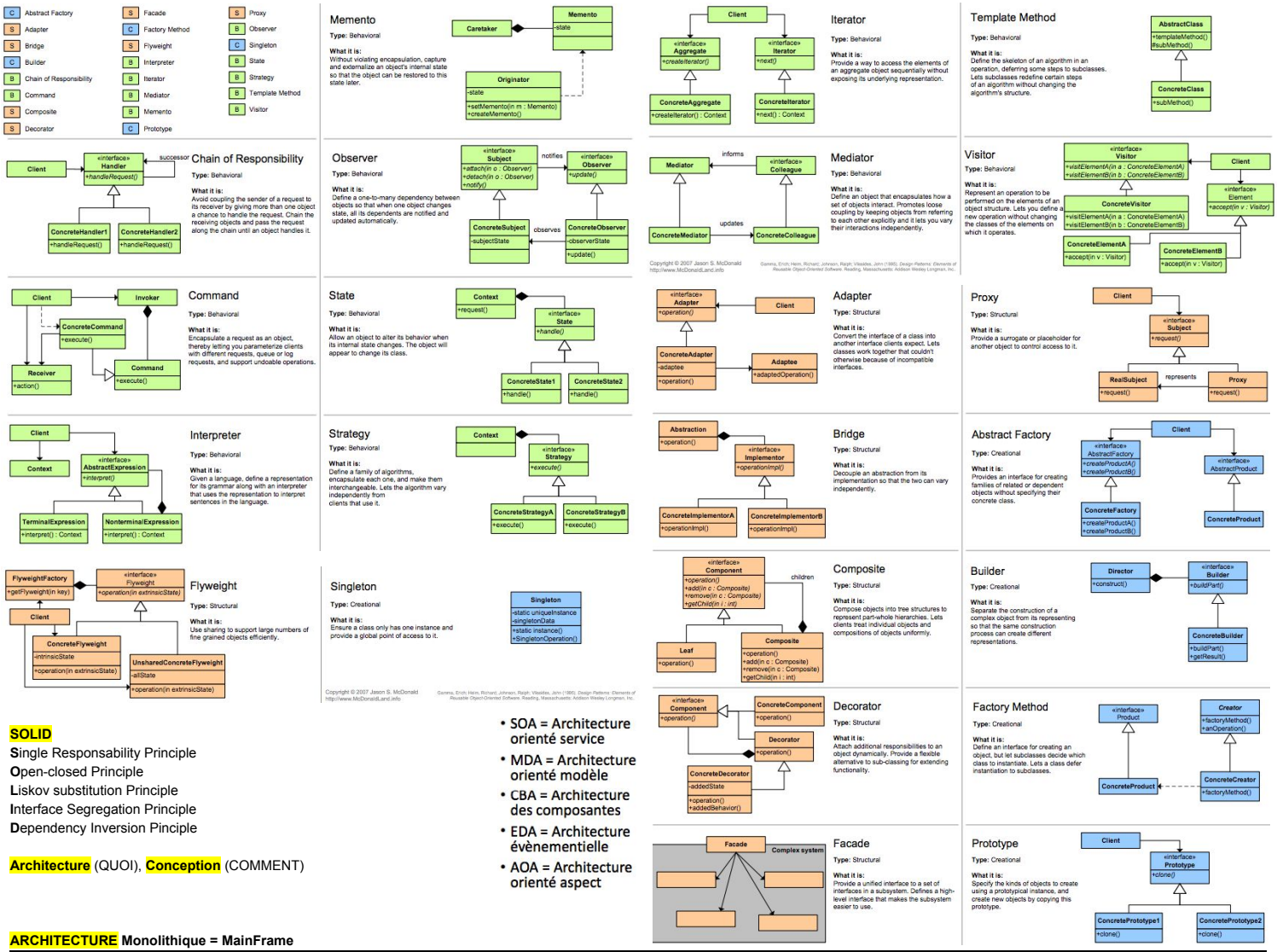


SOLID
Single Responsability Principle
Open-closed Principle
Liskov substitution Principle
Interface Segregation Principle
Dependency Inversion Principle

Architecture (QUOI), **Conception** (COMMENT)

- SOA = Architecture orienté service
- MDA = Architecture orienté modèle
- CBA = Architecture des composantes
- EDA = Architecture évènementielle
- AOA = Architecture orienté aspect

ARCHITECTURE Monolithique = MainFrame

Monolithique (1 seul prog, petite equipe expert)		Modulaire (SRP et DIP respecté)		Distribué (modulaire split sur fournisseurs)		MVC (Modulaire)	
Sécurité max	Pas de réutilisabilité	Dév parallele -> + efficace	Assembler les modules	Transparence	+ complexe	Séparation des resp	+complexité
Efficace (pas de délai)	Évolutivité, extensibilité	Maintenance facile, facile ajouter devs	Besoin de communication	Ouverture	- sécuritaire	Partage du modèle entre divers V et C	Bcp d'alternatifs peut mener à la confusion
Petite taille -> efficacité ++	Maintenance difficile et coûteuse	Unit testing	Connaît pas le système entier	Concurrent -> + performant	Nécessite + de gestion	Cadriciels existants	Re-renders inutiles
Dev, maintenance, évolution (1 responsabilité)	Dur intégrer des nouveaux devs	Réutilisable	Gestion nécessaire	Évolutivité	Effets imprévisibles		

Modulaire

Pipe-Filter		Layered (couches peuvent pas communiquer)		Blackboard		Event-bus (source, channel, bus, listener)	
"Tuyaux" d'opérations, transformations progressives sur données		Architecture en couche (UI-service-domain-persistence)		Un gros travail, un planificateur qui organise le travail entre le sources de connaissances		Si on a plusieurs modules qui veulent communiquer entre eux	
Efficace et parallèles	deadlocks	Maintenance facile	Impossible d'utiliser des couches non immédiates	Grand, décomposables problèmes	Planificateur lourd complexe	Supp/ajout de modules en cours d'execution	Risque de dépassement de capactié si bcp de msgs
Facile d'ajouter des ops	Un seul type d'entrée	Exploiter expertise Devs	Sensibles aux interfaces	Plusieurs expertises	Ordre d'exécution?	Les modules ne gère pas la livraison des msgs	Synchronisation est critique
Filtres peuvent attendre	Les filtres doivent être interchangeables	Couche réutilisables		Faciles d'ajouter nouvelles source	Synchro et contrôle d'accès		
Récursion possible					Impo modifier structure données		

Distribué

Client-server	Multitier (couches peuvent communiquer)		Master-Slave		Peer-to-peer	
Un serveur pour plusieurs clients (réseau). Client-léger ou client-lourd. Trop générique pour avantages/inconvénients Architecture fondamentale de l'internet	Extension de client-server (+de 2 niveaux)		Grand travail séparé par le maître.plusieurs slaves(même programme) exécute le travail		Données partagés par communication directe, non centralisé	
	Vieille technologie robuste	Modules specifique à une technologie (-flex)	Parallélisation	Architecture sensible au maître(master backup possible)	Système robuste aux échecs	Qualité, peformance et sécurité non garanties
	Modules réutilisables	Fonctionnalité et dispo dépend du réseau	Présence du maître = +fiabilité	Esclaves isolés peut mener à de la redondance	Évolutivité augmentée	
	Échange dynamique des modules facile	Sécurité complexe et réduite	Résultat garanti	Problème doit être décomposable		

Distribué (suite)

Semaine 3: Qualité

Broker		Service-oriented architectures (SOA)		REST vs. SOAP		ISO IEC modèle
Courtier choisi le bon service selon la demande du client (établi connexion client-service)		Tout est un service ! (logiciel autonome et atomique)		Simple Acess Object Protocol (remplace broker)		- Pertinence fonctionnelle - Fiabilité - Usabilité - Efficacité - Maintenabilité - Portabilité - Sécurité - Compatibilité
				REST	SOAP	
Dynamiquement ADD et DEL serveurs	Transaction et exception doivent être gérés	Composition, réutilisation, interop	SOAP est très lourd	+facile	Standardisé	
Distribution transparente		Client de dépend pas d'une technologie	Plusieurs technologies ne marchent pas (UUDI)	+simple	Indépendant du lang	
Serveur et client découplés		Abondance d'outils atutomatiques		+haute compatibilité	Bon support des outils	
		Abondance d'options			Gestion des fautes	

Métriques

Taille

LOC	**Lines of code**	Compréhensibilité, Attraction, Analysabilité, Possibilité de modification, Testabilité, Adaptabilité, Facilité d'apprentissage, Stabilité, Remplaçabilité, Interopérabilité, Sécurité, Tolérance aux pannes, Récupérabilité
SIZE2	**Nb attributs + meth**	
NOM	**Nb methodes**	

Complexité

CC	**Complex. Cyclo**	Compréhensibilité, Opérabilité, Attraction, Analysabilité, Possibilité de modification, Testabilité, Adaptabilité, Facilité d'apprentissage, Stabilité, Remplaçabilité, Interopérabilité, Sécurité, Tolérance aux pannes, Récupérabilité
WMC	**Weighted. Met. class**	
RFC	**Response for class**	

Héritage

DIT	**Profond arbre heritage**	Compréhensibilité, Opérabilité, Attraction, Analysabilité, Possibilité de modification, Testabilité, Adaptabilité, Facilité d'apprentissage, Stabilité, Remplaçabilité, Interopérabilité, Sécurité, Tolérance aux pannes, Récupérabilité
NOC	**Number of children**	Compréhensibilité, Opérabilité, Attraction, Analysabilité, Possibilité de modification, Testabilité, Adaptabilité, Facilité d'apprentissage, Stabilité, Remplaçabilité,

Couplage Métrique

CBO	**Couplage between objects**	Compréhensibilité, Opérabilité, Attraction, Analysabilité, Possibilité de modification, Testabilité, Adaptabilité, Facilité d'apprentissage, Interopérabilité, Sécurité, Tolérance aux pannes, Récupérabilité
Ca	**Couplage afférent**	Opérabilité, Attraction, Facilité d'apprentissage, Stabilité, Remplaçabilité, Interopérabilité, Sécurité, Tolérance aux pannes

Cohésion

LCOM	**Lack cohesion method**	Maturité, Compréhensibilité, Attraction, Analysabilité, Possibilité de modification, Testabilité, Adaptabilité, Facilité d'apprentissage, Stabilité, Remplaçabilité
TCC	**Tight class cohesion**	**Quand la LCOM se réduit ou la TCC augmente**

Documentation

LOD	**% d'éléments pas commentés**	Compréhensibilité, Opérabilité, Attraction, Maturité, Analysabilité, Possibilité de modification, Testabilité, Adaptabilité, Facilité d'apprentissage, Stabilité, Remplaçabilité

	Functional suitability	Performance efficiency	Usability	Compatibility
1st choice	CBA	EDA	MDA	SOA|MDA
2nd choice	SOA|AOA	AOA	EDA	CBA|EDA|AOA
3rd choice	MDA|EDA	MDA|CBA	CBA|AOA	
	Reliability	Security	Maintainability	Portability
1st choice	CBA|AOA	CBA	MDA	AOA
2nd choice	MDA	AOA	SOA|CBA	EDA
3rd choice	EDA	EDA	EDA|AOA	SOA|MDA|CBA

Semaine 4 : Mauvaise Conception

N	Architecture by Implication	Autogenerated Stovepipe	Cover your assets	Design by commitee	Jumble	Reinvent the wheel
D	Manque planification, 2 projets pas égaux, ignorer nouvelles technologies	Interfaces du système original inapproprié pour migration Syst. Dist.	Documentation a trop de détails pour couvrir toutes les options et devient compliquée	Tous les membres influence l'archit. Le résultat est trop complexe et trop caractéristiques	Mélange des éléments horizontaux et verticaux	Self explanatory
R	Modulariser la spec en vue et POV. Exprimer à l'aide d'objectifs et questions	Redéfinir les interfaces. Focus surl'interop. Et stabilité	Utiliser abstractions, diagrammes et tables	Définir des rôles claires. Désigner un architecte principal. Prioriser les exigences. Organiser. DevOps	Horizontaux=abstractions verticaux=implémentations	- processus agile - raffiner exigences, décision de conceptions

N	Spaghetti Code	Stovepipe System	Swiss Army Knife	Vendor Lock-in	Missing Abstraction	Multifaceted abstraction
D	Manque de structure et cohérence	Chaque intégration d'un sous-système est spécial. Manque d'un plan d'intégration	Interface très complexe qui expose beaucoup de fonctionnalités (SRP, ISP violé)	Système répond aux logiciels/matériels propriétaires	Groupes de données se produisent ensemble souvent	Abstraction a plus d'une responsabilité (SRP)
R	Utilisation de patrons et bonnes pratiques	Définition d'une interface document (SOA)	- patron facade ou adapter - Documenter usage et implémentation interface	Architecture multi-niveaux	Extraire de la classe	Extraire classe

N	Duplicated abstraction	Deficient encapsulation	Unexploited encapsulation	Broken modularization	Insufficient modularization	Cyclically - dependend modularization
D	Self explanatory	L'abstraction donne plus de permissions que nécessaire	Utilisation des contrôles de type lorsque polymorphisme est disponible	Des données ou méthodes qui doivent êtres groupées sont dans diff. abstractions	Une abstraction a bcp de membres publiques / méthodes complexes (SRP && ISP)	Trop haut couplage
R	Extract from class	Encapsuler les données et donner méthodes d'accès	Replace type checking by polymorphisme	Move method/field. Inline class	Extraire classe / interface	Move methodéfield, inline class

N	Unfactored hiercharchy	Broken hierarchy	Cyclic hierarchy	Duplicated code	Long method	Large class
D	Duplication classes dérivées et classe de base	Classe de base et dérivées ne sont pas "est un"	Classe de base a une association avec classes dérivées	Code existe ailleurs	Longue mtéhode	Classe qui a trop de membres ou trop de responsabilités
R	Extract superclass, pull up method or field	Replace inheritance by delegation OR Inverse relationship	Extract Class, Move method, Inline class, state/strategy pattern	Extract class, decompose conditional, compose method	extract method	Extract class

N	Long parameter list	Divergent Change	Shotgun surgery	Feature Envy	Trop de switch	Message chains
D		Classe change bcp à chaque changement d'exigence	Plusieurs endroits à changer pour 1 changement	Méthode utilise plus de membres d'une autre classe que la sienne	Replace Type code with subclasses or state/strategy.	Longue invocation de méthodes en cascade
R	Parameter object, replace parameter with method	Extract Class	Move method or field, inline class	Move method	Replace conditional with morphism	Extract and move method

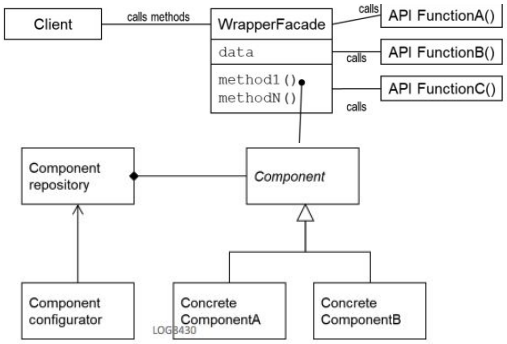
4 Défis architecture: Accès et configuration de service, synchronisation, concurrence, gestion des événements

N	Inapropriate intimacy	Refused Bequest
D	Trop d'associations avec les membres privés d'une autre classe	Classé dérivé pas besoin d'hériter membre classe de bas
R	Move method/field. Extract Class. Hide delegate	Push down method or field

Large Class : LOC > HIGH AND LCOM < 1/3 AND CBO > FEW AND WMC > HIGH
Long Method : LOC > HIGH AND CC > FEW
Feature Envy : CBO > FEW AND LCOM < 1/3
Type Checking : LOC > HIGH AND CC > FEW

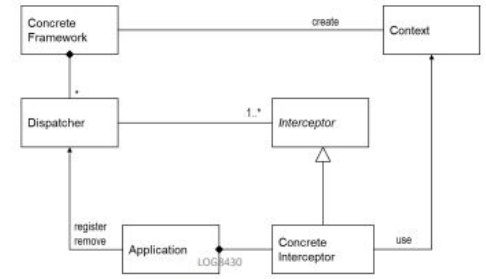
SEMAINE 7 : Architecture Événementielle

Patrons de service access and configuration

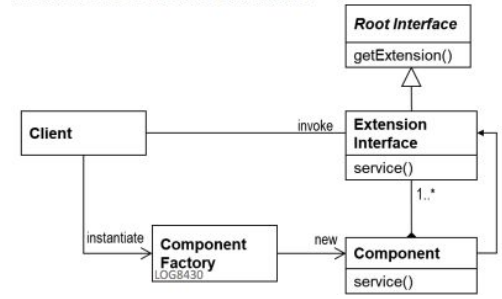


Wrapper Façade: Encapsulation des fonctionnalités de bas niveau dans une interface de haut niveau
+ Des interfaces de haut niveau cohésives et fiables.
+ Portabilité, maintenabilité, modularité et réutilisabilité augmentées.
- Perte de la fonctionnalité.
- La performance est diminuée.
- Des limitations imposées par les langages et les compilateus.

Component Configurator: Changer dynamiquement implémentation, évitant de charger configuration en mémoire
+ Uniformité de la configuration et contrôle de l'interface
+ Administration centralisée
+ Modularité, testabilité et réutilisabilité augmentées
+ Configuration dynamique
+ Optimisation augmenté grâce à plusieurs possibilités de configuration
- Dynamisme augmente l'incertitude à cause de toutes les interactions possibles parmi des composantes configurées dynamiquement
- Sécurité et fiabilité réduites
- Complexité augmentée et performance réduite



Interceptor: Ajouter service au cadriciel de manière transparente et déclenché automatiquement
+ Extensibilité et flexibilité du cadriciel augmentées
+ Séparation des concernes parmi les services
+ Capacité de monitorer et de contrôler les cadriciels
+ Réutilisabilité des intercepteurs
- Difficulté à attendre les évènements qui doivent être interceptés et à séparer les interfaces des intercepteurs
- Les intercepteurs sont des points d'insertions des vulnérabilités ou des fautes
Possibilité d'interception en cascade à cause des changements au cadriciel



Extension interface: Plusieurs interfaces pour garantir ISP
+ Extensibilité des composantes
+ Séparation des concernes parmi les rôles
+ Polymorphisme parmi des classes pas liées
+ Découplage des composantes et des clients
+ Agrégation et délégation des interfaces
- Effort augmenté pour la conception et l'implémentation des composantes
- Complexité des clients augmentées
- Indirection additionnelle et surcharge d'exécution

Synchronization patterns

Scoped Locking		Strategized Locking		Thread safe interface	
- obtenir des verrous pour protéger concurrent code - classe de garde -> acquiert verrou (constructeur) -> libère verrou (destructeur)		**- accès multithread** **- types enfichables**		**- Accès valeur sur multithread** **- tout traiter invocation intra-component selon:** **- méthodes d'interface contrôle les verrous** **- méthodes d'implémentation font confiance aux méthodes publiques que les verrous sont contrôlés**	
+ robuste **+ fiable**	**- destructeur pas toujours appelé** **- récursion problème**	**+ flexibilité, réutilisable** **+ effort réduit de maintenance**	**- verrouillage inclusif** **- sur ingénierie**	**+robuste** **+simple**	**-Blocage potentiel**

CONCURRENCY PATTERNS

Double-checked locking optimization		Active Object		Half-sync half-async	
-réduire les conflits et la surcharge de synchronisation à 1 seul point - On ajoute un bool sur les section critiques pour savoir si on a besoin d'attendre un verrou (On peut ainsi ignorer certaines sections)		Découpler l'exécution de méthode de l'invocation		**Découpler les services async de ceux sync**	
+ minimise surcharge	**-augmentation du nombre de mutex**	**+ concurrence, parallélisation**	**-surcharge, complexe a debuguer**	**+ simple, performant, centralisation communication**	**- pas tjrs bénéfique d'avoir IO async**

SEMAINE 8:SOA

Fournisseur: Fournir le service, responsable du déploiement, disponibilité et qualité du service
Client: Consomme le service, doit suivre les termes d'utilisations imposés par le fournisseur
Réseau: moyen de communication
Service et application: applications normales indépendentes
Interface de service: défini les opérations et données qui peuvent être utilisé par le client (métainfos, peut être consommer par outil ex: stub)
Service stub: code qui rend possible d'accéder le service(généré automatiquement ou librairies)
Protocole de communication: HTTP, SMTP, FTP
SLA: entente entre le fournisseur et le consommateur, spécifie les métriques

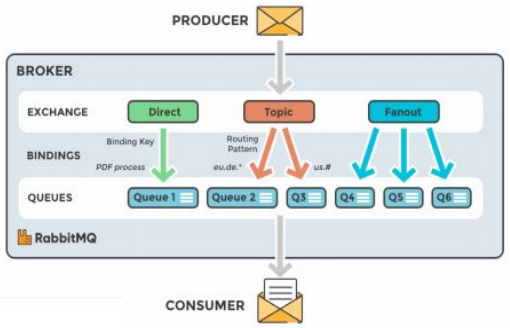
MIcroservice: version plus petite de SOA -. Chaque service est une opération avec une responsabilité (+ réutilisable, grosse partie du DevOps)

SOAP	REST	RPC
Protocole de message en XML(normes WSDL[endpoint, operations, input, output]) **Avantages:** indépendant du protocole, redéfinir normes **Désavantages:** non efficace, interfaces longues et complexes, parcourir fichier est coûteux	**Services web, donc HTTP** **Accessible par URL, GET, PUT, POST, DELETE** **1 demande = 1 réponse** **Avantages:** simple, performant, modifiable, fiable, evoluable, etc .. **Désavantages:**manque de normes et spécifications formelles, fonctionnalités complexes	**Format:** JSON, XML **Appelle n'importe quel le méthode (comme un objet)** **CORBA est RPC** **Avantages et désavantages** Presque comme REST mais avec plusieurs capacité au niveau des méthodes dispo. Et sans l'organisation des données imposées:

Semaine 10: Architecture des Mégadonnées

AMQP (Advanced Message Queuing Protocol)

Courtier	Producteurs et échanges	Consommateurs et Queues	Messages
- implémente interopérabilité - enlève responsabilité des modules - asynchronicité - Peut transformer les messages (ex: changer format) **Garantie**: sécurité, performance, fiabilité, évolutivité	-soumet messages au courtier et aux échanges - Échanges acheminent messages au queues selon algo **Direct:** vers queue avec ID **Fanout:** toutes queues liées a échange **Thème:** selon la clef et un param **Entête:** selon plusieurs params qu'une clef	- consommateur consomme en registrant au queues - pull ou push messages dans la Q - Q agit comme stockage temporaire - Q durable ou pas, stockée dans le disque si survit la relance du courtier - Q peut etre exclusive à une connexion - liés aux échanges par des liaisons	- Peut être persistent -> sauvé dans le disque et survivre la relance - contient: - type et encodage (du contenu) - clef d'acheminement - persistent ou non - priorité - ID producteur - Horodatage et temps d'expiration



RabbitMQ

« Input Systems » = Producteurs

« Output Systems » = Consommateurs

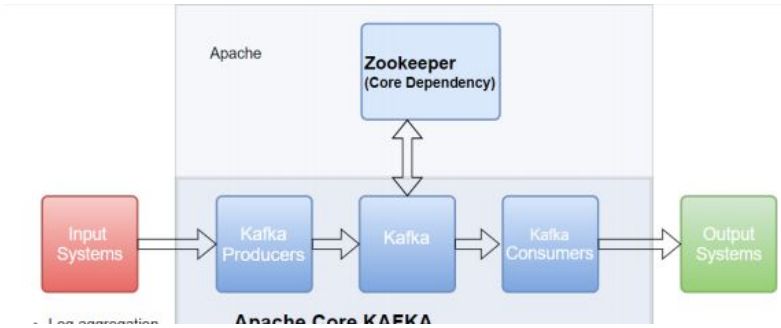
« Kafka Producers » = Échanges

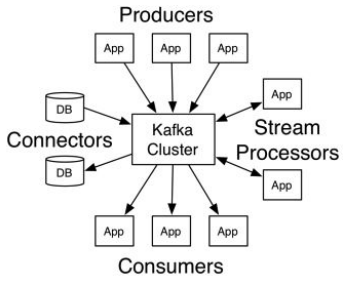
« Kafka Consumers » = Queues

« Kafka » = Courtier

Pour Kafka, les échanges et les queues sont comme le stub est pour les services.

- Ce sont des classes locales qui seront appelées par les systèmes externes.





RabbitMQ : implémentation directe d'AMQP(implémente tout d'AMQP)

RabbitMQ
- Courtier intelligent/Consommateurs simples
- Fiabilité et découplage sont les priorités.
- Plusieurs types d'échange sont supportés.
- Transmission directe ou multidiffusion.
- Sécurité est augmentée.
- La haute performance est possible mais avec des ressources extensives.

Kafka
- Courtier simple/Consommateurs intelligents
- Streaming permet l'utilisation d'une seule connexion avec un taux d'entrée augmentée.
- Juste l'échange par thème est supporté.
- Transmission par diffusion.
- Sécurité peut être un problème.
- La performance est augmentée.

MapReduce : Avantages et Désavantages

Avantages
- Efficacité grâce au parallélisme.
- Disponibilité.
- Tolérance aux pannes.
- Évolutivité.
- Mobilité des données minimale.
  - Le traitement des données peut se produire dans le serveur qui contient les données.

Désavantages
- « Region hotspotting »
  - L'utilisation des clés pour partitionner et partager les données parmi les ouvriers peut charger un ouvrier particulier.
  - On peut résoudre cette situation en utilisant le hachage sur les clés.
- Il n'est pas possible que tous les façons de traitement soient implémentées par MapReduce.

Contient: un algo, une implémentation, une infrastructure. 2 tâches (map et reduce). Map: travail d'analyse, reduce: aggréger les résultats des tâches map individuelless. Architecture master-slave. Workers acceptent des tâches map et ou reduce

2 implémentations de MapReduce : Haddop, Apache Spark, Hadoop implementation directe. Spark fournit son propre gestionnaire fournit d'autre capacité, SparkSQL, etc. 2 Processus (un driver, des executers) RDD (Resilient Distributed Dataset)

Rapide (100x plus que Hadoop), tolérant au pannes (RDD récupérables). Spark streaming (petits flots à intervalle) crée des DStreams(fils de RDD). Kafka Streaming 1 theme = 1 flux, mltithread
Spark: basé sur temps, fausse analyse de stream (intervalle), + stable, + portable. Kafka: flux basé sur événements, + efficace
Apache Kafka: service distribué de streaming, streams au lieu de queues, transmission selon thème uniquement, 1 messages lus par plusieurs consommateurs, toutes les données sont persistantes

Semaine 11: Architectures des Mégadonnées

Base de données distribués: gestion de grands volumes de données et garantir une disponibilitée presque absolue
Théorème **CAP**: **C**onsistency, **A**vailability, **P**artition tolerance(fonctionne après échec d'un noeud) (IMPOSSIBLE DE GARANTIR LES 3)
ACID: **A**tomique(réussi ou échoué), **C**ohérente, **I**solée, **D**urable(résultats permanents même en cas d'échec)
BASE: **BA**sic availability,**S**oft-sate(non-cohérente en écriture, répliques doivent être cohérentes), **E**ventual Consistency(cohérence ultérieure)
NoSQL:

Key-Value	Document	Graph	Wide Column
Données organisées en pairs de clés et de valeurs, Notion d'objet, pas de schéma	Données en tant que fichier(JSON ou XML)	Données en forme de graphe. Noeuds = entités, arrêtes = relations	Similaire aux BD relationnelles, système de tableaux, rangées, colonnes, définition flexible dune colomn(Super column family, row key, column family, column)
Très efficace pour app simple	Possible de demander les données selon les attributs	N'importe quel relation de façon pratique et efficace	Migration depuis BD relationnelles facile
Peut stocker n'importe quoi	Définir par des métadonnées	Existe pour traverser des graphes déjà existants	Flexibilité des famille de colonnes
	Indices primaires et secondaires		Énorme quantité de données
			Supporte facilement MapReduce

Lamda: garantie la disponibilité des données volatile, garanti la cohérence des données immuables
Kappa:Planifier des rollbacks, système orientées messages pour gérer l'entrée des données

COMMENT ON VA SÉLECTIONNER LA SOLUTION NoSQL

Key-value
- Mise en cache
- Lorsque on a des données séquentielles (listes, queues etc.)
- Pour supporter des systèmes publish/subscribe
- Pour maintenir un état

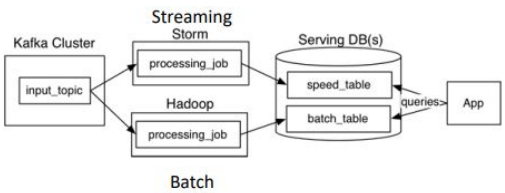
Document
- Avec des données très complexes et imbriquées
- Lorsque on a des clients en JavaScript

Graph
- Lorsque les relations entre les données sont très complexes
- Pour travailler avec des hiérarchies et des taxonomies

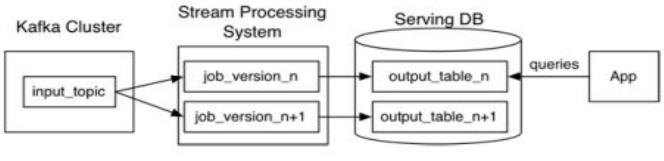
Wide-column
- Lorsque on a des données sans structure, qui ne changent pas beaucoup et qui d stockées pendant longtemps
- Lorsque on a besoin d'un stockage évolutif

	Hadoop	Spark
Performance	o	+
Utilisabilité/Portabilité	+	++
Frais	o	-/+
Compatibilité	+	+
Traitement de données	-	+
Tolérance aux pannes	+	++
Évolutivité	++	+
Sécurité	+	o

LAMDA -> :-)



KAPPA



Lambda
- Les demandes des utilisateurs doivent être servies sur une base ad-hoc en utilisant le stockage de données immuable.
- Des réponses rapides sont nécessaires et le système doit pouvoir gérer diverses mises à jour sous la forme de nouveaux flux de données.
- Aucun des données stockées ne doit être effacée et cela devrait permettre l'ajout de mises à jour et de nouvelles données à la base de données.

Kappa
- Plusieurs événements ou demandes de données sont consignés dans une file d'attente pour être traités en fonction d'un stockage du système de fichiers distribué.
- L'ordre des événements et des demandes n'est pas prédéterminé. Les plates-formes de traitement de flux peuvent interagir avec la base de données à tout moment.
- C'est résilient et hautement disponible car la gestion de téraoctets de stockage est requise pour chaque nœud du système afin de supporter la réplication.