pourra:
 - être complexe/sophistiqué en interne (ex : sécurité , ...).
 - être vu comme une "boîte opaque" rendant un certain service utile (un minimum documenté)
 - être utilisé au sein de test d'intégration (alias "end-to-end") pour vérifier la bonne communication inter-service dès la phase de "pré-released" / "pré-production" du micro-service appelant .
 - prévoir peut être variantes/profils "avec ou sans cluster" durant les phases de tests (avec complexité progressive).

1.7. Factorisation / réutilisation des micro-services

Factorisation et ré-utilisation des micro-services

Trouver le bon niveau de découpage / décomposition sachant que :

- *un service simple/réduit peu plus facilement être réutilisé*
- *trop de délégations peuvent induire un très grand trafic réseau et un ralentissement dû aux sérialisations/dé-sérialisations des requêtes / réponses*
- *certaines données sont plus simples à relier/corréler localement qu'en mode "dispersé" : il faut idéalement modéliser (avec UML ou autre) pour étudier les aspects "cohésion non décomposable" et "indépendances possibles" .*
- *ne pas hésiter à restructurer d'une version à l'autre (le "refactoring" n'est jamais un objectif mais c'est un "moyen souvent nécessaire à mettre en oeuvre").*
- *orchestration simple = "bonne idée de SOA à reprendre"*

1.8. Evolutivité des micro-services

Evolutivité d'une architecture micro-service

Aspects favorisant l'évolutivité de l'architecture :

- "périmètre réduit" , "faible complexité" et "choix technologiques" favorisent les évolutions unitaires (ex : meilleurs performances après ré-écriture ,
ajouts de fonctionnalités opaques (meilleurs heuristiques , ...),
ajouts de traitements annexes)
- "déploiements indépendants" (selon infrastructure hôte telle que "kubernetes") favorisent également l'évolutivité de l'architecture

Contraintes à garder à l'esprit :

- dans l'idéal : ne pas trop changer les structures de données échangées d'un micro-services à l'autre pour ***ne pas remettre en cause les "contrats d'invocation"*** . il doit y avoir une idéale *compatibilité ascendante* . [NB : JSON un peu plus souple que SOAP/XML et certains ajustement sont plus simples/rapides]
- migration de données en cas de changement de format (mysql / mongo) ?

1.9. Résilience des micro-services

Résilience (tolérance panne partielle)

Au sein d'une architecture micro-service , **l'inaccessibilité ponctuelle et les pannes partielles sont considérées comme des phénomènes normaux / courants** .

Pour qu'un micro-service puisse continuer à fonctionner (éventuellement en mode "service légèrement dégradé") dans le cas ou un élément dont il dépend ne fonctionne plus bien , il faudra anticiper des "plans B".

--> ***Choisir des technologies "clusterisables"*** (avec "load-balancing" , "rétablissement de connexions" , ...)

--> ***Bien gérer les exceptions*** (des 2 cotés : client et serveur)

--> ***Bien paramétrer l'infrastructure hôte*** (ex : "*Kubernetes*") qui dit coopérer avec la technologie "clusterisable" (ex : mysql, mongo)

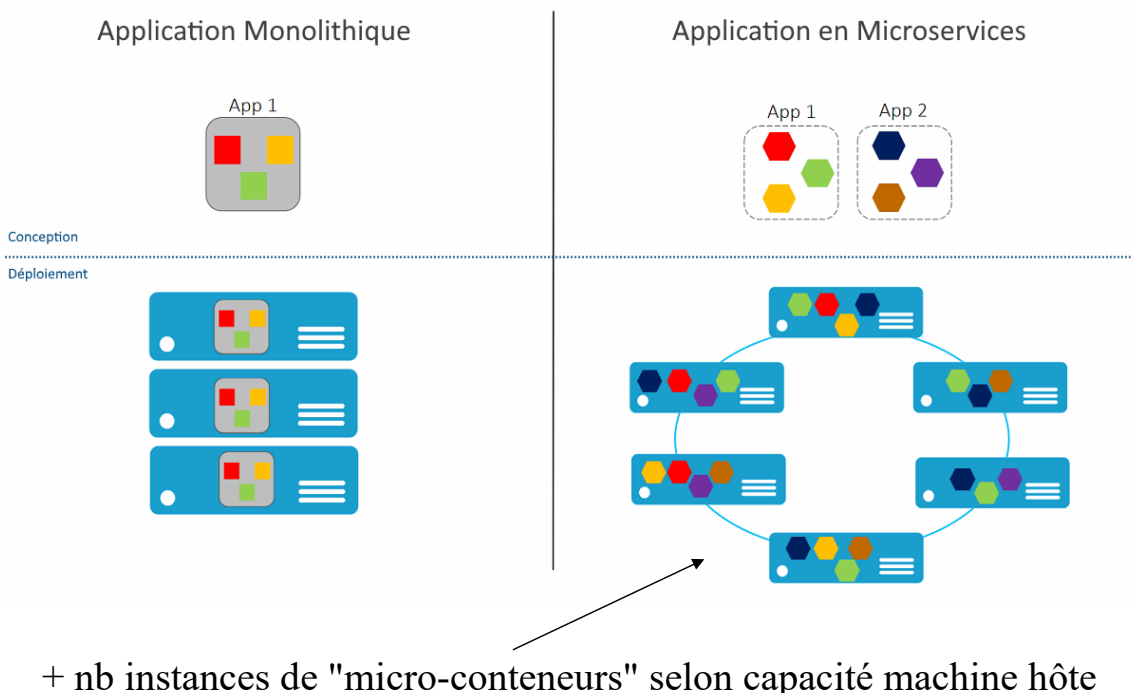
1.10. Bonne exploitation des ressources matérielles

Bonne exploitation des ressources matérielles

L'architecture "micro-service dans micro-conteneur" permet (avec de bons réglages) de généralement mieux exploiter les ressources matérielles (CPU, RAM, ...) qu'une architecture "application mono-bloc" .

- une application mono-bloc (avec plusieurs couches logicielles "GUI" + "Tx" + "Accès données") comporte généralement quelques "goulets d'étranglement" difficiles à gérer/optimiser .
- il faut souvent surdimensionner la puissance de la machine hôte pour être certain d'anticiper certains pics de charge.
- un micro service correspond généralement à un seul "tiers" (soit "GUI" , soit "Api REST" , soit) et la consommation en ressource matérielle est plus régulière/homogène . En multipliant le déploiement de conteneurs identiques sur la même machine on peut ainsi assez bien exploiter la puissance d'une machine sans trop la sous-exploiter .

Exploitation optimisée des ressources matérielles



1.11. Elasticité / scalabilité

Extensibilité/élasticité/scalabilité selon orientation "stateless"

D'emblée prévus pour l'infrastructure "cloud", les micro-services doivent pouvoir s'adapter à des redimensionnement de voilure (changement de la taille d'un cluster).

Ceci ne pose aucun problème pour les micro-services de type "API stateless" (ex : Web Service de calcul, ...) mais pose de très nombreux problèmes sur l'aspect "cohérence des données réparties/dispersées" :

- théorème "CAP" spécifiant que l'on ne peut pas avoir en même temps les aspects "distribué / tolérance de panne", "disponibilité / réponses quasi immédiates" et "transactions parfaites ACID". il faudra privilégier un aspect prioritaire par rapport aux autres.
- intégrité référentielle en mode distribué/dispersé (cohérence entre données fonctionnelles/métiers réparties dans plein de petites bases?)

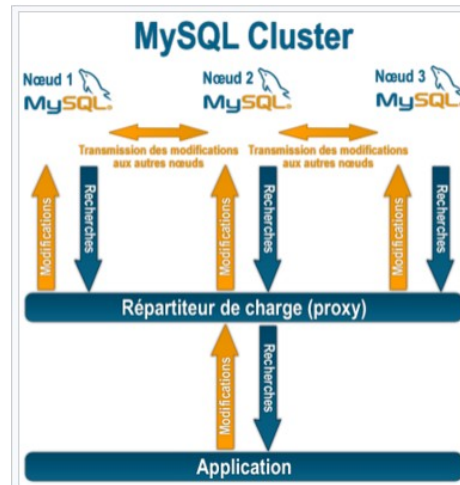
1.12. Cluster de données

Cluster de données avec priorité aux écritures cohérentes

Lorsque l'on fait fonctionner "mysql" en cluster sur différentes machines (avec "load-balancing" et "fail-over") ,

- les *lectures simultanées* peuvent alors être *déclenchées en grand nombre* (relativement rapidement)

- les *écritures* sont par contre *ralenties* (car le système relationnel privilégie la cohérence des données écrites/modifiées via une réplication immédiate synchronisée vers les autres membres actifs du cluster)



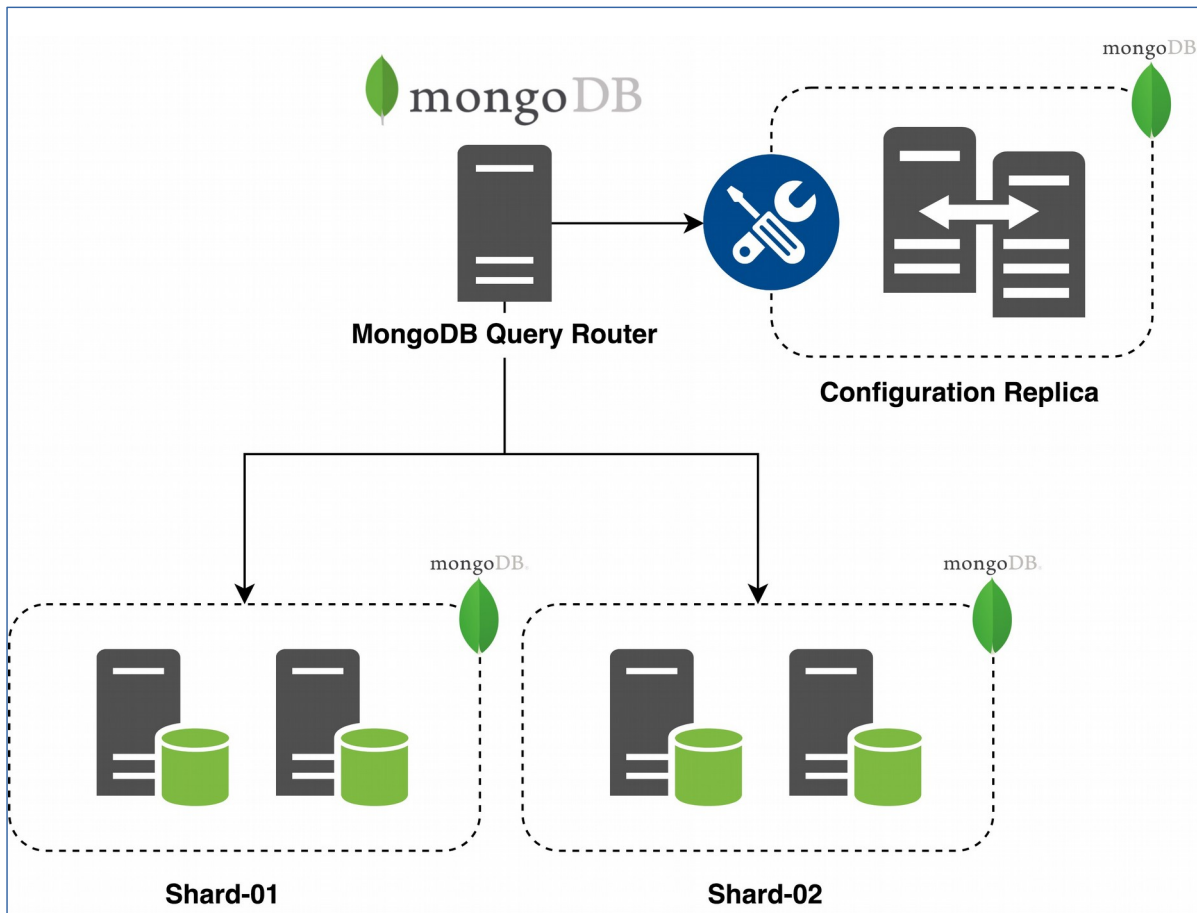
Cluster de données avec priorité aux écritures rapides et lectures immédiates (avec consolidation différée)

Lorsque l'on fait fonctionner certaines technologie "noSQL" (ex : "mongo db") en cluster sur différentes machines (avec "load-balancing" et "fail-over") ,

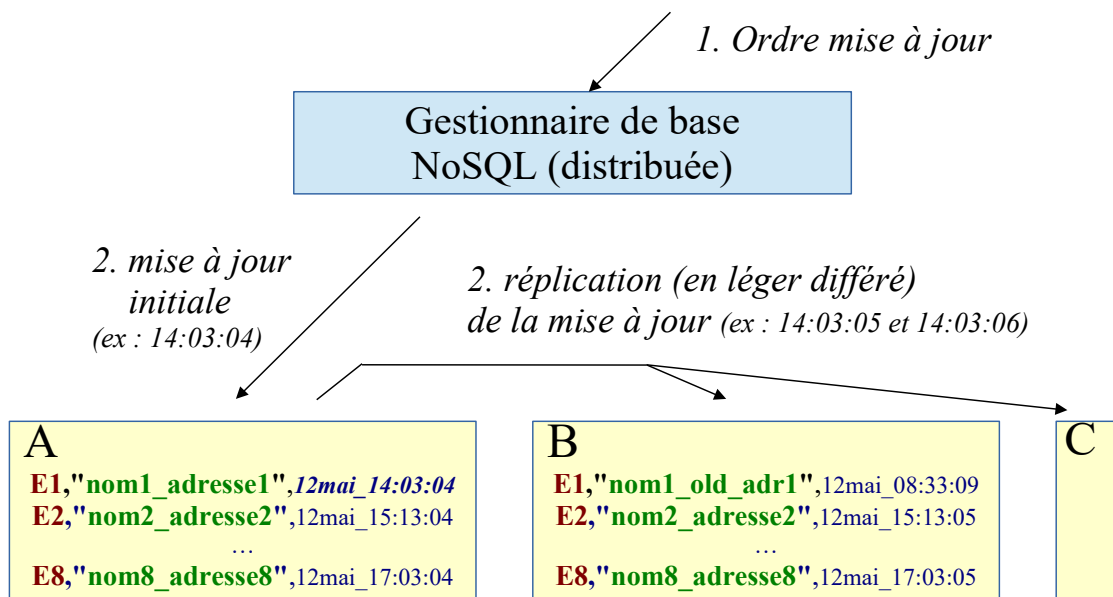
- les *écritures* sont *quelquefois* (selon le niveau transactionnel paramétré) *effectuées relativement rapidement* et *répliquées de façon différée* vers les autres membres actifs du cluster

- des *lectures simultanées effectuées/redirigées* vers différents membres du cluster peuvent *quelquefois retourner des valeurs différentes* (lorsque la réplication différée n'est pas terminée)

NB : ce manque de précision/actualisation des valeurs lues est quelquefois "pas gênante" dans certains contexte (ex : données statistiques , big-data , ...)



Basic NoSQL (**key**, **value**, *timeStamp*)



Les synchronisations/répliquations peuvent s'effectuer dans tous les sens (A-->B,C ou B-->A,C , ...) . L'information technique/cachée "timeStamp" (ex : 12mai_14:03:04) permet de connaître quelle est la version la plus à jour (la plus récente).

1.13. Eventuel mode message pour les micro-services

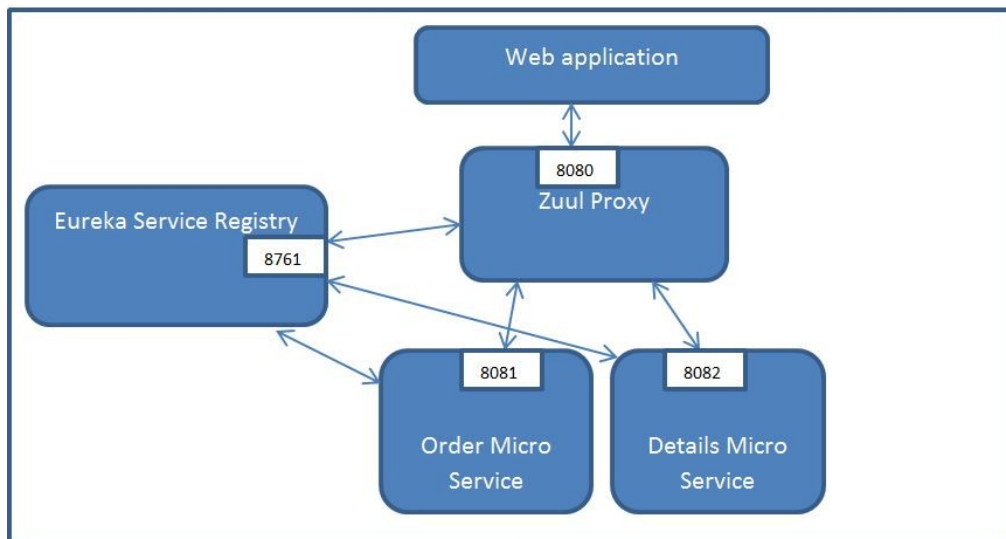
Micro service en mode "message asynchrone / événement"

Bien qu'une api "REST/JSON" soit le type de micro-service le plus répandu , il existe certaines technologies orientées "transports asynchrones de messages" qui sont compatibles avec l'architecture micro-services :

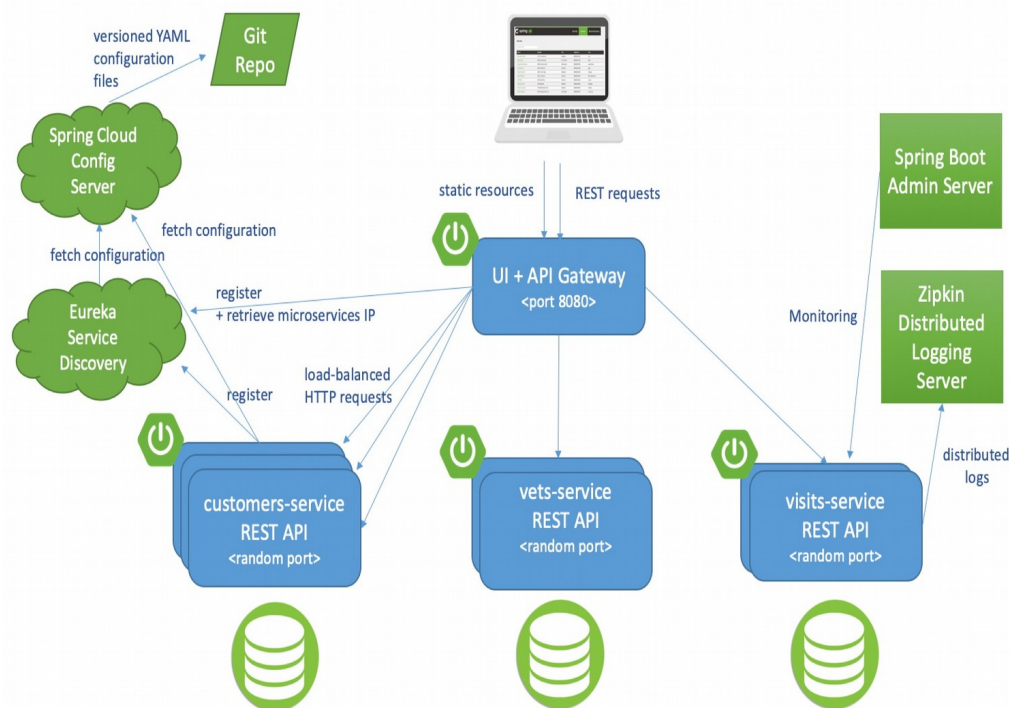
Exemple : Spring-cloud
avec services "Netflix OSS (Eureka, zuul)"
+ RabbitMQ

1.14. Diversité des solutions "cloud/micro-services"

Diversité des solutions "cloud" : Netflix-OSS

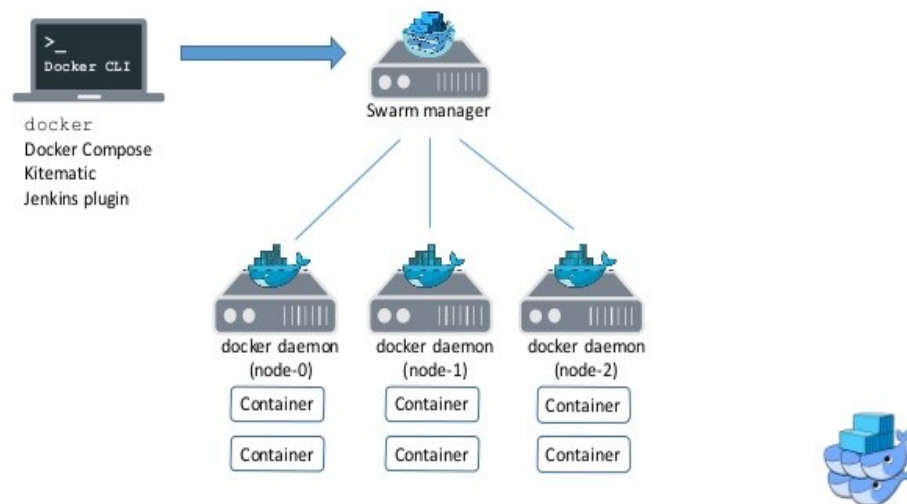


Diversité des solutions "cloud" : Spring-cloud

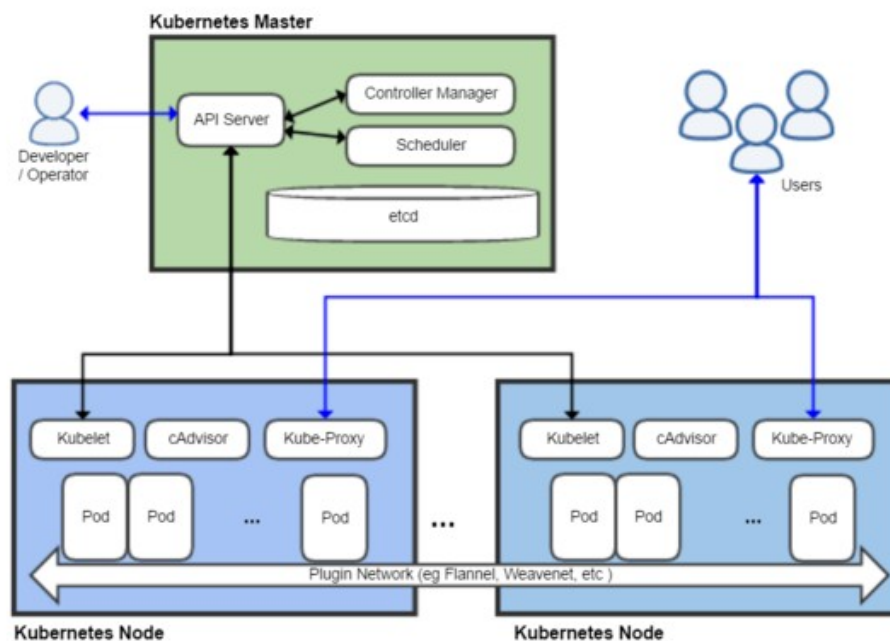


L'extension facultative "spring-cloud" s'appuie sur "netflix-oss"

Diversité des solutions "cloud" : "docker swarm"



Diversité des solutions "cloud" : Kubernetes



NB. : Un "pod kubernetes" est un ensemble de "conteneurs co-localisés" (docker ou autres)

VI - Design/Gestion d'API

1. Design d'une api REST

1.1. Rappels des fondamentaux

Bien respecter les "verbes"

GET pour lecture , recherche , interrogations,
POST pour ajout ou "saveOrUpdate"
PUT pour mise à jour d'un existant
DELETE pour une suppression .

Bien respecter les conventions habituelles :

fin d'url en typeEntité/**idRessource** (ex : produit/1) pour **recherche unique** (par id/pk) .

fin d'url en typeEntité?**critere1=val1&critere2=val2** pour **recherche multiple** (retournant un tableau ou liste de taille 0 ou n) .

Même URL pour recherche , suppression , mise à jour , seul le verbe (méthode HTTP) change :
GET ou DELETE ou PUT .

1.2. Retourner des réponses explicites et des statuts Http précis

Lorsque c'est possible , il vaut mieux retourner les statuts http précis :

"Unauthorized access (401)";
"Forbidden resource can't be accessed (403)";
"resource not found (404)";
"Internal server error (500)";
"Service unavailable (503)";

Ceux ci sont standards et récupérables via plein de technologies (xhr , jquery , jax-rs , spring-mvc, ...) . Assez "génériques" ces statuts peuvent souvent être gérés par du code hautement réutilisable (filtres ,) .

En outre le contenu (partie "body") de la réponse doit idéalement être très explicite.

Il est par exemple conseillé d'accompagner le résultat de la recherche avec une copie en retour des critères de recherches reçus .

Exemple_1 (elevation_api) :

```
https://maps.googleapis.com/maps/api/elevation/json
?locations=39.7391536,-104.9847034
&key=YOUR_API_KEY
```

Api response :

```
{
  "results" : [
    {
      "elevation" : 1608.637939453125,
      "location" : {
        "lat" : 39.73915360,
        "lng" : -104.98470340
      },
      "resolution" : 4.771975994110107
    }
  ],
  "status" : "OK"
}
```

Exemple_2 (fixer.io) :

```
http://data.fixer.io/api/convert
? access_key = API_KEY
& from = GBP
& to = JPY
& amount = 25
```

Api response :

```
{
  "success": true,
  "query": {
    "from": "GBP",
    "to": "JPY",
    "amount": 25
  },
  "info": {
    "timestamp": 1519328414,
    "rate": 148.972231
  },
  "historical": ""
  "date": "2018-02-22"
  "result": 3724.305775
}
```

Il peut également être intéressant de retourner en retour une copie des données effectivement

ajoutées ou mises à jours (POST , PUT) .

--> ceci permet de rassurer le client : la mise à jour s'est bien effectuée .

--> dans le cas d'un ajout (POST) , il est souvent utile de retourner une copie de l'entité complète (avec l'identifiant / clef primaire quelquefois auto incrémentée) .

1.3. POST au sens "save or update"

au lieu d'un double point d'entrée (un en "POST" pour l'ajout et un en "PUT" pour la mise à jour), il est souvent possible de programmer un seul point d'entrée (en mode "POST") avec une sémantique "saveOrUpdate()" :

si id == null alors ajout / insert / persist
sinon update / merge .

1.4. Prise en compte des problématiques de sécurité

.Très souvent HTTPS (rarement HTTP)

.pas de choses confidentielles à la fin d'une URL d'une requête en mode GET

.crypter si nécessaire

.Api key et/ou token d'authentification selon les cas

...

.Séparer si besoin la partie "Consultation" (read) de la partie "Mise à jour" (write)
dans deux WS complémentaires de la même api pour pouvoir simplement effectuer les
paramétrages de sécurité (ex : permitAll() ou)

1.5. Design pattern "DTO" adapté aux web services REST

Bien qu'il soit techniquement possible de directement retourner des représentations "Xml" ou "Json" des entités persistantes (avec @Entity de Jpa) via des ajouts adéquats de @JsonIgnore près des @OneToMany , @ManyToMany,

il est souvent préférable de retourner des structures spécifiquement adaptées aux "contrôleurs" de "web services REST" : classes java rangées dans un package du genre "rest.dto" ou "rest.payload" ou

Il s'agit d'un glissement "web / http / rest" du design pattern "DTO = Data Transfert Object" .

Contrairement aux WS SOAP , les WS REST n'ont pas une logique "RPC" et les DTOs sont plutôt à placer près des "contrôleurs REST" que des services "métiers/business" internes) .

Dans certains cas , les DTOs sont indispensables (ex : dto.OrdreDeVirement pour déclencher serviceInterneCompte.transférer(montant,numCptDeb, numCptCred) .

D'autres fois , lorsque les structures sont (et sont censées rester) simples , une sérialisation directe des entités persistantes peut éventuellement booster les performances et/ou la rapidité de réalisation. A ce sujet, étant donné que le format JSON est très souple (comparé à Xml) , un changement de structure interne du serveur peut souvent rester transparent vis à vis du client effectuant les invocations : réadaptation plus faciles à effectuer en REST/JSON qu'en mode SOAP/XML .

En appliquant les slogans **KISS** (Keep It Simple Stupid) et **DRY** (Don't Repeat Yourself) , on peut souvent commencer simple et complexifier graduellement en fonction des besoins .

Critères importants (indépendance / non-adhérence , réutilisabilité , partage , ...)
--> à doser au cas par cas .

1.6. Autres considérations (bonne pratiques)

Etant donné que d'un point de vue externe , un WS REST est avant tout identifié par l'URL de son point d'accès (endPoint) , il est très fortement conseillé de mettre en place des URIs assez structurés/composés de type :

`https://www.domainXy.com/appliXy/rest/api-zz/public/entityXy .`

La partie "rest/api-zz" ou "api-zz" permettra d'effectuer des réglages/paramétrages de reverse-proxy HTTP dans le cadre d'une mise en production sérieuse .

Exemple :

Appli-angular 6
(index.html + bundles js)

déposée sur serveur HTTP (Apache 2.2 ou Nginx ou ...) .

Navigateur -----> Serveur Http intermédiaire (nginx ou ...)
 --> partie angular (html / js) statique
 ---> partie "api-zz1" déléguée vers -----> Appli-Jee 1
 ---> partie "api-zz2" déléguée vers -----> Appli-Jee 2

Et du coup , moins besoin de paramétrages CORS.

2. Api Key

Un web service hébergé par une entreprise et rendu accessible sur internet a un certain coût de fonctionnement (courant électrique , serveurs ,) .

Pour limiter des abus (ex : appel en boucle) ou bien pour obtenir un paiement en contre partie d'une bonne qualité de service , un web service public est souvent invocable que si l'on renseigne une "api_key" (au niveau de l'URL ou bien au niveau de l'entête la requête HTTP).

Une "api_key" est très souvent de type "**uuid/guid**" .

Critères d'une api_key :

- lié à un abonnement (gratuit ou payant) , ex : compte utilisateur / compte d'entreprise
- ne doit idéalement pas être diffusé (à garder secret)
- souvent lié à un compteur d'invocations (limite selon prix d'abonnement)
- doit pouvoir être administré (régénéré si perdu/volé , ...)
et les modifications doivent pouvoir être immédiatement ou rapidement prises en compte.

Exemple :

Le site **https://fixer.io** héberge un web service REST permettant de récupérer les taux de change (valeurs de "USD" , "GBP" , "JPY" , ... vis à vis de "EUR" par défaut).

Début 2018, ce web service était directement invocable sans "api_key" .

Courant 2018, ce web service est maintenant invocable qu'avec une "**api_key**" **liée à un compte utilisateur** "*gratuit*" ou bien "*payant*" selon le mode d'abonnement (options, fréquence d'invocation,).

URL d'appel sans "api_key" : **http://data.fixer.io/api/latest**

Réponse :

```
{
  "success":false,
  "error":{"code":101,"type":"missing_access_key",
    "info":"You have not supplied an API Access Key. [Required format:
      access_key=YOUR_ACCESS_KEY]"
  }
}
```

URL d'invocation avec api_key valide :

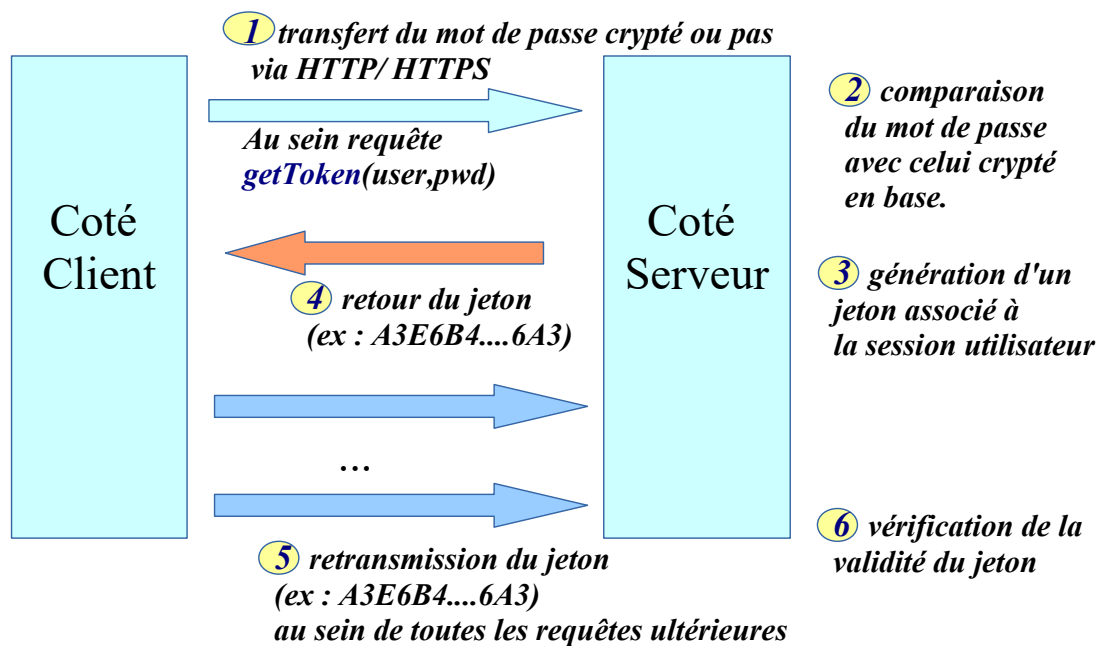
http://data.fixer.io/api/latest?access_key=26ca93ee7.....aaa27cab235

```
{
  "success":true, "timestamp":1538984646, "base":"EUR", "date":"2018-10-08",
  "rates":
  {
    "AED":4.224369,...,"DKK":7.460075,"DOP":57.311592,"DZD":136.091172,"EGP":20.596249,
    "ERN":17.250477,"ETB":31.695652,"EUR":1,"FJD":2.46956,"FKP":0.88584,"GBP":0.879667,.
    ..., "JPY":130.858498,...,"USD":1.15005,...,"ZWL":370.724343}
  }
}
```

3. Token d'authentification

3.1. Tokens : notions et principes

Jeton ("token") d'authentification valide le temps d'une session utilisateur



Plusieurs sortes de jetons/tokens

Il existe plusieurs sortes de jetons (normalisés ou pas).

Dans le cas le plus simple, un **jeton** est **généré aléatoirement** (ex : **uuid** ou ...) et sa **validation** consiste essentiellement à **vérifier son existence** en tentant de le récupérer quelque part (*en mémoire ou en base*) et éventuellement à vérifier une date et heure d'expiration.

JWT (Json Web Token) est un **format particulier de jeton** qui **comporte 3 parties** (une entête technique , un paquet d'informations en clair (ex : username , email , expiration, ...) au format JSON et une signature qui ne peut être vérifiée qu'avec la clef secrète de l'émetteur du jeton.

3.2. Bearer Token (au porteur) / normalisé HTTP

Bearer token (jeton au porteur) et transmission

Le champ **Authorization:** normalisé d'une entête d'une requête HTTP peut comporter une valeur de type ***Basic ...*** ou bien ***Bearer ...***

Le terme anglais "**Bearer**" signifiant "**au porteur**" en français indique que la simple possession d'un jeton valide par une application cliente devrait normalement, après transmission HTTP, permettre au serveur d'autoriser le traitement d'une requête (après vérification de l'existence du jeton véhiculé parmi l'ensemble de ceux préalablement générés et pas encore expirés).

NB: Les "bearer token" sont utilisés par le protocole "O2Auth" mais peuvent également être utilisés de façon simple sans "O2Auth" dans le cadre d'une authentification "sans tierce partie" pour API REST.

NB2 : un "bearer token" peut éventuellement être au format "JWT" mais ne l'est pas toujours (voir rarement) en fonction du contexte.

3.3. JWT (Json Web Token)



Structure jeton "JWT / Json Web Token"



Example:

[eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJ0b3B0YWVwYyZ29tIiwiaXhwIjojNDI2NDIwODAwLCJodHRwOi8vdG9wdGFsLmNvbS9qd3RfY2xhaW1zL2l2X2FkbWluIjp0cnVILCJjb2lwYW55IjoieGV9dGFsIiwiaXdlc29tZSI6dHJlZX0.yRQYnWzskCZUxPwaQupWkiUzKELZ49eM7oWxAQK_ZXw](#)

NB: "iss" signifie "issuer" (émetteur) , "iat" : issue at time
 "exp" correspond à "date/heure expiration" . Le reste du "payload"
 est libre (au cas par cas) (ex : "company" et/ou "email" , ...)

3.4. Mise en oeuvre concrète de la sécurité des WS-REST

- Spring-Security et plugins/extensions utiles

ou bien

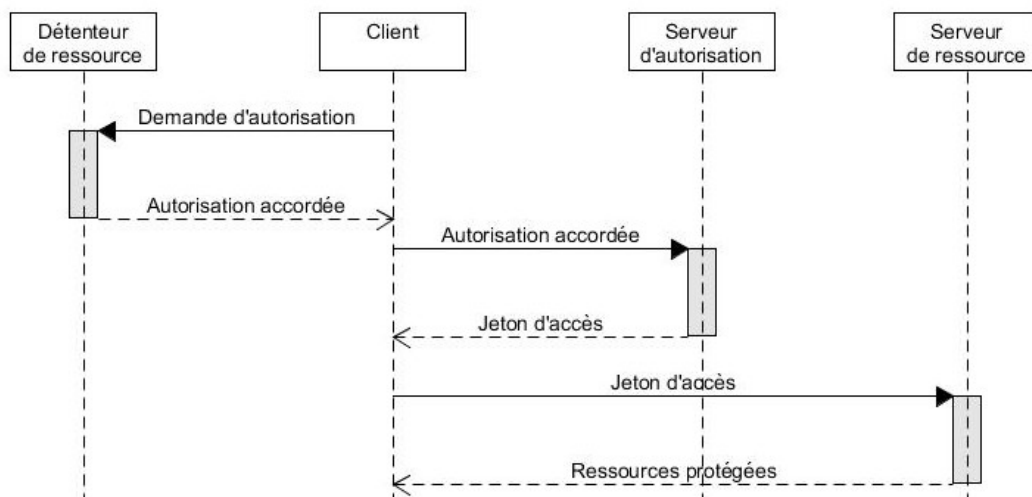
- (pour NodeJs /Express) "**passportjs**" avec **plugins** pour "jeton JWT" et "OAuth2"

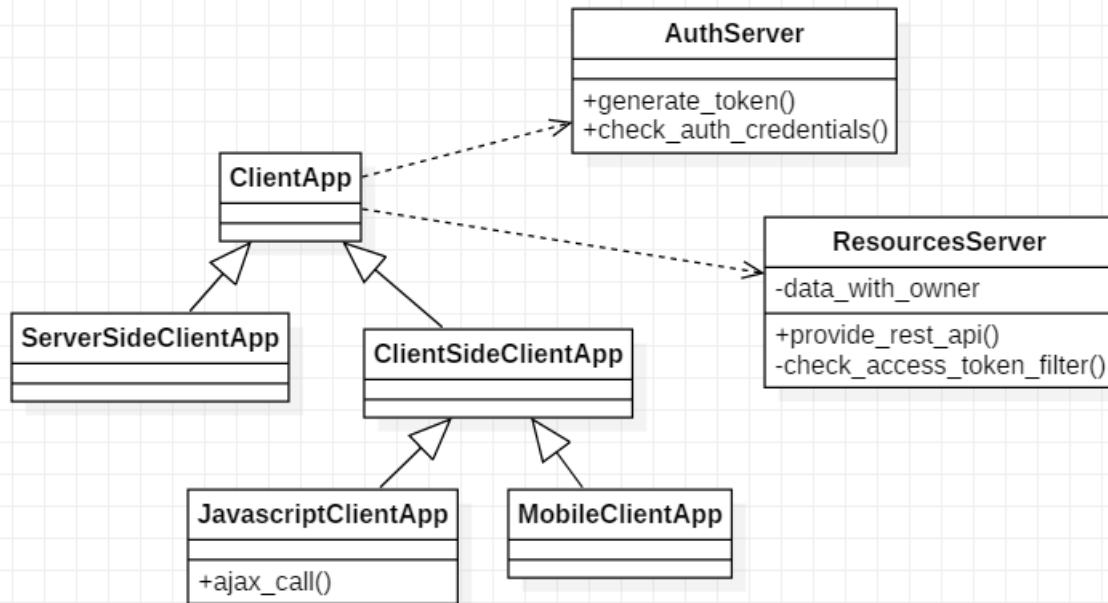
• • • •

4. OAuth2 (présentation)

Norme/Protocole "OAuth2"

OAuth (Open Authorization) existant en versions "1" et "2", est une norme (RFC 6749 et 6750) qui correspond à un **protocole de "délégation d'autorisation"**. Ceci permet par exemple d'autoriser une application cliente à accéder à une API d'une autre application (ex : FaceBook, Twitter, ...) de façon à accéder à des données protégées.



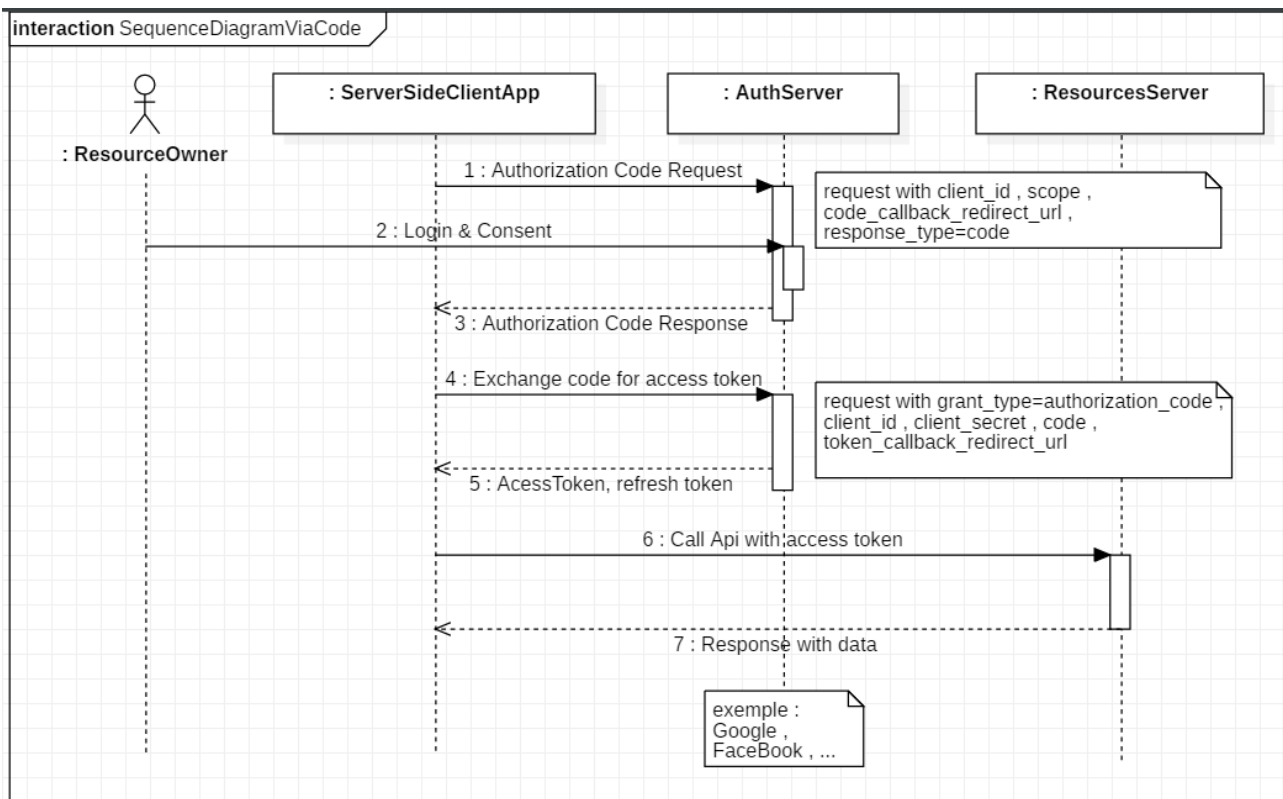


"ClientApp", "AuthServer", "ResourceServer" correspondent à des rôles !!!
 Une même application peut être à la fois "ClientApp et ResourceServer"
 et plein d'autres variantes sont possibles .

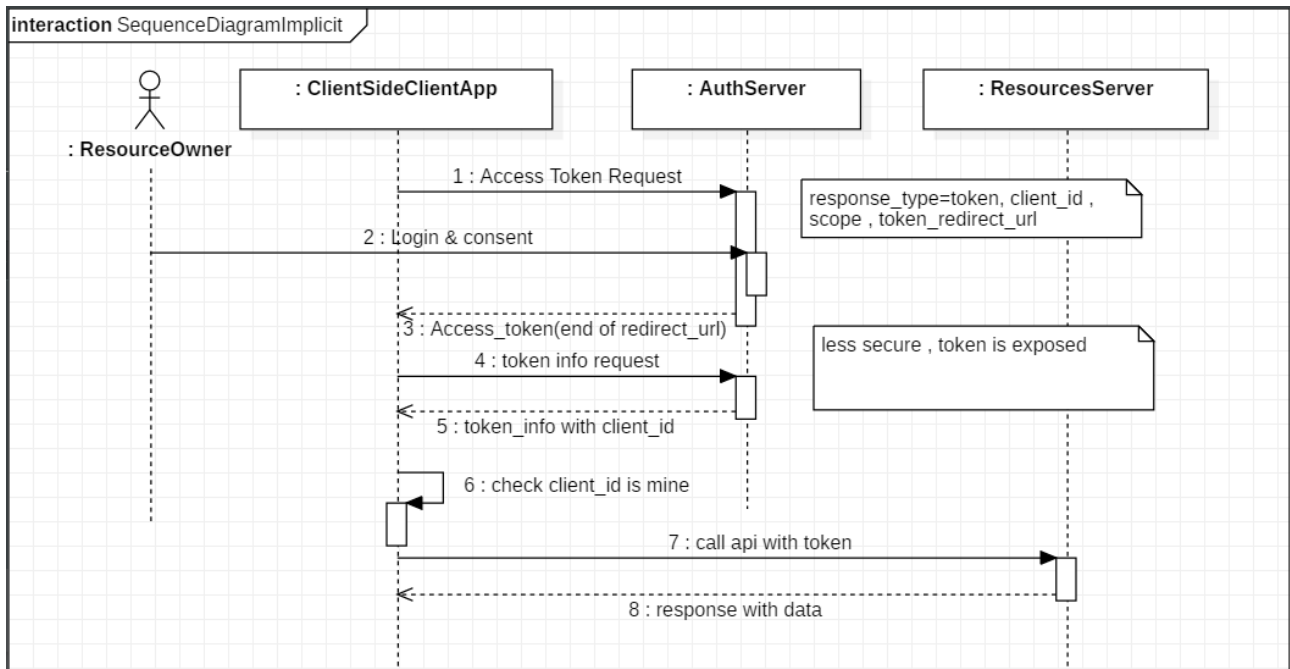
Il existe **4 grandes variantes de OAuth2** (valeurs possibles de **grant_type**).

Ces 4 variantes correspondent aux work**flows** typiques suivants :

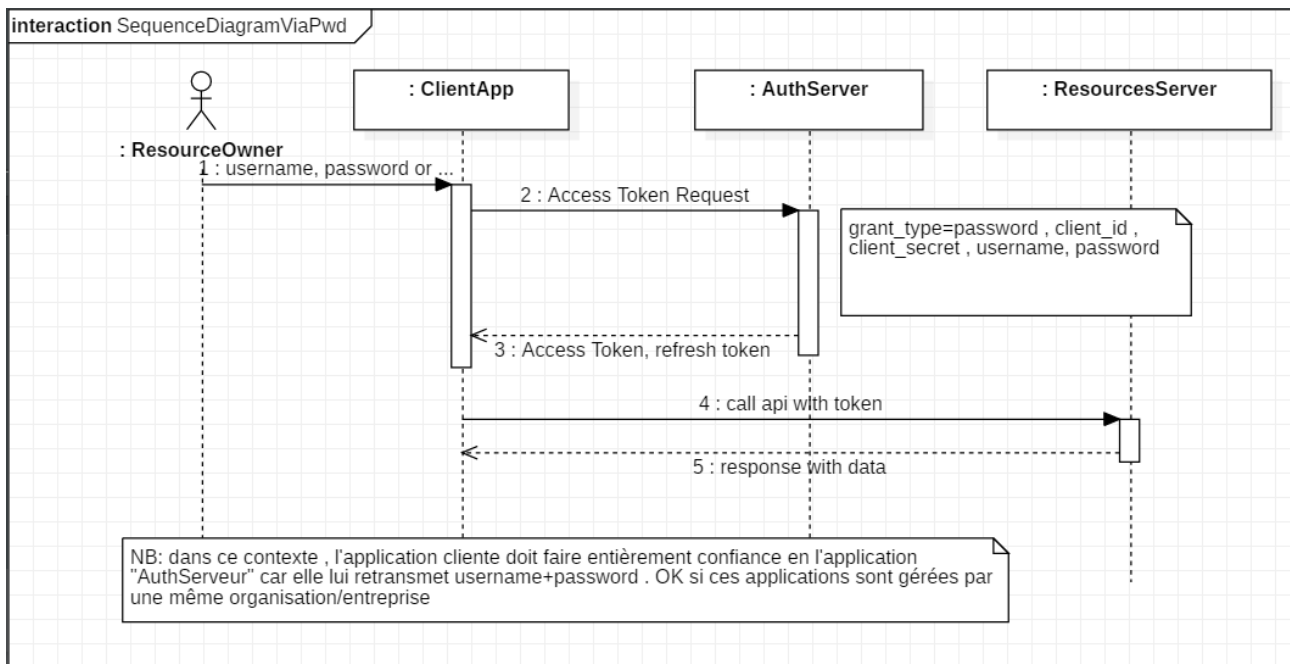
4.1. grant type="code" (for server-side SSO , ...)



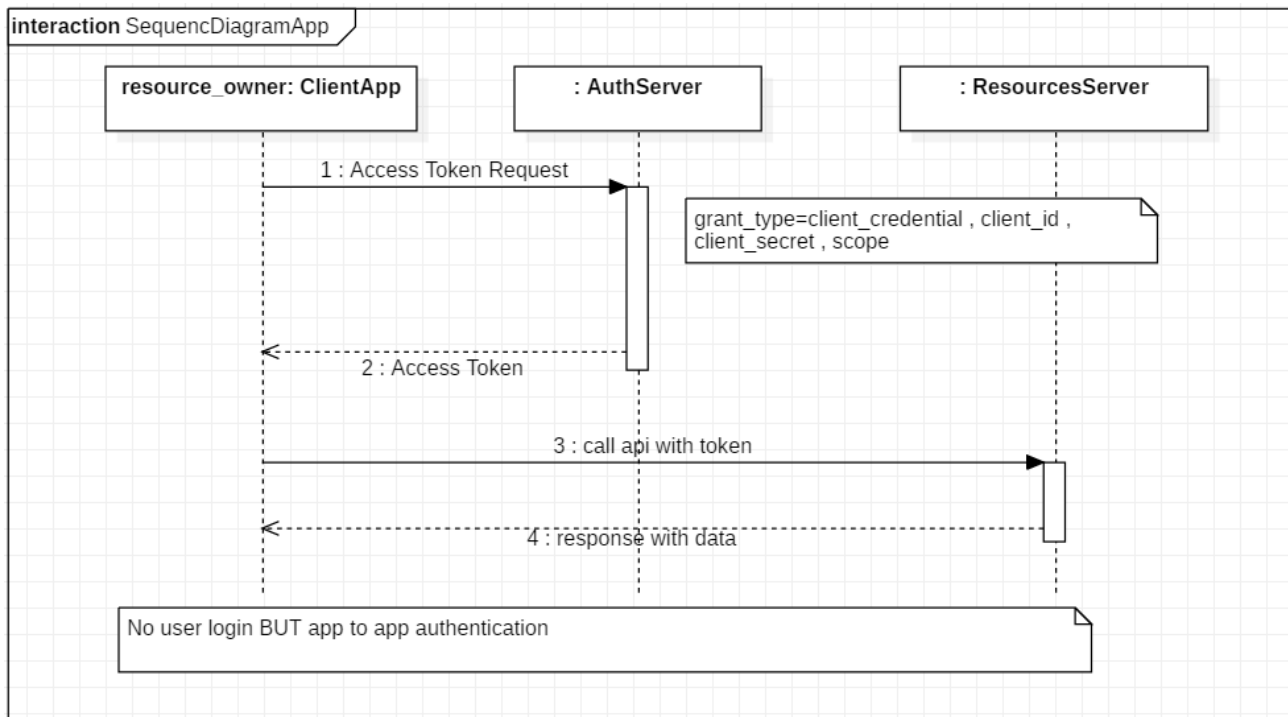
4.2. grant_type="implicit" (for client-side / js / ...)



4.3. grant_type="password" (delegate in same organization)



4.4. grant_type="client_credential" (app to app auth,no user login)



5. Eventuelle Monétisation d'une API REST

Si une entreprise souhaite louer l'utilisation d'une api REST à d'autres entreprises clientes , le mode opératoire classique tourne autour des "API KEY" vus précédemment. Certaines solutions "Gateway Api" gèrent en interne les "api key" et certains éléments associés .

6. Monitoring d'API REST

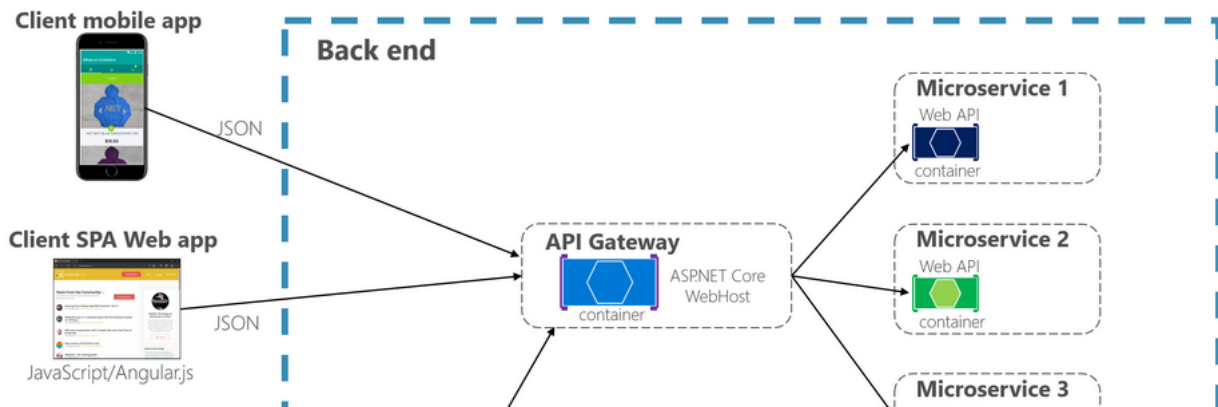
Il est souvent intéressant de surveiller l'activité d'une API REST pour s'assurer du bon fonctionnement et améliorer le marketing associé .

Il existe pour cela plein de solutions en "cloud privé" et "cloud public" .
L'aspect "monitoring" peut souvent être pris en charge par le serveur "gateway" en façade : cette sorte de "ESB exposé à l'extérieur du SI" doit idéalement servir à plein de choses pour éviter "3600 intermédiaires ajoutant une fonctionnalité à la fois" .

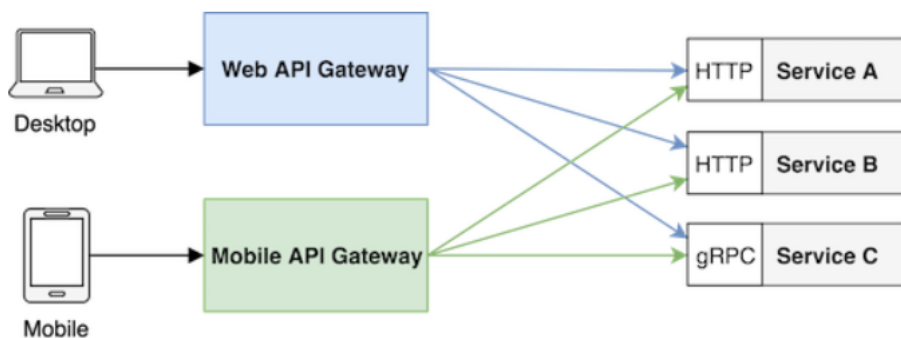
7. Gateway API

Le réseau interne (intranet) d'une entreprise est déjà en soi un domaine à bien sécuriser. Lorsque certains services sont exposés à l'extérieur du "S.I.", il faut prendre encore plus de précautions (attaques de toutes sortes : force brute , déni de service , ...)

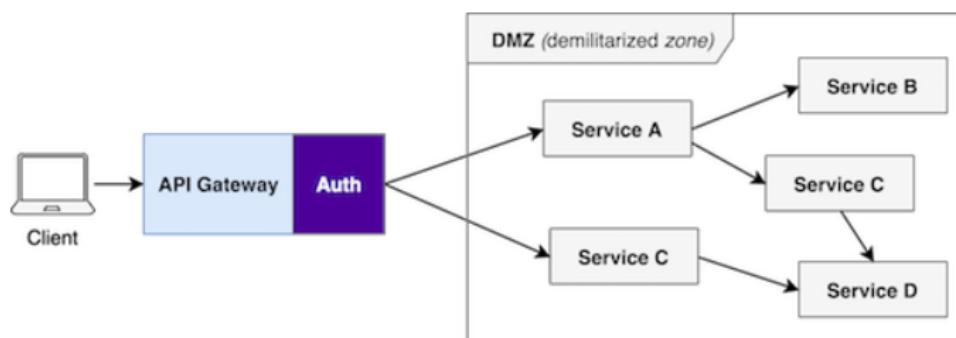
La solution la plus courante utilisée pour protéger l'accès aux services exposés à tout internet consiste à mettre en place des **serveurs d'intermédiation spécialisés appelés "Gateway"** .



On peut envisager des "gateway" spécifiques aux types de clients externes ("desktop" ou "mobiles") :



On peut éventuellement concentrer la sécurisation des web-services sur un "gateway" :



Si le principal rôle d'un "Gateway" est la sécurisation , beaucoup de solutions "gateway" apportent "dans la façade exposée" tous un tas de valeurs ajoutées intéressantes :

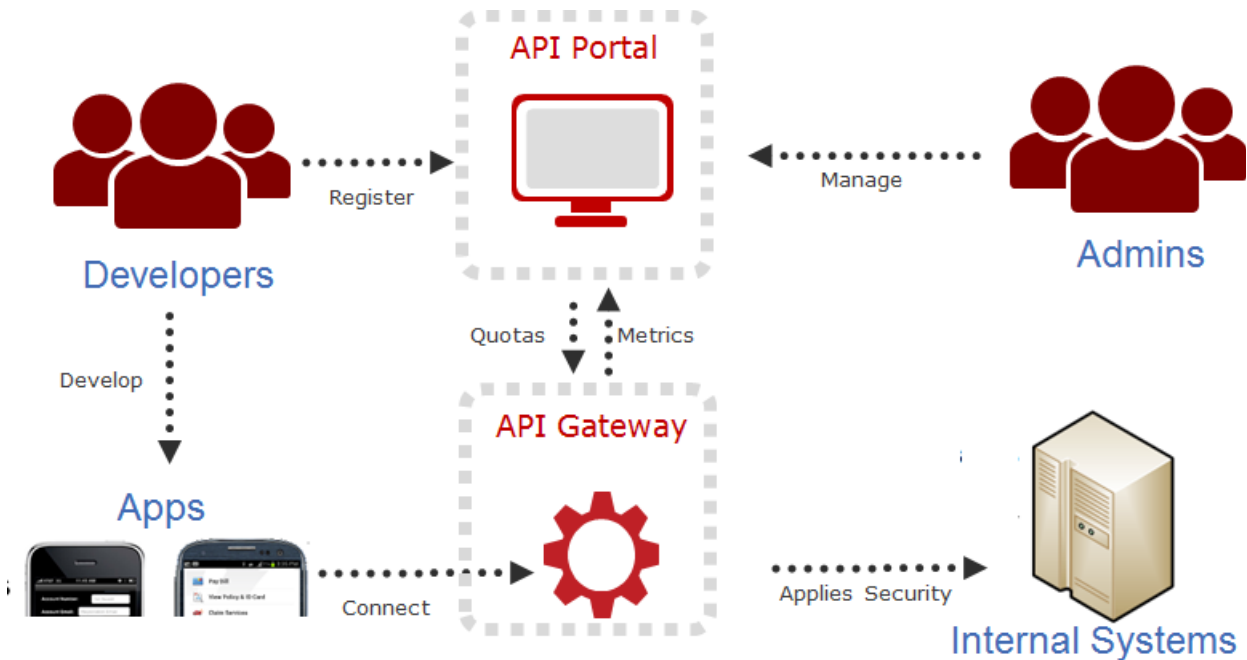
- Authentification et autorisation
- Intégration de la découverte de service
- Mise en cache des réponses
- Stratégies de nouvelle tentative, disjoncteur et qualité de service (QoS)
- Limitation du débit
- Équilibrage de charge
- Journalisation, suivi, corrélation
- Transformation des en-têtes, chaînes de requête et revendications
- Liste verte d'adresses IP

NB : Dans le cadre d'un cloud public , la plateforme d'hébergement "cloud" (ex : Microsoft Azure ou ...) peut directement en charge le "Gateway API" (avec plus ou moins de fonctionnalités associées) .

8. Portail développeur

Un **portail développeur** est un site web :

- exposé au réseau externe (vu par les développeurs des entreprises clientes)
- **documentant à fond la collection d' API exposée par le "API gateway"** (URL , exemples , descriptions SWAGGER , ...)



Un développeur (d'une entreprise "cliente") pourra ainsi :

- Déterminer quelles API sont disponibles
- Parcourir la documentation sur les API
- S'inscrire à leur propre clé d'API, qu'ils reçoivent immédiatement, nécessaire pour créer des applications
- ...

NB : Certaines "plateformes cloud" offre ce genre de service (ex : Amazon Web Services)

ANNEXES

VII - Annexe – Cloud computing (essentiel)

1. Le "Cloud Computing" (présentation)

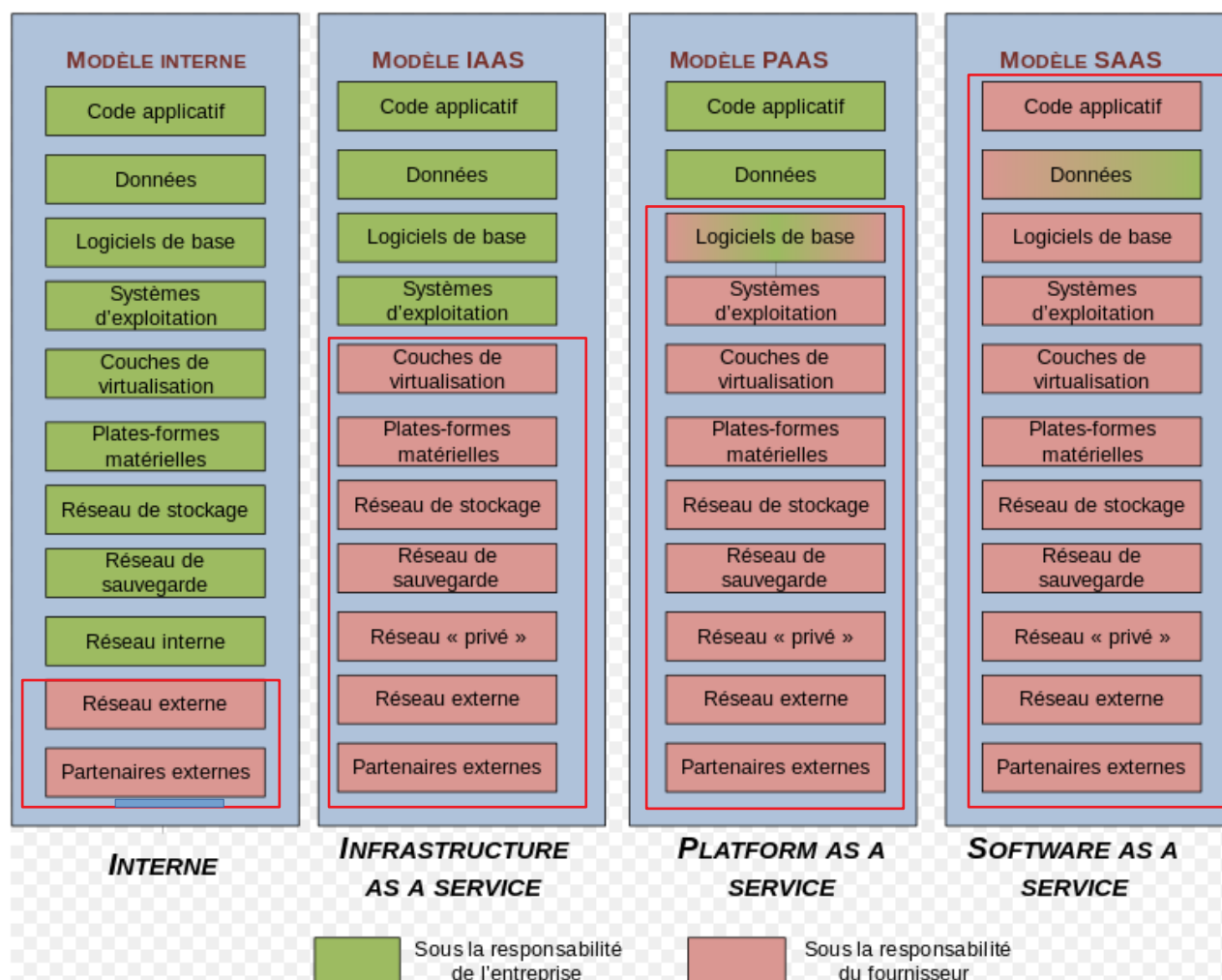
1.1. Architecture / terminologie "Cloud"

Un nuage (anglais *cloud*) est une métaphore désignant un ensemble de matériel, de raccordements réseau et de logiciels qui fournit des services sophistiqués que les individus et les collectivités peuvent exploiter à volonté depuis n'importe où dans le monde.

Le *cloud computing* est un basculement de tendance : au lieu d'obtenir de la puissance de calcul par acquisition de matériel et de logiciel, le consommateur se sert de puissance mise à disposition par un fournisseur via Internet.

Les caractéristiques essentielles d'un nuage sont la disponibilité mondiale en self-service, l'élasticité, l'ouverture, la mutualisation et le paiement à l'usage :

- *ressources en self-service*, et adaptation automatique à la demande. La capacité de stockage et la puissance de calcul sont adaptées automatiquement au besoin d'un consommateur. Ce qui contraste avec la technique classique des hébergeurs où le consommateur doit faire une demande écrite à son fournisseur en vue d'obtenir une augmentation de la capacité - demande dont la prise en compte nécessite évidemment un certain temps. En *cloud computing* la demande est automatique et la réponse est immédiate.
- *ouverture*. Les services de *cloud computing* sont mis à disposition sur l'Internet, et utilisent des techniques standardisées qui permettent de s'en servir aussi bien avec un ordinateur qu'un téléphone ou une tablette.
- *mutualisation*. La mutualisation permet de combiner des ressources hétérogènes (matériel, logiciel, trafic réseau) en vue de servir plusieurs consommateurs à qui les ressources sont automatiquement attribuées. La mutualisation améliore la scalabilité et l'élasticité et permet d'adapter automatiquement les ressources aux variations de la demande.
- *paiement à l'usage*: la quantité de service consommée dans le *cloud* est mesurée, à des fins de contrôle, d'adaptation des moyens techniques et de facturation



Niveaux (<i>aas=as a service</i>)	Caractéristiques
SAAS (Software aas)	<p>On loue un logiciel entièrement géré par le fournisseur (ServApp + Base_données ,).</p> <p>On a simplement besoin de postes clients avec des navigateurs internet .</p> <p>Seule la sauvegarde des données (copies locales) est facultativement à gérer.</p>
PAAS (Platform aas)	<p>On loue une plate-forme logicielle d'un certain type (ex : Serv App Java/Jee Tomcat" + bases MySql ou) ou bien "Node-js" + "MongoDB") entièrement gérée par un fournisseur d'hébergement puis l'on déploie de code d'une application à faire fonctionner</p>
IAAS (Infrastructure aas)	<p>On loue des machines virtuelles auprès d'un fournisseur d'hébergement (ex : Gandi , OVH) avec par exemple un O.S. "linux" , puis on installe la base de données et le serveur d'applications que l'on préfère (ex : Mysql et Tomcat7) pour ensuite installer et faire fonctionner des applications.</p>
Tout en interne	On achète les ordinateurs "serveurs" , les "systèmes

d'exploitation" , les logiciels "SGBDR" , "compta" , "...". On administre tout nous même.
--

Plus on s'appuie sur une couche basse , plus on a de liberté pour choisir l'O.S. , les logiciels que l'on préfère mais on doit gérer soit même plein de choses.

Plus on s'appuie sur une couche haute , moins on a de choses à gérer mais plus on devient dépendant de l'offre (large ou pas) et de la qualité de service (bonne ou pas) du fournisseur.

La maîtrise d'œuvre sera souvent intéressée par une offre Iaas ou Paas . Cela lui permettra de fournir sa "valeur ajoutée" sous forme de logiciel à déployer et faire fonctionner de façon à ensuite satisfaire les besoins d'une maîtrise d'ouvrage.

Spécificité importante d'une application Saas :

- bien gérer les sauvegardes des données
- bien gérer la sécurité et la disponibilité
- bien gérer la séparation des données "client1" , "client2" , ... , "clientN" si une application Saas est partagée/louée par plusieurs clients

1.2. Technologies pour le "cloud computing"

Windows Azure (en tant que plate-forme),

→ Plateforme "Cloud" de *Microsoft*

Amazon Web Services (AWS)

→ Offre cloud de la société Amazon (historiquement une des premières offres)

Google App Engine

→ API ...

1.3. Exemples d'hébergements "cloud"

Gandi.net	Essentiellement Iaas , un peu Paas (php , node-js, ...)
Windows Azure (hébergement)	Microsoft
"CloudLayer / SoftLayer"	IBM

VIII - Annexe – DevOps

1. DevOps

1.1. Présentation de devops (origine, concepts, ...)

DevOps

Devops est la concaténation des trois premières lettres du mot anglais development (*développement*) et de l'abréviation usuelle ops du mot anglais operations (*exploitation*), deux fonctions de la gestion des systèmes informatiques qui ont souvent des objectifs contradictoires (*ex : nouvelles fonctionnalités apportant de l'instabilité / fiabilité exigée en exploitation*)

Le **devops** est un mouvement en ingénierie informatique et une pratique technique visant à l'**unification** du développement logiciel (*dev*) et de l'administration des infrastructures informatiques (*ops*), notamment l'administration système. (*source wikipedia*)

Origine et principes de DevOps

Apparu autour de 2007 en Belgique avec Patrick Debois, le mouvement Devops se caractérise principalement par la promotion de l'automation et du suivi (monitoring) de toutes les étapes de la création d'un logiciel, depuis le développement, l'intégration, les tests, la livraison jusqu'au déploiement, l'exploitation et la maintenance des infrastructures.

Conférences "DevOpsDays" (la première en 2009 à Gand / Belgique)

Autre terminologie: "**Agile Infrastructure**"

Les principes Devops soutiennent des cycles de développement plus courts, une augmentation de la fréquence des déploiements et des livraisons continues, pour une meilleure atteinte des objectifs économiques de l'entreprise. *(source wikipedia)*

Avant "DevOps"

Univers "Dev" et "Ops" traditionnellement séparés dans les années 1995-2010 pour les raisons suivantes :

- beaucoup de complexités à gérer → besoin de se spécialiser .
- solutions à bases de serveurs (ex : ServApp JEE) à administrer et dans lesquels on déploie des applications (à développer).
- peu d'automatisation , contextes différents (O.S. , ...)

Objectifs "dev"

. apporter les changements nécessaires au moindre coût et le plus vite possible
souvent au détriment de la qualité lorsque des retards viennent mettre les délais du planning en péril

Objectifs "ops"

. garantir la stabilité du système.
. se concentrer sur contrainte qualité,
. besoin de temps et de moyens
. contrôler sévèrement les changements apportés au système

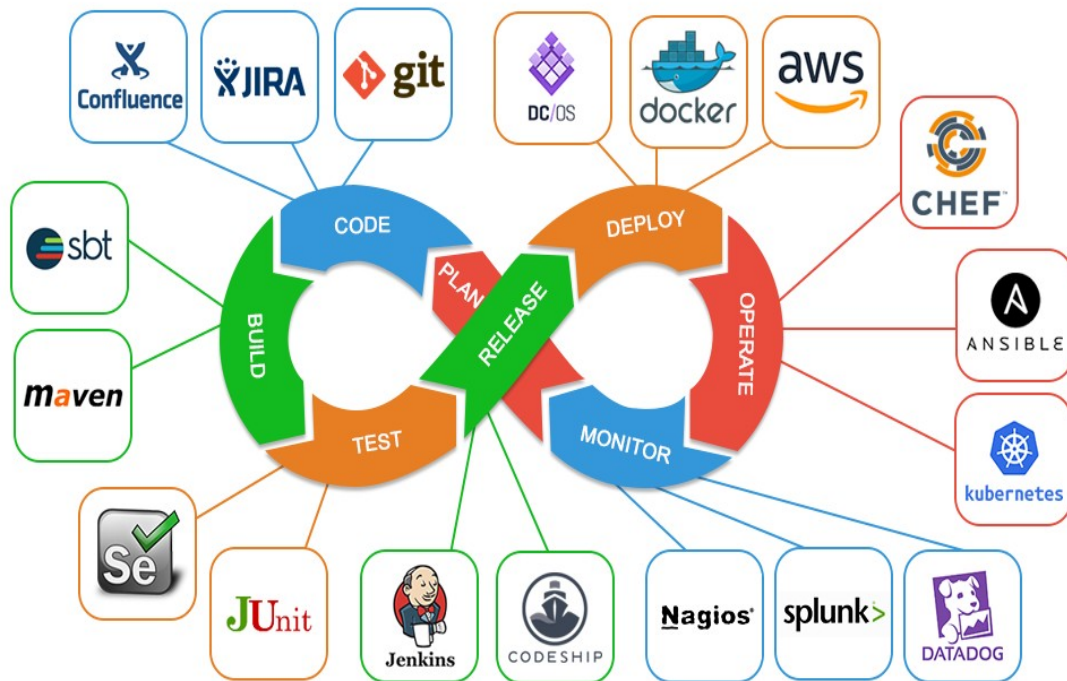
Vers "DevOps"

- changement de paradigme ("séparation" → "unification")
- les équipes "dev" et "ops" ne doivent pas se rejeter la faute en cas de problèmes mais doivent collaborer et s'entraider de façon à ne former qu'une seule "méga-équipe" .
- automatisations (intégration continue , déploiements, ...)
- cycles courts (dev, tests, intégration , mise en prod, ...)
- bonnes métriques / supervisions
- applications des méthodes agiles (tests , communications, ...)
- alignement avec les technologies "cloud" et "web"



1.2. cycle de vie devops

Cycle de vie "DevOps"



Première phases du cycle "devops"

1. Planification (et modélisation , suivi)



2. Codage/implémentation (avec tests unitaires)



Phases suivantes du cycle "devops"

3. Build (de "dev" ou "release/prod") (assemblage , pré-released)

maven

éventuels profils de 'dev' / 'prod'
avec "bases" et ...
... plus ou moins simplifié(e)s



4. Tests d'intégration (intégration continue)

Exécution automatique des tests dans le but d'avoir un feedback concernant la qualité du code produit en cours de déploiement



JUnit



Phases suivantes du cycle "devops"

5. Release (de ou vers "prod")

maven

profile de 'prod'
avec vraies "bases" / "serveurs" / ...



Environnement au sein duquel l'identification
des anomalies doit idéalement pouvoir se faire aisément.

6. Deploy (packaging & déploiement)



Dernieres phases du cycle "devops"

7. Operate (fonctionnement en "production")



fonctionnement en cluster ,
...

8. Monitoring (surveillance & métriques)

surveillance/exploitation ,
remontées/interprétations des mesures , gestion des logs,
Redimensionnement éventuel (élasticité cloud, ...)

1.3. Devops , concrètement ...

Concrètement "DevOps" c'est avant tout :

- De l'agilité au niveau de l'infrastructure (et pas que sur le papier en théorie)
- Ça se traduit par des automatisations efficaces (intégration continue et déploiement de conteneurs "docker") de façon à ce que les développeurs puissent assez facilement construire des "briques" prêtes à être déployées sans trop de d'ajustements/paramétrages .
- Et inversement , certains informaticiens (très techniques) peuvent packager des environnements sophistiqués (avec sécurité , ...) correspondant aux contraintes réelles de production sous forme d'images "docker" de façon à ce que le développeur puisse facilement y intégrer et tester de nouvelles fonctionnalités.

IX - Annexe – Docker (présentation / essentiel)

1. Docker : Vue d'ensemble

Docker est une technologie de conteneur logiciel (nouvelle déclinaison plus optimisée de la virtualisation) .

1.1. Docker et notion de conteneur

Micro-conteneurs "Docker"



Technologie légère et efficace de virtualisation
avec comme principaux apports :

- **Coûts optimisés** (car moins consommateur)
- **Déploiements rapides**
- **Portatibilité** (linux/windows , isolation interne , compatibilité externe)

Principes de la "conteneurisation"



Vis à vis de plusieurs conteneurs co-localisés, permet un partage d'un système d'exploitation hôte unique, avec ses ressources (*fichiers binaires, librairies, pilotes de périphériques, mémoire vive, ...*)



Conteneur = **environnement d'exécution** logicielle **complet** comportant :

- micro o.s. (debian, centos ou ...)
- librairies / dépendances
- logiciel de base (ex : mysql, node, ...)
- configuration logicielle
- le code d'une application ou d'un service

1.2. Introduction, présentation

Docker permet de créer des environnements (appelées containers) de manière à isoler des applications.

Docker repose sur le *kernel* **Linux** et sur deux de ses grandes fonctionnalités :

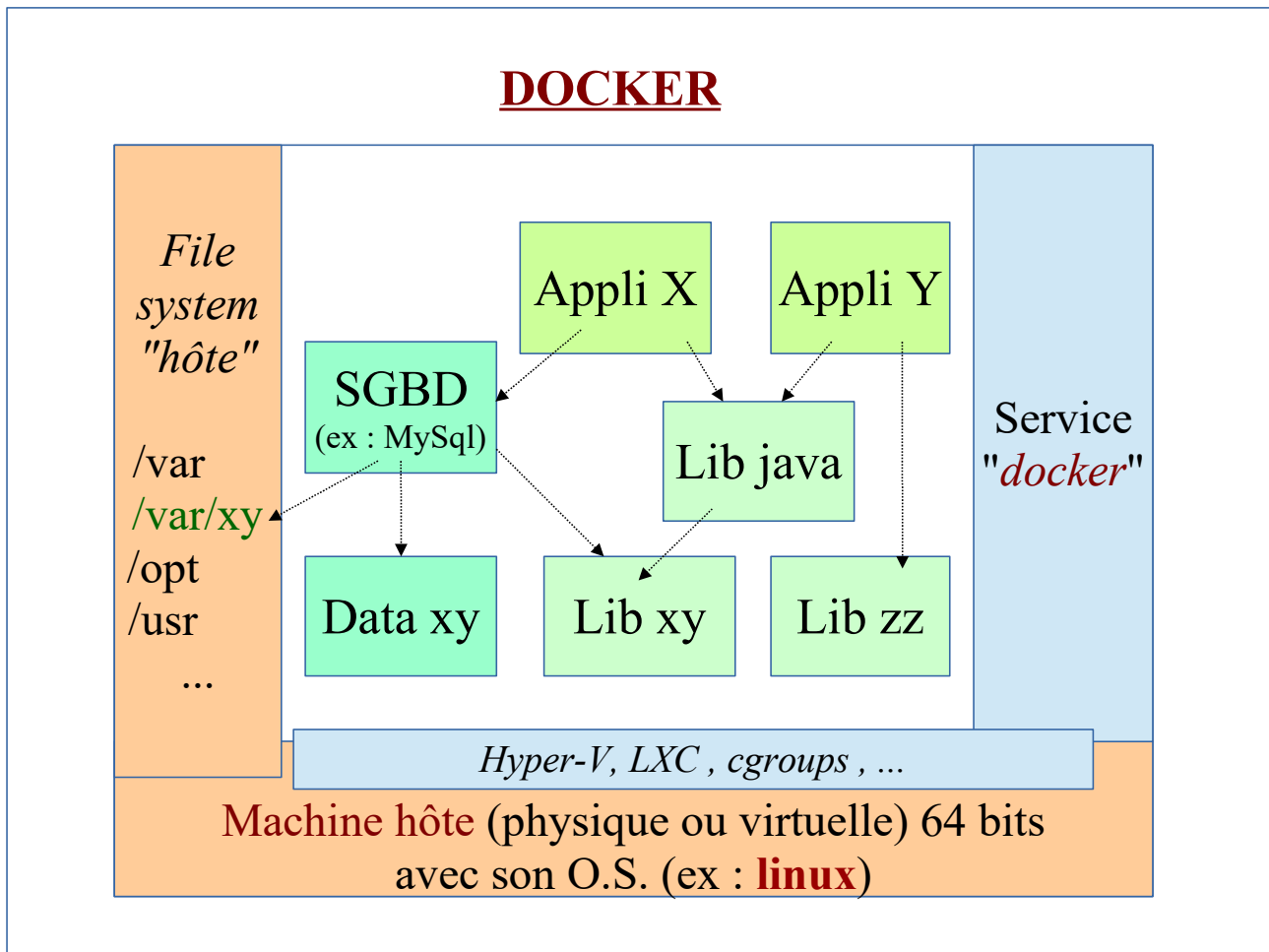
- * les containers LXC : L'idée est d'exécuter une tâche dans un environnement isolé.
- * **cgroups** qui va avoir pour objectif de gérer/partager les ressources (utilisation de la RAM, CPU entre autres).

Historique: **Docker** est un produit développé par la société du même nom. Initialement développé par un ingénieur français, **Solomon Hykes**, le produit a été dévoilé en mars 2013 .

En seulement 3 ans, Docker est devenue une technologie incontournable pour la mise en œuvre du "cloud computing" et est utilisée (avec quelques variantes) par tous les géants de l'informatique (Google, Microsoft, ...).



l'icône de docker est une baleine (Moby Dock) .



Les premières versions de "**Docker**" ne fonctionnaient que sur "**Linux**".

Au fil des années qui passent, "**Docker**" est de **mieux en mieux intégré sur le système "windows" de Microsoft**.

Historiquement, en 2015, pour faire fonctionner Docker sur windows 10 Home (ou autre), il fallait utiliser une machine virtuelle spéciale "**Docker ToolBox**" (basée sur VirtualBox et "**boot2docker**" : un noyau linux minimaliste mais suffisant pour faire fonctionner en RAM un grand nombre de conteneurs "docker"). Ceci n'était pas prévu pour la production mais pour un environnement de développement.

Aujourd'hui "**Docker**" fonctionne nativement sur la version "**professionnel**"/"**serveur**" de **windows 10 et partiellement sur un "windows 10" basique**.

Ceci dit "Docker for windows" est prévu pour faire parfaitement fonctionner des images "dockers pour windows".

Faire fonctionner une image "docker/linux" sous windows est pour l'instant partiellement réalisable (avec plein de restrictions).

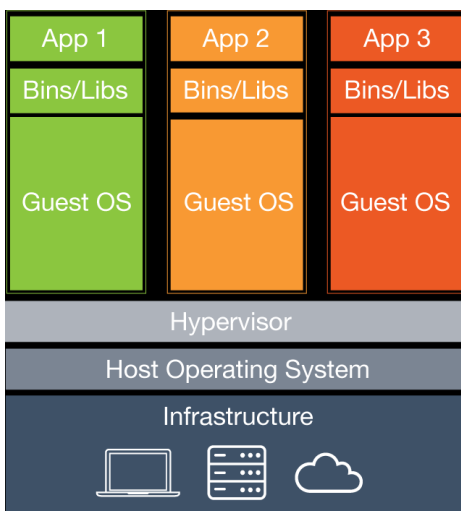
Les futures versions de l'OS "windows" de Microsoft vont évoluer (en se rapprochant de linux) de façon à ce qu'un conteneur "docker/linux" puisse à terme fonctionner sur "windows" ;

1.3. "Container" vs "VM"

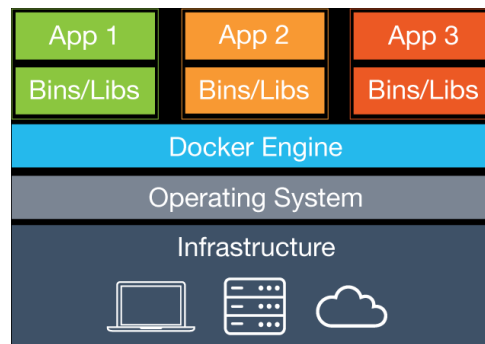
Bien que proche d'une VM (Machine Virtuelle) , **Docker n'est pas une VM** . Sa technologie de "Container" permet de bien mieux partager les ressources d'une machine physique (RAM , CPU , ...)

Evolution de la virtualisation ("VM" → "Conteneur")

V.M. (lourdes)
avec "**O.S. complets internes**"
gérées par hyperviseur
(ex : VmWare , VirtualBox)



Conteneurs (très légers)
avec "applications et dépendances"
gérées moteur de conteneurs
(ex : Docker)



A bas niveau l'ordinateur (cpu/ ram / *bios*, ...) doit être prévu pour partager les ressources matérielles entre plusieurs conteneurs lorsqu'il sera en partie contrôlé par l' **hyperviseur "docker"** .

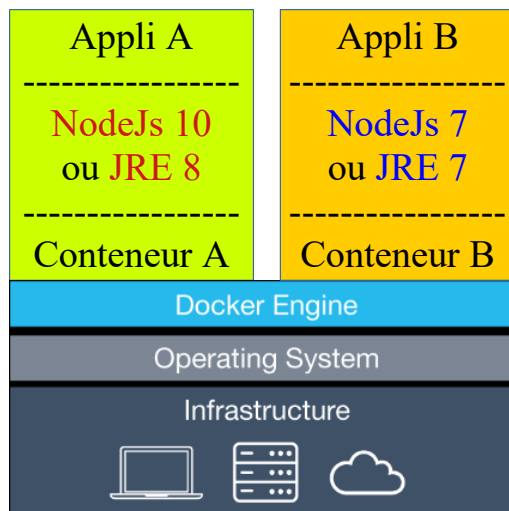
Chaque conteneur sera isolé des autres d'un point de vu "logique" et "fonctionnement interne". Par contre, les différents "conteneurs" qui vont s'exécuter sur un même ordinateur vont partager très efficacement les ressources de l'ordinateur (RAM, CPU, ...) :

- **très faible sur-consommation mémoire** (comparée à celle d'une VM)
- **démarrage très rapide** (comparé à celui d'une VM)

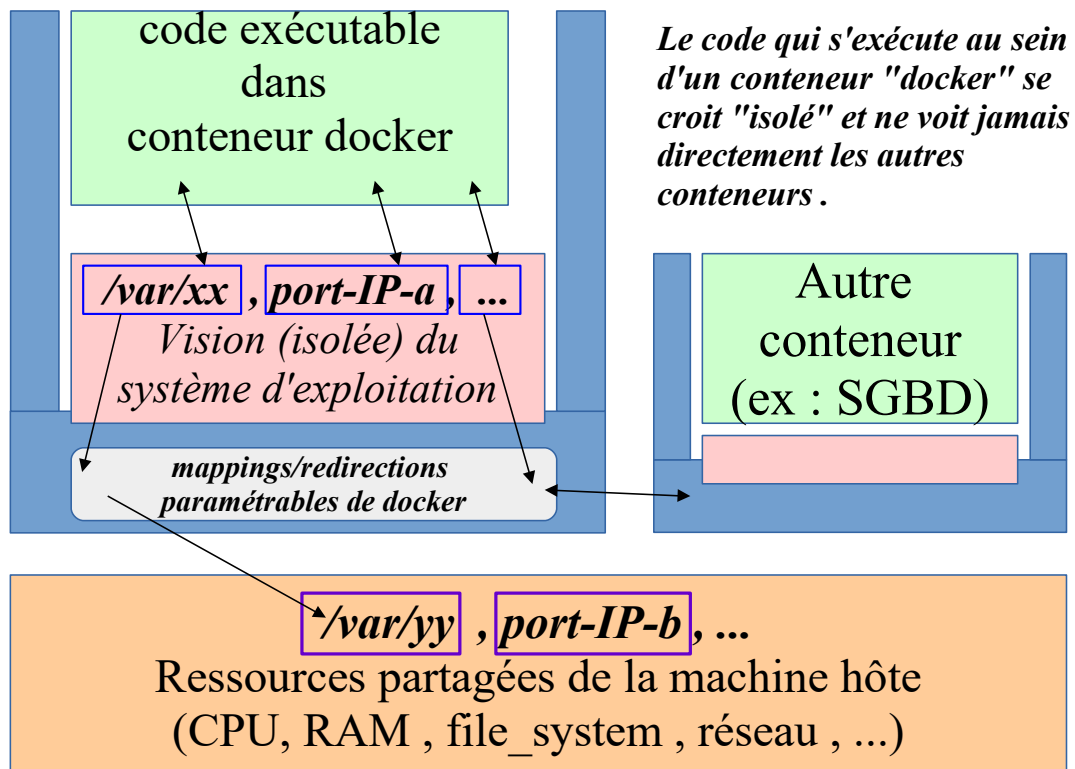
NB : Etant donné que toutes les distributions linux partagent un même type de noyau , un conteneur docker "debian" peut sans problème fonctionner sur un système "redHat" ou "centOs" et vice versa.

1.4. Isolation des conteneurs

Isolation des conteneurs



"Isolation virtuelle"

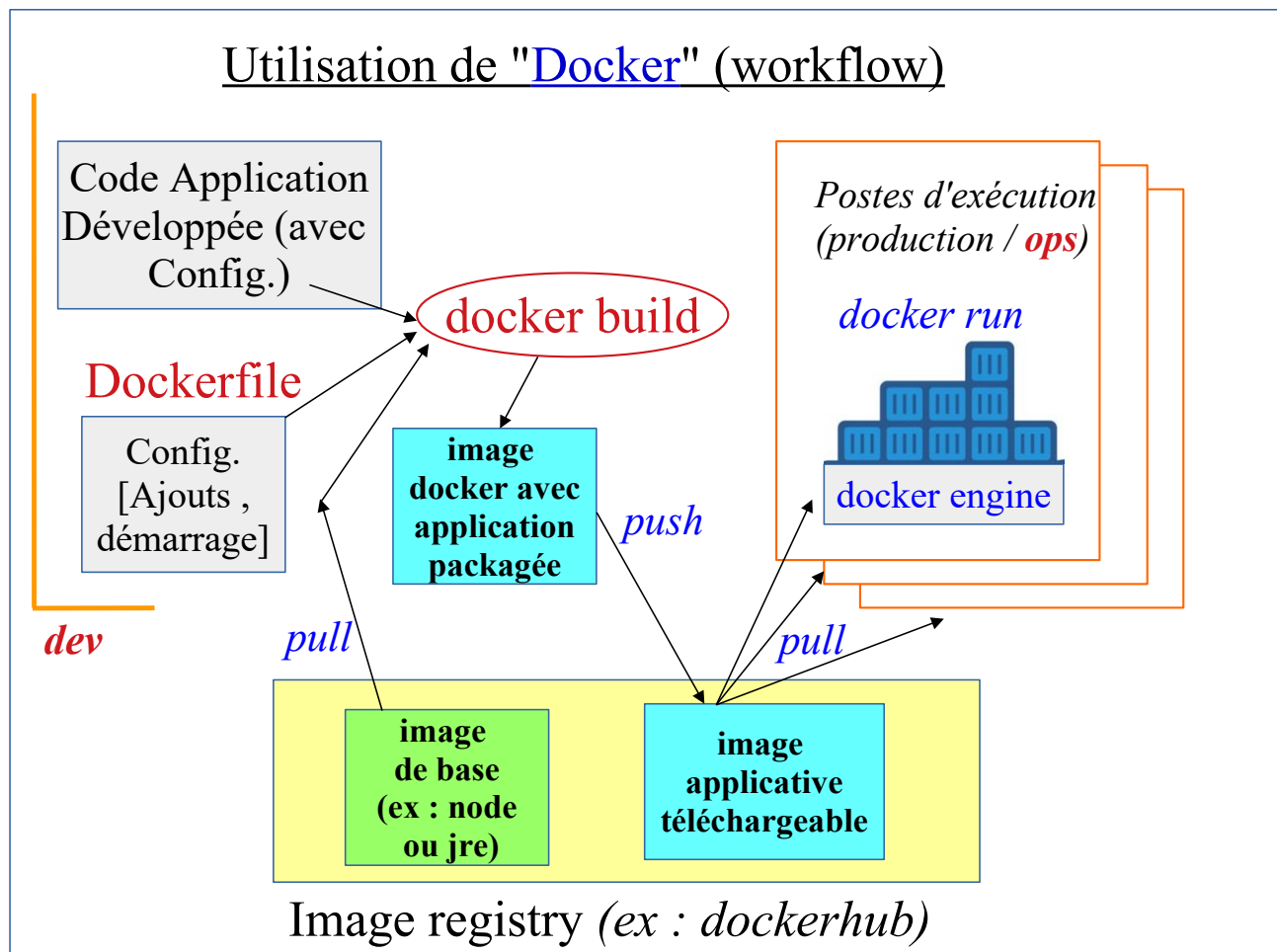


1.5. Référentiel d'images "docker" prêtes à l'emploi

Le site <https://hub.docker.com> permet de télécharger (gratuitement) toute une **série d'images "docker" prêtes à être installée sur n'importe quelle distribution linux** (fedora, redhat, debian, ubuntu, ...).

On y trouve une centaine d'images officielles ("debian", "mysql", "postgresql", "mongo", "node", "tomcat", "jenkins", ...) et une multitudes d'images "non officielles".

1.6. "Docker workflow" (utilisation classique)



1.7. Spécificités de "docker"

Spécificités d'un conteneur "docker"

Points forts :

- démarrage assez rapide
- instanciations , démarrages , arrêts , suppressions facilement contrôlables à distance (via api REST) .
Ce qui apporte beaucoup de souplesse dans la gestion des ressources (vite allouées, vite libérées)
- isolation , portabilité (tous linux et bientôt windows)

Points délicats (à avoir en tête)

- tailles de certaines images quelquefois importantes (presque 1Go)
- nécessite réseau à relativement haut débit
- attention aux conflits potentiels entre les numéros de ports (heureusement reconfigurables) entre les éléments internes de différents conteneurs co-localisés.

Les 2 versions actuelles de "docker"

- Docker CE : Community Edition
- Docker EE : Enterprise Edition

1.8. Commandes de "docker"

Avant la version 1.13 certaines commandes étaient pas très structurées (ex : "docker ps" pour afficher la liste des conteneurs et "docker images" pour afficher la liste des images .

A partir de la version 1.13 , on peut :

- soit lancer les commandes historiques (des toutes premières versions) "docker ps, dockers images"
- soit lancer des commandes plus structurées respectant le format

"docker typeElement nomAction" :

exemples :

docker container ls , docker container run

docker image ls , docker image pull ...

Actions les plus courantes (applicables ou pas en fonction du type d'élément):

ls	lister
rm	remove (supprimer)
start / stop	démarrer / arrêter
run	lancer (créer et démarrer) (souvent un container)
inspect	afficher les détails
exec	lancer une commande (selon le contexte)
logs	afficher les logs

docker --help

Usage: docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Options:

- config string Location of client config files (default "/root/.docker")
- D, --debug Enable debug mode
- H, --host list Daemon socket(s) to connect to
- l, --log-level string Set the logging level ("debug"|"info"|"warn"|"error"|"fatal") (default "info")
- tls Use TLS; implied by --tlsverify
- tlscacert string Trust certs signed only by this CA (default "/root/.docker/ca.pem")
- tlscert string Path to TLS certificate file (default "/root/.docker/cert.pem")
- tlskey string Path to TLS key file (default "/root/.docker/key.pem")
- tlsverify Use TLS and verify the remote
- v, --version Print version information and quit

Management Commands:

- builder Manage builds
- config Manage Docker configs
- container Manage containers
- engine Manage the docker engine

image	Manage images
network	Manage networks
node	Manage Swarm nodes
plugin	Manage plugins
secret	Manage Docker secrets
service	Manage services
stack	Manage Docker stacks
swarm	Manage Swarm
system	Manage Docker
trust	Manage trust on Docker images
volume	Manage volumes

Commands:

attach	Attach local standard input, output, and error streams to a running container
build	Build an image from a Dockerfile
commit	Create a new image from a container's changes
cp	Copy files/folders between a container and the local filesystem
create	Create a new container
diff	Inspect changes to files or directories on a container's filesystem
events	Get real time events from the server
exec	Run a command in a running container
export	Export a container's filesystem as a tar archive
history	Show the history of an image
images	List images
import	Import the contents from a tarball to create a filesystem image
info	Display system-wide information
inspect	Return low-level information on Docker objects
kill	Kill one or more running containers
load	Load an image from a tar archive or STDIN
login	Log in to a Docker registry
logout	Log out from a Docker registry
logs	Fetch the logs of a container
pause	Pause all processes within one or more containers
port	List port mappings or a specific mapping for the container
ps	List containers
pull	Pull an image or a repository from a registry
push	Push an image or a repository to a registry
rename	Rename a container
restart	Restart one or more containers
rm	Remove one or more containers
rmi	Remove one or more images
run	Run a command in a new container
save	Save one or more images to a tar archive (streamed to STDOUT by default)
search	Search the Docker Hub for images
start	Start one or more stopped containers
stats	Display a live stream of container(s) resource usage statistics
stop	Stop one or more running containers
tag	Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
top	Display the running processes of a container
unpause	Unpause all processes within one or more containers
update	Update configuration of one or more containers
version	Show the Docker version information
wait	Block until one or more containers stop, then print their exit codes

Run 'docker COMMAND --help' for more information on a command.

X - Annexe – WS-REST (NodeJs/Express)

1. Ecosystème node+npm

node (nodeJs) est un **environnement d'exécution javascript** permettant essentiellement de :

- compartimenter le code à exécuter en **modules** (import/export)
- exécuter du code en mode "**appels asynchrones non bloquants** + callback" (sans avoir recours à une multitudes de threads)
- exécuter directement du code javascript sans avoir à utiliser un navigateur web

npm (*node package manager*) est une sous partie fondamentale de node qui permet de :

- **télécharger et gérer des packages utiles à une application** (bibliothèques réutilisables)
- télécharger et utiliser des utilitaires pour la phase de développement (ex : grunt , jasmine , gulp , ...)
- **prendre en compte les dépendances entre packages** (téléchargements indirects)
- générer éventuellement de nouveaux packages réutilisables (à déployer)
-

node est à peu près l'équivalent "javascript" d'une machine virtuelle java.

npm ressemble un peu à maven de java : téléchargement des bibliothèques , construction d'applications.

Un **projet basé sur npm** se configure avec le fichier *package.json* et les packages téléchargés sont placés dans le sous répertoire **node_modules** .

Principales utilisations/applications de node :

- application "serveur" en javascript (répondant à des requêtes HTTP)
- application autonome (ex : StarUML2 = éditeur de diagrammes UML , ...)
-

2. Express

Express correspond à un des packages téléchargeables via npm et exécutables via node.

La **technologie "express"** permet de répondre à des requêtes HTTP et ressemble un peu à un Servlet java ou à un script CGI .

A fond **basé sur des mécanismes souples et asynchrones** (avec "routes" et "callbacks") , "**express**" permet de coder assez facilement/efficacement des applications capables de :

- **générer dynamiquement des pages HTML** (ou autres)
- mettre en œuvre des **web services "REST"** (souvent au format "JSON") .
- prendre en charge les détails du protocoles **HTTP** (authentification "basic" et/ou "bearer" , autorisations "CORS" ,)
-

"express" est souvent considéré comme une technologie de bas niveau lorsque l'on la compare à d'autres technologies "web / coté serveur" telles que ASP , JSP , PHP , ...

"express" permet de construire et retourner très rapidement une réponse HTTP (avec tout un tas de

paramétrages fin si nécessaire) . Pour tout ce qui touche au format de la réponse à générer , il faut utiliser des technologies complémentaires (ex : templates de pages HTML avec remplacements de valeurs) .

3. Exemple élémentaire "node+express"

first_express_server.js

```
//modules to load:
var express = require('express');

var app = express();

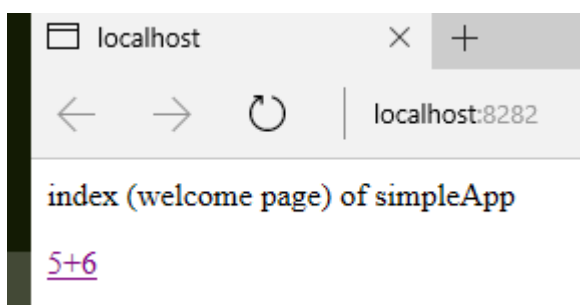
app.get('/', function(req, res , next) {
  res.setHeader('Content-Type', 'text/html');
  res.write("<html> <body>");
  res.write('<p>index (welcome page) of simpleApp</p>');
  res.write('<a href="addition?a=5&b=6">5+6</a>');
  res.write("</body></html>");
  res.end();
});

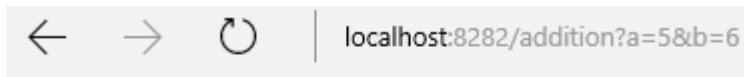
//GET addition?a=5&b=6
app.get('/addition', function(req, res , next) {
  a = Number(req.query.a);    b = Number(req.query.b);
  resAdd = a+b;
  res.setHeader('Content-Type', 'text/html');
  res.write("<html> <body>");
  res.write('a=' + a + '<br/>');  res.write('b=' + b + '<br/>');
  res.write('a+b=' + resAdd + '<br/>');
  res.write("</body></html>");
  res.end();
});

app.listen(8282 , function () {
  console.log("simple express node server listening at 8282");
});
```

lancement: node first_express_server.js

via <http://localhost:8282> au sein d'un navigateur web , on obtient le résultat suivant :





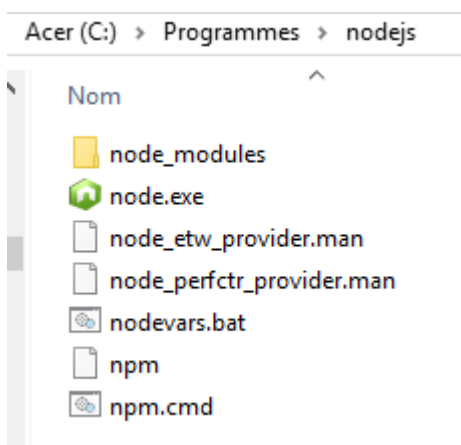
4. Installation de node et npm

Téléchargement de l'installateur **node-v10.15.3-x64.msi** (ou autre) depuis le site officiel de nodeJs (<https://nodejs.org>)

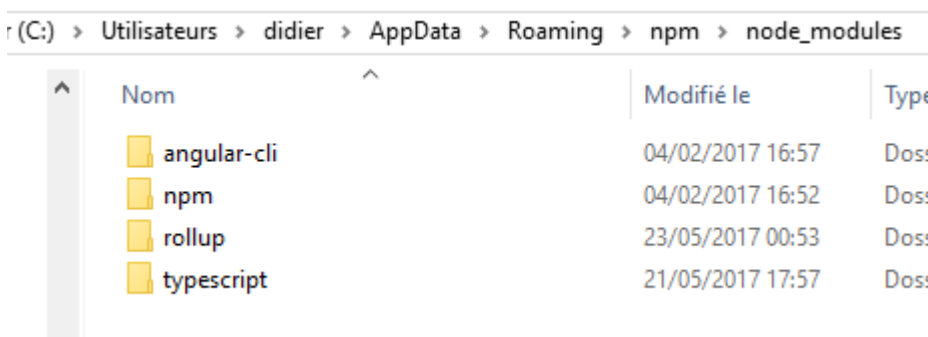
Lancer l'installation et se laisser guider par les menus.

Cette opération permet sous windows d'installer node et npm en même temps .

Sur une machine windows 64bits , nodejs s'installe par défaut dans **C:\Program Files\nodejs**



Et le répertoire pour les installations de packages en mode "global" (-g) est par défaut **C:\Users\username\AppData\Roaming\npm\node_modules**



Vérification de l'installation (dans un shell "CMD") :

node --version
v10.15.3 (ou autre)

npm --version
6.4.1 (ou autre)

5. Configuration et utilisation de npm

5.1. Initialisation d'un nouveau projet

Un développeur utilise généralement npm dans le cadre d'un projet spécifique (ex : xyz). Après avoir créé un répertoire pour ce projet (ex : C:\tmp\temp_nodejs\xyz) et s'être placé dessus, on peut lancer la *commande interactive* **npm init** de façon à **générer un début de fichier "package.json"**

Exemple de fichier *package.json* généré :

```
{
  "name": "xyz",
  "version": "1.0.0",
  "description": "projet xyz",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "didier",
  "license": "ISC"
}
```

5.2. installation de nouveau package en ligne de commande :

```
npm install --save express
npm install --save mongoose
```

permet de télécharger les packages "express" et "mongoose" (ainsi que tous les packages indirectement nécessaires par analyse de dépendances) dans le sous répertoire **nodes_modules** et de mettre à jour la liste des dépendances dans le fichier **package.json** :

```
.... ,
"dependencies": {
  "express": "^4.17.0",
  "mongoose": "^4.11.0"
},
....
```

Sans l'option **--save** (ou son alias **-s**), les packages sont téléchargés mais le fichier **package.json** n'est pas modifié.

Par défaut, c'est la dernière version du package qui est téléchargé et utilisé.

Il est possible de choisir une **version spécifique** en la précisant après le caractère @ :

npm install --save mongoose@4.10
ou bien (autre exemple) :
npm install --save mongodb@2.0.55

Autre procédure possible :

- 1) **éditer** le fichier **package.json** en y ajoutant des dépendances (au sein de la partie "dependencies") :
exemple :

```
"dependencies": {  
  "express": "^4.15.3",  
  "markdown": "^0.5.0",  
  "mongoose": "^4.10.8"  
}
```
- 2) lancer **npm install** (ou *npm update* ultérieurement) **sans argument**

Ceci permet de lancer le téléchargement et installation du package "mardown" dans le sous répertoire **node_modules** .

Installation de packages utilitaires (pour le développement) :

Si l'on souhaite ensuite expliciter une dépendance de "développement" au sein d'un projet , on peut utiliser l'option **--save-dev** de **npm install** de façon ajouter celle ci dans la partie "devDependencies" de package.json :

npm install --save-dev grunt

```
.... ,  
"devDependencies": {  
  "grunt": "^1.0.1"  
}  
...
```

5.3. Installation en mode global (-g)

L'option **-g** de **npm install** permet une installation en mode global : le package téléchargé sera installé dans **C:\Users\username\AppData\Roaming\npm\node_modules** sous windows 64bits (ou ailleurs sur d'autres systèmes) **et sera ainsi disponible (en mode partagé) par tous les projets** .

Le mode global est souvent utilisé pour installer des packages correspondant à des "utilitaires de développement" (ex : grunt) .

Exemple :

npm install -g grunt

6. Utilisation basique de node

hello_world.js

```
console.log("hello world");
```

node *hello_world.js*

7. WS REST élémentaire avec node+express

7.1. Récupérer des données entrantes au format JSON

La récupération des valeurs JSON véhiculés en mode **POST** ou **PUT** dans le corps (*body*) de la requête entrante s'effectue avec la syntaxe **req.body** et nécessite la préparation et l'enregistrement d'un "*bodyParser*" :

```
...
var bodyParser = require('body-parser');
//import * as bodyParser from 'body-parser';
...
//support parsing of JSON post data
var jsonParser = bodyParser.json() ;
app.use(jsonParser);
...
//POST ... with body { "firstname" : "Jean" , "lastname" : "Bon" }
app.post('/xyz', function(req : Request, res : Response) {
  var user = req.body ; //as javascript object
});
```

NB : il existe une variante de la méthode `app.post()` où l'on peut passer un "*bodyParser*" spécifique en second paramètre (pour certains cas pointus/spécifiques):

```
// POST /login gets urlencoded bodies :
app.post('/login', urlencodedParser, function (req, res) {
  res.send('welcome, ' + req.body.username)
```

```

}))

// POST /api/users gets JSON bodies :
app.post('/api/users', jsonParser, function (req, res) {
  // use user in req.body
})

```

Dans le cas d'une api homogène (quasiment tout en JSON) , il est tout de même plus simple de paramétrer l'utilisation par défaut d'un bodyParser JSON via app.use() :

```

var jsonParser = bodyParser.json() ;
app.use(jsonParser)

```

7.2. Renvoyer des données/réponses au format JSON

La fonction prédéfinie **res.send(jsObject)** effectue en interne a peu près les opérations suivantes :

```

res.setHeader('Content-Type', 'application/json');
res.write(JSON.stringify(jsObject));
res.end();

```

Cette méthode ".send()" est donc tout à fait appropriée pour retourner la réponse "JSON" à un appel de Web Service REST .

7.3. Renvoyer si besoin des statuts d'erreur (http)

Via éventuelle fonction utilitaire :

```

function sendDataOrError(err,data,res){
  if(err==null) {
    if(data!=null)
      res.send(data);
    else res.status(404).send(null);//not found
  }
  else res.status(500).send({error: err});//internal error (ex: mongo access)
}

```

Via "errorHandler" :

...

7.4. Exemple

```

var express = require('express');
var myGenericMongoClient = require('./my_generic_mongo_client');

```

```

var app = express();
...

function sendDataOnError(err,data,res){
    if(err==null) {
        if(data!=null)
            res.send(data);
        else res.status(404).send(null);//not found
    }
    else res.status(500).send({error: err});//internal error (ex: mongo access)
}

// GET (array) /minibank/operations?numCpt=1
app.get('/minibank/operations', function(req, res,next) {
    myGenericMongoClient.genericFindList('operations', { 'compte' :
    Number(req.query.numCpt) },
        function(err,tabOperations){
            sendDataOnError(err,tabOperations,res);
        });
});

// GET /minibank/comptes/1
app.get('/minibank/comptes/:numero', function(req, res,next) {
    function(req, res,next) {
        myGenericMongoClient.genericFindOne('comptes',
            { '_id' : Number(req.params.numero) },
            function(err,compte){
                sendDataOnError(err,compte,res);
            });
    });
});

app.listen(8282 , function () {
    console.log("rest server listening at 8282");
});

```

NB : `req.query.pxy` récupère la valeur d'un paramètre http en fin d'URL (`?pxy=valXy&pzz=valZz`)
`req.params.pxy` récupère la valeur d'un paramètre logique (avec:) en fin d'URL

dans cet exemple , `myGenericMongoClient.genericFind....()` correspond à un élément d'un module utilitaire qui récupère des données dans une base mongoDB .

8. Avec mode post et authentication minimaliste

```

var express = require('express');
var bodyParser = require('body-parser'); //dépendance indirecte de express via npm
var app = express();

var myGenericMongoClient = require('./my_generic_mongo_client');

var uuid = require('uuid'); //to generate a simple token

```

```

app.use(bodyParser.json()); // to parse JSON input data and generate js object : (req.body)
app.use(bodyParser.urlencoded({ extended: true}));

...

// POST /minibank/verifyAuth { "numClient" : 1 , "password" : "pwd1" }
app.post('/minibank/verifyAuth', function(req, res,next) {
  var verifAuth = req.body; // JSON input data as jsObject with ok = null
  console.log("verifAuth :" +JSON.stringify(verifAuth));
  if(verifAuth.password == ("pwd" + verifAuth.numClient) ){
    verifAuth.ok= true;
    verifAuth.token=uuid.v4();
    //éventuelle transmission parallèle via champ "x-auth-token" :
    res.header("x-auth-token", verifAuth.token);
    //+stockage dans une map pour verif ulterieure : ....
  }
  else {
    verifAuth.ok= false;
    verifAuth.token = null;
  }
  res.send(verifAuth); // send back with ok = true or false and token
});

// GET /minibank/comptes/1
app.get('/minibank/comptes/:numero',
  displayHeaders, verifTokenInHeaders /*un peu sécurisé*/,
  function(req, res,next) {
    myGenericMongoClient.genericFindOne('comptes', { '_id' : Number(req.params.numero) },
      function(err,compte){
        sendDataOnError(err,compte,res);
      });
  });

function sendDataOnError(err,data,res){
  if(err==null) {
    if(data!=null)
      res.send(data);
    else res.status(404).send(null);//not found
  }
  else res.status(500).send({error: err});//internal error (ex: mongo access)
}

//var secureMode = false;
var secureMode = true;

function extractAndVerifToken(authorizationHeader){
  if(secureMode==false) return true;
  /*else*/
  if(authorizationHeader!=null ){
    if(authorizationHeader.startsWith("Bearer")){
      var token = authorizationHeader.substring(7);
      console.log("extracted token:" + token);
    }
  }
}

```

```

        //code extremement simplifié ici:
        //idealement à comparer avec token stocké en cache (si uuid token)
        //ou bien tester validité avec token "jwt"
        if(token != null && token.length>0)
            return true ;
        else
            return false;
    }
    else
        return false;
}
else
    return false;
}

// verif bearer token in Authorization headers of request :
function verifTokenInHeaders(req, res, next) {
    if( extractAndVerifToken(req.headers.authorization))
        next();
    else
        res.status(401).send(null);//401=Unautorized or 403=Forbidden
}

// display Authorization in request (with bearer token):
function displayHeaders(req, res, next) {
    //console.log(JSON.stringify(req.headers));
    var authorization = req.headers.authorization;
    console.log("Authorization: " + authorization);
    next();
}

...
app.listen(8282 , function () {
    console.log("minibank rest server listening at 8282");
});

```

9. Autorisations "CORS"

```

var express = require('express');
var app = express();
...

```

```
// CORS enabled with express/node-js :
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
  //ou avec "www.xyz.com" à la place de "*" en production
  res.header("Access-Control-Allow-Headers",
    "Origin, X-Requested-With, Content-Type, Accept");
  next();
});

...
...
app.get(...) , app.post(...) , ...

app.listen(8282 , function () {
  console.log("rest server with CORS enabled listening at 8282");
});
```

XI - Annexe – Essentiel XML/XSD (pour SOAP)

1.1. XML

XML signifiant *eXtended Markup Language* est un **langage standardisé (w3c)** permettant d'**encoder des données structurées** de *manière hiérarchique*.

XML est **eXtensible** car il permet (contrairement à HTML) d'utiliser des noms de balises quelconques (ex: chapitre , region ,) .

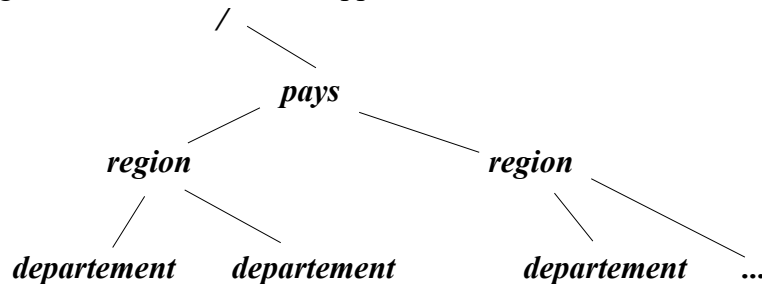
Exemple:

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<?xml-stylesheet type="text/xslt" href='fic_style.xslt' ?>
<!-- commentaire -->
<pays nom='France' capitale='Paris' >
  <region nom='Haute-Normandie' >
    <departement nom='Eure' num='27' prefecture='Evreux' />
    <departement nom='Seine Maritime' num='76' prefecture='Rouen' />
  </region>
  <region nom='Picardie'>
    <departement nom='Oise' num='60' prefecture='Beauvais' /> ...
  </region> ...
</pays>
```

Remarques:

Ce document xml comporte des **données** dont les **significations** sont **clairement indiquées** par les **attributs** ou par les **noms des balises englobantes**.

La **structure** globale de ce document s'apparente à celle de l'**arbre** suivant:



Aucune mise en forme (grs, italique , couleur , alignement , ...) n'apparaît directement dans le fichier xml. Par contre l'**application d'une feuille de style externe (CSS ou XSL)** permet d'obtenir facilement une présentation de ce genre:

France

Haute-Normandie

num	departement	prefecture
27	Eure	Evreux
76	Seine Maritime	Rouen

1.2. Principales caractéristiques d'XML

XML – principales caractéristiques

XML est un **META-LANGAGE** à partir duquel on peut *dériver* tout un tas de *langages particuliers* (*XHTML* , *SVG*, *MathML*, ...).

Les langages dérivés d'XML (et normalisés) sont généralement *identifiés par des namespaces* (ex: <http://www.w3.org/1999/xhtml>) .

- C'est un **LANGAGE DE DESCRIPTION DE DONNEES** comportant non seulement des **valeurs textuelles** mais également **les sémantiques de celles-ci** . Les *significations des données* sont renseignées au niveau des *noms de balises* d'encadrement (ou bien au niveau des noms de certains attributs).

- *Simple à encoder et à ré-interpréter* sur tout environnement (*Unix* , *Windows*, ...) et depuis n'importe quel langage (*C/C++*, *Java*, *Perl/Php* , ...) , XML est une technologie clef permettant de garantir une bonne **interopérabilité** entre différents systèmes informatiques.

(X)HTML est un plutôt un langage de présentation (pour affichage à l'écran) .

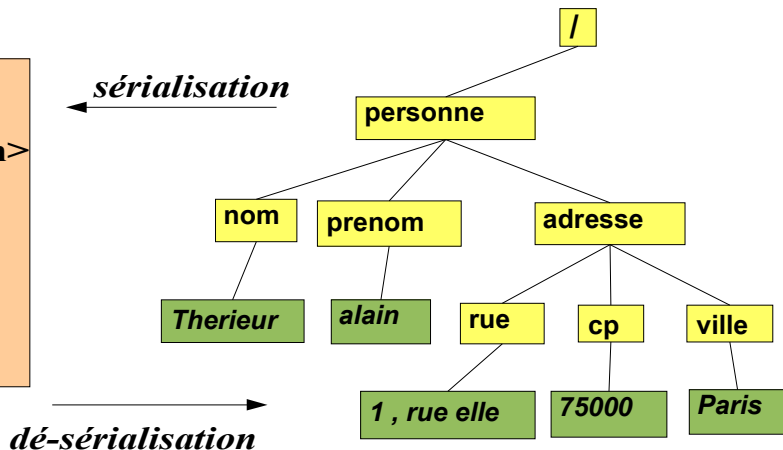
XML est plutôt un **langage de communication entre différentes applications** (des re-traitement sont presque toujours nécessaires).

XML est une **technologie transverse** que l'on retrouve partout (Présentation via feuilles de styles, Echange de données , Services WEB , B2C, B2B , Bureautique , ...).

Document XML = arbre de données

pers1.xml

```
<personne>
  <nom>Therieur</nom>
  <prenom>alain</prenom>
  <adresse>
    <rue>1, rue elle</rue>
    <cp>75000</cp>
    <ville>Paris</ville>
  </adresse>
</personne>
```



XML permet d'encoder une **structure de données arborescente** en une **suite de caractères séquentiels** (*paquet d'octets dans un fichier ou flux réseau*) .

1.3. Namespaces XML

Namespace XML et documents composés

Un **namespace XML** est un **identificateur de langage dérivé d'XML**

Cet ID prend généralement la forme d'un **URI** (*Exemple:*

'<http://www.yyy.com/Année/Norme>') [*sans téléchargement*].

Les **namespaces** permettent d'encoder des **documents XML composés** dont les différentes parties sont exprimées dans des langages distincts .

```
<h:html xmlns:h='http://www.w3.org/1999/xhtml'
  xmlns:b='http://www.yyy.com/2004/biblio' >
  <h:head>
    <h:title> bibliographie sur XML </h:title>
  </h:head>
  <h:body>
    <h:p> <b:title> XML et XSLT </b:title> </h:p>
  </h:body>
</h:html>
```

Différentes interprétations

Syntaxe générale: **xmlns:prefixe_local='nom_du_namespace'**

1.4. Documents XML bien formés

XML – documents bien formés

Un document XML est dit « **bien formé** » s'il est **syntactiquement correct** ; il doit pour cela vérifier les principales règles suivantes:

- **Toute balise ouverte doit être fermée:**

~~<p> paragraphe~~ mais <p> paragraphe </p>
 ligne ~~
~~ mais ligne
</br> ou ligne

 sachant que **<elt_vide a1='v1' ></elt_vide> <==> <elt_vide a1='v1' />**

- **Une seule balise de premier niveau englobant toutes les autres:**

~~<pers>....</pers><pers> </pers>~~ mais
<liste_pers><pers>....</pers> <pers>...</pers></liste_pers>

- **Pas de chevauchement de balises mais des imbrications claires:**

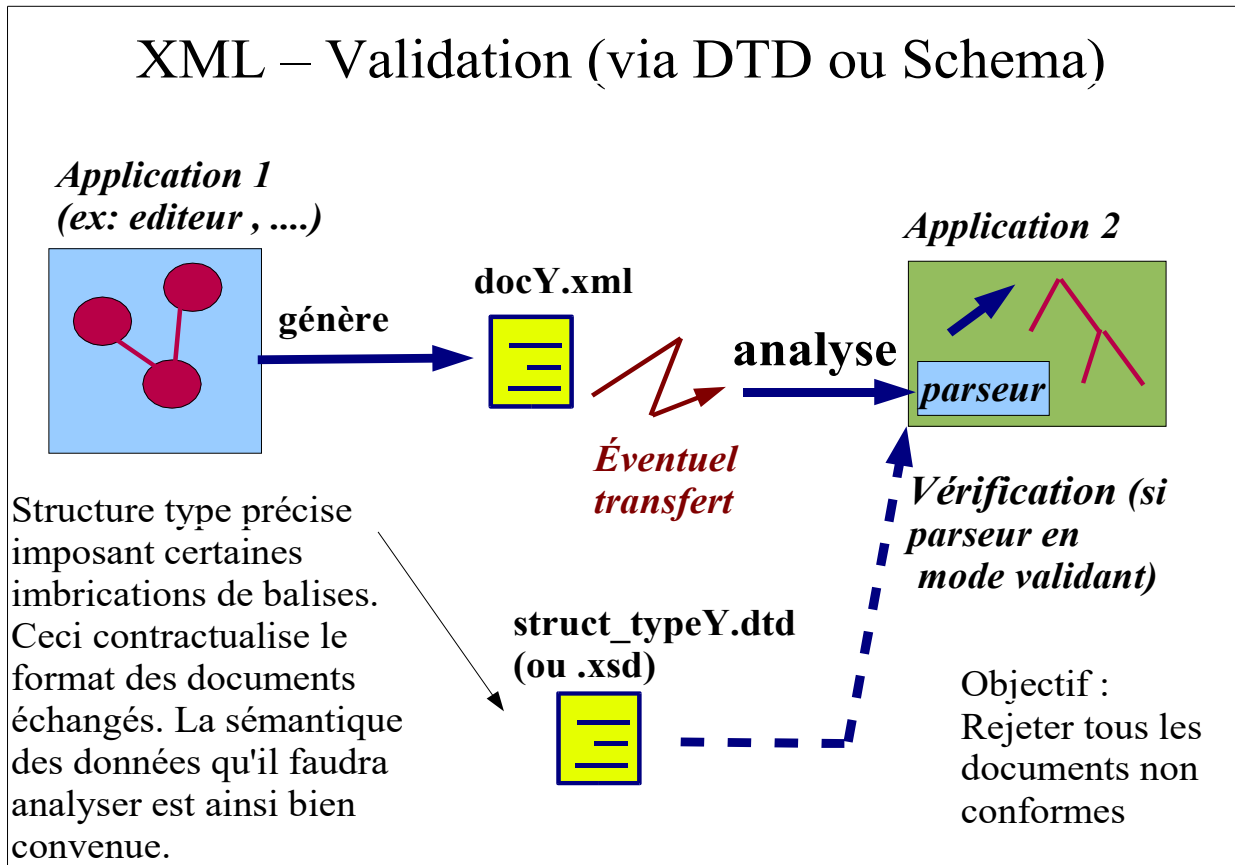
~~<i><g>blabla</i></g>~~ mais <i><g>blabla</g></i>

- **Les valeurs des attributs sont obligatoirement entre simple ou double quote:**

~~<pers age=35>...~~ mais <pers **age='35'**>... ou <pers **age="35"**>...

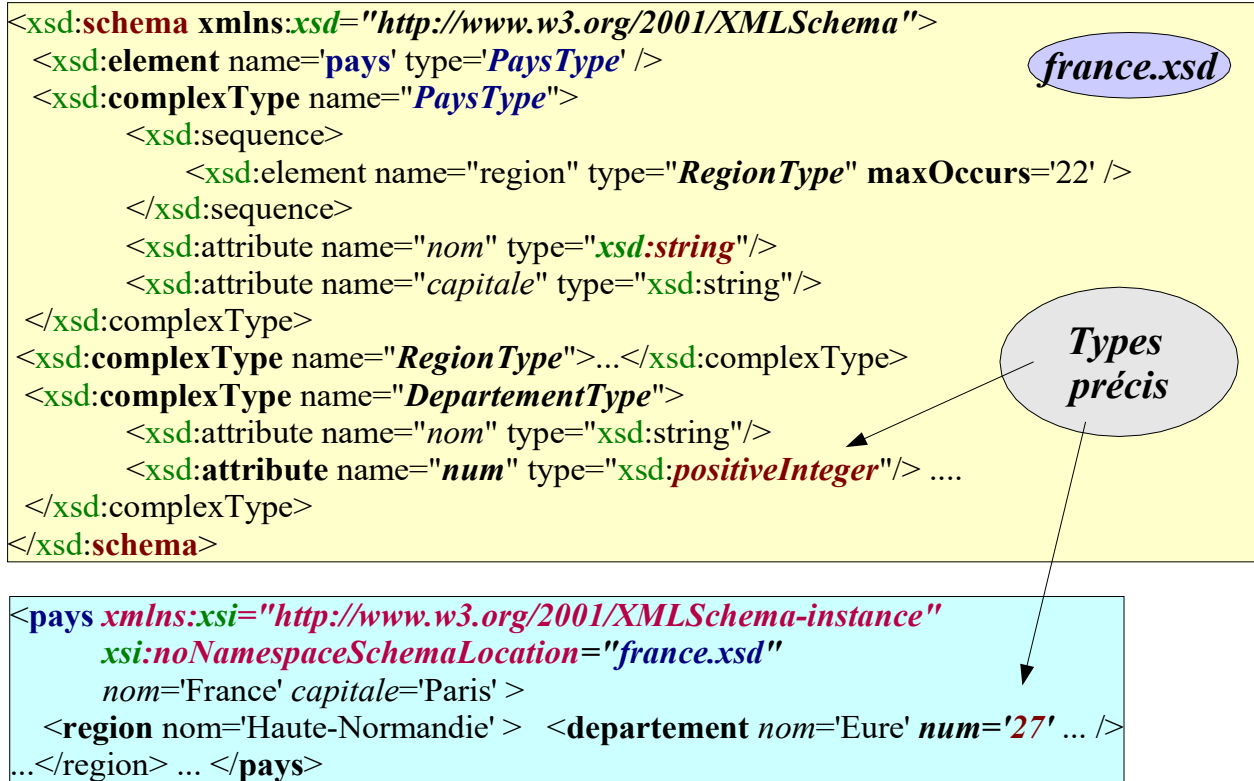
NB: Un document XML est par défaut encodé avec le jeu de caractères **UTF-8**.

1.5. Documents valides



1.6. Schéma W3C

Schéma XML (.xsd) plus évolués que DTD



XII - Annexe – BPMN

1. BPMN (essentiel)

1.1. Présentation de BPMN

BPMN signifie ***B**usiness **P**rocess **M**odel and **N**otation*

- Il s'agit d'un formalisme de modélisation spécifiquement adapté à la modélisation fine des processus métiers (sous l'angle des activités) et prévu pour être transposé en BPEL ou jBpm.
- Un diagramme **BPMN** ressemble beaucoup à un diagramme d'activité UML . Les notions exprimées sont à peu près les mêmes.

Les différences entre UML et BPMN sont les suivantes:

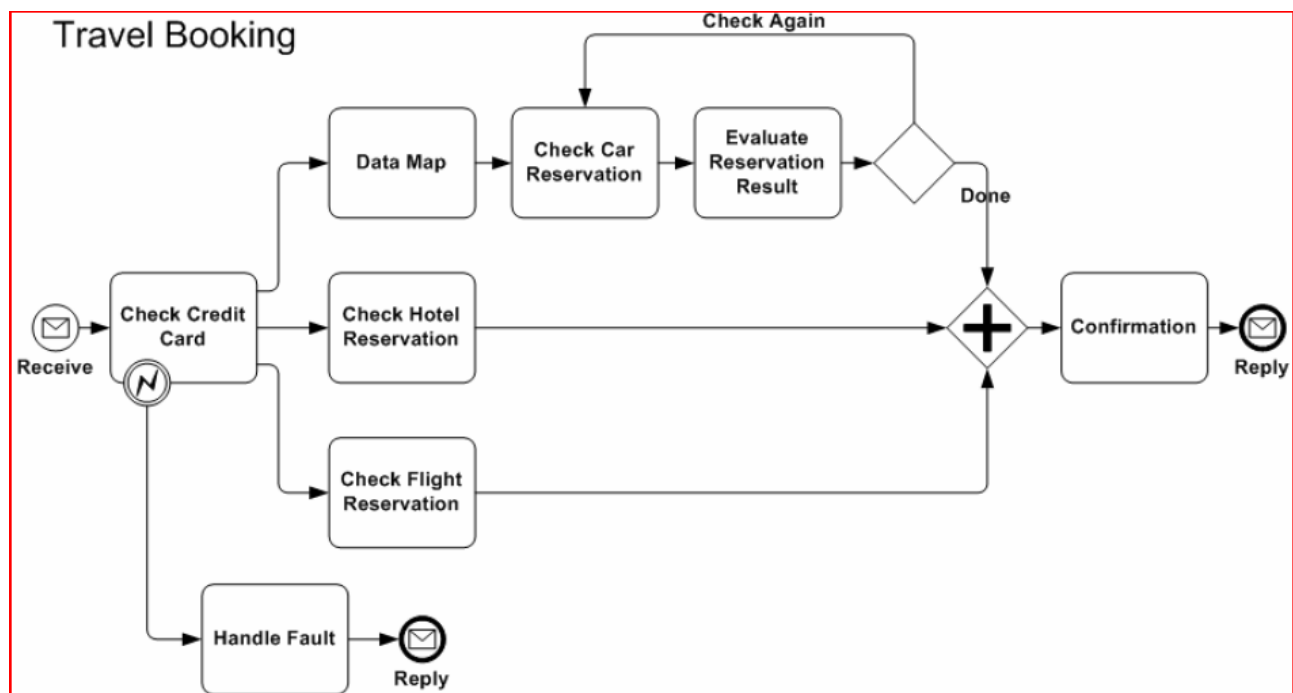
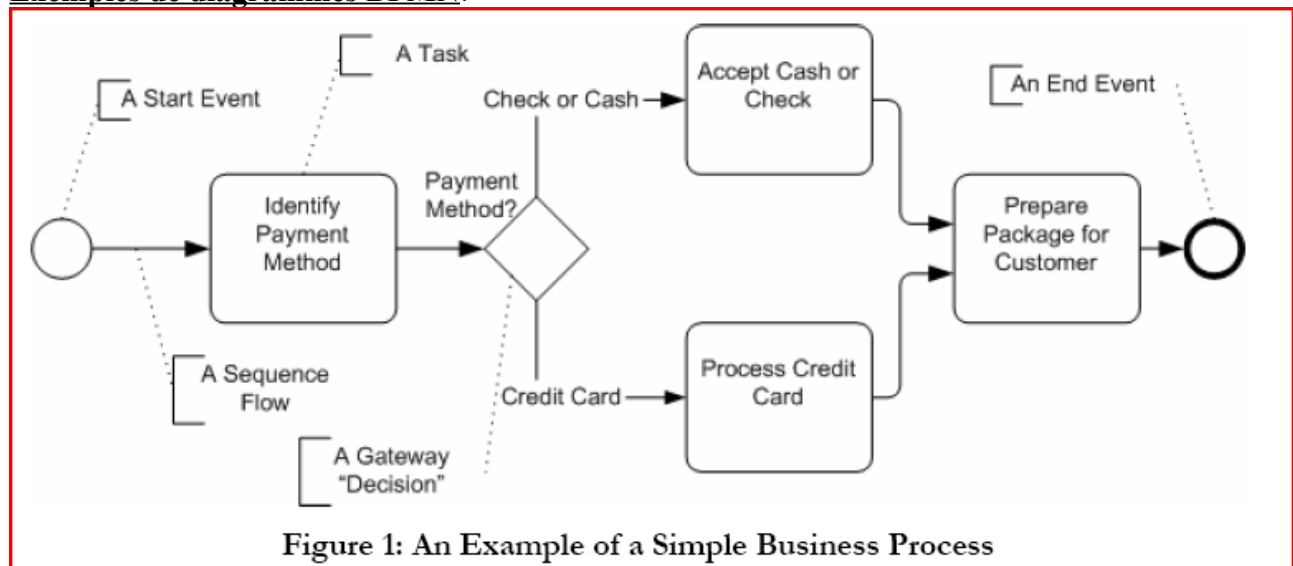
- la syntaxe des diagrammes d'activités UML est dérivée des diagrammes d'états UML et est plutôt orientée "technique de conception" que "processus métier"
- à l'inverse la syntaxe des diagrammes BPMN est plus homogène et plus parlante pour les personnes qui travaillent habituellement sur les processus métiers.
- UML étant très généraliste et avant tout associé à la programmation orientée objet, il n'y a pas beaucoup de générateurs de code qui utilisent un diagramme d'activité UML
- à l'inverse quelques éditeurs BPMN sont associés à un générateur de code BPEL et la version 2 de Bpmn peut être accompagnée d'extensions "java / jbpm ou activiti)" pour le rendre exécutable. Les éléments fin d'une modélisation BPMN ont été pensés dans ce sens.

Outils concrets pour la modélisation BPMN :

- *Intalio BPMN* .
- *Bizagi Process Modeler* (version gratuite)
- **Editeur BPMN2 intégré à l'IDE eclipse** (drools / jbpm5).
- **Yaoqiang bpmn(2) editor** (très bien : versions gratuites et payantes).

NB : Certains anciens outils BPMN 1.x (tels que **Bizagi**) étaient capables d'effectuer des imports/exports au format **XPDL** . **XPDL** signifie "***X**ml **P**rocess **D**efinition **L**anguage*" : c'est une sorte de sérialisation Xml de BPMN.

Des outils BPMN récents (supportant **BPMN 2**) peuvent quelquefois directement utiliser xml comme format natif (ex : **Jboss drools/jbpm5**, **activiti bpmn** , **yaoqiang bpmn editor**) .

Exemples de diagrammes BPMN:

1.2. Principales notations BPMN :

Flow objects :

Event (événement) :

- **Start** (début)
- **Intermediate** (intermédiaire)
- **End** (fin)






Activity (activité au sens large) :



- **Task** (tâche)
- **Sub-Process** (sous processus)



Gateway (décision / contrôle du flow)




Le losange (gateway) peut comporter alternativement plusieurs types/marqueurs :

<p>Exclusif (selon état des variables/données internes au processus)</p> <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;">  ou bien </div> <div style="text-align: center;">  (selon outil) </div> </div> <p>parallel</p> <div style="text-align: center;">  </div>	<p>Le mode exclusif désigne une seule route de sortie possible (selon condition/décision)</p> <p>Les conditions sont portées par les "sequenceFlow" de sorties .</p> <p>parallèlement (=fork <i>au début</i>) , aucune condition ne sera analysée.</p> <p>(sémantique "= join" à la fin) → suite que si toutes les branches concurrentes/parallèles sont finies/terminées .</p> <p>Le mode inclusif désigne plusieurs (au moins 2) routes de sortie possibles (par exemple: une</p>
--	---



<p>inclusif</p> 	<p>branche principale/obligatoire et d'autres branches facultatives) qui se rejoignent généralement par la suite .</p> <p>Comportement au début (fork selon conditions portées par les "sequenceFlow")</p> <p>Comportement à la fin (join en attendant que la fin des exécutions commencées)</p>
<p>eventBased</p> 	<p>(exclusivement) selon (premier) événement qui sera ultérieurement reçu.</p> <p>Les événements qui suivent un "eventBasedGateway" ne peuvent être que de type "<i>intermediateCatchEvent</i>" (souvent Timer ou Message)</p>

NB: Lorsque le processus effectue plusieurs **activités concurrentes (en //)**, la technologie interprétant Bpmn gère alors plusieurs "**jeton d'exécutions**" pour suivre et synchroniser les différents **états d'avancement**.

Connecting objects (connexions) :

- **sequence flow** (séquence ordonnée) 
- **message flow** (entre 2 participants) 
- **association** (entre tâche et données) 

Swimlane (couloir/partition d'activités)

- **pool** (*un par participant : Processus ou Partenaire ou ...*) 
- **lane** (*sous partition*) 

Data Objects :

- **Data** (*document xml ,objet en mémoire , ...*)
- **Data Store** (*extraction/persistance en base,...*)

1.3. Structure BPMN2

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:activiti="http://activiti.org/bpmn"
xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
xmlns:omgdc="http://www.omg.org/spec/DD/20100524/DC"
xmlns:omgdi="http://www.omg.org/spec/DD/20100524/DI"
typeLanguage="http://www.w3.org/2001/XMLSchema"
expressionLanguage="http://www.w3.org/1999/XPath"
targetNamespace="http://www.activiti.org/test">
  <message id="asyncResponseEvent" name="asyncResponseEvent"></message>
  <!-- logique (importante) du processus : -->
  <process id="myProcess" name="My Process" isExecutable="true">
    <startEvent id="startevent1" name="Start"></startEvent>
    <scriptTask id="scripttask1" name="SendAsyncRequest" activiti:async="true"
      scriptFormat="javascript" activiti:autoStoreVariables="false"><script>.... </script>
    </scriptTask>
    <sequenceFlow id="flow1" sourceRef="startevent1" targetRef="scripttask1"></sequenceFlow>
    <eventBasedGateway id="eventgateway1" name="Event Gateway"></eventBasedGateway>
    <intermediateCatchEvent id="timerintermediatecatchevent1" name="TimerCatchEvent">
      <timerEventDefinition>
        <timeDuration>PT1M</timeDuration>
      </timerEventDefinition>
    </intermediateCatchEvent>
    <intermediateCatchEvent id="...." name="MessageCatchEvent">
      <messageEventDefinition messageRef="asyncResponseEvent"></messageEventDefinition>
    </intermediateCatchEvent>

    <endEvent id="endevent1" name="End"> </endEvent>
  </process>

  <!-- coordonnées graphiques (non importantes) du processus : -->
  <bpmndi:BPMNDiagram id="BPMNDiagram_myProcess">
    <bpmndi:BPMNPlane bpmnElement="myProcess" id="BPMNPlane_MyProcess">
      <bpmndi:BPMNShape bpmnElement="startevent1" id="BPMNShape_startevent1">
        <omgdc:Bounds height="35.0" width="35.0" x="60.0" y="101.0"></omgdc:Bounds>
      </bpmndi:BPMNShape>
      ...
      <bpmndi:BPMNEdge bpmnElement="flow1" id="BPMNEdge_flow1">
        <omgdi:waypoint x="95.0" y="118.0"></omgdi:waypoint>
        <omgdi:waypoint x="140.0" y="151.0"></omgdi:waypoint>
      </bpmndi:BPMNEdge>
    </bpmndi:BPMNPlane>
  </bpmndi:BPMNDiagram>
</definitions>

```

1.4. Types précis d'événements BPMN2 :

Catching event (en réception):

Le processus est bloqué tant qu'il n'a pas reçu l'événement attendu.

L'icône interne est "non colorié" (laissé à blanc)

Throwing event (en envoi):

Le processus déclenche (envoi/soulève) un événement

L'icône interne est "colorié" (rempli de noir)

Définitions d'événements (Bpmn) :




La "définition d'un événement bpmn" englobe :










- une sémantique (fonctionnelle) de l'événement
(nom logique de l'événement , description , ...)
- un type d'événement (ex : Timer , Message , Signal , ...)
- d'éventuels paramétrages précis (ex : "timeDuration" pour un Timer)



Sémantiques des types (de définition) d'événements de BPMN :

Signal	Signal (avec un nom logique) qui peut être simultanément traité/reçu par plusieurs processus . Un signal envoyé par un processus est potentiellement diffusé vers plusieurs processus destinataires par la technologie prenant en charge l'exécution BPMN . Un signal n'a pas de paramètres .
Message	Message avec un " nom logique " et un " payload " (paquets de paramètres / arguments) . Un message "bpmn" est envoyé vers un seul destinataire .
Error	Exception métier / business (un peut comme message négatif)
Timer	À date ou heure fixe ou bien après période écoulée ou bien de façon cyclique .
Rule	règle métier vérifiée (extension pour technologie avec moteur de règles telle que "drools" ou ...)
...	...

Principaux types d'événements (bpmn) :

Start	None 	Selon contexte de déclenchement externe
	Signal 	A la réception d'un certain signal (un déclenchement de signal peut quelquefois servir à démarrer plusieurs processus au même moment)
	Message 	A la réception d'un message (avec d'éventuels paramètres/arguments) (Un processus peut éventuellement avoir plusieurs "MessageStartEvent" si

	<p>Timer </p> <p>Error </p> <p>...</p>	<p>plusieurs façons de démarrer).</p> <p>A une heure précise (éventuellement de manière périodique) ou ...</p> <p>Pour un démarrage de sous-processus en cas d'exception métier .</p>
Intermediate Catching	<p>Signal </p> <p>Message </p> <p>Timer </p> <p>...</p>	<p>Attente d'une réception de signal</p> <p>Attente d'une réception de message</p> <p>Attente d'une période écoulée ou ...</p>
Intermediate Throwing	<p>None </p> <p>Signal </p> <p>Compensation </p> <p>...</p>	<p>Juste pour indiquer (localement) qu'une tâche est finie (utile si techniquement associé à "listener d'événement" activi ou jbpmp → stats (KPI , BAM))</p> <p>déclenchement/envoi d'un signal</p> <p>pour déclencher une demande de compensation/annulation (à rattraper via "compensation boundary catching event " ailleurs)</p>
End	<p>None </p>	<p>sans "résultat" en retour</p> <p>fin (souvent au sein d'un sous-processus)</p>

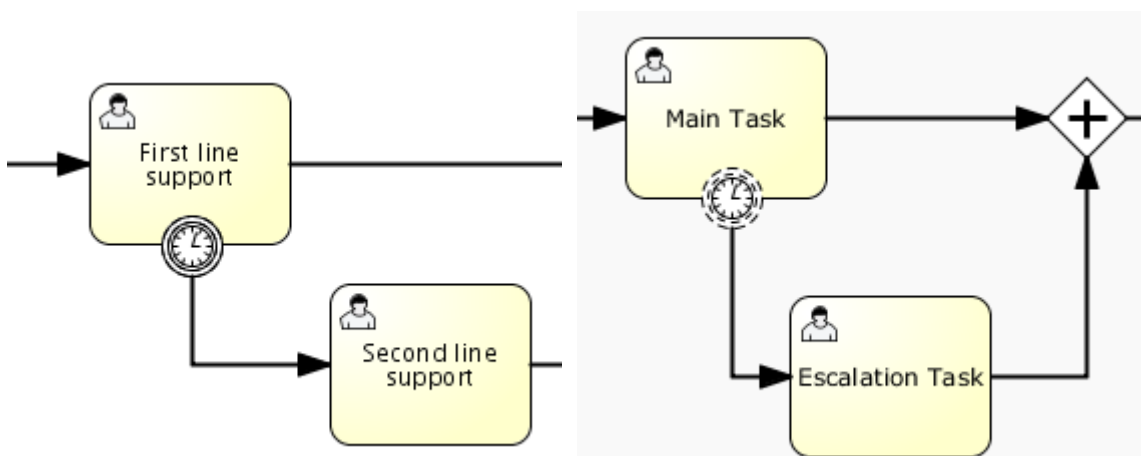
	<p>Error </p> <p>Cancel </p> <p>... selon outils/technologies "Message", "Signal", "..." au niveau du "endEvent" ou bien au niveau d'une "Task" préalable .</p>	<p>en envoyant un événement d' erreur devant être rattrapé par un "intermediate boundary error event" du niveau englobant</p> <p>Retourné (en mode transactionnel) par un sous-processus de façon à ce que le processus parent puisse le rattraper via un "cancel boundary event" lui même associé à un déclenchement de compensations.</p>
--	---	--

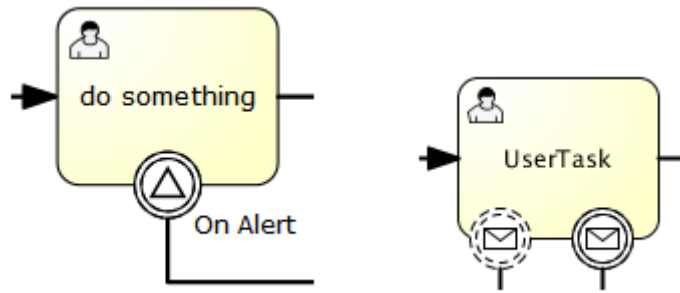
Boundary Events :

Les "**Boundary events**" sont une sorte de "**catching events**" qui sont **attachés à une activité**. Lorsque l'activité est en cours d'exécution, l'événement (en écoute) est potentiel déclenché .

Lorsque l'événement est attrapé , l'activité est interrompue et la séquence qui suit l'événement déclenché est alors exécutée .

Nouveauté de "BPMN2" : Lorsque les 2 cercles périphériques d'un "boundary event" sont en **pointillés** , l'activité **n'est pas interrompue** et on ne fait qu'effectuer des **opérations supplémentaires ("escalation")**. En xml , `cancelActivity="false"` si pointillés.





Tous les "boundary events" sont définis de la même manière :

```
<boundaryEvent id="myBoundaryEvent" attachedToRef="theActivity">
  <XXEventDefinition/>
</boundaryEvent>
```

Type de "Bondary Events":

Timer	Au bout du "timeout" , activité interrompue (ou pas) et suite après "evt"
Error	Erreur remontée par l'activité (sous-processus , call-activity , ...)
Signal	Signal reçu (émis depuis un endroit quelconque) au moment de l'exécution de l'activité
Message	Message reçu au moment de l'exécution de l'activité (en mode interruption ou pas)
Cancel	Issue potentielle d'une activité (en mode "transactionnal")
Compensation	Pour attacher un "gestionnaire de compensation" à une activité
...	

Détails du certains types (de définition) d'événements :

Paramétrage d'un **timerEventDefinition** :

timeDate (ex : 2011-03-11T12:13:14 YYYY-MM-DD T(ime) hh:mm:ss)

ou bien







timeDuration (ex : P10D pour une Période de "10 Day")

ou bien

timeCycle (ex : R3/PT10H/... Repeating 3 times / ...)

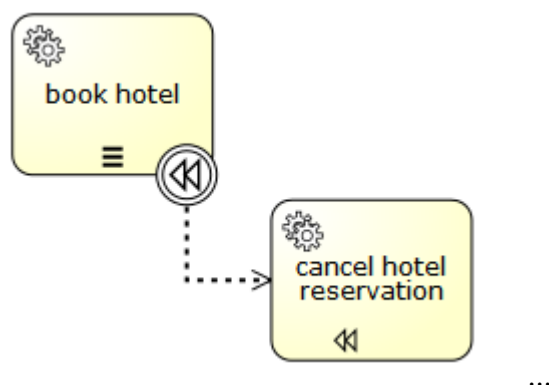
Les valeurs des paramétrages correspondent à la norme **ISO 8601**

1.5. Principaux types de Tâches "BPMN"

ServiceTask 	Tâche automatique (via appel de service web ou via classe java ou ...)
ScriptTask 	Tache automatique (via script "javascript" ou "groovy" ou ...)
UserTask 	<p>Tâche effectué par un utilisateur (souvent un employé appartenant à un certain groupe) avec généralement une console (web ou ...) : Les actions effectuées par l'utilisateur dépendent des valeurs du processus (selon tâches précédentes) et les valeurs saisies par l'utilisateur seront récupérées par le processus et influenceront les tâches ultérieures.</p> <p>Asynchronisme automatique (attente tant que tâche pas complètement effectuée)</p>
ReceiveTask 	<p>Réception d'un message (attente bloquante)</p> <p>Alternative fréquente : "intermediate Message Catching event"</p>
ManualTask 	Référence à une tâche entièrement manuelle (sans interaction avec le processus), pas gérée (ignorée) par le moteur d'exécution bpmn (ex : activiti).
BusinnessRuleTask 	Applications de règles métiers (souvent en s'appuyant sur "drools")
Autres (spécifiques selon technologies / extensions)	EmailTask , WsTask , EsbXyTask , ShellTask , ...

1.6. Gestionnaire de compensation :

Activité spéciale prévue pour compenser (via action inverse ou palliative) une activité (déjà effectuée) suite à un problème détecté tardivement .



1.7. Sous-processus BPMN

Un sous processus est une activité (non atomique) qui comporte d'autres activités , "gateway" , événements , Le sous-processus est lui même une partie d'un processus plus grand.

Un "sous-processus" BPMN ordinaire est entièrement défini dans le processus parent et a

donc une sémantique de "embedded sub-process".

Deux grands intérêts :

- **modélisation hiérarchique** (avec +/- pour visualiser/cacher les détails)
- **sous-processus = nouvelle portée pour certains événements** (*certaines événements soulevés par les éléments du sous-processus sont rattrapés via des "boundary event" du sous-processus*) .

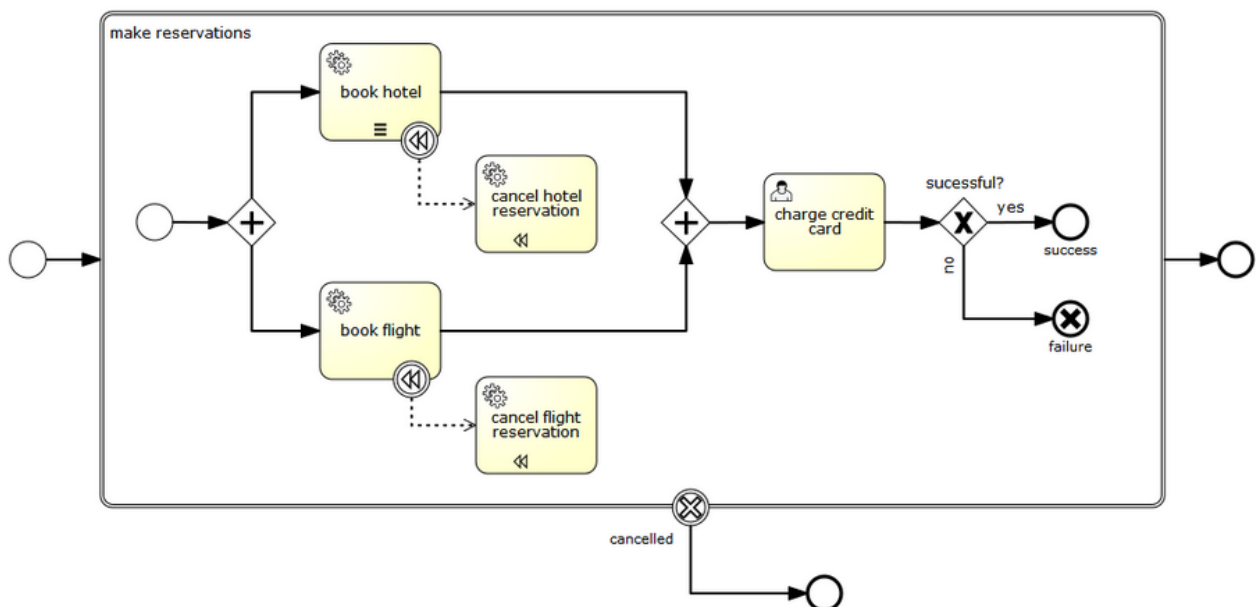
Contraintes (BPMN1 et certains moteurs BPMN2) :

- Un sous-processus doit comporter un seul sous "(none) startEvent"
- Un sous-processus doit comporter au moins un "endEvent"

Structure Xml :

```
<subProcess id="subProcess">
  <startEvent id="subProcessStart" />
  ... other Sub-Process elements ...
  <endEvent id="subProcessEnd" />
</subProcess>
```

Transactional (sub-process) :



délimité par <transaction> ... </transaction> en xml , double contour en notation graphique.

Un processus transactionnel comporte tout un tas de sous-activités qui seront toutes "réussies" ou toutes "annulées / éventuellement compensées en interne" .

L'issue globale de la transaction sera "success" (suite par défaut) ou bien "cancel" (vu comme un "cancel boundary event") . Dans certains cas rares , une erreur interne technique (hazard) sera gérée comme un troisième type d'issue (sans compensation mais souvent "log à étudier/gérer") .

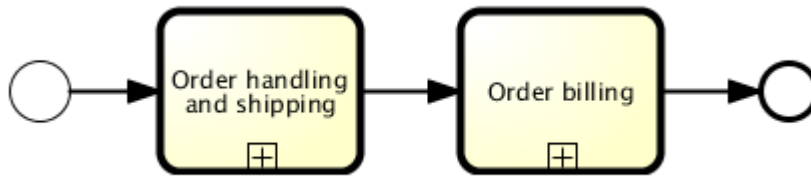
Attention : "transactional" (au sens transaction longue) est assez récent (et pas encore parfaitement interprété/géré partout) .

Call activity (au sens "call external subProcess") :

Le (sous) processus appelé est alors défini (de façon externe) dans un autre fichier "bpmn" ce

qui permet une plus grande ré-utilisabilité .

Le code du (sous-)processus appelé est censé être chargé (ou pré-chargé) dynamiquement .



Certains (rares) outils permettent une visualisation développée (en ouvrant/analysant le sous fichier ".bpmn")

```
<callActivity id="callCheckCreditProcess" name="Check credit"
calledElement="checkCreditProcess" />
```

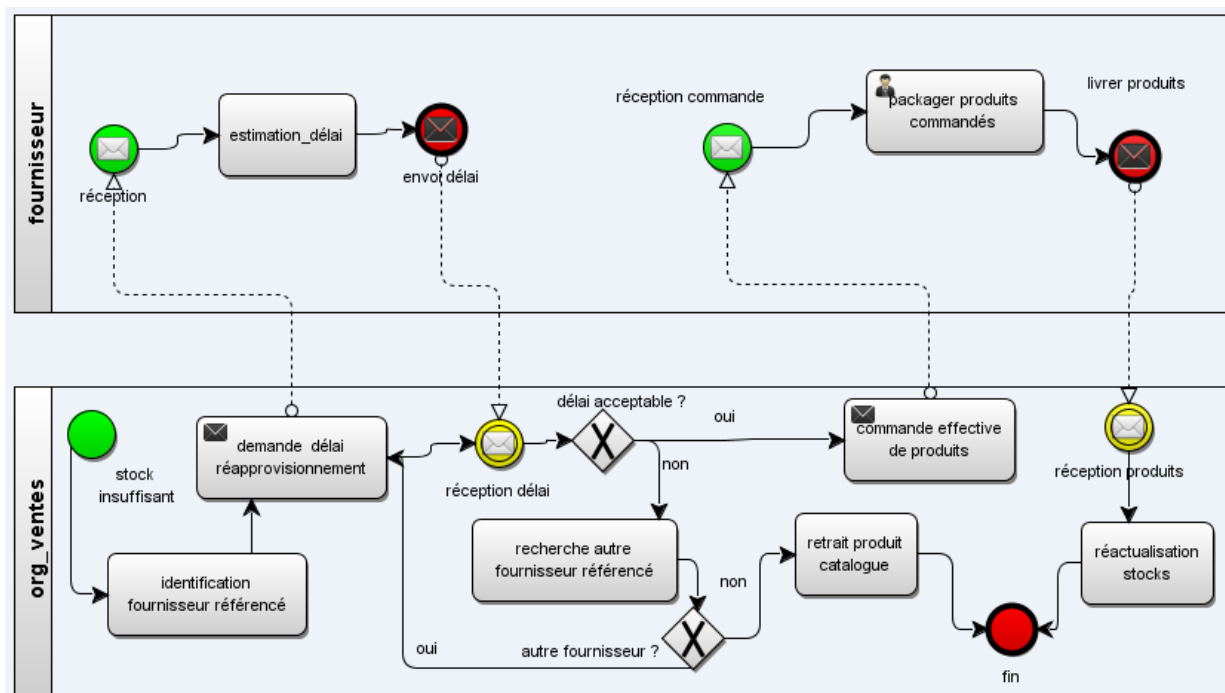
Le (sous-) processus appelé est **identifié par son nom logique** (identifiant interne au fichier ".bpmn").

Possibilité de passer des contenus de variables lors de l'appel d'un sous-processus (via certaines extensions telles que "activiti" , ...) .

1.8. Exemples de diagrammes BPMN

Exemple "approvisionnements fournisseurs" (édité avec "yaoqiang-bpmn-editor") :

Exemple de diagramme BPMN



Exemple BPMN "pizza" (tiré de la norme officielle de l'OMG) :

