Introduction au métier d’Architecte Solutions

Université de Nantes
M1 MIAGE 2024-2025

Copyright
Bertrand Florat

Bertrand Florat

Architecte Solutions

bertrand@florat.net

Avis / suggestions

Objectifs

- présenter le rôle d’un Architecte IT en général et **Architecte Solutions** en particulier ;
- donner des pointeurs ;
- **faire un retour des tranchées ;**
- présenter les **règles du jeu** ;
- se mettre en **condition réelle** ;
  - via un modèle dossier d’architecture
  - via des méthodes et des outils du terrain

Thèmes au choix (gras : obligatoire)

- **L'architecture IT, c'est quoi ?**
- Le zoo de l'architecte technique
- Introduction aux ENF
- L'outillage et les diagrammes
- Les règles du jeu
- L'architecture agile
- Les grandes typologies d'architectures

Thèmes au choix

https://forms.gle/hppGFnpWhaRyTfuZA

L’architecture, c’est quoi ?

(et ça sert à quoi ?)

“

Dictionnaire Hachette

architecture n. f.
1. Art de construire des édifices selon des proportions et des règles **déterminées par** leur caractère et **leur destination**
2. Disposition, ordonnance, style d’un bâtiment

“

1962 F. Brooks in W. Buchholz in Planning Computer Systems:

ii. 5 : Computer architecture, like other architecture, is **the art of determining the needs of the user**...and then designing to meet those needs as effectively as possible.

Qu’est ce que l’architecture ?

1. La conception de ce qui est **structurant** pour un projet, ce qui est **difficile à changer**
   - la **taille et la forme des pièces** de la maison, pas la marque des prises de courant
   - le **découpage en couches**, pas le choix de l’IDE

Qu’est ce que l’architecture ?

2. Les éléments **partagés par les experts du projet** *(shared understanding)*
   - Les schémas ou informations **que les experts vous montreraient** pour expliquer comment le système fonctionne
   - Plus grand **commun dénominateur** dans la **compréhension** du projet

Vision forcement partielle, différents points de vue

“

Martin Fowler :
Architecture is about the **important stuff**.
Whatever that is.

Sans architectes pour donner une vision d'ensemble du projet...

modules incompatibles

écart au besoin

développements inutiles

Sans architecte d'entreprise/urbaniste pour donner une vision SI...

processus métier entravés

problèmes d'interopérabilité entre projets

projets sans cohérence SI

l’architecture solutions, c’est la synthèse...

des exigences fonctionnelles

doit permettre de saisir sa déclaration d’impôts en ligne en moins de 10 mins

des exigences non fonctionnelles (ENF)

doit tenir 1M d’utilisateurs par jours, doit être accessible aux non-voyants…

des contraintes

doit coûter moins de 1M€, doit utiliser les technologies déjà en place...

des règles de l’art

dans les années 2020, choisir plutôt une architecture Web que du client-serveur.

Solution d'Architecture

L'architecte peut intervenir lors des différentes phases du projet

en amont

- aide à faire **émerger les besoins et exigences**
- obtention de **consensus**
- connaît les **contraintes d'urbanisation**
- Fait des **POC**, plusieurs scénarios ...

Voir liste des ENF dans la section "Introduction aux ENF"

en développement

- **bonnes pratiques**
- patterns
- expertise technique
- benchmarks
- contrôle de la sécurité …

après la MEP

- feedback
- performances et sécurité : jamais achevé

Les types d'architectes principaux

- Architecte SI / d'entreprise (EA)/urbaniste : la loi

  Défini les règles globales du SI (fonctionnelles, applicatives, techniques)

- Architecte logiciel / applicatif : le guide

  Aide à la conception dans le code, usine logicielle, frameworks, bonnes pratiques ...

- Architecte d'infrastructure / technique : l'expert

  Expert en infrastructure (réseau / système / matériel / middleware). Dimensionnement, performances.

- Architecte solutions : le couteau suisse

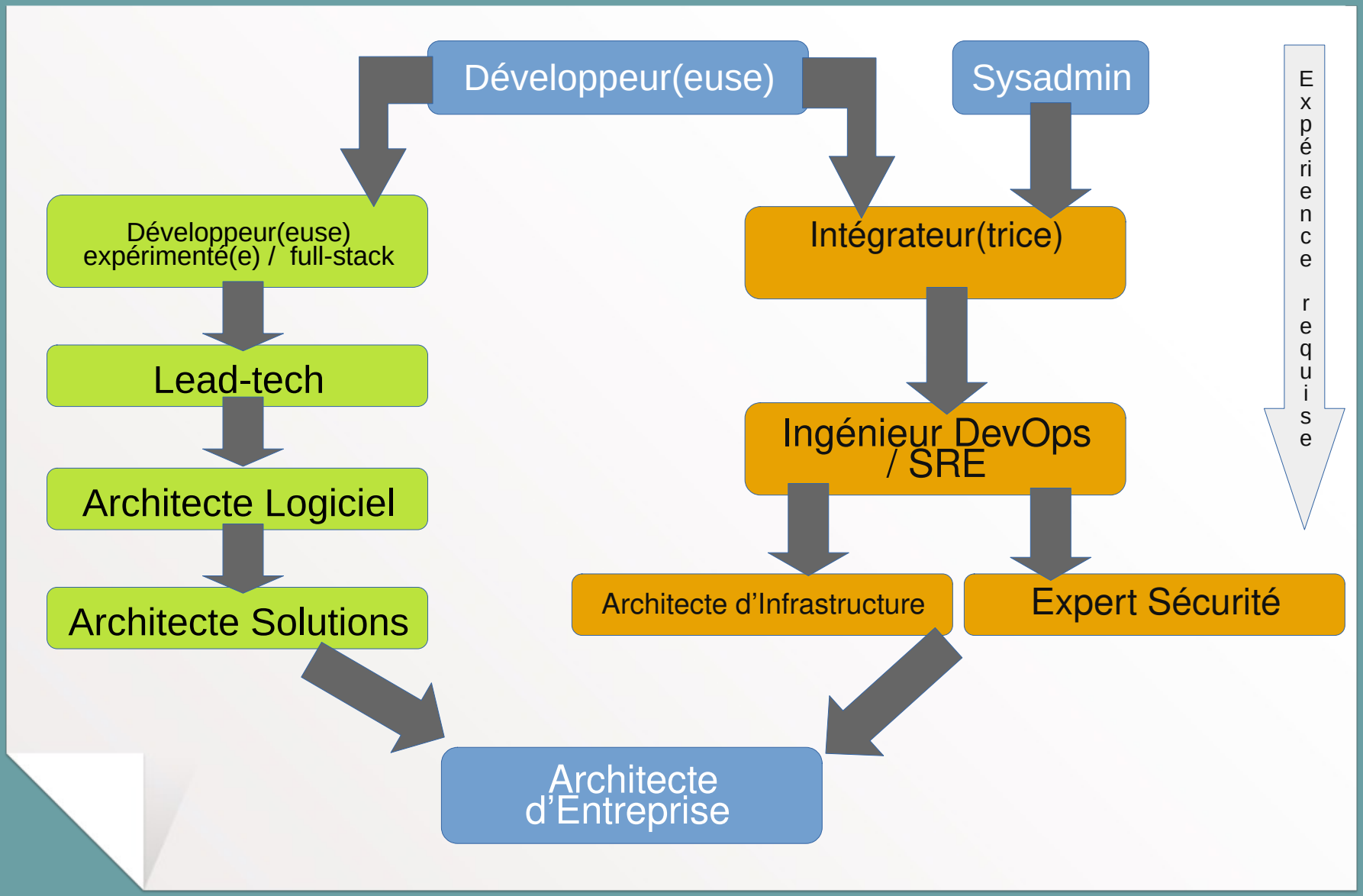
  Agrège les besoins et **propose une solution globale** (transverse, sans être expert)

La terminologie et le périmètre varie entre organisations

Les qualités conseillées : un bon architecte solutions...

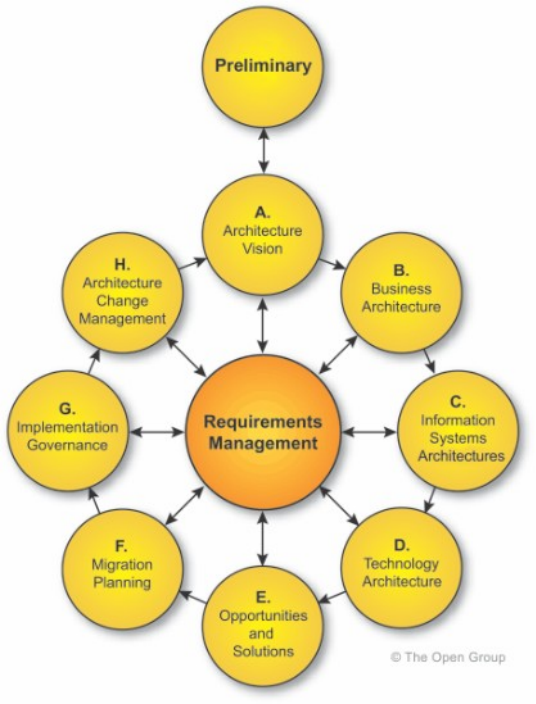
- Possède un esprit **d'analyse** et de **synthèse ;**
- A besoin de **Soft skills** : négociation, persuasion, diplomatie, patience, bonne **gestion du stress ;**
- **Est passionné(e)** de technologie et de méthodologie ;
  - n'a pas peur de la nouveauté et d'expérimenter.
- Tend à aller au **fond des sujets** pour les comprendre ;
- Doit être **responsable**, réaliste**, rigoureux(se)** et prudent(e) ;
- Aime partager ses connaissances et expériences ;
  - sessions de **RETEX**, animations de **conférences**, enseignement, **meetups**...
  - **méthode Feynman**: expliquer pour comprendre des sujets complexes.
  - **documentation** : sait apprécier ce qui doit (ou ne doit pas) être documenté, aime la rédaction de **dossiers** et la réalisation de **diagrammes**.

Trajectoire type d'un(e) architecte



L’Architecture d'Entreprise (urbanisation)

- Réflexion au niveau SI (ville), pas seulement projet (bâtiment)
- Aborder le **SI de manière globale**, **mutualiser**
- Découpage du SI en blocs régis par des règles
- Permet de connaître **son SI**
  - Cartographies
  - Études d’impacts
  - Connaître ses données principales
- Permet de **prévoir** :
  - Hiérarchiser les priorités
  - Suivre le portefeuille applicatif
  - Anticiper les besoins
- Frameworks
  - Le plus utilisé : TOGAF



L’architecte n’est PAS :

- Un concepteur
- Un expert produit
- Un manager / chef de projet

Les erreurs les plus courantes chez les architectes

- - Être un architecte tout d'ivoire.
- - Travailler à la chaîne.
- - Tomber dans le syndrome du "Golden Hammer".
- - Succomber au Résumé-Driven Developement.
- - Essayer d'être Nostradamus.
- - Être attrapé par le chant des sirènes des éditeurs.
- - Être mal utilisé.
- - Négliger les aspects métier, fonctionnels ou juridiques.
- - Négliger l'évolutivité et la réversibilité
- - Être trop léger sur la gouvernance et la documentation.
- - Ignorer les contraintes de l'organisation.
- - Manquer de compétence dans la collecte et l'intégration des ENF.

Voir [cet article] pour le détail et les solutions possibles.

Les architectes entre eux

- Chaque architecte a sa vision, son background
- Pour chaque projet, il y a une **infinité de solutions possibles**, autant d'architectures que d'architectes

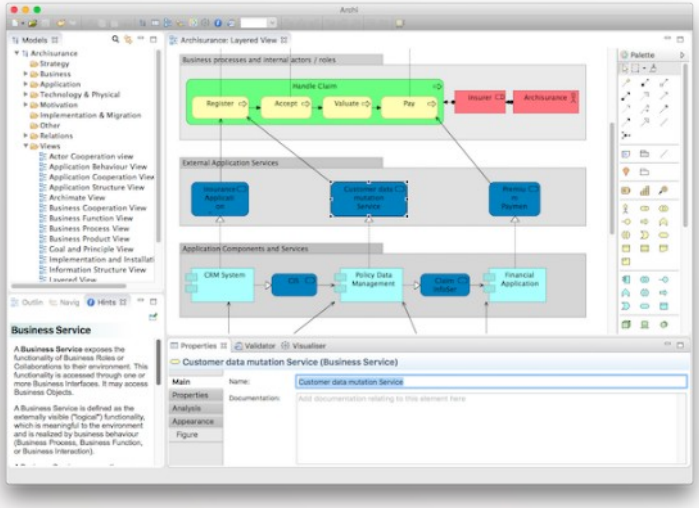
  dans la rue, les maisons sont-elles semblables ?

- Un architecte s'épanouit en communauté d'échanges d'expérience et de connaissance
- Doit avoir un **pouvoir de décision important** sur son projet pour être efficace

L’outillage et les diagrammes

Les diagrammes

- Moyen de communication principal (avec les tableaux)
- Peut être un langage **formel** (UML 2, Archimate) mais **pas forcement** : important est de transmettre les idées principales de façon claire
- Mettre les fonctions **sur les flèches** et compléter de façon textuelle dans le dossier si besoin
- Voir Archimate (OpenGroup)
- Ex d'outil OpenSource : archi



https://www.archimatetool.com/

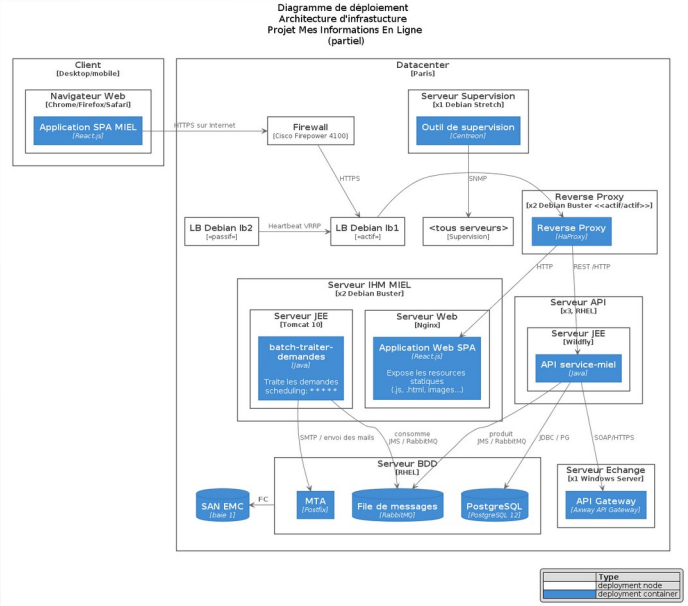
Les diagrammes générés

- Voir Graphviz , **Plantuml**, Mermaid
- « **Diagrams as a code** » : DSL textuel simple
- **Nombreux avantages** (gestion version, gain de temps, diff, factorisation...)

```
@startuml
Node("client","Client","Desktop/mobile"){
   Node("nav1","Navigateur Web","Chrome/Firefox/Safari") {
     Container("spa","Application SPA MIEL","React.js")
   }
}


spa -> r2 : HTTPS sur Internet
r2 --> lb1 : HTTPS
…
@enduml
```

plantuml

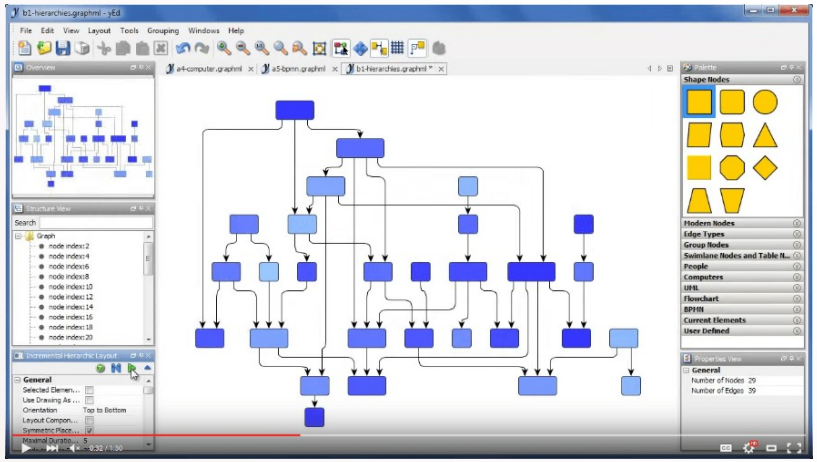


Outil de diagrammes gratuits ou Open Source

- Pour des schémas rapides et beaux : **yEd** (freeware), version cl oud disponible.
- Draw.io (cloud ou stand-alone)
- Diagrammes C4 avec Plantuml ou https://structurizr.com/ sur le cloud.

Parfait pour une présentation ou des études d'architecture

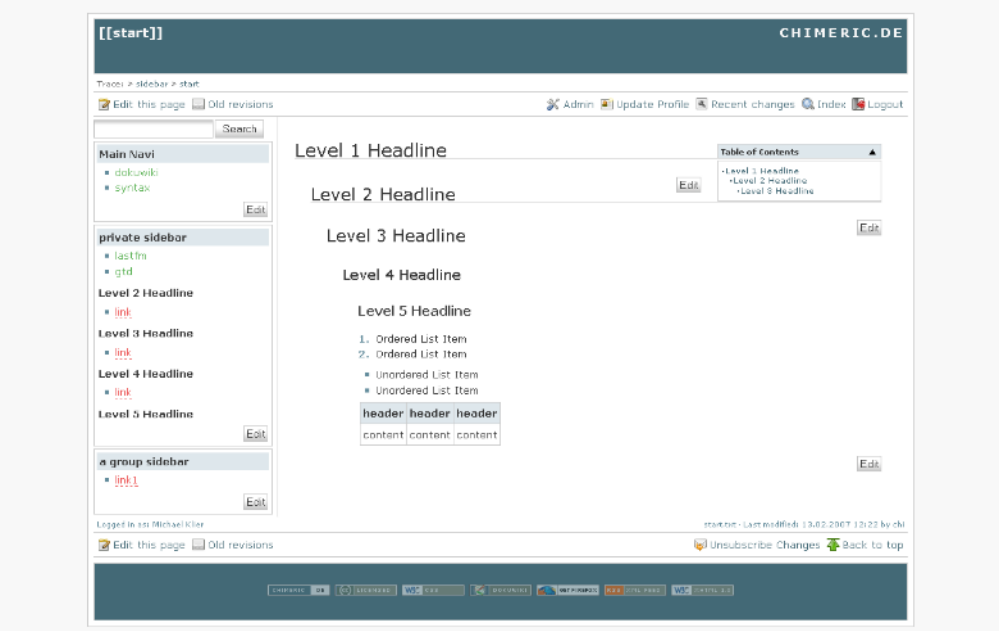
Idéal pour diagrammes à conserver / maintenir



yEd : https://www.yworks.com/products/yed

Les wiki

- Bien pour **centraliser un référentiel d'architecture**
- Wikis simples (sans base de donnée) : **DokuWiki**, jspWiki

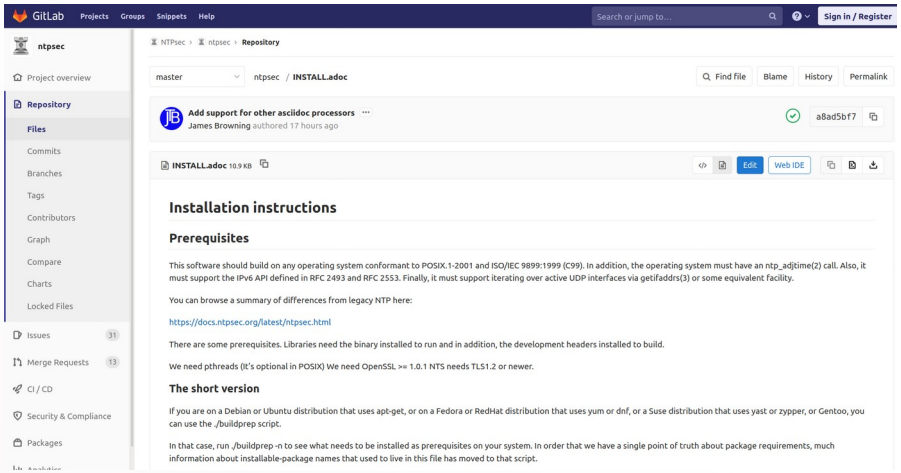


Privilégier maintenant les forges Git pour des documentations d'architecture
Voir page suivante

http://dokuwiki.org

Les plateformes Git (Gitlab, Github, ...)

- Wiki en mieux : rendu similaire mais **versioning comme des sources avec Git** et outillage associé (**Merge Requests**, ...)
- Texte en **Markdown** (pour docs simples) ou **Asciidoc** (pour véritable documentation)
- Possibilité de cloner les sources et d'utiliser un **IDE** avec nombreux plugins (ex: éditeur Asciidoc sous Visual Studio Code)

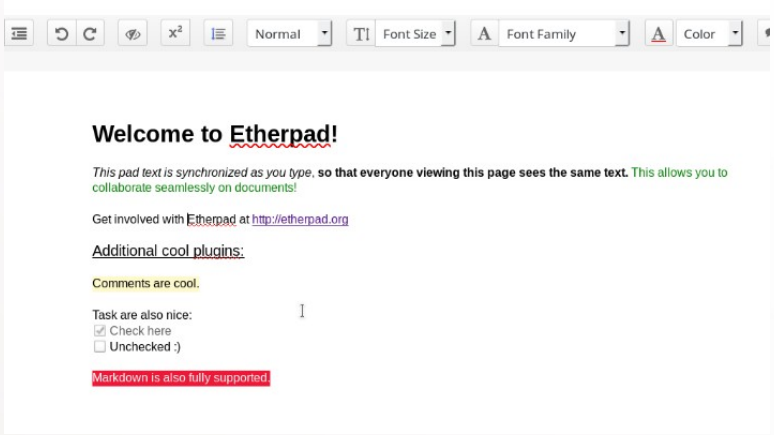


Exemple de documentation sous Gitlab

Exemple de stack : Asciidoc + Plantuml + Gitlab + VSCode

Les documents partagés

- Parfait pour la **phase exploratoire**, partage entre architectes
- **Web**, dynamique
- Mettre le **lien** dans les autres outils
- Solutions :
  - **ONLYOFFICE** (intégrable dans NextCloud par exemple)
  - **Etherpad** (Open Source)
  - Office 365
  - Google Docs...



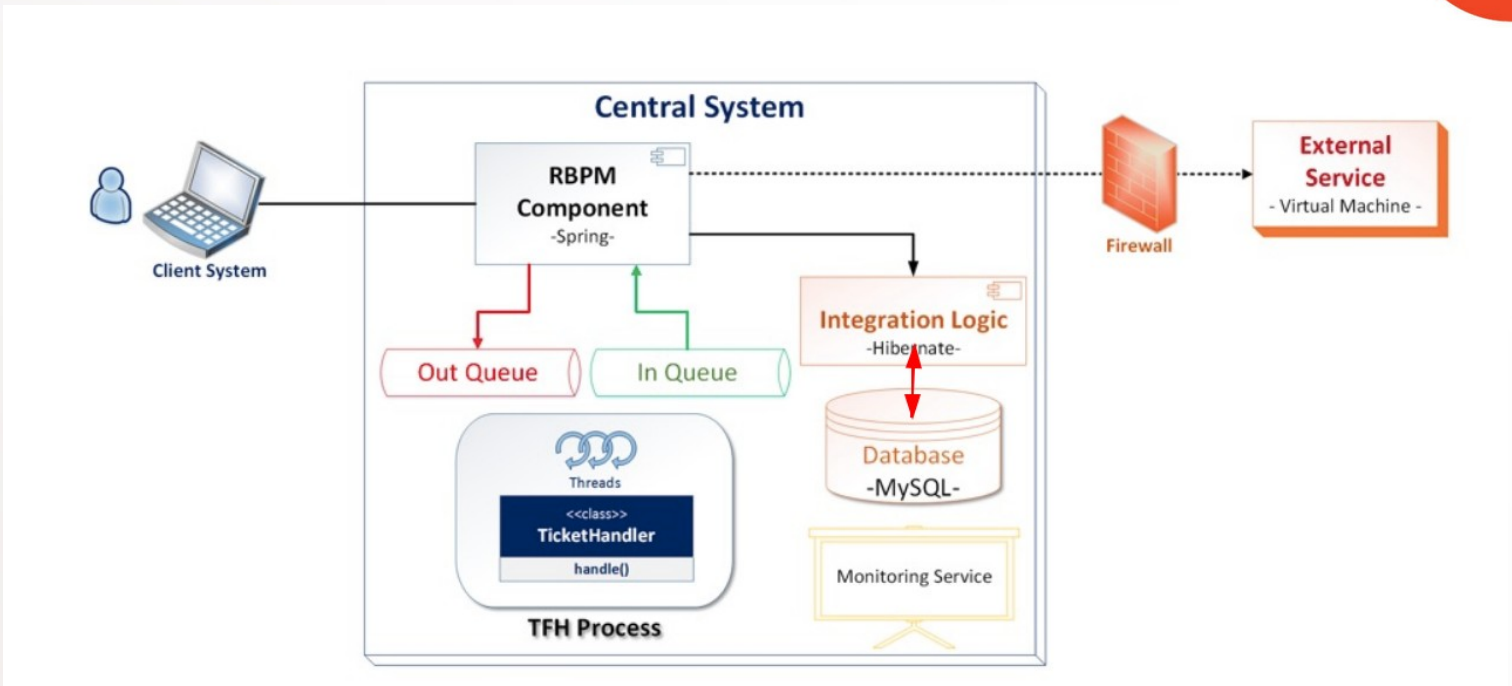
Qu’est ce qu’un bon diagramme ?

- **Auto-porteur** (légende, titre)
- **Explicatif** (doit répondre à plus de questions qu’il n’en pose)
- Limiter la **charge cognitive** (pas plus d’une dizaine de flèches, éviter les flèches qui se croisent)
  - Au besoin, faire plusieurs schémas segmentés par niveau d’abstraction, fonctionnalité, temporalité, etc.
- **Généré** automatiquement si possible (Graphviz, Plantuml, Mermaid, …), surtout si nombreux diagrammes à maintenir
- **Cohérent** : Ne pas mélanger des notions de vues différentes (exemple : une composant logiciel et un serveur) et utiliser toujours les mêmes termes.
- **Versionné** : mettre les sources des diagrammes dans un VCS comme Git

Conseils pour de bons diagrammes

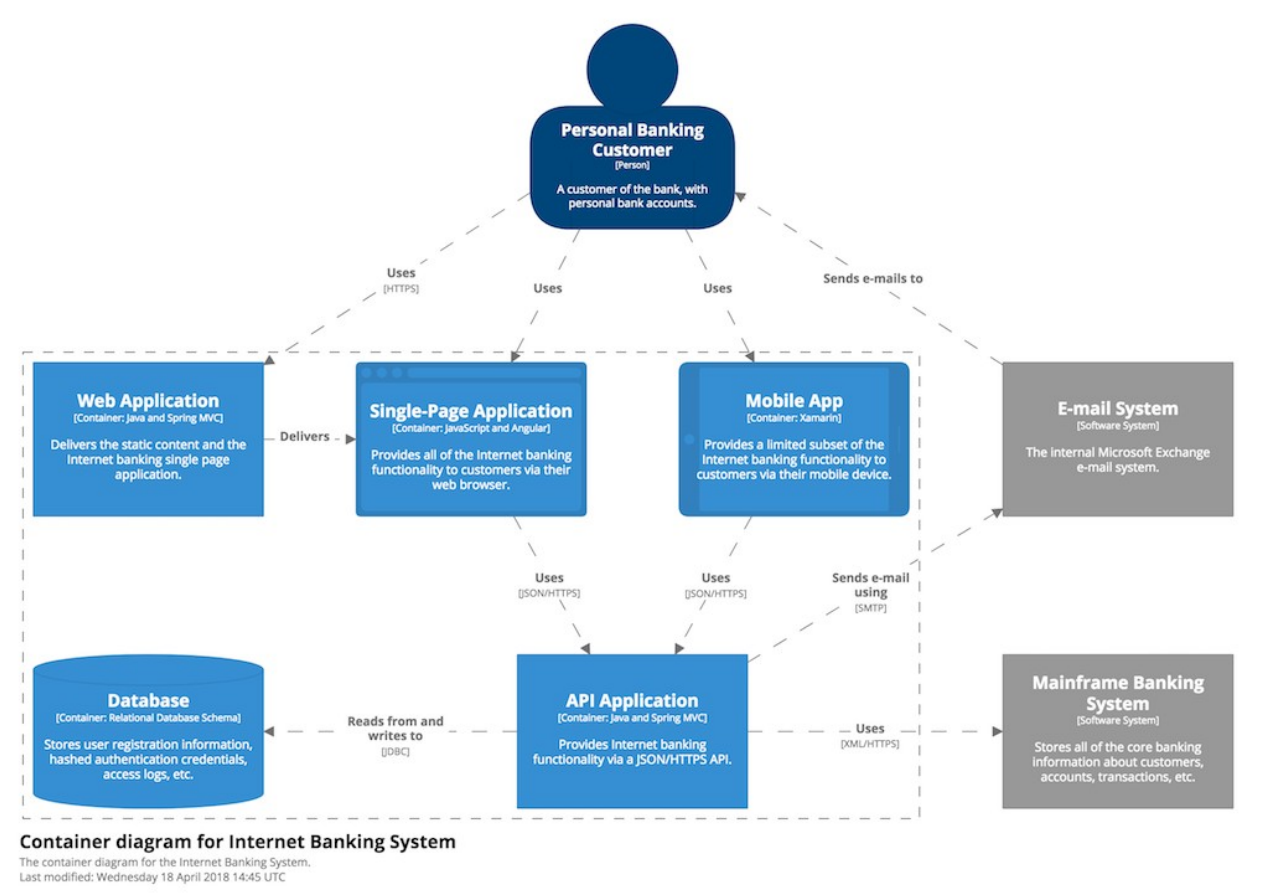
- Limiter le **nombre de couleurs** / **formes** différentes (max cinq couleurs, cinq formes)
- Expliquer clairement le **sens des flèches** (sens des données ou sens de l'appel ?)
- Donner du **contexte** (types d'utilisateur, zone réseau, ...)
- **Jamais de bidirectionnel** (faire deux flèches unidirectionnelles)
- Toujours un **titre** sur une flèche
- Si besoin, fournir une temporalité (**étapes** sous la forme 1, 2.1, 2.2.1, …). Temporalité + identification du flux dans un dossier.
- Ajouter le **texte explicatif dans les boites**. Si besoin, compléter par une explication textuelle sous le diagramme faisant référence aux étapes.

Mauvais diagramme d’architecture



https://www.infoq.com/articles/crafting-architectural-diagrams/

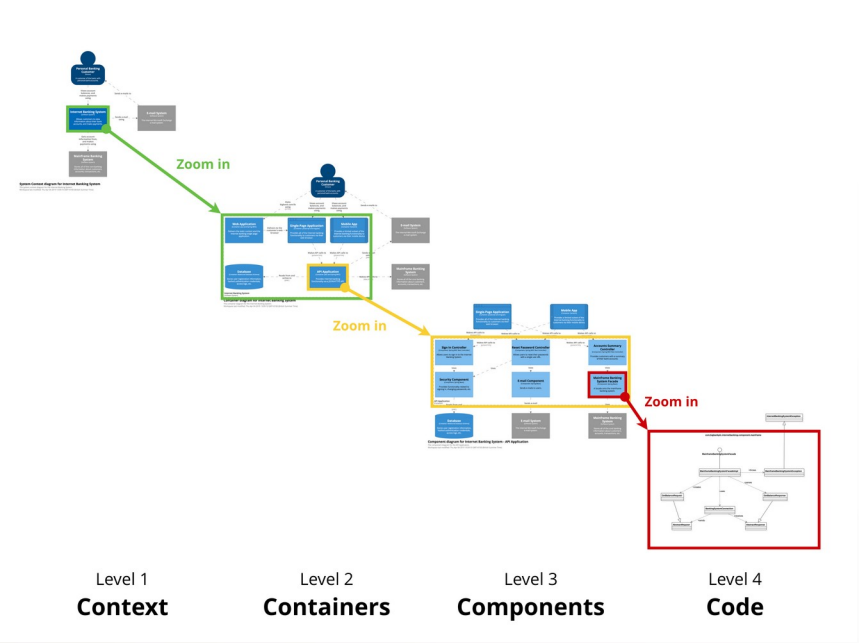
Bon diagramme d’architecture



Le modèle C4

- Quatre niveaux de détail d'un système :
  - Diagrammes de **System Context** (architecture applicative générale) / System landscape
  - **Diagrammes de containeur** (blocs techniques qui héberge du code ou des données)
  - **Diagrammes de composants** (classes principales)
  - Diagrammes **UML 2**
- Excellente dichotomie des points de vue d'architecture, limite la charge cognitive
- Nombreux plugins pour : Plantuml, draw.io, VS Code, ...

https://c4model.com/



Les règles du jeu

Les « eight fallacies of distributed computing »

- Le réseau est fiable.
- Le temps de latence est nul.
- La bande passante est infinie.
- Le réseau est sûr.
- La topologie du réseau ne change pas.
- Il y a un et un seul administrateur réseau.
- Le coût de transport est nul.
- Le réseau est homogène (même latence/ bande passante pour tous les clients).

Les principes fondamentaux :
Les twelve factors des applications cloud native (Heroku)

Les 12 facteurs

I. Base de code
Une base de code suivie avec un système de contrôle de version, plusieurs déploiements

II. Dépendances
Déclarez explicitement et isolez les dépendances

III. Configuration
Stockez la configuration dans l'environnement

IV. Services externes
Traitez les services externes comme des ressources attachées

V. Build, release, run
Séparez strictement les étapes d'assemblage et d'exécution

VI. Processus
Exécutez l'application comme un ou plusieurs processus sans état

VII. Associations de ports
Exportez les services via des associations de ports

VIII. Concurrence
Grossissez à l'aide du modèle de processus

IX. Jetable
Maximisez la robustesse avec des démarrages rapides et des arrêts gracieux

X. Parité dev/prod
Gardez le développement, la validation et la production aussi proches que possible

XI. Logs
Traitez les logs comme des flux d'évènements

XII. Processus d'administration
Lancez les processus d'administration et de maintenance comme des one-off-processes

Voir cours
conception logicielle, chapitre
« applications Cloud Native »

https://12factor.net/fr/

Notions de temps de réponses

- 1 CPU cycle 0.3 ns 1 s
- Level 1 cache access 0.9 ns 3 s
- Level 2 cache access 2.8 ns 9 s
- Level 3 cache access 12.9 ns 43 s
- Main memory access 120 ns 6 min
- Solid-state disk I/O 50-150 µs 2-6 days
- Rotational disk I/O 1-10 ms 1-12 months
- Internet: SF to NYC 40 ms 4 years
- Internet: SF to UK 81 ms 8 years
- Internet: SF to Australia 183 ms 19 years
- OS virtualization reboot 4 s 423 years
- SCSI command time-out 30 s 3000 years
- Hardware virtualization reboot 40 s 4000 years
- Physical system reboot 5 m 32 millenia

Crédit : « Systems Performance: Enterprise and the Cloud »

Les bonnes pratiques
Comment concevoir un système performant

- **Services multivalués** (avec taille adéquate)
- Asynchronisme
- Parallélisation
- **Co-localisation** des composants si possible
- **Stateless**
- Scalabilité : horizontale, verticale, diagonale
- Tunning (GC, BDD : index, SGA…)
- Bon usage des ORM
- **Borner/limiter** (voir plus loin)

Les bonnes pratiques

Comment concevoir un système robuste ?

- Produire des **logs** et des traces
- **Borner** (voir plus loin)
- Gestion de la cohérence
  - utiliser les transactions (ACID) voire le XA dans cas bien précis
  - architectures Event-Driven : systèmes « eventually consistent », patterns SAGA.
- **Points de reprise** des batchs (JSR352)
- Gestion des **rejeux** et des erreurs intermittentes
- Types **d'erreurs** : fonctionnelles et techniques. Que faire dans chaque cas ?
- **Idempotence** pour éviter les doublons

Les bonnes pratiques

Borner les systèmes informatiques

- Limites **maîtrisées plutôt que subies**
- **Rate limiting** (nombre max d'appel par intervalle de temps)
- Max payload
- Coupe-circuits
- QoS par jetons
- Timeouts (décroissants avec la chaîne de liaison)
- Taille pools (threads/connexions…)
- Mémoire (Xmx, ...)
- Limites CPU, mémoire des conteneurs
- **Taille des transferts de fichier, uploads ...**

Les principes fondamentaux :
Le théorème de CAP

- Un système distribué ne peut garantir à un moment donné que deux de ces trois propriétés :
  - **C**onsistency = Cohérence : tous les nœuds voient les mêmes données ;
  - **A**vailability (Disponibilité) : toutes les requêtes ont une réponse ;
  - **P**artition Tolerance (clustering) : le cluster fonctionne même si les nœuds ne communiquent plus entre eux.

Exemple de choix suivant les contraintes: CA: Mysql ou PostgreSQL, CP: Apache HBase, AP: Cassandra

Introduction aux ENF

Définition et utilité

Une Exigence Non Fonctionnelle est une **exigence portant sur une aptitude d'un système informatique** (exemple: la confidentialité).

Les exigences **fonctionnelles** précisent ce que doit faire le système (le ***quoi***) : règles de gestion, IHM, traitements, ...) alors que les **ENF** précisent les **attributs de qualité** du système.

Les ENF sont un **entrant majeur pour concevoir une architecture IT**. Une unique ENF peut conditionner toute une architecture.

Les ENF sont **négociées** et **recensées** par l'architecte puis **suivies** (dans le dossier d'architecture par exemple).

Un programme qui fonctionne, ça ne suffit pas !

- La conception/dev : seulement une partie du projet
  - **maintenance : 50 %** (van Vliet [2000])
- Le produit doit répondre aux exigences non-fonctionnelles

C'est bon ! il y a un toit et une porte

Il y a un toit, des portes, on peut accueillir 5000 personnes en même temps, il y a des sorties de secours et une garantie décennale

Liste exhaustive dans la norme ISO 25000

ENF principales par familles

Famille (cyber)Sécurité

- Confidentialité
- Intégrité
- Non répudiation

Famille Performances

- Rapidité
- Scalabilité
- Élasticité
- Efficacité

Famille Fiabilité

- Disponibilité
- Robustesse
- Résilience

Famille Maintenance

- Évolutivité
- Testabilité

Famille Exploitation

- Exploitabilité
- Auditabilité

ENF de (cyber)sécurité

Moyens techniques, organisationnels, juridiques et humains permettant d'empêcher l'utilisation non-autorisée, le mauvais usage, la destruction ou le détournement du SI

ENF	***Exemples de dispositifs***
Confidentialité La confidentialité est le fait de s'assurer que l'information n'est accessible qu'à ceux dont l'accès est autorisé	- Chiffrement cryptographique (ex: AES) - Identification et authentification par login/mot de passe ou devices - Gestion des autorisations (droits)
Intégrité Durabilité, justesse et niveau de confiance dans les données de l'application	- Utiliser un moniteur transactionnel et des bases de données ACID - Sauvegardes - Empreintes cryptographiques (ex: SHA)
Non-répudiation Aptitude à prouver la réalisation d'un traitement dans un contexte donné *Exemple : prouver que M. Durand a bien signé notre contrat en ligne*	- Signature électronique - Jetons d'horodatage - Archivage de données, documents, logs

ENF de performances

ENF	*Exemples de dispositifs*
Rapidité Aptitude d'un système à présenter un temps de réponse (TR) acceptable pour un niveau charge donné. *Exemple: opération REST répondant en moins de 100 ms pour une charge de 2 TPS.*	- Optimisation du code - Utilisation de caches - Utilisation d'index en base de données
Scalabilité Aptitude d'un système à conserver des temps de réponse acceptables quand la sollicitation augmente *Exemple: opération REST répondant en moins de 100 ms pour une charge de 2 TPS et en moins de 150 ms pour une charge de 20 TPS.*	- Scalabilité verticale: on ajoute des CPU et de la mémoire au serveur - Scalabilité horizontale: on ajoute des serveurs derrière un répartiteur de charge - Scalabilité diagonale : les deux
Élasticité (= « super scalabilité ») *Aptitude d'un système à provisionner /dé-provisionner automatiquement des ressources en fonction de la charge afin de conserver des temps de réponses acceptables tout en optimisant le coût* *Exemple: opération REST répondant en moins de 100 ms pour une charge de 2 TPS et 1 POD Kubernetes et en moins de 110 ms pour une charge de 20 TPS et 4 POD*	- Cluster Kubernertes en auto-scaling - Cluster OpenStack en auto-scalling pour arrêter/démarrer des machines virtuelles
Efficacité *Aptitude d'un système à maitriser la quantité de ressource utilisée* *Exemple: Consommer maximum 200KWH/mois pour 1000 utilisateurs*	- Déploiement dans datacenters avec PUE proche de 1 - Utilisation de CDN (Content Delivery Network) au plus proche de l'utilisateur

ENF de fiabilité

ENF	*Exemples de dispositifs*
Disponibilité % de temps où un système fonctionne de façon nominale durant une plage d'ouverture du service. (On parle de HA (Haute disponibilité) quand >= 99 %) *Exemple : base de donnée disponible en « quatre neufs » = 99.99 % du temps (sur plage d'ouverture).*	- Clusters (note: le clustering répond à la fois aux exigences de performance et de disponibilité) - Hardware redondant (RAID, …) - Hébérgement de l'application dans un datacenter hautement disponible (ex : Tier 4)
Robustesse Aptitude d'un système à continuer à fonctionner correctement suite à des erreurs techniques (erreur système, stress important, requêtes incorrectes, attaques...) *Exemple : Arrêt gracieux d'un batch sans plantage et avec logs complets suite à un dépassement mémoire.*	- Bonne gestion des erreurs dans le code - Rejeux automatiques sur erreurs techniques intermittentes - Gestion des reprises sur erreurs des jobs
Résilience *Aptitude d'un système à revenir à son état initial suite à des erreurs techniques* *Exemple: Disposer à tout moment et même après une panne d'une partie du système d'une capacité de traitement constante.*	- Déploiement Kubernetes fixant un état désiré et un nombre de POD - Système de failback sur un cluster (un slave promu master redevient slave après rétablissement)

Il existe une ENF plus récente, l'**anti-fragilité** (sorte de « super résilience ») : le **système apprend** et sort plus fort de l'événement
voir aussi le « chaos computing »

ENF de maintenance

ENF	*Exemples de dispositifs*
Évolutivité Aptitude d'un système à implémenter de nouvelles fonctionnalités à un coût reproductible *Exemple : changer le logo de l'entreprise sur le réseau social prend 1 JH du dev à la production, pas 100 JH.*	- Principalement des principes d'architecture logicielle (ex : SOLID, découplage du code par architecture hexagonale, bounded contexts DDD, ...) - Architecture modulaire à base de contrats comme les architectures microservices
Testabilité Aptitude d'un système à être testé (en architecture technique, on parle surtout de tests d'intégration et de tests système) *Exemple : possibilité de bouchonner facilement une application SPA en remplaçant une API REST par un simple stub qui renvoie des données statiques.*	- Principalement des bonnes pratiques de développement comme le TDD (Test first) - Cohésion forte des modules, faible couplage permettant de les bouchonner facilement

La testabilité est un **pré-requis à l'évolutivité !**
un système doté de nombreux tests automatisés peut être modifié avec confiance qu'on n'y introduira pas de régressions

ENF d'exploitation

ENF	*Exemples de dispositifs*
Exploitabilité *Aptitude d'un système à être supervisé, lancé et à remonter des alertes pertinentes* *Exemple : courriel envoyé quand il ne reste que 10% de mémoire disponible*	- Sondes de supervision système, traces applicative - Système d'hypervision qui consolide les métriques - Ordonnenceur de batchs - Système d'alerting sur seuils pré-programmés
Auditabilité *Aptitude d'un système à tracer les événements pertinents* *Exemple : détection d'intrusion (IDS) ou traces métier de l'activité d'agents pour détecter des fraudes*	- Logs centralisés dans une stack ElasticSearch Fluentd Kibana - Enregistrement des traces métier dans une base NoSql

l’architecture agile (introduction)

Les patterns principaux

A faire

Le **design émerge** par la collaboration et l'exécution

Concevoir le **système le plus simple** qui puisse fonctionner

Baser sa décision sur la mise en ouvre (**POC**)

Concevoir pour la **testabilité**

Design itératif et évolutif

A éviter

Reproduction de solutions existantes ou imposées

Optimisations prématurées

Faire confiance aux éditeurs ou aux modes du moment

Système uniquement testable par l'IHM

Design gravé dans le marbre, prévu des mois à l'avance

La réversibilité

- Enrico Zaninotto et Martin Fowler :
  Le travail le plus important d'un architecte IT est de produire de la **réversibilité**
  - **API évolutive** (ex : propriétés génériques via une map de clés valeurs)
  - **Modèles de données flexibles** (ex : nosql vs relationnel)
  - **Réversibilité de l'infrastructure** (cloud et conteneurs)

On voit souvent le contraire (règles gravées dans le marbre, difficile à changer)

Le fail-fast

- L'architecte agile aide à révéler les problèmes au plus tôt
  - **POC** (Proof Of Concept) et POV (Proof Of Value). Aller suffisamment loin.
  - Se **former**/s'auto-former aux technologies pour **juger correctement** avec son expérience
  - Projets pilotes **jusqu'à l'exploitation** pour évaluer tous les aspects
  - Grand rôle des **pairs** (réseau, séminaires, conférences, blogs, coding sessions...)
  - Lutter contre le syndrome de Stockholm IT

La documentation

- **Web** et **collaborative** (wiki), moins de bureautique
  - Au moins pour les normes et référentiels
- Capitaliser les échanges sur des **documents partagés** (voir « Outillage et diagrammes »)
- Diagrammes affichés à tous
- Diagrammes générés à partir de texte (voir « Outillage et diagrammes »)
- Mettre en place un référentiel d'Architecture Decision Record (**ADR**)
- **Spécifications exécutables** (vivantes), voir BDD

Exemple d’ADR (dans un wiki par exemple)

Décisions **notables** uniquement, **dates** et **raisons** du choix

```
0004-verouillage-dossiers-pendant-expertise.md  (fichier texte Markdown ou Asciidoc déposé dans un dépôt Git)


## Statut
VALIDE en atelier du 08/11/2018 avec HUI


## Contexte
* Un dossier peut être modifié par : les utilisateurs (référent pouvant être logué
sur plusieurs machines en même temps) et par le batch.
* Aucun dispositif de merge de dossiers n'est encore prévu (voir #69)
* L'expert est 'prioritaire', aucune des données qu'il saisi ne doit être
perdue ou écrasée.


## Décision
* Le dossier sera verrouillé en lecture seule sauf par l'expert entre le statut
 EXPERTISE_EN_COURS et le statut EXPERTISE_ENVOYEE.


## Conséquences
* Aucune modification dans TMPP ne sera prise en compte (mais sera loguée)
tant que l'expertise ne sera envoyées.
* L'IHM empêchera toute mise à jour du dossier.
```

En savoir plus

Le design

- Toujours **revenir au besoin** et aux exigences
- Donner les règles mais toujours donner **au moins un exemple**
  - S'il n'y a **pas d'exemple**, c'est qu'il n'y a **pas de besoin**
  - **Behavioral Driven Development** : spécifications par l'exemple
- **Laisser le design émerger** naturellement
- KISS, YAGNI, Knuth (voir « Les règles du jeu »)
- Penser à la **testabilité**
- Le **métier** avant tout (Domain Driven Development)

Le rôle de l'architecte agile

Pas/plus :

- Imposer des choix
  - *Dossier avec le MPD et 60 diagrammes de séquence*
- DA à la chaîne de projets en projets

Mais plutôt :

- Guider, donner des directives générales
  - *« Plutôt du Java, pas du .Net ici »*
- Faire profiter de son expérience
  - *« Attention aux performances avec les dblinks »*
  - pair-programming
- Être disponible, suivre le projet, s'enrichir de feedback, être un artisan (craftmanship)

le zoo de l'infrastructure

L'architecte solutions assemble des composants d'infrastructure

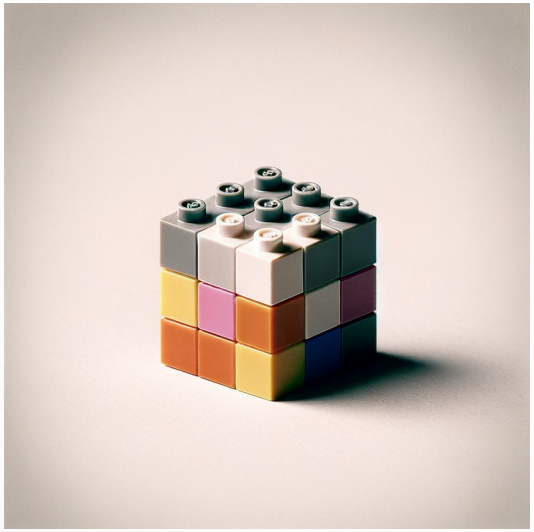
Ces composants d'infrastructure sont exécutés ou sont utilisées les modules applicatifs

Exemples :

- *Un serveur d'application Tomcat qui exécute une application Java ;*
- *Une base de données MariaDB qui persiste les données d'un module ;*
- *Un serveur SMTP qui permet à un module d'envoyer des courriels.*

L'architecte solutions **choisit** et **assemble** les briques, l'intégrateur les **configure**.

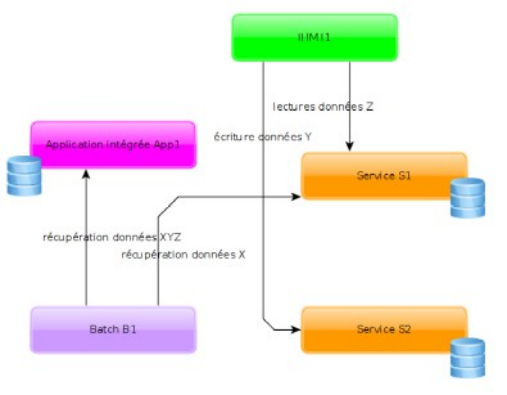
Il est recommandé à un architecte solutions de **participer à l'intégration** pour être s'enrichir des retours terrain en terme de sécurité et performances notamment.



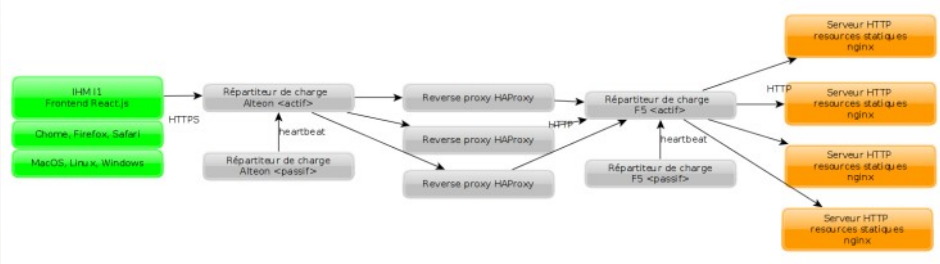
Impédance entre les points de vue d'architecture

On en montre pas la même chose en fonction de l'interlocuteur (notion de '**point de vue**'). Un point de vue est constitué de **vues**.

Quelques exemples de vues courants :

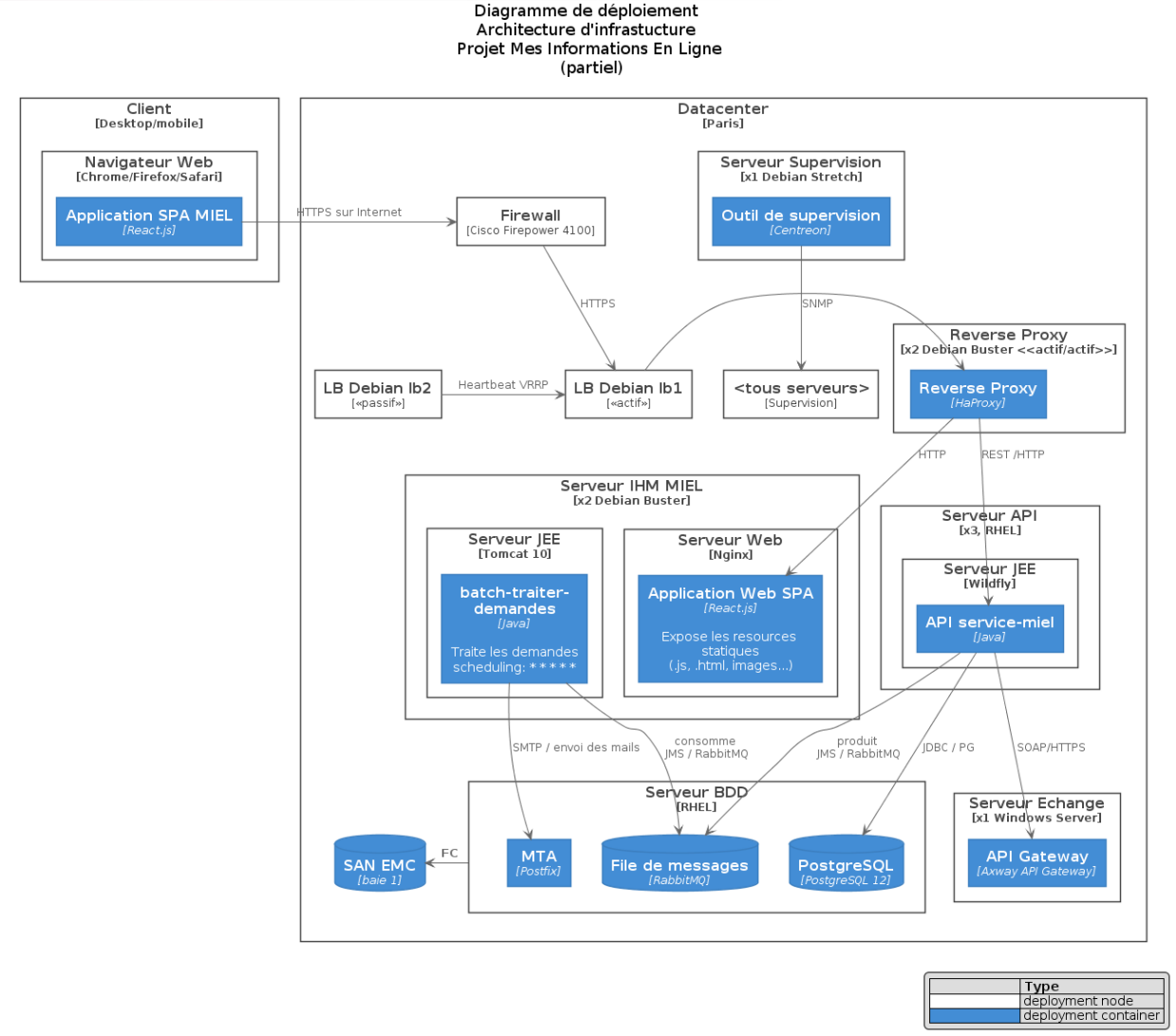


Vue infrastructure : déploiement des composants d'infrastructure avec leurs serveurs cibles et le détail de leur instanciation en clusters



Vue applicative : modules, composants et flux principaux. Pas de notion technologique ni de déploiement.

Un exemple d'architecture d'infrastructure



Critères de choix généraux

Compétences internes
- Équipes formées ?
- Adhésion, gestion du changement

Intégration avec existant
- Outillage de déploiement, exploitation, supervision, sauvegardes...
- Compatibilité infrastructure réseau, stockage
- Compatibilité système (OS)
- Support sur les clouds

Prise en main
- Documentation
- Support
- Formations

Coût
- Certains composants propriétaires en M€ (ex: SGBD Oracle : 30K€ / CPU min)
- Penser aux besoins de licence sur tous les environnements, pas que production
- Coût des prestataires sur le marché

Performances
- Toujours qualifier la solution via des benchmarks en amont et aval

Ubiquité
- Privilégier les solutions mainstream à l'excellence technique
- Privilégier les solutions Open Source
- Facilité de trouver des compétences

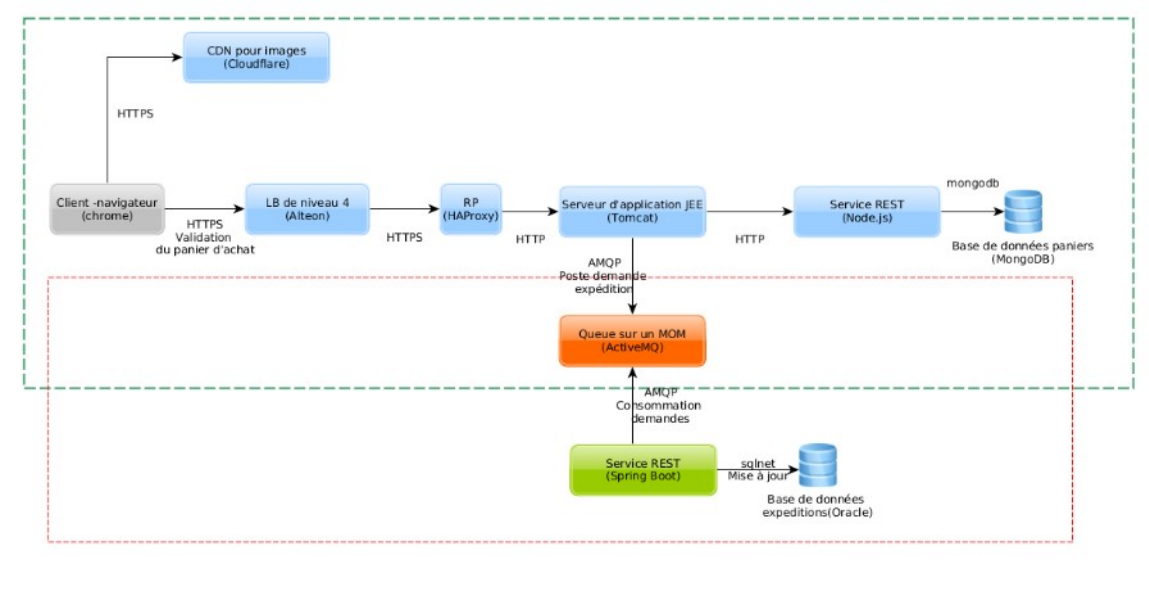
La frontière entre les catégories est souvent ténue

- HAProxy sert souvent à la fois de **load balancer** et de Reverse Proxy
- NGINX peut servir de Reverse proxy, load balancer de niveau 7 ou **serveur Web**
- Une **API Gateway** propose des fonctionnalités comprenant celles d'un reverse proxy
- Un **Reverse proxy** ou un forward proxy servent de **cache** mais pas seulement
- ...

Notion de chaîne de liaison

- Une chaîne de liaison regroupe **tous les composants impliqués dans un traitement effectué de façon synchrone**
- La disponibilité d'une chaîne de liaison = la disponibilité du **maillon le plus faible**
- **L'asynchronisme** (via queues par exemple) améliore grandement la robustesse des chaînes de liaison en les segmentant (découplage temporel) au prix d'une plus grande complexité

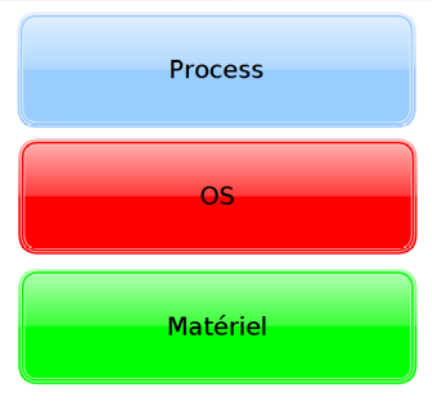
Exemple de deux chaînes de liaison simples, l'une servant de source à l'autre :



Les OS (systèmes d'exploitation)

Rôles

- Permet d'**exécuter des processus sur un hardware physique ou virtuel**, donne accès aux ressources (IO, réseau, graphique...)
- Fourni de nombreuses **librairies** et utilitaires (libc, CLI, shells...)



Exemples

- Certains orientés desktop, serveur ou embarqué
- Grandes familles (comme Unix ou Windows) ; sous familles (Unix : Gnu/Linux, MacOS-Darwin, AIX, Solaris...) et distributions (Redhat, Debian...)

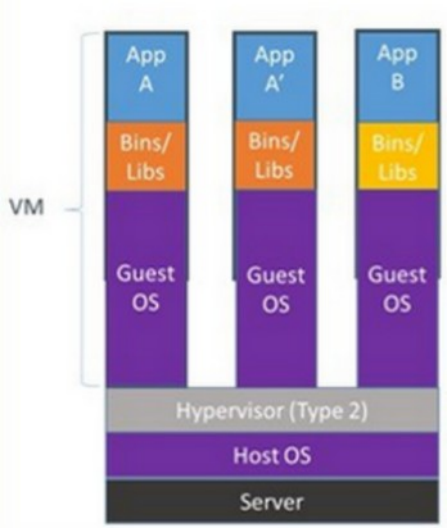
Quelques critères de choix

- Besoin spécifiques (ex : CoreOS)
- Compatibilité avec cloud (IaaS)

Hyperviseurs et machines virtuelles

Rôles

- Un hyperviseur **permet un ou plusieurs OS invités depuis un OS hôte**
- L'hyperviseur émule un hardware virtuel et configurable
- Chaque OS invité est complètement isolé (en théorie)



Source : ZDNet

Apports

- **Optimisation de l'utilisation des serveurs** par complémentarité des invités : moins de serveurs physiques
- Grande flexibilité dans la scalabilité verticale des serveurs
- Snapshots pour les backups

Limites

- Consolidation des invités rapidement limitée par la mémoire réservée
- Coûts des solutions propriétaires

Exemples

- OSS : KVM, Xen, Qemu, VirtualBox
- VMWare
- Microsoft Hyper-V

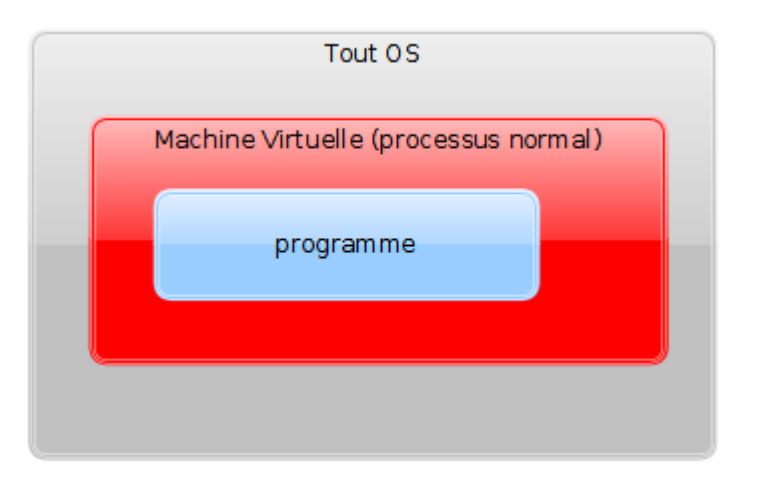
Quelques critères de choix

- Open source / propriétaire
- Performance / limitations

Machines virtuelles logicielles

Rôles

- Fournir un **environnement d'exécution universel à un programme**
- Exemple : une classe Java exécutée dans une JVM, quelque soit l'OS (Linux, MacOS, AIX, Windows...) et le matériel (X86, AMD64, ARM, Sparc...)



Apports

- **Write once, run everywhere** : le runtime produits est agnostique de l'OS et du matériel sous-jacent
- Meilleur sécurité (sandbox)
- Limites (Xmx..)
- Bonnes performances grâce au Just In Time

Limites

- Utilisation mémoire souvent importante
- Overhead significatif sur certains cas
- Installation/mise à jours des VM logicielles

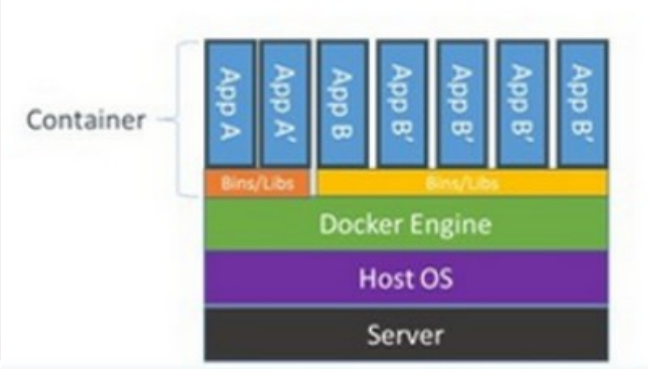
Exemples

- JVM
- Framework .NET

Containeurs

Rôles

- **Exécution d'un process (en général un démon) dans un environnement isolé mais partageant le même kernel** que les autres containeurs
- Instanciation d'une image préparée
- S'appuie sur des fonctionnalités du noyau Linux : namespaces (isolation), cgroups (limites accès aux ressources), FS de type union (stockage images)...



Source : ZDNet

Apports

- Permet de **pousser beaucoup plus loin l'optimisation des serveurs** car la mémoire est partagée entre containeurs
- Rend l'installation des composants logiciels très simple
- Approches **DevOps et IaC** via les images (voir cours Intégration)

Limites

- Risque sécurité si mauvaise gouvernance (exemple : images du DockerHub sans contrôle)

Exemples

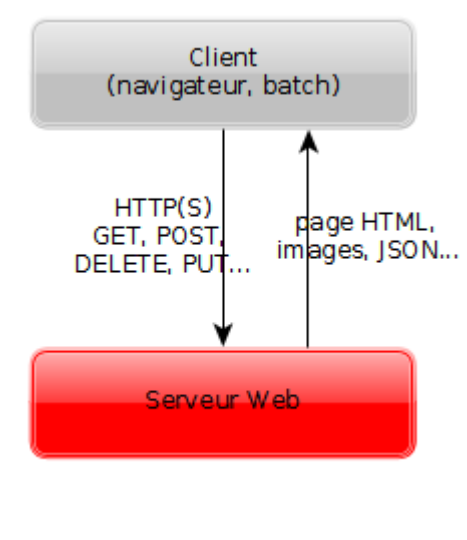
- Docker, containrd, rkt (Redhat), LXD, V-Server, Hyper-V

Serveurs Web

Protocole : **HTTP(S)**

Rôles

- **Servir en HTTP(S) des ressources statiques** (images, .html, .js...) et éventuellement des pages dynamiques (PHP, CGI...)
- Propose souvent des fonctions complémentaires de reverse proxy, compression, sécurité, traces, FTP...



Exemples

- **Apache 2** (46% des sites webs), **NGINX** (39%), Microsoft IIS

Critères de choix

- Outillage d'automatisation déjà utilisé dans l'entreprise
- Facilité de **configuration**
- **Extensions** disponibles

Note: les CDN (Content Delivery Networks)

tendent à remplacer les serveurs Web 'on premise' pour le contenu statique des sites publics. Ce sont des serveurs HTTP situés au plus prêt des clients et disposant d'énormes bandes passantes

NAS (Network Attached Storage)

Protocoles :
NFS, CIFS, SFTP, Webdav sur HTTP(S)...

Rôles et fonctionnement

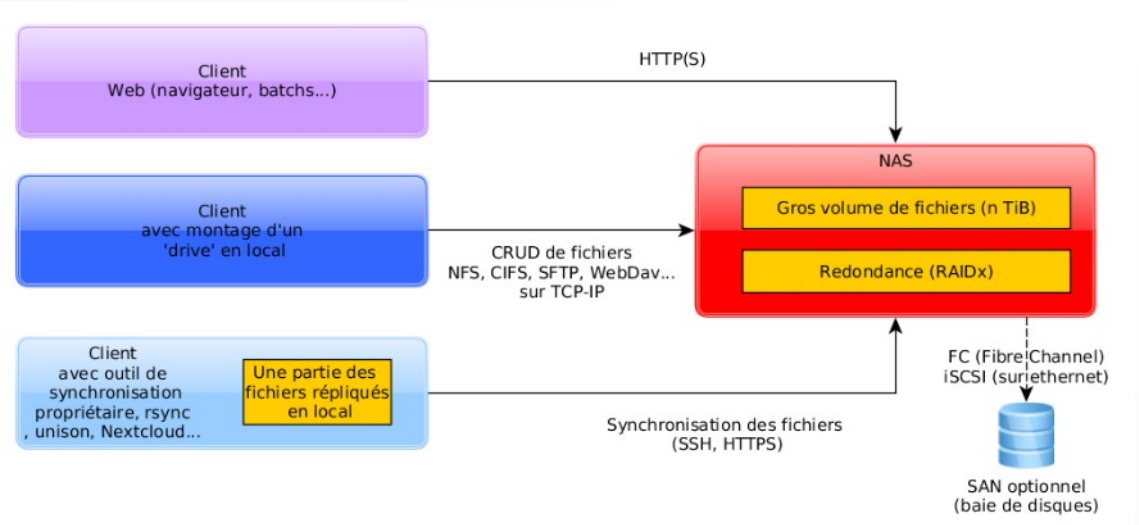
- **Stocker et servir depuis le réseau** de grands volumes de données
- Petits NAS : stockage dans des disques internes en RAID (RAID5 ou RAID1 plus coûteux)
- NAS grands comptes : stockage sur baie de disque SAN (Storage Area Network)
- Nombreuses offres NAS IaaS (Infrastructure As A Service)
- Montage en local ou synchronisation
- Nombreux protocoles

Exemples

- Synology, Cisco S-Series...
- Cloud : NAS OVH, Amazon S3, Glacier pour archivage...

Critères de choix

- Mode d'utilisation (**montage ou synchronisation**)
- Performances, sécurité (RAID par ex), confidentialité, autorisations
- Appliance / serveur (rare) / cloud



Protocole : **TCP-IP**

LB (Load Balancer) - répartiteur de charge

Rôles

- Router requêtes vers **plusieurs serveurs vus comme une seule machine** (**VIP** = Virtual IP)

Fonctionnement

- Plusieurs types de LB : niveau 3 ou 4 (réseau) et 7 (applicatif), souvent couplés
- Plusieurs **algorithmes de répartition**, les plus courants : **round-robin** (rr), weighted rr, **least-connection**, hash ip source/url
- Persistance de session (**affinité** par IP source, redirection vers serveur final, lecture de cookie, ajout de cookie)
- Réponses : repassent par LB (NAT / tunneling) ou vers clients (Direct Routing)

Apports

- Assure la haute disponibilité
- Permet la scalabilité horizontale

Limites

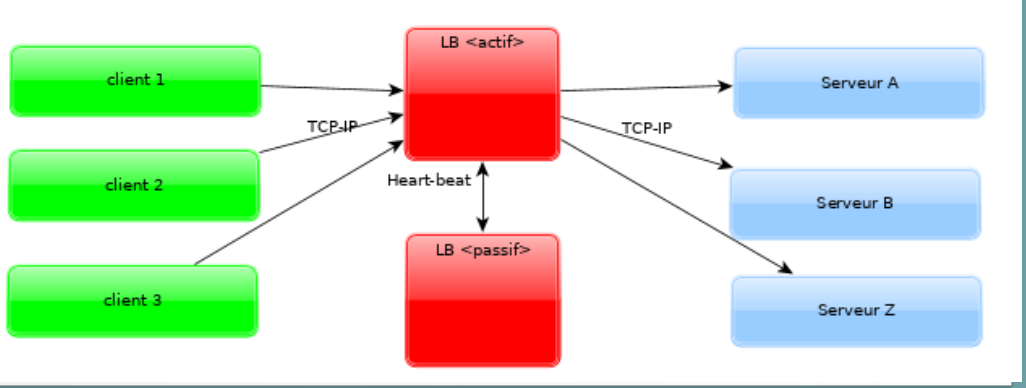
- LB de **niveau 3 ou 4 plus performants** que niveau 7 mais moins configurables
- Penser au HA du LB lui-même pour ne pas créer de **SPOF**

Exemples

- Niveau 3/4 : LVS, F5 Big-IP, Google Cloud LB
- Niveau 7 : HAProxy, NGINX, F5 Big-IP, Alteon

Critères de choix

- Appliance ou serveur
- Sondes healthcheck disponibles
- Algorithmes de routage
- Dispositifs de failover entre répartiteurs



Alternative (LB « du pauvre ») :
le **round robin DNS**
mais pas de persistance de session,
pas de health checks, quid des caches ?

Reverse proxy (RP)

Quelle différence avec un LB de niveau 7 ?
Ne fait pas que distribuer les requêtes : les modifie en entrée et sortie.

Rôles

- Sert de **façade à un ou plusieurs services de niveau 7**, en général **HTTP(S)**. Les clients ne voient que le RP.
- Le RP contrôle les requêtes (caching, analyse Headers, analyse anti-XSS/CSRF, firewall, anti-DDOS, persistance de session)
- Le RP modifie les requêtes (terminaison TLS, décompression, réécriture URL...) et refait la requête pour le compte du client
- Répartition de charge optionnelle
- Le RP modifie les réponses (compression gzip/deflate, TLS, modification de headers)

Apports

- Gros **apport en sécurité**
- Apporte beaucoup de **flexibilité** (réécriture URL)
- **Performances** (caching, compression)

Limites

- Beaucoup plus consommateur qu'un LB car traite les contenus, on le place souvent derrière des LB de niveau 3, 4.
- Penser au HA du RP lui-même pour ne pas créer de **SPOF**. Exemple: heartbeat avec Keepalived ou derrière LB de niveau 3, 4.

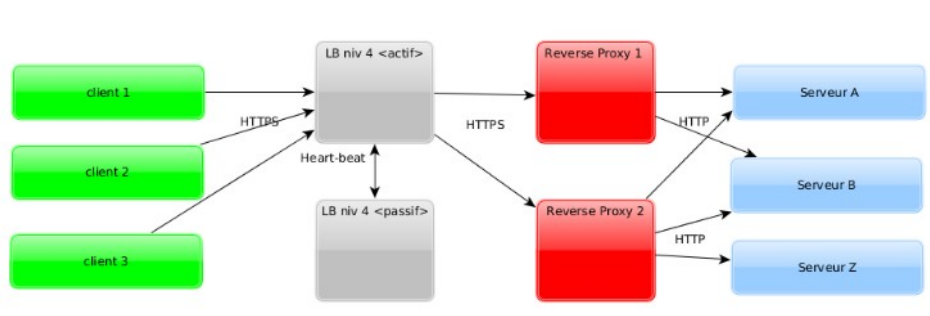
Exemples

- HAProxy, NGINX
- Pour les containeurs : Traefik, Envoy
- Varnish : Cache RP (cache les pages dynamiques)

Critères de choix

- Appliance ou serveur
- Fonctions nécessaire et non redondantes

Exemple de typologie classique :



Proxy (ou «Forward proxy»)

Quelle différence avec un reverse proxy ? Proxy dans le sens système d'information (SI) vers Internet, RP dans le sens Internet vers SI

Rôles

- Sert de **mandataire pour les accès à Internet**, utilise le SNAT pour masquer l'IP interne du client et lui transmettre la réponse
- Permet des **contrôles d'accès** (black/white lists d'URL, traces)
- Met les ressources (HTML, JS, images, vidéos...) en **cache**

Apports

- Sécurité
- Performances (caching)

Limites

- Penser à la haute disponibilité, une défaillance d'accès à Internet est aujourd'hui très problématique pour les utilisateurs et pour les composants
- Complexifie de nombreuses configurations (maven, mises à jours...)

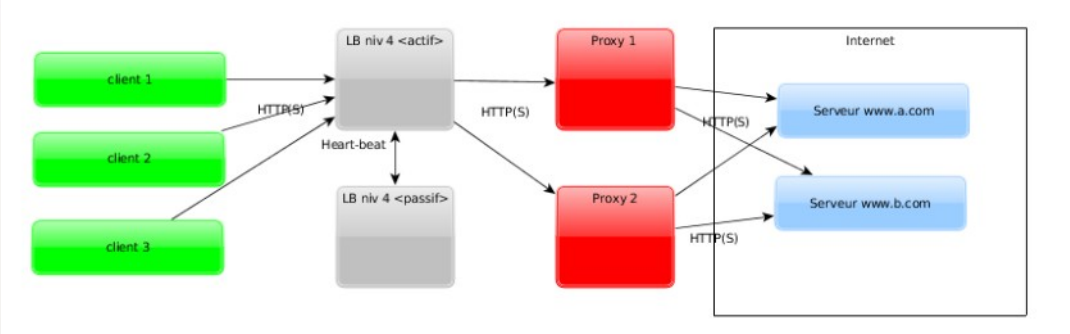
Exemples

- Apache Squid, NGINX, BlueCoat

Critères de choix

- Appliance ou serveur

Exemple de typologie classique :



Serveur DNS (Serveur de noms)

Protocole : **DNS**

Rôles et fonctionnement

- Permet de récupérer l'**IP V4 ou V6 correspondant à un nom de domaine**, exemple : `www.florat.net -> 37.187.120.199`
- Presque toutes les communications (mêmes internes LAN) passent par un DNS
- **Structure arborescente** : racine (`.`) > TLD (`com`) > domaine (`thoughtworks`) > sous domaine (`www`) donne `www.thoughtworks.net.`
- Un domaine peut correspondre à **plusieurs IP** (Round Robin DNS)
- Plusieurs **enregistrements** (record) par domaine, principaux : **A** (IP V4), **MX** (serveur de courriel), **CNAME** : alias vers un A,...

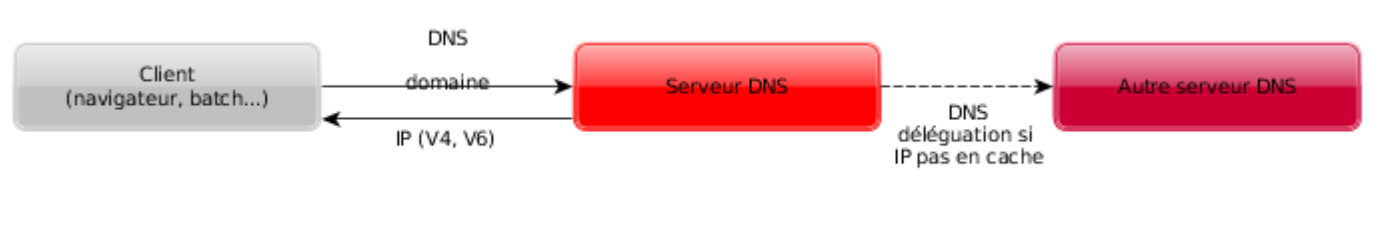
Limites

- Attention aux nombreux **caches** (exemple : JVM)
- Attention au TTL de prise en compte des modifications

Exemples

- Bind, CoreDNS dans clusters Kubernetes
- Microsoft DNS dans ActiveDirectory

Utiliser des IP en dur est une très mauvaise pratique, utilisez des noms de domaine, même en interne



Annuaire LDAP

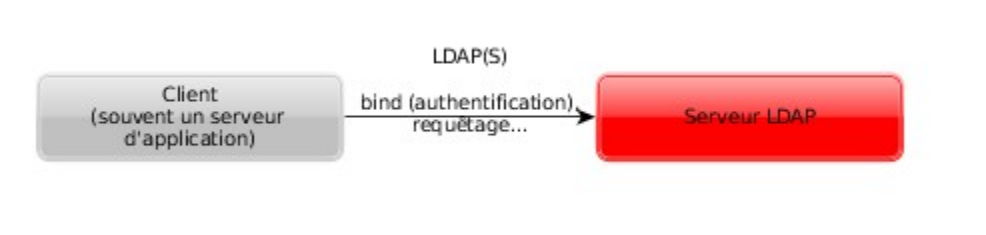
Rôles et fonctionnement

- Stocke sous **forme arborescente des listes d'objets** (en général des organisations et des personnes ou des parcs informatiques)
- Exemple de DN (Distinguish Name), identifiant unique :

  `cn=Martin Fowler,dc=thoughtworks,dc=com`
- Sert souvent à **stocker les identifiants** (login/mots de passe) et les **autorisations**. Utiliser en priorité le LDAPS (LDAP sur TLS)
- Exemple type d'utilisation : serveur d'application vérifiant des identifiants fournis dans une requête cliente.

Exemples

- OpenLDAP
- Oracle Internet Directory
- Microsoft ActiveDirectory

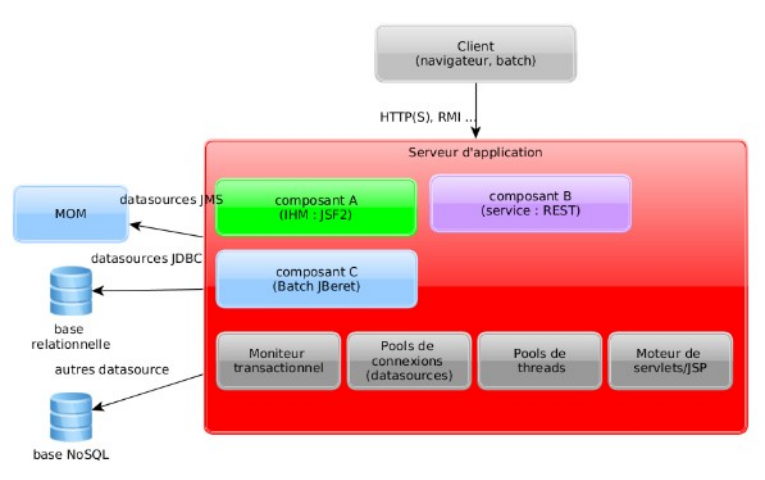


Serveurs d'application

Protocole :
SOAP/REST sur HTTP(S)
RMI/IIOP

Rôles

- Proposer un **contexte d'exécution à des modules** (IHM, services ou batchs)
- Exemple de fonctionnalités : moniteur transactionnel, moteur de servlets, gestion des datasources, pools, EJB, batchs...
- Peut être utilisé derrière un serveur Web qui sert les ressources statiques (de moins en moins utilisé)



Limites

- A la base, conçus pour exécuter **plusieurs modules** (ear, war par ex) mais la pratique évolue pour les applications cloud-native (un composant applicatif par serveur applicatif)
- Nombreuses normes (JSR) mais attention au **vendor locking**
- Peut être complexe à prendre en main

Exemples

- Serveurs Java/**JEE** : Tomcat, Spring Boot, JBoss Wildfly, Weblogic, Websphere
- Serveur Web avec extension PHP (stack **LAMP** par exemple)
- **Node.js** (EcmaScript coté serveur)

Critères de choix

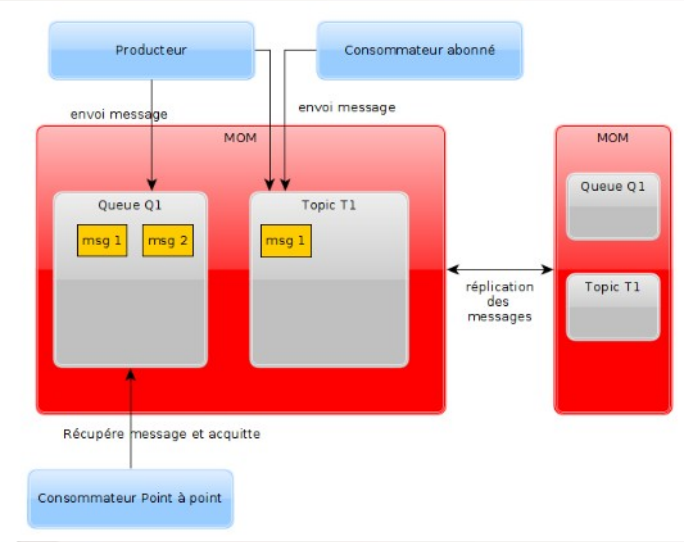
- Facilité de configuration
- Facilité d'exploitation

MOM (Message Oriented Middleware) ou «Message Broker»

Protocole : **MQ, AMQP ...**

Rôles

- **Stocker des messages** à traiter au fil de l'eau
- **Très haut débit**, on lisse la charge des consommateurs
- Forte **robustesse** contre la perte de message (acquittements, réplication..)
- Deux typologies : **Point à Point** ou **Topic** (abonnement)



Avantages

- Permet le **découplage par asynchronisme** entre deux modules (ex : arrêt du consommateur sans bloquer le producteur de données)
- Facilite la transformation des messages

Limites

- Dans la plupart des MOM, l'**ordre de consommation n'est pas assuré** (JMS)
- Peut être complexe à mettre en œuvre (surtout MOM anciens)

Exemples

- JMS : RabbitMQ, ActiveMQ
- Kafka, Apache Pulsar
- Propriétaire : IBM MQ (ex-MQSeries)

Critères de choix

- Intégration avec existant (JMS si JEE)
- Capacité de HA
- Performance
- Facilité de configuration

ESB (Enterprise Service Bus)

Rôles

- **Colonne vertébrale** d'un SI SOA
- Nombreuses fonctionnalités : routage, service discovery, transformation de message, agrégation de services, orchestration, suivi BAM...
- **Adaptateur universel** : permet d'interconnecter la plupart des technologies

Apports (si bien utilisé)

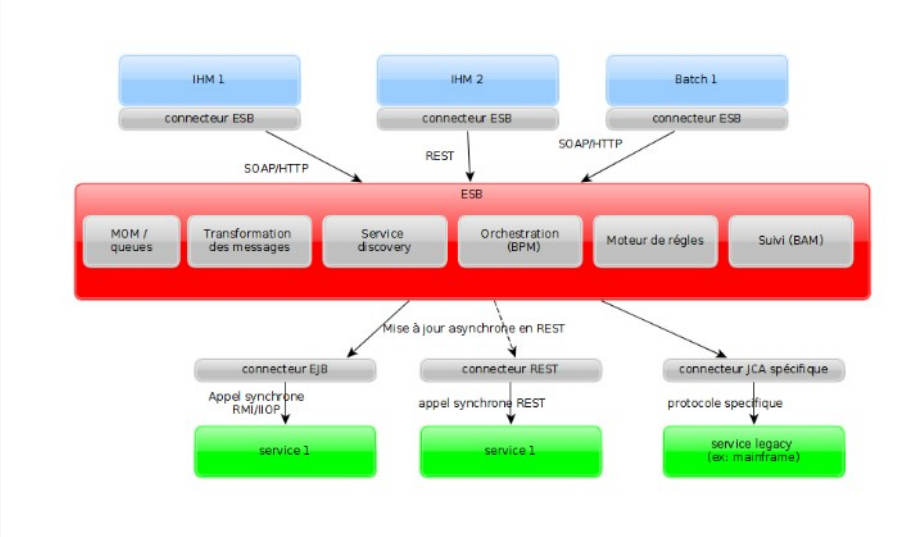
- Simplifie l'intégration de SI hétérogènes
- Intègre de nombreuses fonctionnalités build-in : BAM, BPM, logging, métrologie...

Exemples

- Apache ServiceMix, Mule ESB, Petals, Fuse ESB
- Oracle OSB, Talend ESB, Microsoft Biztalk, TIBCO, IBM Websphere ESB ...
- Composants d'intégration : Apache Camel

Limites

- Nombreux échecs SOA. Effet "silver bullet", manque de gouvernance.
- Très cher (propriétaire), complexe
- Ajoute de la **complexité**, nécessite des compétences pointues
- Possibles **SPOF**
- N'est pas obligatoire ! Certains grands SI SOA n'en utilisent pas (avec succès)
- **De moins en moins utilisé avec l'émergence des micro-services**



API Gateway

Rôles

- Point d'entrée d'une architecture microservices depuis des consommateurs d'API
- Gère la sécurité, l'adaptation des appels de services et réponses pour limiter les appels et la taille des réponses, la documentation, la sécurité, le circuit breaking, la compression etc...
- Bref : **point central des aspects techniques** des services exposés

Apports

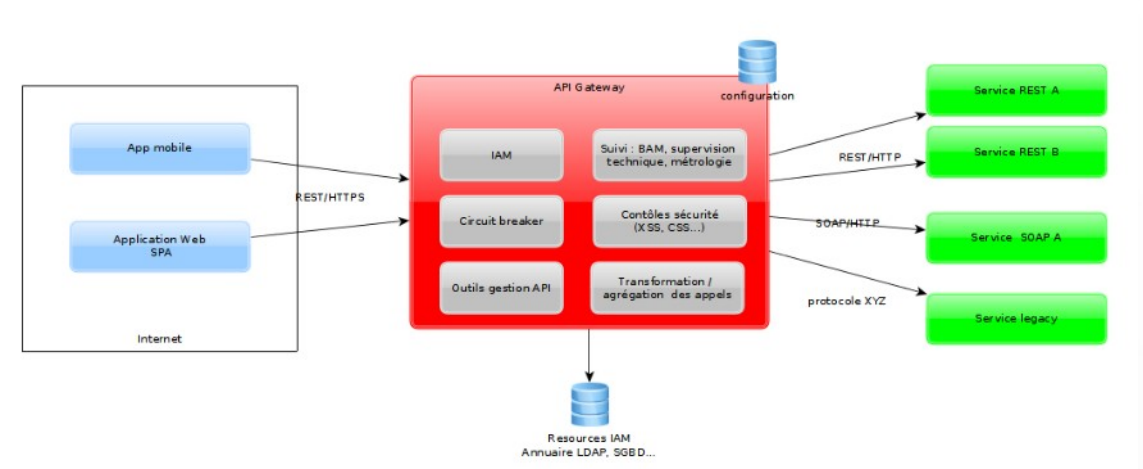
- **Meilleure sécurité** car centralisée, plus à gérer dans chaque service
- Découple le SI des API exposées, **flexibilité**
- Permet un très bon **suivi** des appels

Limites

- Peut ajouter un **overhead** mais souvent négligeable. Si bien utilisé, peut même améliorer les performances vues du client en factorisant les appels.
- Nécessité de compétences (**complexité moyenne**)
- Peut conduire à des problèmes si mal maîtrisé (ex: doublons causés par failover)
- Encore un composant d'infrastructure à gérer mais en général, bon ROI

Exemples

- Gravitee, Netflix API Gateway, Axway API Gateway, Kong ...



Note : de nombreux composants (proxy, RP, serveurs DNS, bases de données, kernel…) offrent des fonctions de cache embarquées. Nous décrivons ici les composants entièrement dédiés à cet usage.

Caches / data grids

Rôles et fonctionnement

- Un cache sert à **stocker temporairement des réponses à une requête** (HTTP, DNS, BDD, ...) pour éviter de la refaire inutilement
- **Cache Hit** : donnée disponible. **Cache miss** : le cache doit réémettre la requête.
- Stockage clé/ valeur
- **Stockage temporaire** (l'appelant doit s'attendre à ne plus trouver la donnée)
- Divers **algorithmes d'éviction** (Least Frequency Used, Least Recently Used...)
- Stockage mémoire (ex : memcached), disque (SSD) ou les deux
- Peut former un cluster (**data grid**) : **caches distribués**

Apports sur les performances

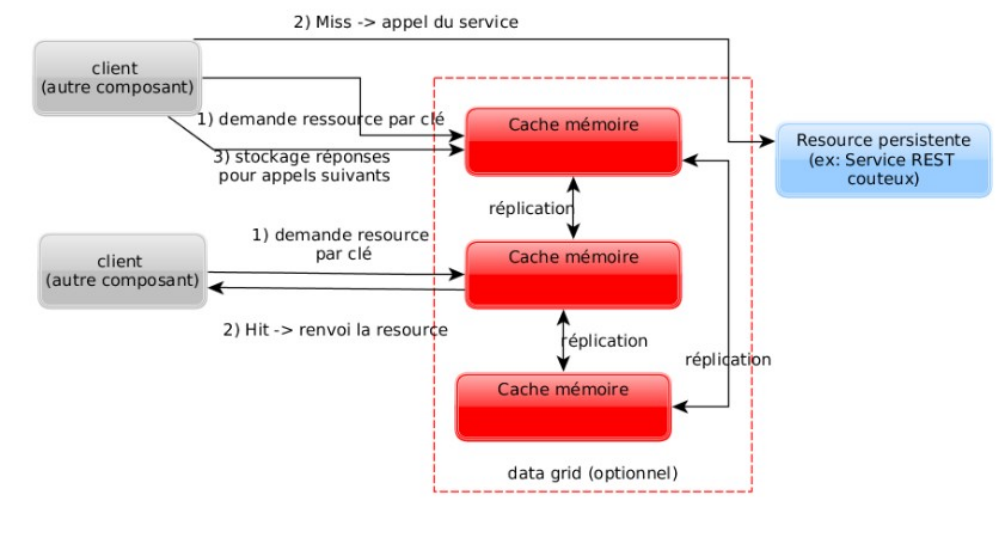
- **Diminue fortement les besoins en bande passante**
- Améliore souvent les temps de réponse
- Peut améliorer la scalabilité

Limites

- **Complexifie l'architecture**, surtout les caches distribués,
- Impacts applicatifs (ex: entêtes HTTP Cache-Control)
- **Peut être contre-productif** sur les temps de réponse si inadapté
- Risque de **SPOF** si mal sécurisé

Exemples

- Caches mémoires : Redis , Memcached
- Caches mémoire distribués : JBoss Infinispan, Hazelcast, Oracle Coherence



Techniquement proche des caches mais : persistants, très sécurisés et usage différent

Services clés-valeurs de coordination

Rôles et fonctionnement

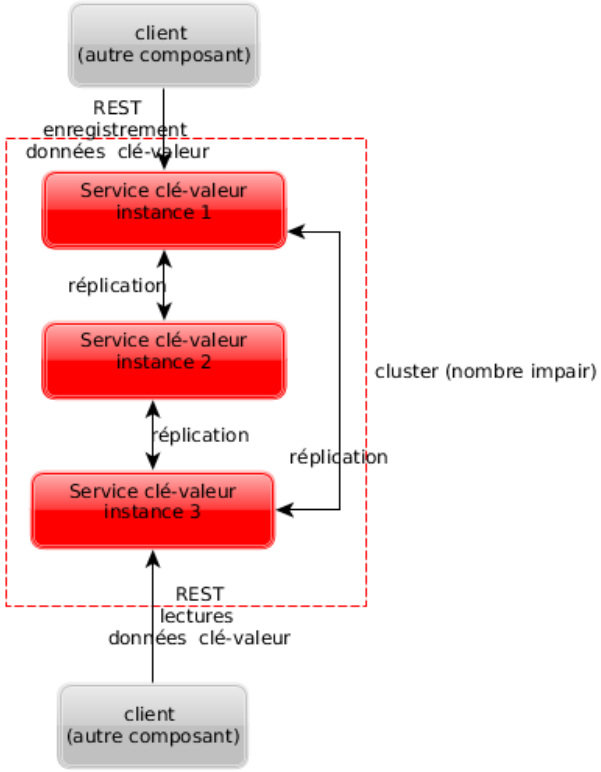
- A émergé récemment avec les **containeurs, les architectures micro-services et Cloud Native**
- Sert à stocker :
  - Les **données de configuration** système ou applicatives (ex : port d'une base de donnée)
  - Les informations de service discovery (ex: service rest `xxx` accessible à `https://serveur-yyy/xxx`)
  - Des **données de travail partagé** (ex: construction d'un rapport par plusieurs machines)
- Stockage clé-valeur persistent (contrairement à un cache) et source unique d'information
- Exemple : un orchestrateur de containeurs lance un serveur Web sous forme de containeur et enregistre son URL d'accès dans la base clé-valeur (service discovery)

Apports

- Composant **simple**
- **Sécurisé** (certificats)
- **Haute disponibilité (HA)**
- Annuaire central des architectures hautement distribuées et éphémères (ex: Kubernetes)

Exemples

- Consul
- Etcd (système de persistence de Kubernetes)
- Apache ZooKeeper



Bases de données

Protocoles :
postgresql, mongo, mysql, sqlnet (Oracle)...

Rôles

- **Persister des informations durablement**
- Gérer l'**intégrité des données** et les accès concurrents (lecture/écriture)
- Chaque BD possède son protocole d'accès (mongodb, postgres, sqlnet...) en général utilisé derrière un driver standard (ex: JDBC)

Type

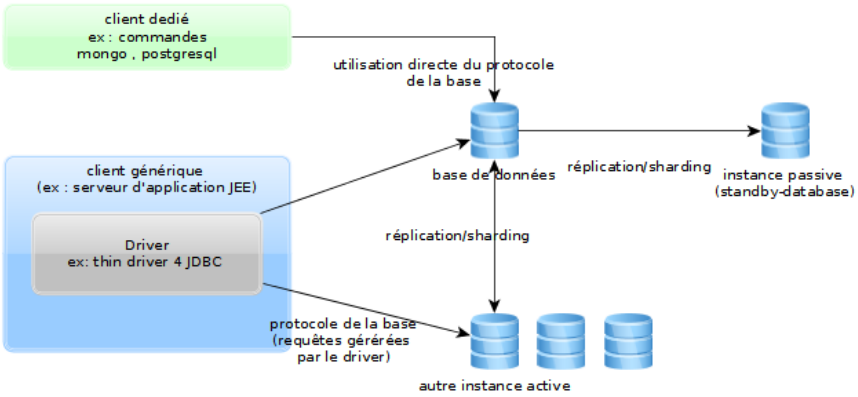
- **Relationnel** (SGBDR) basé sur des tables (tableaux à deux dimensions) et des relations entres tables. Requêtes en SQL. Gestion des transactions en mode ACID.
- **NoSQL** (Not Only Sql) : bases orientés documents (JSON), graphes ou key-value stores. Requêtage en SQL-like ou spécifique, gestion des transactions en mode "eventually consistent, cf. théorème CAP.
- Autres types spécifiques : base objet, time series ...

Exemples:

- Relationnel : Oracle, Mysql/MariaDB, PostgreSQL, DB2...
- NoSQL : MongoDB, CouchDB, Cassandra, OrientDB ...

Critères de choix

- Relationnel pour les modèles complexes (jointures), peu distribués, volumes faibles
- NoSQL adapté à des modèles simples, peu de relations, distribution forte, volumes importants
- Besoins spécifiques : utiliser solutions dédiées (ex : bases time series pour données de supervisions)



Composants emails (MTA, MDA, MUA)

Protocoles :
SMTP, POP3, IMAP sur TLS ou pas

Rôles

- MUA (**Mail User Agent**) : envoyer / recevoir des courriels
- MTA (**Mail Transfert Agent**) : Router les mails vers un autre MTA en SMTP (sur TLS ou pas)
- MDA (**Mail Delivery Agent**) : exposer les courriels (en IMAP / POP3 sur TLS ou pas)

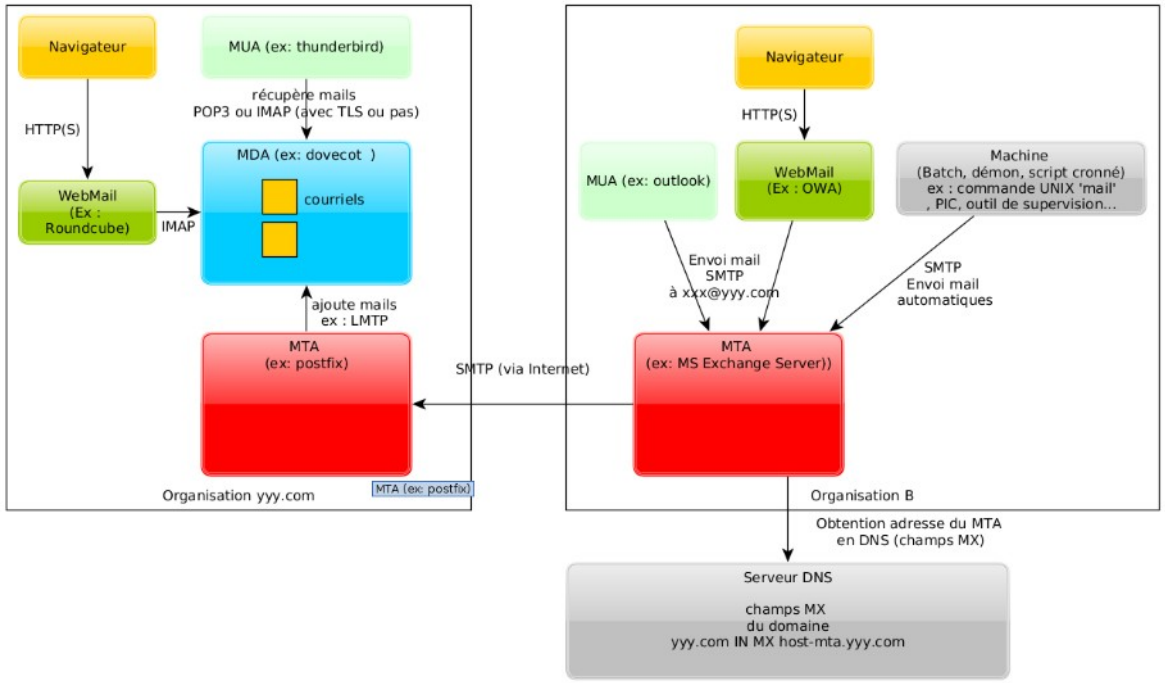
Exemples:

- MTA : Sendmail, Postfix
- MDA : Dovecot
- MUA lourds : Thunderbird, Outlook, IBM Lotus Notes
- Webmail : OWA, Roundcube, Zimbra, Rainloop, Gmail (sur cloud Google)
- MTA + MDA : Microsoft Exchange Server, IBM Lotus Domino

Critères de choix

- Support du TLS
- Approche intégrée ou toolbox
- Scalabilité, volumes

Envoi d'un courriel depuis l'organisation B vers xxx@yyy.com



ETL (Extract Transform Load)

Rôles

- Composant d'intégration au niveau données
- **Extrait (Extract) des données** depuis une source, **la transforme (Transform) et déverse le résultat** dans cible **(Load)**
- En général utilisé en BI pour alimenter un datawarehouse ou un datamart depuis un datawarehouse
- **Mode batch**, pas adapté au fil de l'eau

Apports

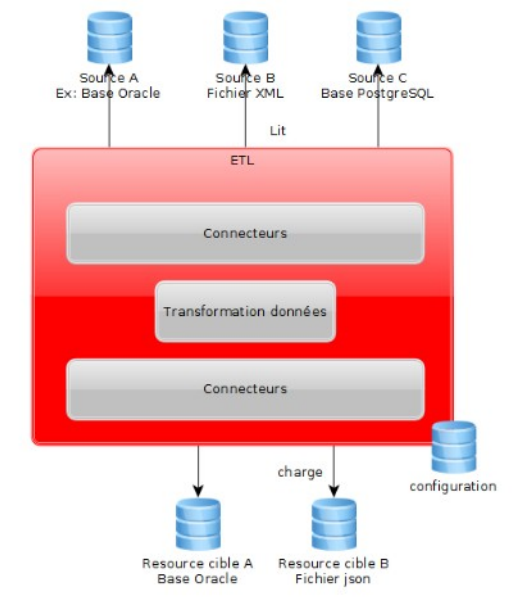
- Évite de développer des batchs complexes, utilisation de connecteurs sur étagère, transformation décrites via un DSL textuel ou graphique
- Centralise les traitements de chargements

Limites

- Souvent chers et complexes (surtout les outils propriétaires), compétences pointues

Exemples

- Open Source : Talend ETL, Pentaho Data Integration...
- IBM Datastage, SAS Integration, Informatica PowerCenter ...



ESP (Event Stream Processing)

Rôles

- Nouvelle génération d'ETL dans le monde du BigData mais pas seulement (logs, métier, backups, IoT...)
- **Event** : génération d'un événement formaté avec métadonnées (timestamp...), **Stream** : flux d'événements à traiter, **Processing** : traitement des événements
- Non plus mode batch mais **flux continu**, s'appuie sur des composants MOM (typiquement Kafka)

Apports

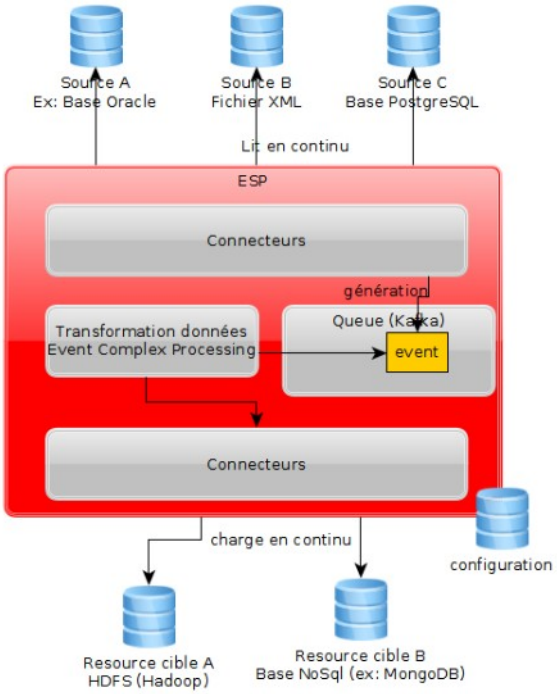
- Plus simples et performant que les ETL
- Intégration des données en temps réel
- Forte robustesse, reprise sur erreur, complétude ... grâce au messaging
- Beaucoup moins cher que les ETL (composants Open Source)

Limites

- Pas adapté aux calculs nécessitant de passer plusieurs fois sur une ensemble complet de données (ex : machine learning)

Exemples

- Logstash, Fluentd, Apache Storm, Apache NIFI



Ordonnanceur

Rôles

Lance périodiquement les traitements (batchs principalement). On distingue :

- les simples **planificateurs** (cron sous Unix, AT ou schtasks sous Windows, Quartz en Java…) qui se contentent de lancer une commande suivant une expression temporelle (exemple : `2,13 15 2 FEB` : tous les 2 février à 15:02 et 15:13)

- les ordonnanceurs qui permettent en plus de définir de véritables **workflows** comme un lancement conditionnel en fonction du code retour d'une commande précédente. Vient avec des **studios** permettant de définir graphiquement les workflows/périodicités (le « plan de production ») , des interfaces Web de r**eporting d'exécution et d'alerting** (couriels, SMS…)

Apports

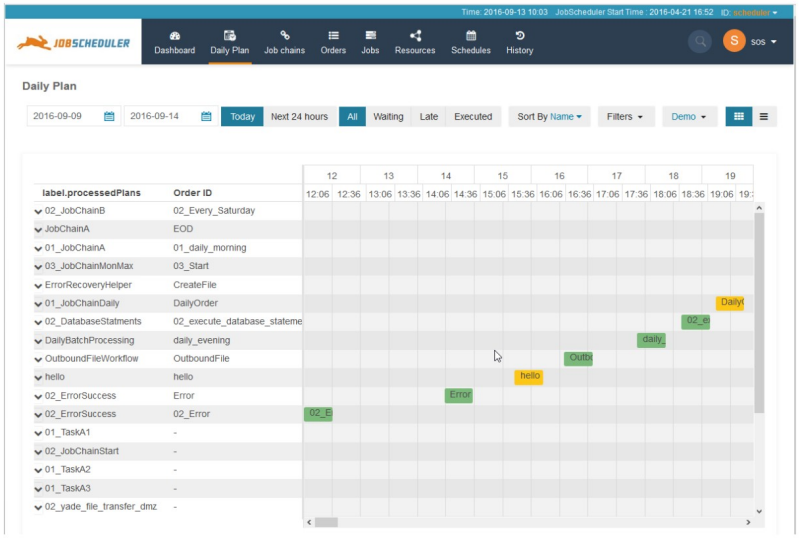
- Vision consolidée des traitements programmés, de leur état, outils graphiques, recherches, traces...
- Formalisation / contractualisation des périodicités
- Indispensable pour un service d'exploitation efficace

Limites

- Utilisation souvent complexe, nombreuses options
- **Pas nécessaire pour des besoins simples.** Exemple : une purge peut se faire avec un simple planificateur et des logs.

Exemples

- Exemples : JobScheduler ou Rundeck en Open Source, Axway Automator, IBM TWS coté propriétaire



Autres composants (1/4)

CMS (Content Management System) / Wiki

Sorte de serveur d'application permettant de **gérer et produire facilement du contenu Web**. Principalement en technologie PHP (Drupal, Wordpress) et Java (Jahia, exo Platform, Liferay...). Certains intègrent des wikis et des portlets. Propose en standard de **nombreux widgets** (météo, bourse...) et permet d'en développer de nouveaux.

Wiki : CMS orienté page. Permet d'éditer et de créer facilement des pages Web. Principalement technologie PHP (Mediawiki, Dokuwiki) et Java (Xwiki, JSPWiki...).

GED (Gestion Électronique de Documents)

Sert à **gérer d'énormes quantités de documents** (PDF, open document, office...), à les indexer, à fournir des outils de recherche. Ils s'appuient en général sur des **moteurs d'indexation** et bases de données dédiés (Apache Solr ou Elastic Search)

Exemples : Filenet, Alfresco, Nuxeo, Nextcloud

Autres composants (2/4)

Portails

Sorte de serveur d'application spécialisé implémentant en général la spécification JSR 168 et permettant de **produire facilement des sites web composites constitués de portlets**.

Exemple : EXo Platform, JBoss Portail, Liferay

Moniteur de transfert de fichiers

Composant servant à **transférer des fichiers** et à en assurer le **suivi** (relances sur erreur, planification, gestion des logs, outils de suivi graphiques, gestion des accès, alerting, etc...).

Exemple : Axway CFT

Protocoles : CFT, SFTP , SCP, ...

Vaults d'entreprise

Coffre fort centralisé servant à gérer les secrets (mots de passe, clés d'API, certificats, clés privées, etc.). Nombreuses fonctionnalités comme l'authentification multi-facteurs, la rotation automatique des secrets, la traçabilité des accès, l'historisation, la vérification de fuite sur Internet, l'accès par API, la vérification des politiques de mot de passe, etc.

Exemples : HashiCorp Vault, CyberArk, Bitwarden (Open Source)

Autres composants (3/4)

Moteur BPM

Composant permettant de **gérer des workflows humains ou d'orchestration de services**. Les workflows sont décrits en BPMN. Peut s'appuyer sur un moteur de règles (voir plus loin) pour certaines décisions. Intégré sous forme de libraire ou de services Web.

Exemples : IBM Business Process Manager, JBPM, Bonita

Moteur de règles (BRMS)

Permet le **calcul de règles complexes** basées sur la programmation par contraintes comme le calcul de l'itinéraire le plus cours, de plannings complexes... Les règles décrites graphiquement ou par un DSL textuel.

Exemples : Drools, IBM JRules

Intégré sous forme de libraire ou de services Web.

Autres composants (4/4)

Protocoles :
WBT (RDP), ICA (Citrix), RFB (VNC) ...

Serveur d’affichage déporté / VDI (Virtual Desktop Infrastructure)

Permet d'affichage d’applications ou d'écrans complets à distances. Utile dans des cas particuliers de gestion des clients lourds qui ne peuvent être installés sur le poste. Sensé permettre de se contenter de "thin clients". Souvent couplés à des hyperviseurs (exemple : Citrix + Xen)

Exemple : Citrix, Microsoft RDP, VNC

Serveur d’exécution de commande à distance

Composant serveur permettant d’exécuter des commandes shell depuis une autre machine (en général un autre serveur). Authentification par échanges de clés RSA ou par mot de passe.

Exemple : OpenSSH, IBM rexec

Protocoles :
SSH, rexec

Excellente vulgarisation

Comment faire une infra prête pour la prod

Voir cette video de Cocadmin

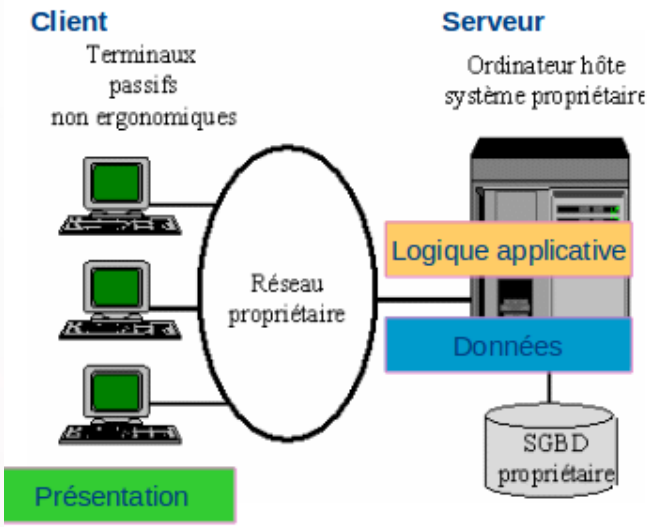
Grandes typologies d’architectures

Les types principaux d'architecture
Par ordre chronologique d'apparition

- Mainframe
- Client-serveur
- n-tiers
- SOA
- Micro-services

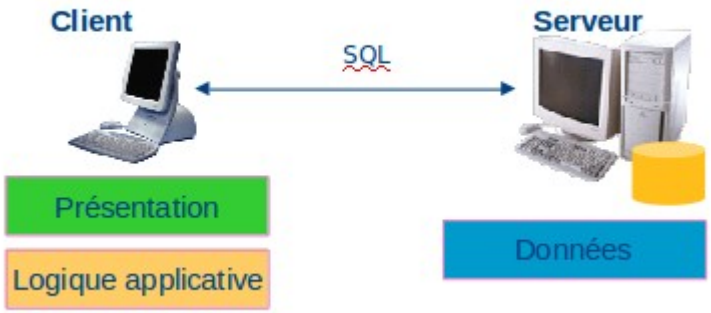
Mainframe

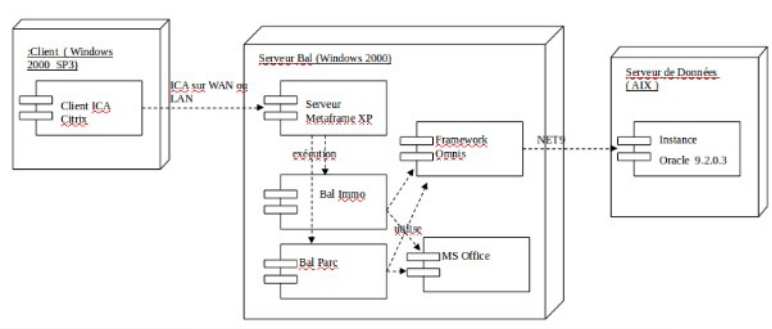
- fin 1960
- Un serveur central virtualisé, des batchs et des terminaux passifs (TN3270, VT220...)
- Encore très utilisé en France :
  - GCOS de Bull
  - Z/OS = MVS d'IBM
- programmes en différents cobols
- très fiable, très coûteux, très fermé
- bases hiérarchiques IDS2, DL1, Adabas
- moniteurs transactionnels (CICS, TUXEDO)
- peut s'intégrer avec des drivers (JTDS + TNVIP SE par exemple)



Client-serveur

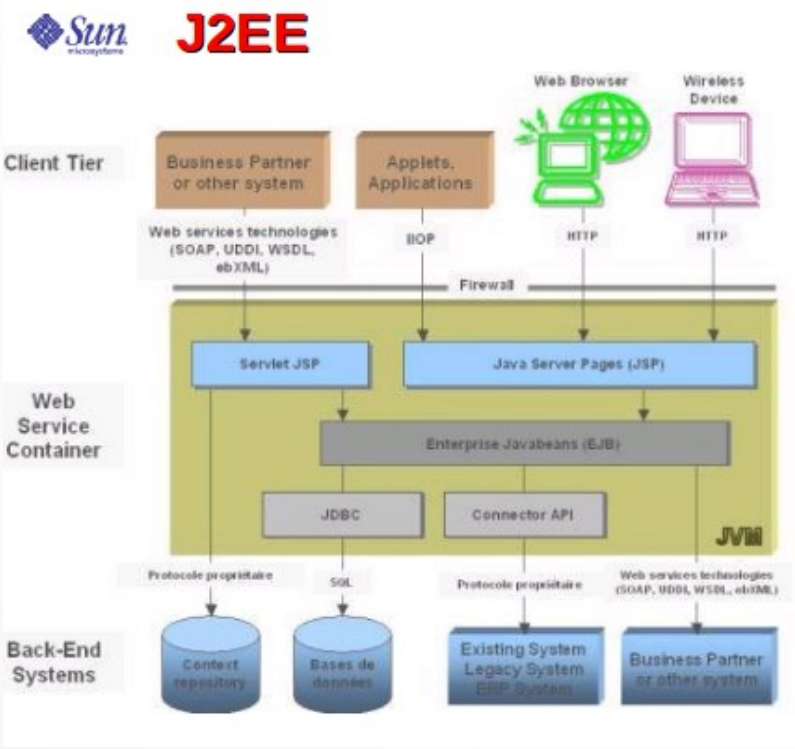
- années 1980, 1990
- une base de données, des clients lourds (en général sous Windows)
- de moins en moins utilisé
- exemples d'AGL : Oracle Forms, PowerBuilder, FMP, WinDev, Omnis ...
- beaucoup moins cher que le mainframe mais...
- forte dépendance au modèle de données, complexité du déploiement des clients lourds, faibles performances sur WAN, sécurité
- peut s'intégrer par affichage déporté (CITRIX dans applet java par exemple)



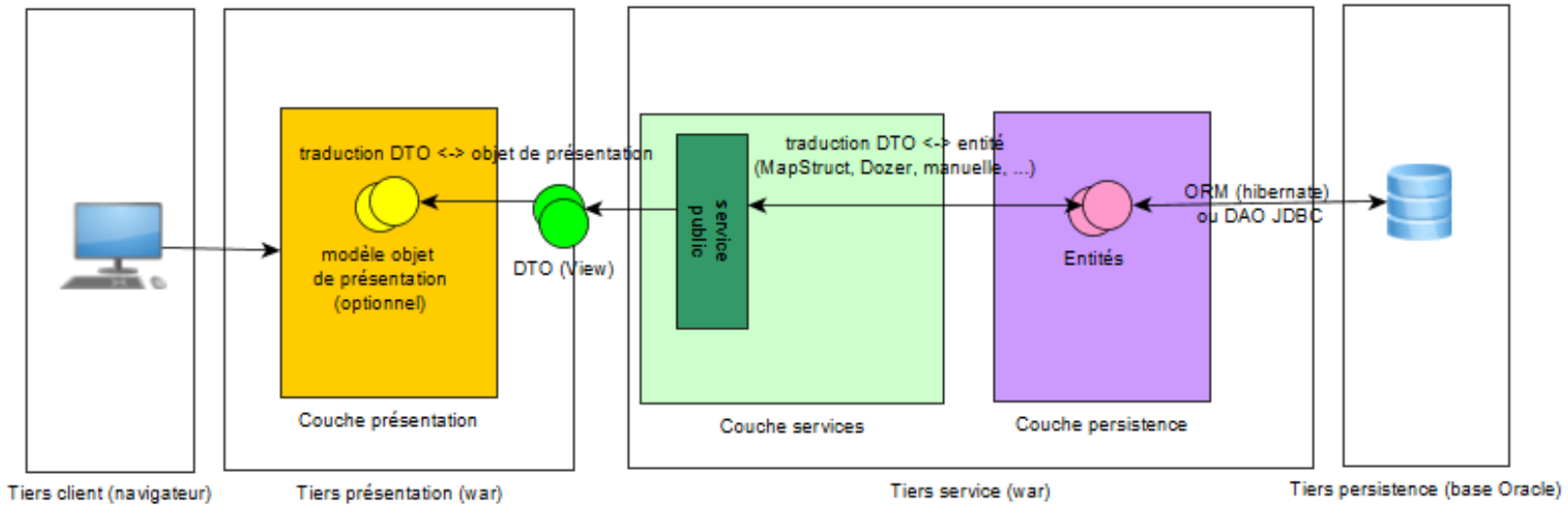


n-tiers

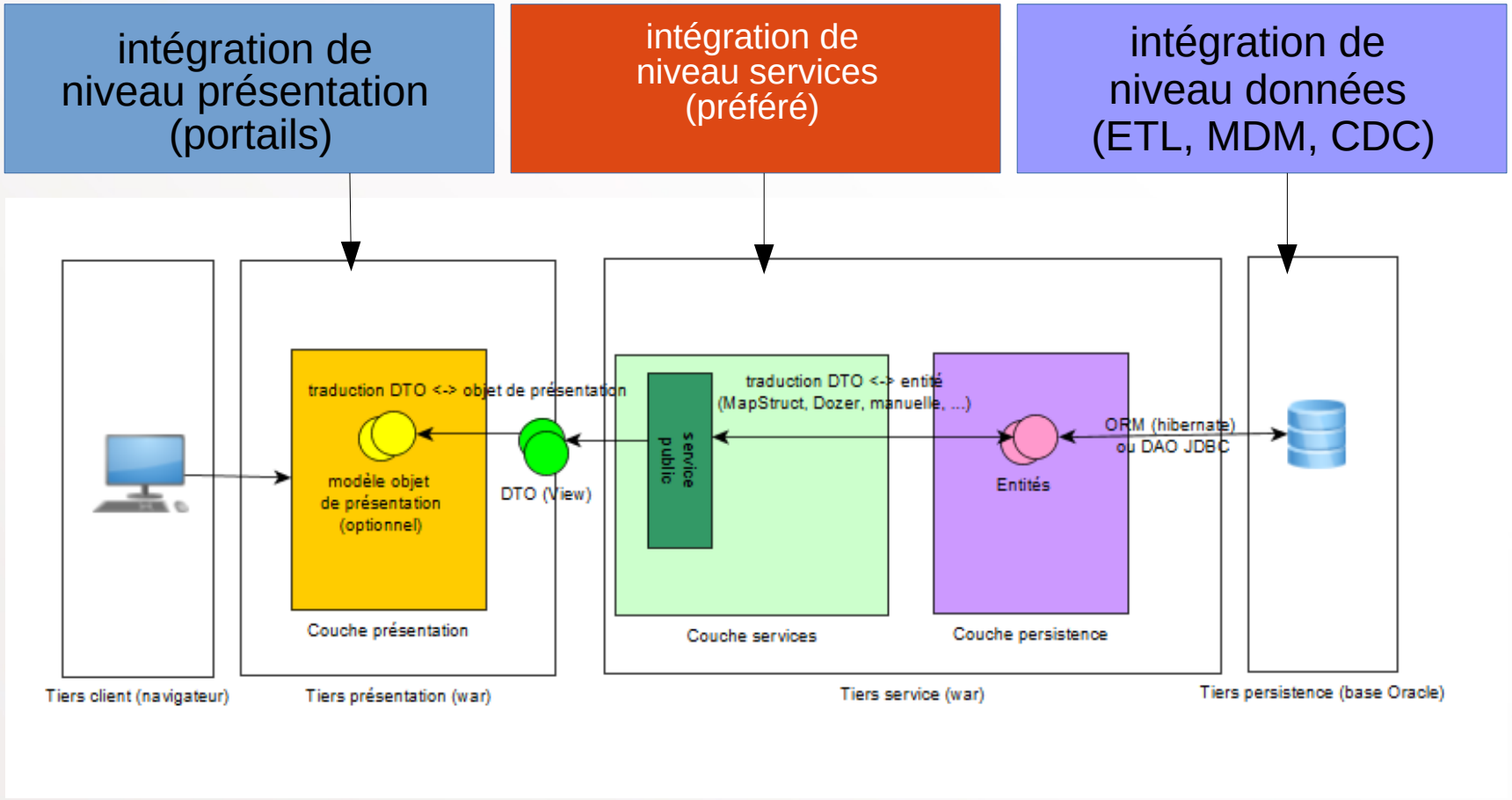
- Fin années 1990
- IHM Web, un middleware, des ressources (base de données, mainframe...)
- une des architectures actuellement les plus utilisées (JEE, .Net, PHP...)
- réutilisation de l'existant
- découplage
- interopérabilité (technologies standards et ouvertes comme TCP et HTTP)
- mais en général grosses applications en silo (monolithiques)



Le découpage en couches en n-tiers

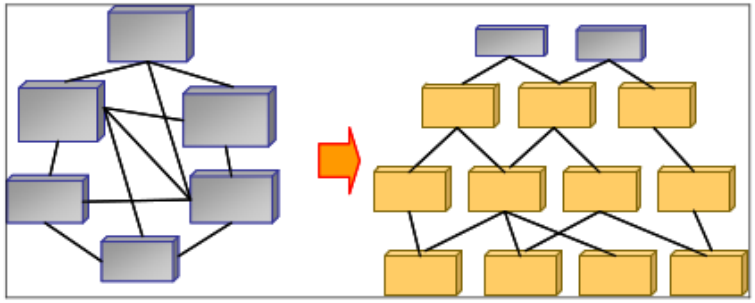


Les trois niveaux d'intégration



SOA (Service-Oriented Architecture)

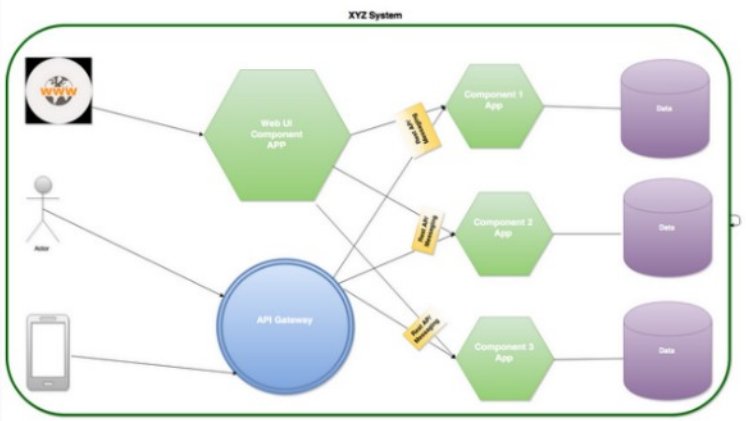
- années 2000
- réutilisation des services au niveau du SI
- intégration des applications entre elles par les services
- fort besoin l'urbanisation et de gouvernance
- standards ouverts (SOAP, XML, WS-*)
- contractualisation
- deux styles d'architecture SOA : avec ou sans bus



crédit : Gilbert Ra

Micro-services

- années 2010 ? ou avant ?
- concept encore en gestation, issu d'un certain rejet de la complexité du SOA
- proche du (vrai) SOA mais :
  - orienté fonction métier
  - les services sont regroupés dans des composants très fins encapsulant leur petite zone de persistance
  - Protocole unique (REST) plutôt que adaptateur universel (SOAP, RMI, protocole spécifique)
  - favorise la programmation polyglotte
  - en off : approche plus 'jetable'
  - aspects intégration : processus autonomes, orientés containeurs, haute scalabilité
  - équipes multi-disciplinaires plus agiles orientées produit, pas projets, DevOps (« You build it, you run it »). Petites équipes « two pizzas team ».



Micro-services, qu'en penser ?

En général, viser du micro-service avec approche monolith-first
(on ne fait du MSA que sur les bounded contexts / domaines bien identifiés)

Forces - simplicité - bonne testabilité - fortement compatible avec les méthodes agiles (Scrum en particulier) - focus métier - time to market incomparable	**Faiblesses** - l'approche tout composant conduit à des appels distribué → problème de performance du au réseau et au marshaling/unmarshaling - problème des « jointures » SOA **- nécessite une refonte de l'organisation, difficile pour une organisation structurellement silotée**
Opportunités - construire un SI pas à pas, éviter l'effet Louvois - intégrer plus rapidement les nouvelles technologies - trouver plus facilement et rapidement des compétences et des talents	**Risques** - composants jetables, confondre agile et « à la rache (tm) » - multiplication de technologies, risque de perte de maîtrise technologique - on fait quoi quand l'expert Golang est parti chez le concurrent ? - plus difficile à sécuriser

Références

- Performance des architectures IT - 2e ed. - Pascal Grojean
- [GOF] Design Patterns: Elements of Reusable Object-Oriented Software de Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides
- Christophe Longépé « Le projet d'Urbanisation du SI »
- Site Martin Fowler : https://www.martinfowler.com/
- Brendan Gregg : « Systems Performance: Enterprise and the Cloud »
- Robert Martin : «The Clean Coder: A Code of Conduct for Professional Programmers »
- Richard Monson-Haefel et autres : « 97 Things Every Software Architect Should Know »