Formation Architecture logicielle 2023

[I. Qu’est-ce que c’est ? 1](#_Toc782060500)

[II. Implémentation 4](#_Toc1262110038)

[III. Modèles d’architecture 5](#_Toc1030475585)

[1. ARCHITECTURE ORIENTEE OBJET - AOO 5](#_Toc1540068403)

[2. ARCHITECTURE ORIENTEE COMPOSANTS - CBA 6](#_Toc130408242)

[3. DOMAIN-DRIVEN DESIGN (DDD) 6](#_Toc1786173317)

[4. CLIENT / SERVER 8](#_Toc320556347)

[5. DISTRIBUEE PAR NIVEAU 10](#_Toc175531583)

[6. EVENEMENTIELLE 10](#_Toc703083077)

[7. SERVICE ORIENTED ARCHITECTURE - SOA 11](#_Toc1701321459)

[8. MICRO-KERNEL (Noyau - Plugin / Modulaire) 14](#_Toc340445244)

[9. PIPELINE 14](#_Toc1994277759)

[10. BASÉ SUR L’ESPACE (SBA) 14](#_Toc1336432988)

[11. SERVERLESS 15](#_Toc1221909847)

[12. MICROSERVICES 15](#_Toc538144776)

[Monolithique vers Microservices 15](#_Toc1979768708)

[\* PATTERN STRANGLER 17](#_Toc697632939)

[\* PATTERN DE CLOISONS 17](#_Toc437126387)

[\* PATTERN SIDE-CAR 17](#_Toc300849186)

[\* MODELES DE DB 17](#_Toc1460152006)

[Microservices - CQRS 19](#_Toc576004619)

[API Gateway 19](#_Toc1093190320)

[IV. Design Patterns 19](#_Toc800562155)

[V. Cloud 22](#_Toc1894320021)

[VI. Big Data 24](#_Toc1204189036)

[VII. IoT Internet of Things 26](#_Toc974655411)

# **Qu’est-ce que c’est ?**

Ex: Waterfall software: on ne passe pas à l’étape suivante sans la précédente finie.

-> très axé Sécurité (prendre en compte l’intégralité dans son ensemble dès le début), aviation, embarqué, etc...

Vs Agile : permet de boucler sur le dev et ajouter des fonctionnalités au fur et a mesure

[International Organization for Standardization](https://www.iso.org/home.html)

[*https://www.iso.org/standard/50508.html*](https://www.iso.org/standard/50508.html) *: 42010 Systems and software engineering*

*Test intégration : Dans le monde du développement informatique, le* ***test d'intégration*** *est une phase de tests, précédée par les tests unitaires et généralement suivie par les tests de validation, vérifiant le bon fonctionnement d'une partie précise d'un logiciel ou d'une portion d'un programme (appelée « unité » ou « module ») ; dans le test d’intégration, chacun des modules indépendants du logiciel est assemblé et testé dans l’ensemble Pour cela, le test d’intégration a pour cible de détecter les erreurs non détectables par le test unitaire. (Avant le test fonctionnel)*

*-> intégration du code dans un env fini (pentest, api, test sde charge, etc...)*

*End 2 end : End-to-end testing is a software testing technique that verifies the functionality and performance of an entire software application from start to finish by simulating real-world user scenarios and replicating live data.*

Architecture : Concepts ou propriétés fondamentaux d'un système dans son environnement, incarné dans ses éléments, ses relations et dans les principes de sa conception et de son évolution -> TENIR COMPTE DE L’ENVIRONNEMENT

-> connaître les notions de performance, charge, contraintes dès le début de l’analyse

*SPOF : A single point of failure (SPOF) is a potential risk posed by a flaw in the design, implementation or configuration of a circuit or system. SPOF refers to one fault or malfunction that can cause an entire system to stop operating.*

*Book: Software architecture in practice (O’reilly)*

[*https://www.oreilly.com/library/view/software-architecture-in/9780136885979/*](https://www.oreilly.com/library/view/software-architecture-in/9780136885979/)

Architecture logicielle est une ABSTRACTION (définition et détails PUBLICS des élements). Ne concerne que les choses importantes. Pas de docs sur les implémentations concrètes

* **Qualité** *(charge, sécurité)***, Longévité (***maj de techno, changement futurs)***, Utilité** (architecture pertinente au vue de l’utilisation) + MESURES

Nécessité d’avoir des éléments MESURABLES et TESTABLES de la qualité pour éviter les retouches futures

Language commun pour éviter les notions subjectives des personnes impliquées (clarté, point de référence)

**Importance de la doc**

**-> la qualité d’architecture est liée à la qualité de la transmission / explication**

**REFERENCE COMMUNE POUR TOUTE L’EQUIPE !**

* Clarté et concision pour les dev (techno) et les métiers (clarté d’utilisation user)
* Images explicatives et belles à voir
* Concret (droit au but, pas d’explication inutiles, sujet - action)
* Courtoisie dans la forme. Valoriser le lecteur
* -> système d’écoute, messagerie, communication
* CLARIFIER LES TERMES

Contrat d’interface: format data (json), headers (sécurité), metadatas, monétisation (AWS) -> permet d’être MOCKÉ

-> permet de séparer les différents éléments en contrats d’interface, et séparer les différentes équipes

-> commencer par la contrainte finale (container, mémoire, etc... de la prod) pour savoir vers quelle architecture se tourner)

Ex: pour apprendre une nouvelle techno, trouver sur quoi cette techno s’insère, pourquoi elle est utilisée, dans quel env, etc..)

* **Fonctionnelle** (login, mdp, chiffrage) vs **non-fonctionnelle** (charge, serveur, sécurité, environnement) “ça marche sur mon poste” -> (applicative vs matérielle / environnementale)

Vision commune : architecture logicielle est contre le changement / agilité. Contrainte et coût de dev

Au contraire : - par ex: changement d’un élément (contrairement sur monolithique), une combinaison d’éléments. - Réduit la complexité du système car vecteur de travail commun. - Tient le rôle du chef d’orchestre

-> Permet de construire des briques REUTILISABLES

-> Aide à la gestion du projet sur les questions (durée, délai, coût de dev, répartition des tâches, etc...)

---

2 approches de gestion :

+ Descendante “chef de projet”, résultat : début par les livrables finaux, puis décomposition des tâches

+ Ascendante “devs et techs”, fonctionnel : tâche très spécifique (Product Owner: défini tous les besoins de l’app “Brainstorm agile”) puis création de lots et dégager les besoins nécessaires MESURES CONCRETE D’EVALUATION

Archi se situe et fait le pont entre les deux. Il a un rôle de formateur / veille techno pour les devs et de conseiller pour le métier. “Softskill” transmettre facilement et rapidement l’info

<https://asana.com/fr/resources/top-down-approach>

*Book: https://www.amazon.fr/Mythical-Man-Month-Software-Engineering-Anniversary/dp/0201835959*

L’architecture n’est pas une solution miracle, mais devra toujours choisir entre des compromis, avantages / inconvénients

Archi est un leader technique avec les intervenants : users, analyse commerciaux, experts audit, qa, gestion financière, devs, etc..

**-> Importance des softskills / transmission / ECOUTE / Communication**

*Savoir répondre à la question “Pourquoi ça fonctionne / ça ne fonctionne pas ?”*

*Le FinOps est un modèle informatique et financier créé pour répondre à la structure volatile du cloud et à son coût en forte hausse.*

[*https://fullscale.io/blog/tech-lead-vs-team-lead/*](https://fullscale.io/blog/tech-lead-vs-team-lead/)

**Methods ATAM** : Architecture Tradeoff Analysis Method

*In software engineering,* ***architecture tradeoff analysis method*** *(****ATAM****) is a risk-mitigation process used early in the software development life cycle* ***(SDLC)****.*

*ATAM was developed by the Software Engineering Institute at the Carnegie Mellon University. Its purpose is to help choose a suitable architecture for a software system by discovering trade-offs and sensitivity points.*

<https://www.geeksforgeeks.org/architecture-tradeoff-analysis-method-atam/>

-

*Software Development is the development of software for distinct purposes. These 6 stages are discussed below.*

*1: Planning And Requirement Analysis 2: Defining Requirements 3: Designing Architecture 4: Developing Product 5: Product Testing and Integration 6: Deployment and Maintenance Of Product*

<https://en.wikipedia.org/wiki/Architecture_tradeoff_analysis_method>

<https://www.cvedetails.com>

---

# **Implémentation**

Contraintes :

* Conception (dev)
* Déploiement (ops)
* Livraison (comment le code est utilisé, package, consommation du produit, intégration)
* Intégration, Débogage, Robustesse (Spof)

- Archi Orienté Objet, Basé composants, Domaine, Client / server, Couche, Evènements, Orienté service

*Utilisation de Data Science (hadoop)*

++ Non-fonctionnel

Sécurisé, fiabilité

++ Qualité de service

Disponible, (SLA Service Level Agreement)

Minimum à 3 neuf : 99,9% du temps d’utilisation

*A service-level agreement (SLA) is* ***a contract between a service provider and its customers that documents what services the provider will furnish and defines the service standards the provider is obligated to meet****. A service-level commitment (SLC) is a broader and more generalized form of an SLA.*

[*https://www.techtarget.com/searchitchannel/definition/service-level-agreement*](https://www.techtarget.com/searchitchannel/definition/service-level-agreement)

* Tester puis libérer les ressources pour MONITORER les infos

IAS Infrastructure As Code (K8s, Terraform, Ansible)

<https://www.opengroup.org/>: Architect Case Study

**Créer un Modèle logiciel** : proposer l’idée

* Nom, problème, contexte (état initial)
* Force / inconvénients / avantages
* Solutions existantes (ex statique pour présentation)
* Résultat vécu, exemples, modèles alternatifs
* Evolutions possibles (héritage legacy)

*directeur des systèmes d'information* (*DSI*)

---

1. **Modèles d’architecture**

<https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013?gi=451d7619461e>

## **ARCHITECTURE ORIENTEE OBJET - AOO**

* Object interopérable encapsulant les fonctionnalités
* Très modulable
* Peu de couplage
* Réutilisable

+ Etat et comportement, Modularité, Typage, Concurrence

3 fondamentaux

+ Abstraction (masquage d’informations)

*le concept d'****abstraction*** *identifie et regroupe des caractéristiques et traitements communs applicables à des entités ou concepts variés ; une représentation abstraite commune de tels objets permet d'en simplifier et d'en unifier la manipulation*

***L'abstraction*** *est le processus de masquage des détails d'implémentation d'un objet afin qu'il puisse être utilisé sans comprendre comment il fonctionne.*

+ Encapsulation

*L'encapsulation est le processus de masquage d'informations dans un objet afin qu'elles ne soient pas accessibles directement depuis l'extérieur de l'objet.*

+ Héritage

*L'héritage est la capacité d'une classe à hériter des attributs et des méthodes d'une autre classe*

<https://fr.wikipedia.org/wiki/Architecture_orient%C3%A9e_objets>

Principes S.O.L.I.D.

Designs Patterns

1. **ARCHITECTURE ORIENTEE COMPOSANTS - CBA**

Archi monolithique vs Composant 1 (plusieurs classes) <=> Composant 2 <=> C3...

Injection dépendance, localisation...

-> abstraction et découplage des modules

## **DOMAIN-DRIVEN DESIGN (DDD)**

*Focusing on modeling software to match a domain according to input from that domain's experts.* Under domain-driven design, the structure and language of software code (class names, class methods, class variables) should match the business domain.

-> Utile avec le BigData ou données complexes

*il faut mettre en place un* ***langage ubiquitaire****, c'est-à-dire un langage structuré autour du modèle du domaine et utilisé par tous les membres de l'équipe, aussi bien techniques que métiers*

*Chaque domaine doit dépendre du noyau, mais le noyau ne doit dépendre d'aucun domaine*

Exigence Business => MODELE <= Devs, archi et techs

1. Se concentrer sur le domaine de base. Chercher les sous-domaines (espaces de problème, permet de résoudre le problème principal, langages de développements peut-être différents) du business
2. Modéliser le domaine (concept, vocabulaire “diagrammes” “contraintes”, problèmes et relations entre les entités) -> Langage et Pseudo-code
3. Evolutions (commentaires avec les experts du domaine)

* Contexte : objectifs fondamentaux des problèmes commerciaux (ex: context “point de vue de la Banque”, contexte “vue du Client”...)
  + Relation entre les contextes (action commune factorisables ou séparées ?)
* Entités : Acteurs du modèle (ex: client-id,nom) MUABLE (id unique)
* Objets de valeur : IMMUABLE (sans id)
* Encapsulation des données
* Racines / entrées communes
* Services de domaine : encapsule une logique de domaine, responsables de l’orchestration de la logique métier (ex: effectuer une transaction)
* Factory : séparer l’usage d’un objet de sa création (simplification en cachant toute la structure de build de l’objet)
* Repository : séparation du type de data, de la persistance / récupération des aggregates. Séparation domain métier / donnée

*The repository pattern is* ***a design pattern that isolates the data layer from the rest of the app****. The data layer refers to the part of your app, separate from the UI, that handles the app's data and business logic, exposing consistent APIs for the rest of your app to access this data*

Clarifier les flux logiques

<https://rusyasoft.github.io/ddd/2018/05/10/ddd-repository/>

Ex: Contexte de la bulle (app + data propre) pour les systèmes legacy, les isoler pour les améliorer (couche “anti-corruption”) avec une synchro des données en asynchrone entre la bulle (legacy+db) et (new+db)

*FinTech: Financial technology (better known as fintech) is used to describe* ***new technology that seeks to improve and automate the delivery and use of financial services****.*

<https://medium.com/microtica/the-concept-of-domain-driven-design-explained-3184c0fd7c3f>

Création de doc : Lister les modules / contextes / entités... (breadcrumb ex: MDN)

Ex: entreprise.responsable.bounded\_context.domain.module

## **CLIENT / SERVER**

*Request / Response (Applications Web)*

*- Client : demande le service -> envoi une requête*

* *Lourd : Process côté client*
* *Léger : Process côté serveur*
* *Se poser la question d’accès aux datas ? Orm / Odm ? ...*

*- Serveur : fourni le service en permanence selon les demandes du client*

* *Hiérarchiser les requêtes*
* *Droits de sécurité*

*- Messages Client <=> Serveur : SSL / SMTP / FTP / HTTP...*

*+ Sécurité accrue, accès centralisé, maintenance plus facile*

*- Problème de scalabilité, server est un Spof, moins de fiabilité*

*Architecture en couches*

* *Evolue bien sur le spécifique*
* *Difficile à modifier sur une grande appli ou rapidement*

[*https://fr.wikipedia.org/wiki/Architecture\_en\_couches*](https://fr.wikipedia.org/wiki/Architecture_en_couches)

*A 2 niveaux*

* *Client (view) <> Serveur (logique, data)*
  + *Sécurité (centralisé), accès central, meilleure maintenance*
  + *Quand ? “Volonté centrale” (stockage, administration...) ou Prise en charge de beaucoup de clients*
  + *Limites ? Fiabilité, scalabilité, bande passante, sécurité unique*
    - *-> détacher la logique en plusieurs niveaux*
* *Client <> Master - Slave1, Slave2... ( ! Séparation data, latence..)*
* *Client <> File d’attente de serveurs (messagerie)*
* *Peer 2 Peer : Client / Serveur symétriques (à la fois C ou S) décentralisés*
  + *Très grande tolérance aux pannes / Simple à mettre en œuvre*
  + *! Sur-utilisation de bande passante réseau ! (ex : Gnutella)*
  + *Hybride : Client/Server + semi-P2P (ex: Catalogue réseau Napster)*
  + *! Sécurité sur port TCP (*Transmission Control Protocol) *!*

*A 3/n niveaux*

* + *Client - Présentation*
  + *Serveur(s) - Applicatif(s) (ex: séparation sur serveurs +/- performants)*
  + *DB / API – Data, stockage*
* *Evolutif, - de fonction côté client / sécurité triple, séparation design / data*
* *Diviser la logique applicative*
* *Divise la charge réseau en serveurs séparés (ex: livraison, panier...)*

*Distribuée*

*L’****architecture distribuée*** *ou l'****informatique distribuée*** *désigne un système d'information ou un réseau pour lequel l'ensemble des ressources disponibles ne se trouvent pas au même endroit ou sur la même machine. Ce concept, dont une version peut être une combinaison de transmissions du type client-serveur, s'oppose à celui d'informatique centralisée.*

*- Avec le haut-débit et les requêtes mutualisées nécessite un monitoring de traitement des transactions entre les serveurs de DB (ex: monitor de replication..)*

*3 niveaux*

*Model View Controller*

* *Model : Encapsule la data, les fonctions de base, la logique*
* *View : Html, jQuery..*
* *Controller : Suivi events côté user*

*Model View Presenter*

*Conçu pour les tests unitaires automatisés*

* *Vue passive : Interaction user mapé à un Presenter*
* *Presenter : MAJ de la vue*

*M-V-VM*

* *View : Interface user (html) représente les data*
  + *Component actif, events, liaisons data*
* *View-Model : Tampon entre affichage/donnée. Méthodes, commandes*
  + *Maintient le model en fonction de la vue*
  + *Déclenche des events et MAJ la vue*
* *Model : Logique métier*
* *Frontière de tests claire, code réutilisable, flexibilité de changement de view*

*Single Page Application SPA*

*+ Tout est dans un bundle.js et un index.html vide. Le js s’occupe de tout*

*-> Pref : Utiliser le lazy loading ou le SSR (hydratation des datas js dans le html)*

## **DISTRIBUEE PAR NIVEAU**

*Distribuées sur plusieurs machines*

* *Remote Procedure Call, RMI (Remote Method Invocation*)*, Ent Java Bean*
* *Disponible et évolutif*

*SEPARATION OF CONCERN (séparer les préoccupations)*

1. **EVENEMENTIELLE**

Besoin d’une approche pro-active / asynchrone, en réaction à une demande

One - to – one : Plusieurs éléments basés autour d’un noyau central

1. Systèmes distribués
2. Contextes délimités

Avantages : Fiabilité, couplage très lâche, très customisable

* Composants intégrés via une messagerie asynchrone, les messages sont stockés dans une file d’attente par sujets.

1.Générateur

Envoi d’un message

2.Gestionnaire d’évènements

-> Broker

3.Ecouteurs

Broker -> listener1

-> listener2

-> listener3...

***Commands/APIs : Synchro*** *Has an intent, Directed to a target (archi client-server)*

**Events : Asynchro** It’s a fact, Happened in the past (archi par messagerie de stockage)

MOM (Messagerie Oriented Architecture) message queuing

Contrainte de réactivité / mise à l’échelle

*Ex Twitter, Watsapp, réservation, app de stock..*

*PB: si le server ne dépile pas les messages aussi vite qu’il ne les reçoit*

*Use* ***Kafka*** *(twitter) /* ***AWS Kinesis*** *ou SQS workflow, Apache* ***ActiveMQ*** *par exemple*

*Modèle PUBlish / SUBscribe*

*ESB ou Enterprise Service Bus peut se définir comme un ensemble d’outils qui* ***garantissent la sécurisation des échanges entre les sources et cibles d’un système d’information****.*

[*https://aws.amazon.com/fr/event-driven-architecture/*](https://aws.amazon.com/fr/event-driven-architecture/)

## **SERVICE ORIENTED ARCHITECTURE - SOA**

*L'****architecture orientée services*** *ou* ***AOS*** *(calque de l'anglais* ***service-oriented architecture****,* ***SOA****) est une forme d'architecture de médiation qui est un modèle d'interaction applicative qui met en œuvre des* ***services*** *(composants logiciels) :*

* *avec une forte cohérence interne (par l'utilisation d'un format d'échange pivot, le plus souvent XML ou JSON) ;*
* *des couplages externes « lâches » (par l'utilisation d'une couche d'interface interopérable, le plus souvent un service web WS-\*).*

*L'architecture orientée services (SOA) est une méthode de développement de logiciels qui utilise des composants logiciels appelés services pour créer des applications métier. Chaque service offre une capacité commerciale, et les services peuvent également communiquer entre eux à travers les plateformes et les langues.*  *Les développeurs utilisent la SOA pour réutiliser des services dans différents systèmes ou combiner plusieurs services indépendants pour effectuer des tâches complexes.*

*- Librairie : collection de “Services” : découvrable, réutilisables (Interface + Implémentation)*

*- Le SOA est un ensemble de codes/services réutilisables*

*- Possibilité d’utiliser : IHM <=> Orchestrator <=> s1, s2, s3...*

*+ Evolution indépendante des services*

*- Importance de la standardisation ! Les services doivent avoir le même schéma / méthode d’appel*

*! Difficulté : Risque d’avoir beaucoup de services interdépendants (gros monolithique pseudo-séparé) !*

*-> utilisation d’un service messagerie (ex: kafka) pour séparer/découpler les services entre eux*

*-> courtier / broker de data*

*-> création de façades de services : (services connectés à l’orchestrator)*

*! Coûts de dev / maintenance*

*-> faire des actions / transactions ATOMIQUES (oui ou non, mais pas de demi-action) : besoin d’une restauration des opérations lors d’un échec !*

*<différent> des microservices d’appel de data pure*

*-> penser à la refactorisation de services (ajout de fonctionnalité n’est pas splité entre différents services)*

*-> sécurité : “filtrage des messages”*

*-> auth : confiance centralisé (service central / token de gestion utilisateur)*

*Peut être combinée à*

* *L’archi* évènements*.*
* *La virtualisation (VM, container...)*

*Logique itérative: client -> service1 -> service2... (ajout de fonction à court terme)*

*Endpoints are* ***devices that connect to and exchange information with a computer network****. “ex <host>/<endpoint>/<params>*

*La notion de* ***granularité*** *définit la taille du plus petit élément, de la plus grande finesse d'un système. Quand on arrive au niveau de granularité d'un système, on ne peut plus découper l'information.*

*Par exemple dans une* [*population*](https://fr.wikipedia.org/wiki/Population)*, la granularité est l'individu.*

*En informatique, la granularité est le plus petit niveau de détail géré.*

*Un SOA :*

1. *Besoin défini*
2. *Exposer le service*
3. *Composition*
4. *Exploiter le service*
5. *Enrichir celui-ci, etc...*

*Modèle de rappel de service asynchrone : afin de ne pas attendre la disponibilité du service appelé : ajouter un message avec l’adresse de retour du service appelant “endpoint ou autre”)*

*-> validation BigData (effectue l’action puis envoie un message au service appelant lors de la fin du traitement), système boursier, réservation, messagerie*

*\* services “lourds” doivent être en fin de chaîne !*

*-> ex: monitoring (tous les 4 appels on check...)*

*+ Le système de perd pas de communication*

*! Nécessité d’avoir un système coûteux de rappel*

*Catalogue de service (métadatas) : -> ex: K8s*

* *Découverte et accessibilité de chaque service* 
  + *Check-health de chaque service*
* *Limiter la redondance (scalabilité)*

*SOA vs microservice:*

*- La SOA est conçue pour partager les ressources entre les services., L'accent est mis sur l'optimisation de la réutilisation des services d'application. Les unités business sont dépendantes.*

*- Les applications de microservices dédient principalement une base de données ou un autre type de stockage aux services qui en ont besoin. Plus axé sur le découplage. Les unités business sont indépendantes les unes des autres.*

*Résumé : Principe <-> Concrétisation*

* *Contrats standardisés : Facade / Format + protocole / Rappel / Bus kafka*
* *Abstraction de service (transparence) : Facade / Refactorisation / Orchestre*
* *Autonomie de service : Broker Kafka*
* *Association de services : Courtier d’auth (soulage réseau) / Orchestrator*
* *Découverte et accessibilité : Catalogue centralisé métadata (sur K8s)*
* *Maintenance de découplage : Facade / Evènement et message Kafka*
* *Service agnostique : Atomicité et rollback*

*! ATTENTION AU SPOF DE BROKER / MESSENGER !*

[*https://fr.wikipedia.org/wiki/Architecture\_orient%C3%A9e\_services*](https://fr.wikipedia.org/wiki/Architecture_orient%C3%A9e_services)

[*https://www.guru99.com/microservices-vs-soa.html*](https://www.guru99.com/microservices-vs-soa.html)

[*https://aws.amazon.com/fr/what-is/service-oriented-architecture/*](https://aws.amazon.com/fr/what-is/service-oriented-architecture/)

## **MICRO-KERNEL (Noyau - Plugin / Modulaire)**

*Plugin1, Plugin2 <=> Core <=> Plugin3, Plugin4*

*+ Idéal pour un Minimum Viable Product*

*+ Tests, agilité, performance*

*- Difficile à scaler (mettre à l’echelle) un plugin*

*- Spof, soucis de stabilité car point central*

*ex: VSCode, Eclipse (IDE), Application...*

## **PIPELINE**

*Modèle monolithique unidirectionnel*

*Tâche 1 => Tâche 2 => Tâche 3...*

*But : Décomposer une tâche complexe en différents points (bloquants ou non)*

*Ex: DevOps (test -> build -> deploy -> …)*

## **BASÉ SUR L’ESPACE (SBA)**

*Mise à l’échelle horizontale de DB*

*On met à l’échelle les unités de traitement*

* *Utilisé en information distribuée*

*A* ***Space-based architecture*** *(****SBA****) is an approach to distributed computing systems where the various components interact with each other by exchanging* ***tuples*** *or entries via one or more shared spaces.*

[*https://en.wikipedia.org/wiki/Space-based\_architecture*](https://en.wikipedia.org/wiki/Space-based_architecture)

## **SERVERLESS**

Utilise l’infrastructure d’un tiers, on paye pour le service donné (scalabilité, sécurité)

Ex: AWS Lambda

## **MICROSERVICES**

Architecture distribuée. Synchrone ou async au choix.

*N’utilise pas de service de messagerie / broker / liaison*

- Précis, avec 1 objectif spécifique

- 1 fonction business

- Logique métier plus clair

(tests d’INTEGRATION = intégration dans un server / environnement spécifique)

Attention :

* Problèmes de performance
* Doit résister à la data-loss (rollback ?)
* Nécessite une AUTOMATISATION du déploiement car chronophage
  + Monitoring pour éviter les goulots d’étranglements des microservices
  + Utiliser des combinaisons de métrics / alerting pertinent
* MicroServices = DevOps + tests obligatoire
* Demande une bonne compréhension du business de la société

### **Monolithique vers Microservices**

* Diviser l’applicatif (décomposition)
  + Fonctions
  + Datas
  + Communications

\* Décomposition en logique métier

1. Besoins commerciaux
2. User Story : Repérer la STABILITE COMMERCIALE de la société
   1. µService1 + µService2...

\* Décomposition en sous-domaines

1. DDD
2. Entreprise en tant qu’espace de problèmes
   1. CORE : + important
   2. Support : lié à l’activité
   3. Générique : pas spécifique au domaine

\* Décomposition par transactions (chaque à sa propre base de données)

1. Chaque service : soit succès <-> soit rollback

ou

2. Modèle SAGA

*Les Sagas permettent la mise en œuvre de transactions métiers distribuées de longue durée, exécutant un ensemble d'opérations sur plusieurs microservices, appliquant une sémantique cohérente du tout ou rien.*

* + *Les transactions compensables sont des transactions qui peuvent être inversées en traitant une autre transaction avec l’effet opposé.*
  + *Une transaction pivot est le point go/no-go dans une saga. Si la transaction pivot est validée, le saga s’exécute jusqu’à la fin. Une transaction pivot peut être une transaction qui n’est ni compensable, ni renouvelable, ou qui peut être la dernière transaction compensable ou la première transaction renouvelable dans la saga.*
  + *Les transactions renouvelables sont des transactions qui suivent la transaction pivot et dont la réussite est garantie.*

“Chorégraphie” entre services

! Grande consommation réseau entre services !

! Eviter de trop découper les services -> refactoriser !

*Mais faire attention à ne pas tomber sur du Pseudo-monolithique.*

**\* PATTERN STRANGLER**

Réécrire une appli complexe, ou problème de migrations

-> Solution : Utiliser une façade “Strangler” (contrat d’api) permettant de migrer lentement vers une application

1. Façade (APPLICATION) + api
2. Façade APPlication + Api
3. Façade appli + (API)

### **\* PATTERN DE CLOISONS**

Objectif : Isoler les éventuelles pannes – Eviter les défaillances en cascade, goulots d’étranglement..

* *Exemple : un service spécifique pour chaque problème (afin d’éviter de trop charger les autres)*
* *Exemple : limiter les threads par lot (ex 25 pour le service 1+2+3)*

### **\* PATTERN SIDE-CAR**

Composant parallèle => même cycle de vie que le parent. Dépendant de celui-ci

Utile pour des fonctionnalités plus complexes

* Logs séparés
* Surveillance
* Conf + services Proxy

! Si panne : impacte tous les services

! Ajoute de la latence

*Ex: Ajouter de la fonctionnalité dans des conteneurs*

[*https://grafana.com/oss/prometheus/exporters/*](https://grafana.com/oss/prometheus/exporters/)

### **\* MODELES DE DB**

*DB partagées :*

*DB non partagées :*

*SQL :*

*Atomic (tout ou rien) Cohérence (des données) Isolation (R/W en opérations simultanées) Durabilité (transactions persistantes / log, même si erreurs)*

*-* Relationnel entre plusieurs tables, entre différents ids

! Perte de perfs dans une grande base. Beaucoup de mémoire pour peu de choses.

! Ex : Si tout dans une seule table : La simple petite requête récupèrera TOUTE la tabl. Les formats ne correspondront pas (donc il y aura des NULL un peu partout, qui occuperont quand même de la mémoire)

- BUT :

MOINS de colonnes possibles

MOINS de NULL possibles

(en gros moins de ressources possible à chaque requête)

-> Normalisation

1NF : chaque champ ne doit stocker qu'une valeur, ne doit pas avoir deux champs qui stockent des informations similaires

2NF : réduire les redondances, stocke des informations qui nous renseignent sur la clé primaire

3NF : pas de dépendance transitive (lorsque la colonne B de votre base de données dépend de la colonne A, qui dépend de la clé primaire)

[*https://appmaster.io/fr/blog/quest-ce-que-la-normalisation-des-donnees*](https://appmaster.io/fr/blog/quest-ce-que-la-normalisation-des-donnees)

N(ot)O(nly)SQL :

CAP : Consistence (même data entre les serveurs) Availability (à haute fréquence, la data sera disponible R/W) Partition (chaque node en réseau fonctionne indépendamment)

*Basically (souplesse, non garanti, à peu près dispo. Système distribué -> premier client arrivé, premier servi) ASE*

- N’importe quel texte JSON sera inséré comme clé => document\_json

- Très grand nombre d’INDEX (sur chaque clé)

Pas de notions de jointures : Tout est indexé dès l’inscription de la donnée (chaque clé)

Sharding : duplication de l’unité de donnée qui contient la data (sur chaque node)

<https://www.lebigdata.fr/sharding-definition-avantage>

*CAN USE :*

*- User <=> NOSQL (redis) fast -> relié à SQL (data moins utiles rapidement)*

*Transition Monolith -> Microservice*

*1. Une DB partagée par tous les services (service1 -> DB <- service2...)*

*+ Cohérence, ACID...*

*- SPOF, si SQL : modif de data -> changement de toutes les tables*

*2. Une DB par microservice*

*+ Evolution de schema facilité*

*+ Pas de spof (distribuable, scalable)*

*+ Utiliser une DB temporaire pour les tests*

*- Problème de synchro entre les bases*

*-> Peut nécessiter un “tampon” entre les services (s1 <> api <> s2)*

### Microservices - CQRS

*Ségrégation des responsabilités : Commande Query Responsability Segregation*

* Comment associer les données entre différents services
* Améliorer les performances de l’accès à la data

> Séparation de la lecture de l’écriture

Command : Create Update Delete

Query : Read

### API Gateway

* Granularité d’api alors que les clients ont différentes exigences

> Api Gateway fait le pont (authentification, proxy, translation... ) entre les API

> Utilisé dans le Cloud

1. **Design Patterns**

* Créationel : Résumer l’instanciation d’un objet
* Structurel : Création de structure plus grande. Cohérence, ajout dynamique
* Comportemental : Attribution de responsabilité, encapsuler comportement dans un objet

**Création**

BUILDER

* Construit un objet complexe par étapes

METHODE USINE / ABSTRAITE

* *Permet de cacher la méthode de création d’un objet*
* Instancier par les sous-classes

***Factory*** *Method pattern hides the construction of a single object whereas* ***Abstract Factory*** *hides the construction of a family of related objects. Abstract factories are usually implemented using (a set of) factory methods.*

PROTOTYPE

* Copier un objet plutôt que le créer de nouveau

SINGLETON

* Une seule instance possible
  + Peut “cacher” les dépendances -> préférer une interface ?

**Structure**

ADAPTATEUR (WRAPPER)

* Pont entre deux interfaces incompatibles

BRIDGE

* Sépare l’abstraction de l’implémentation pour varier indépendamment
  + ! Ajoute de la complexité au code !

***Adapter*** *pattern is used after the application components are designed so that we can use them without modifying the source code. This is in contrast to the* ***Bridge*** *pattern, which is used before the components are designed*

***Strategy*** *Pattern is used for Behavioural decisions, while* ***Bridge*** *Pattern is used for Structural decisions.*

COMPOSITE

* *Traiter ensemble et objet d’une seul manière*

**Comportement**

CHAINE DE RESPONSABILITE

* *Découpler l’objet client du récepteur de la requête*
  + *! Soucis de chainage indétectable*

COMMANDE (DEMANDE DIFFEREE)

* *Action / transaction*
* *Encapsule les requêtes dans un objet*

INTERPRETER

* *Interprète des instructions écrites sous forme d’expression*

*Ex: Menus d’application*

MEDIATEUR

* Interaction entre différents éléments sans connaître leur structure

Maintenabilité et réutilisation du code

*Ex: Menus d’application aussi*

*The* ***Mediator*** *pattern restricts direct communications between objects.*

MEMENTO

* Capturer et stocker l’état d’un objet pour réutilisation

OBSERVEUR

* One to Many entre objets. Informe les objets dépendants d’une modification

*Ex: Ecouter des flux de données (succession de cliques ou clavier)*

STATE

* One to Many. SI un état change, l’objet associé change aussi et informe les composants reliés
* Tout est relié au DataStore
  + Ex: chrysalide (chenille, papillon : même animal, pas les mêmes datas)

STRATEGY

* Moyen simple d’accéder à un algorithme donné.
  + 1.myMethod() ou 2.myMethod()

TEMPLATE

* Défini un algo en permettant d’en modifier certaines étapes
  + Abstract Class.myMethod() -> Imp.myMethod()

***Template Method pattern does it through inheritance whereas Strategy Pattern does it through composition****. The template method provides a concrete method that 'owns' the algorithm whereas in strategy there is no “owner” of the algorithm.*

VISITOR

* Effectue de nouvelles opérations sans modifier la structure de l’objet de base

***Strategy is based on coding to an interface.******Visitor is based on coding to an implementation****. Strategy implements one operation in multiple ways. Visitor implements multiple operations.*

1. **Cloud**

Objectif : Fournir des demandes à la demande

* Diminution du coût de maintenance (uniquement coût d’exploitation)
* Facilité de DevOps en InfraAsCode (infra déployé automatique ex: Ansible)
* On premise : Tout en local
* Infra as a Service : Network + Virtualization on Cloud
* Platform AAS : Iaas ans OS
* Software AAS : Paas and Application
* Daas : Data on demande

**FinOps** : Analyse le coût des infra on demand

**RGPD <> Cloud Act** (Clarifying Lawful Overseas Use of Data)

*permettant aux instances de justice de contraindre les fournisseurs de services établis sur le territoire des États-Unis, à fournir les données relatives aux communications électroniques des "US Persons" c'est-à-dire des citoyens US ou des résidents US, stockées sur des serveurs, qu'ils soient situés aux États-Unis ou dans des pays étrangers*

- Choix de la société (GCP, AWS...)

*Exemple: Si Netflix héberge 100% chez AWS, puis quelques mois après AWS sort sa propre solut° vidéo (on peu prendre le code grâce aux passwords dans l’infrastructure) -> toujours avoir une crypto asymétrique en cloud*

*Gestion totale : AWS Lightsail / Developer Tool*

**CLOUD = $** (dépensé pour être gagnant) + **Légal** (lois en vigeur des services)

Stratégies de migration

*Prendre en compte la connaissance des équipes, le coût / financement, les infras en place ...*

*-> effectuer une formation / culture AVANT la migration vers le Cloud*

*-> cartographier l’app pour analyser quelle partie devra être migré / refactoré / supprimé / nesoin de license définie, ce qui* ***consomme****, ce qui est compatible avec le Cloud, ce qui peut être* ***monitoré****...*

-> Remise à plat total de l’application / infrastructure

[*https://aws.amazon.com/fr/cloud-migration/*](https://aws.amazon.com/fr/cloud-migration/)

*\** Lift and shift *:*

*faire migrer une application d'un env à un autre sans procéder à une refonte*

* **reLocate + reHost + rePlatform($)**
* Mode de migration le plus rapide : migrer une partie après l’autre
* ! Avoir des images containers légères (ex: docker FROM ...) !

<https://hub.docker.com/_/scratch>

*\** Cloud Native *:*

***- r*eFactor + rePurchase($)** (achète iso-local dans le Cloud)

- Réarchitecturer pour être envoyé vers le Cloud

+ On commence par serverless (applicatif sur IaaS), puis propre infrastructure

- Nécessite une bonne connaissance du Cloud (équipe / tech) avant de migrer

Security Data Outpost (toute l’infrastructure Cloud sur un serveur à domicile)

<https://aws.amazon.com/fr/outposts/>

Cloud Hybride / Multicloud : Permet de Gateway plusieurs Cloud (AWS, GCP, Azure...)

- CASB Cloud Access Security Broker

1. **Big Data**

Point principal : Quelles sont les **décisions** que je vais faire avec ces données (schéma, tendances, analyse) qu’on ne peut pas percevoir en tant qu’humain.

- Analyses pertinentes

- Modèles de calcul

-> charge de travail

-> formats

4V:

* Velocity : Vitesse d’accumulation des datas
* Volume : Echelle / phase de croissance de stockage des données (mo, go, to ?)
* Variété : Données structurées (rentrer dans du SQL) ou non (images, vidéos..)
* Véracity : Exactitude / confiance de la donnée
* (value : transformer la data en une valeur utile)

<https://blog.avanci.fr/le-big-data-et-les-4-v-de-la-donnee>

Cycle de vie des datas

* Découverte : Définir la problématique business (quoi comme donnée, comment..)
* Préparer : Extract-Tranform-Load. (<https://www.talend.com/fr/>)
* Modèles de planification : méthodes, flux
* Création de modèles : Dégager un ensemble de datas, flux travail, env d’execution
* Communication des résultats : valeurs, critiques, risques => visualisation
* Mettre en opération

Data Science : Processus de nettoyage + analyse pour tirer des informations utiles

Application : HADOOP

Gouvernance des données :

* Politique d’optimisation, sécurité
* Rentabilité, conformité règlementaires RGPD

*HDFS* ***est un système de fichiers distribué qui gère de grands ensembles de données s'exécutant sur du matériel de base****. Il est utilisé pour faire évoluer un seul cluster Apache Hadoop vers des centaines (voire des milliers) de nœuds*

Avant de stocker : Utiliser un extracteur multisource HDFS (filtre, enrichi, formate)

! Difficulté à faire du temps réel - Beaucoup d’étapes et de traitements !

! Eviter les pertes : Nécessité de faire des copies de data !

! Infra super complexe. Réseau sécurisé ajoutés + gouvernance (monitor) !

Afin de traiter plus efficacement : Utiliser un router

- Stocker les données en multi-destination (un peu Mondo, un peu S3..)

- Aussi des langages de requête par node : Hive, Pig...

- Utiliser un convertisseur de protocol

MapReduce : Implémentation de traitement avec algo en parallèles

* Map : Filtrer et trier
* Reduce : Assembler

<https://aws.amazon.com/fr/emr/>

<https://fr.cloudera.com/> + HortonWorks

Architecture Lambda :

- Equilibre le débit et la tolérance aux pannes

- Diminuer la latence de MapReduce

3 couches : Batch (pré-calcul résultats) Stream (vues instantanées) Vue (Mise à jour par lot)

<https://powerbi.microsoft.com/fr-fr/>

DataLake :

*Un Data Lake* ***contient des données de manière non structurée****. Il n'y a pas de hiérarchie ou d'organisation entre les différents éléments de données. Les données sont conservées sous leur forme la plus brute et ne sont ni traitées ni analysées. (Like AWS S3)*

<https://www.teradata.fr/>

1. **IoT Internet of Things**

Bcp d’appareils avec :

- Petites capacité mémoire et calculs (ex : capteurs, alarmes…)

Il faut récupérer ces données et les analyser (hôpitaux, process d’usines, logistique, maisons, agriculture..)

Fonctionne aussi sans connexion internet :

1. Envoi les données dès qu’internet est dispo
2. connexion variable (data stockées, état de l’appareil)
   * connectivité
   * messagerie
   * Mise a jour
   * Flotte d’appareils
3. Sécurité + analyse (éviter le bruit parasite)

<https://www.freertos.org/>

Protocole de messagerie : MQTT

<https://www.hivemq.com/blog/mqtt-essentials-part-4-mqtt-publish-subscribe-unsubscribe/>

<https://www.oasis-open.org/>

*AWS IoT / Cloud IoT Core*

IoT Alert : Vérifie qu’on ne s’éloigne pas des métriques attendues

* Audit
* Surveillance d’appareils
* Signaler les connexions
* Anomalies

Microcontrolleur : puissance augmentée

* On peut confier de l’analyse directement dans l’objet IT

Si encore plus de puissance (ex using temps réel d’alerte)

* Plugins d’appareils

Comment lier tous le sappareils ?

-> Service d’identité : IoT Core

-> Autorité de certification (SigV4)

Utiliser :

-> Passerelle d’appareils (WebSocket, TLS..)

-> Broker de messages : récupérer avec router fiable et faible latence

- Aide à la santé de l’objet

-> Moteur de règles : Aide au pré-traitement

-> Objet “shadow” : Objet servant de référence commune aux autres objets connectés (même hors-ligne)

-> Registre des appareils

Analyse de données: Difficile à faire

- Pertes de connexions

- Données non structurées ou variable (genre un ping d’alerte ou un “bip”)

-> Compréhensibles uniquement dans un contexte donné

Solut° : Il faut enrichir la donnée reçue

-> Date, heure, localisation, logique d’utilisation...

-> Cela permet de nettoyer les fausses lectures, combler les lacunes

-> En place un code /signal personnalisé (pour tests et maintenance)

**Exemple : Construire une solution IoT industrielle**

Réduire les coûts énergétique, détecter et résoudre les problèmes d’équipement..

Termes :

* Data de technologie opérationnelle (machinerie => micro-arrêts)
* Système de cycle de vie du produit
* Analyse des causes profondes (machine en panne)

-> Représentation cohérente (usine) => partager ces données (serveur central)

! Souvent les datas restent sur site, puis sont jetées sans utilisation

-> Mesures de perf - efficacité globale de l’équipement

-> Créer des apps pour analyser et prévenir une maintenance trôp coûteuse

1. Ingérer les datas d’équipement ou de services legacy
2. Organiser ces datas et les analyser
3. Lire des datas sur site avec des protocoles de normes industrielles de communic.
4. Représentation visuelle des flux de data sur les acifs physiques
5. Mesures de perf industrielle
6. Trouver une représentation visuelle (tableaux de bord) de ces perf.
7. Exploiter les données d’actifs finales pour améliorer ces maintenance prédictives

<https://aws.amazon.com/fr/quicksight/>

<https://prometheus.io/>

Jumeaux numériques ( Digital Twin) : Réplique numerique / 3d d’une machine industrielle sue laquelle on superpose des datas réelles

---

### Appréhender clairement les exigences

1. Réfléchir à chaque composant
2. Diviser votre architecture en tranches
3. Créer un prototype
4. Identifier et quantifier les exigences non fonctionnelles