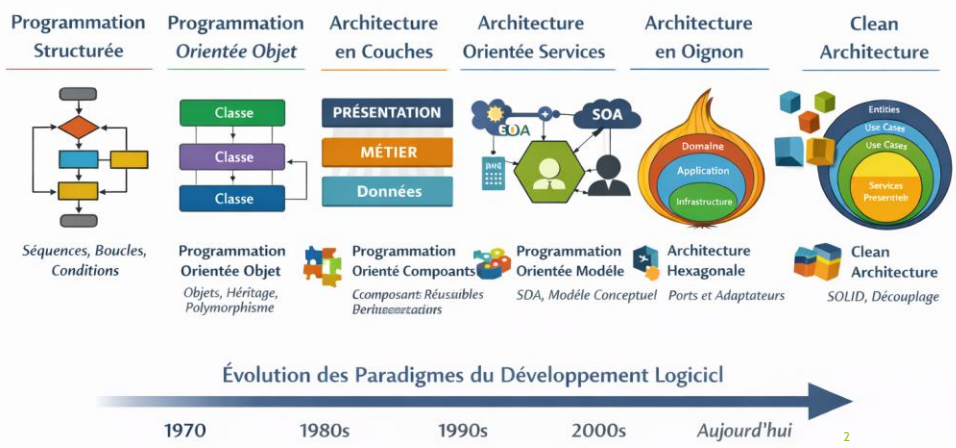
Architectures Logicielles

Chapitre 1
Introduction et généralités

Paradigmes de programmation



Programmation structurée (années 1970)

- Contexte: Avant, on programmait surtout en **code spaghetti** (GOTO partout).
- Difficile à lire, tester, maintenir.
- Principe: La programmation structurée impose :
  - Séquences, Sélections (if / else), Itérations (for, while), Découpage en procédures / fonctions
- Objectif: Améliorer la **lisibilité**, la **correction** et la **maintenabilité**.
- Langages : **C, Pascal**
- **Limites:**
  - Données et traitements peu liés
  - Mauvaise modélisation des systèmes complexes du monde réel

3

Programmation orientée objet - POO (années 1980-1990)

- Pourquoi?
  - Les systèmes deviennent grands et complexes
  - Besoin de mieux représenter le réel.
- Principes: le logiciel est vu comme un ensemble d'objets qui :
  - encapsulent **données + comportements**
  - interagissent par **messages**
- Concepts clés: Encapsulation, Héritage, Polymorphisme et Abstraction
- Objectif: Réutilisabilité, Extensibilité, Maintenance facilitée
- Langages : Java, C++, C#,…, UML devient central
- Limites:
  - Couplage fort
  - Difficultés pour les systèmes distribués

4

Objets distribués (années 1990)

- Pourquoi?
  - Explosion des réseaux et des systèmes répartis.
- Principe: Les objets peuvent être :
  - sur **des machines différentes,** communiquer via **le réseau,** comme s'ils étaient locaux
- Technologie: CORBA (Common Object Request Broker Architecture), RMI (Remote Method Invocation) (Java) et DCOM (Distributed Component Object Model)
- Objectif: Transparence de la distribution
- Limites: Complexité, Problèmes de performance et Faible passage à l'échelle

Technologie	Standard	Langage	Indépendance plateforme
CORBA	OMG	Multi-langage	Oui
RMI	Java	Java uniquement	Non
DCOM	Microsoft	Principalement Windows	Non

5

Programmation orientée agent (années 1990-2000)

- Pourquoi?
  - Certains systèmes sont : autonomes, dynamiques ouverts (ex : systèmes intelligents, multi-agents)
- Principe: Un **agent** est :
  - autonome
  - capable de perception, décision et action
  - parfois coopératif ou compétitif
- Concepts: Autonomie, Proactivité, Réactivité, Communication entre agents
- Domaine: Intelligence artificielle, Simulation, Systèmes distribués intelligents
- Limites: Peu de standards industriels, Adoption limitée en industrie

6

Agent réactif Vs Agent proactif

Réactivité	**Proactivité**
Répond à un stimulus	Initie une action
Basée sur perception immédiate	Basée sur objectifs
Court terme	Moyen / long terme
Comportement déclenché	Comportement planifié

- Exemple:
- **Agent réactif:** Un thermostat
  - Il détecte que la température baisse
  - Il allume le chauffage
  - Réaction immédiate à un stimulus
- **Agent proactif:** Un système de gestion énergétique intelligent
  - Il sait que la température va chuter cette nuit
  - Il anticipe et ajuste progressivement le chauffage
  - Action planifiée orientée objectif

7

Programmation orientée composant (années 2000)

- Pourquoi?
  - Besoin de **réutilisation à grande échelle.**
- Principe: Le logiciel est assemblé à partir de **composants** :
  - indépendants
  - avec interfaces bien définies
  - déployables séparément
- Exemples: EJB, COM+.NET Components
- Objectif: Réutilisation, Déploiement modulaire et Maintenance simplifiée
- Limites: Toujours fortement couplé au middleware et Interopérabilité limitée

8

Architecture orientée services – SOA (années 2000-2010)

- Pourquoi?
  - Interopérabilité entre systèmes hétérogènes.
- Principe: Fonctionnalités exposées sous forme de **services** :
  - faiblement couplés
  - accessibles via des protocoles standards
- Caractéristiques:
  - Autonomie des services, Découvrabilité et Réutilisation métier
- Technologie:
  - Web Services (SOAP), REST (Representational State Transfer), Microservices (évolution moderne)
- Objectifs: Agilité, intégration, évolutivité

9

Ingénierie / Programmation orientée modèle – MDE / MDD / IDM (années 2010+)

- Pourquoi?
  - La complexité logicielle continue d'augmenter
  - Le code devient trop bas niveau.
- Principes: **Le modèle devient l'artefact central**, pas le code.
  - Modèles → transformations → code
  - Automatisation
  - Séparation des préoccupations
- Concepts clé: Modèle, métamodèleTransformation de modèlesPIM / PSM (MDA – OMG)
- Avantages: Productivité, Qualité, Portabilité, Traçabilité
- Domaine: Systèmes complexes, Logiciels critiques

10

Synthèse

Paradigme	Centre d'intérêt
Structuré	Algorithmes
Orienté objet	Objets
Objets distribués	Objets + réseau
Orienté agent	Autonomie et intelligence
Orienté composant	Réutilisation
Orienté service	Interopérabilité
Orienté modèle	Abstraction et automatisation

11

Architecture logicielle

- Une **architecture logicielle** est la **structure fondamentale d'un système logiciel**, qui décrit :
  - ses **composants principaux**
  - les **relations et interactions** entre ces composants
  - les **principes et règles** guidant sa conception et son évolution
- C'est en quelque sorte le **plan d'un bâtiment**, mais pour un logiciel.
- Selon la norme ISO/IEC/IEEE 42010 :
  - L'architecture d'un système est l'ensemble des structures nécessaires pour raisonner sur le système, comprenant les éléments logiciels, leurs relations et les propriétés de ces éléments.
- L'architecture logicielle répond à la question : Comment le système est-il organisé pour satisfaire ses exigences fonctionnelles et non fonctionnelles ?

12

Architecture logicielle

- Que contient une architecture logicielle?
  - **Les composants**
    - Modules, services, classes, bases de données, interfaces, etc.
  - **Les connecteurs:** Mécanismes d'interaction :
    - appels de méthodes, API REST, messages, événements et bus de communication
  - **Les contraintes:**
    - Performance, Sécurité, Scalabilité, Fiabilité et Maintenabilité

13

Architecture logicielle

- Pourquoi est-elle importante?
  - L'architecture logicielle permet de :
    - Gérer la complexité
    - Garantir les qualités non fonctionnelles
    - Faciliter la maintenance
    - Permettre l'évolution
    - Réduire les risques techniques
    - Sans architecture claire : **dette technique rapide**

14

Patterns d'architecture logicielle

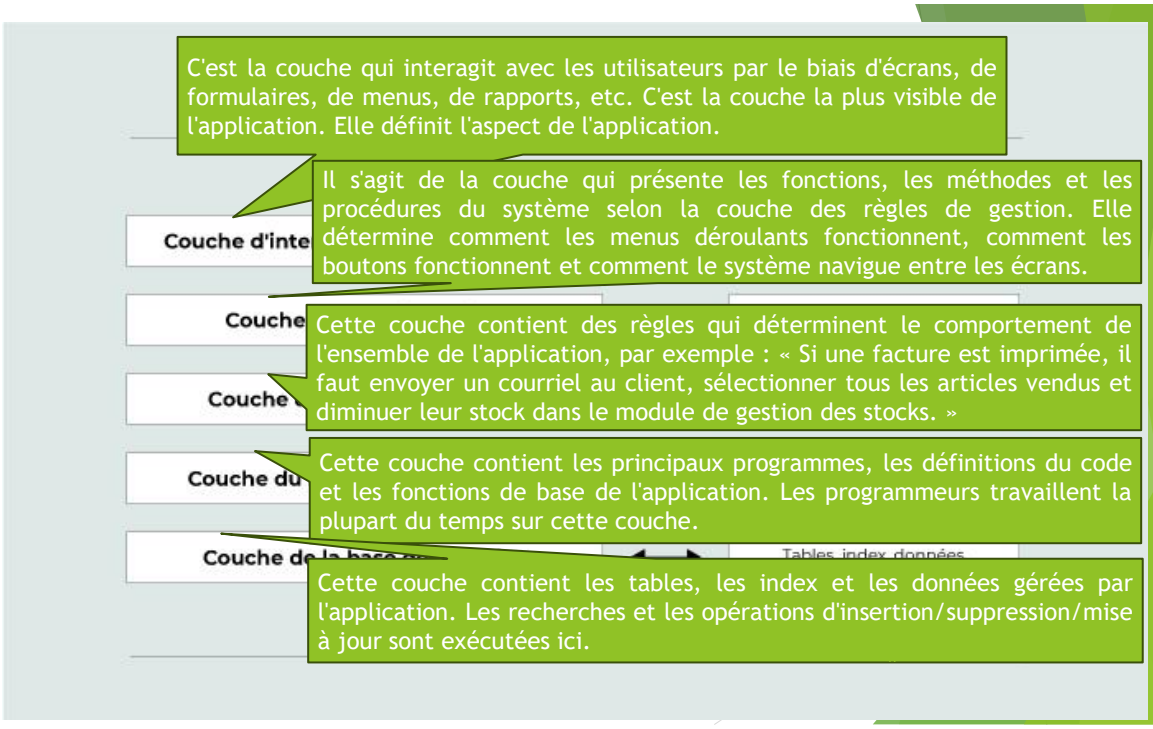
- Layered Architecture
  - MVP (*Model-View-Presenter)* pattern: dérivé de MVC
- Oignion Architecture
- Event-driven Architecture
  - CQRS
- Monolithic Architecture
- Microkernel Architecture (Plug-in Architecture)
  - Plugins dans l'IDE Eclipse
- Microservices
  - Exemple: Netflix
- SOA
- Clean Architecture
- …

15

Layered Architecture (Architecture en couches)

- L'architecture en couches consiste à organiser un système logiciel en couches hiérarchiques, où chaque couche regroupe des responsabilités homogènes et fournit des services à la couche supérieure.
- Les dépendances sont unidirectionnelles :une couche ne peut utiliser que les services de la couche située immédiatement en dessous.
- Architecture 3-tiers: variante
  - Présentation + Métier + Données
  - Très répandue dans les applications web

16



Layered Architecture (Architecture en couches)

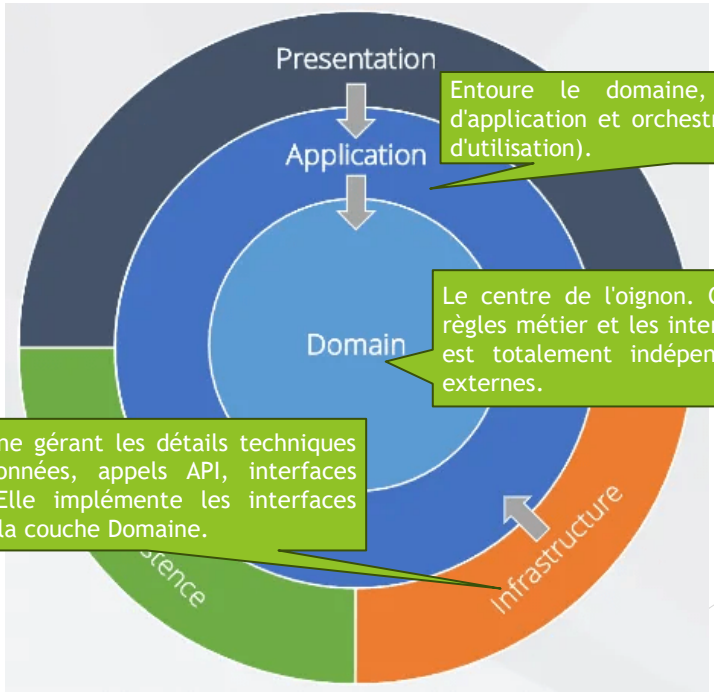
- **Avantages:**
  - Architecture simple et compréhensible
  - Forte séparation des responsabilités
  - Facilité de test unitaire
  - Maintenance facilitée
  - Évolution localisée
- **Inconvénients:**
  - Rigidité structurelle
  - Surcoût de communication
  - Performances parfois dégradées: Pour les petites applications, de nombreuses couches créent un problème de performance et sont très difficiles à maintenir.

18

Oignion Architecture

- L'architecture en oignon, introduite par Jeffrey Palermo en 2008, est un modèle de conception logicielle structuré en couches concentriques, plaçant la logique métier (domaine) au centre.
- Elle repose sur le principe de dépendance inversée : les couches externes dépendent des couches internes, mais jamais l'inverse, assurant un faible couplage, une haute testabilité et une indépendance vis-à-vis de l'infrastructure (bases de données, frameworks).

19



Entoure le domaine, contient les services d'application et orchestre la logique métier (cas d'utilisation).

Le centre de l'oignon. Contient les entités, les règles métier et les interfaces de repository. Elle est totalement indépendante des technologies externes.

Couche externe gérant les détails techniques (bases de données, appels API, interfaces utilisateur). Elle implémente les interfaces définies dans la couche Domaine.

20

Oignion Architecture: avantages et inconvénients

- **Avantages :**
  - **Faible couplage :** Les modifications de la base de données n'impactent pas la logique métier.
  - **Testabilité accrue :** La logique métier peut être testée sans dépendance externe.
  - **Indépendance technologique :** Facilite le remplacement de frameworks ou de bases de données.
- **Inconvénients:**
  - **Complexité initiale élevée**
    - L'architecture en oignon impose : plusieurs couches bien distinctes, des règles strictes de dépendance, une bonne maîtrise de l'injection de dépendances.
    - **Conséquence** : Pour un projet simple ou de petite taille, elle est souvent surdimensionnée.

21

Oignion Architecture: avantages et inconvénients (suite)

- **Courbe d'apprentissage importante**
  - Elle nécessite de comprendre :
  - l'inversion de dépendances,
  - les interfaces,
  - la séparation domaine / application / infrastructure.
- **Moins adaptée aux applications très simples ou CRUD**
  - CRUD basique,
  - peu de logique métier,
  - courte durée de vie,
  - **Conséquence** : effort inutile par rapport aux bénéfices, une architecture en couches classique ou MVC suffit largement.

22

Architecture en couches et architecture en oignon: que choisir ?

- **Choisir l'architecture en couches si :**
  - application simple ou CRUD,
  - règles métier peu complexes,
  - petite équipe ou projet court,
  - objectif : rapidité de développement.
- **Choisir l'architecture en oignon si :**
  - règles métier complexes ou évolutives,
  - projet à long terme,
  - besoin fort en testabilité,
  - équipe expérimentée,
  - volonté d'indépendance technologique.

23

Logique métier Vs logique technique

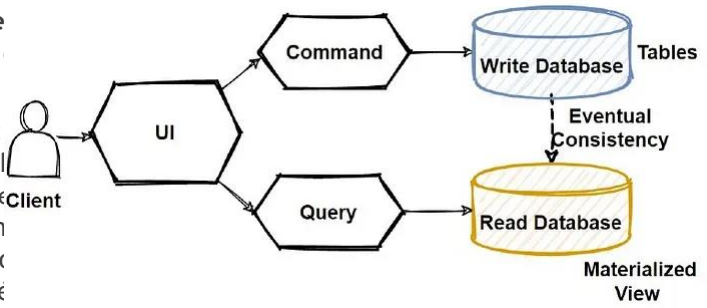
- La logique métier correspond aux règles et décisions propres au domaine que le logiciel doit respecter.

Logique métier	Logique technique
Règles du domaine	Implémentation
Indépendante des technologies	Dépendante des outils
Stable dans le temps	Change fréquemment
Au cœur de l'architecture	En périphérie

24

Style architectural Vs Pattern architectural

- Un **style architectural** décrit la **structure macro** d'un système.
- **Un patte** pouvant résoudre
- Exemple: **Responsi** Command opération système de donn opérations d'écriture).



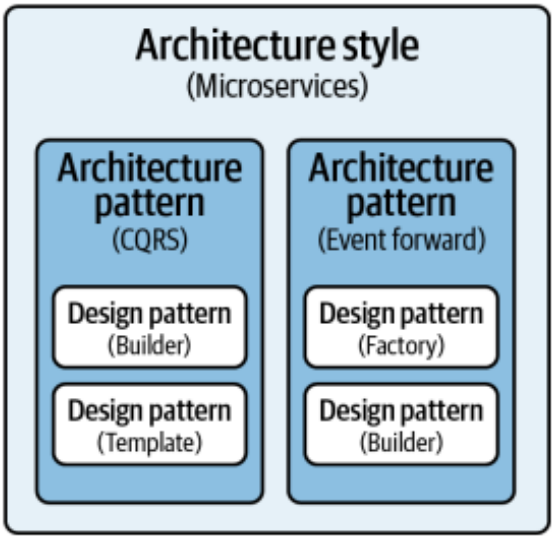
- Ce pattern architectural peut être appliqué à n'importe lequel des styles architecturaux afin d'optimiser les requêtes et les mises à jour de la base de données.

25

Pattern architectural Vs Design pattern

- Les patterns architecturaux, à leur tour, se distinguent des design patterns en ce qu'un pattern architectural impacte l'aspect structurel d'un système, tandis qu'un design pattern influence la manière dont le code source est conçu.
- Par exemple, on peut utiliser le design pattern *Builder* pour implémenter le pattern architectural CQRS (Command Query Responsibility Segregation), puis utiliser le pattern CQRS comme bloc de construction au sein d'une architecture microservices.

26



- Les **styles architecturaux** peuvent être composés de **patterns architecturaux**, qui, à leur tour, peuvent être composés de **design patterns**.

27

- Les **design patterns** et les **patterns architecturaux** sont généralement combinés pour former une solution complète.
- Les **styles architecturaux** fonctionnent de la même manière : ils peuvent également être combinés lors de la conception de solutions logicielles afin de constituer une solution complète.
- Les **styles architecturaux hybrides** sont fréquents dans le monde réel, car aucun style architectural ne peut résoudre tous les problèmes métier.
- Parmi les hybrides courants, on trouve :
  - Microservices orientés événements (event-driven microservices) : événements échangés entre microservices
  - Architecture microkernel orientée événements (event-driven microkernel) : événements entre le système central et des composants plug-in distants
- Bien que la création d'hybrides soit une pratique courante, il est essentiel de **comprendre chaque style architectural individuellement**, ainsi que ses forces et faiblesses, avant de les combiner.

28

Styles architecturaux

- Les **styles architecturaux** permettent d'utiliser des structures existantes et bien connues qui supportent certaines caractéristiques architecturales (également appelées **attributs de qualité non fonctionnels**, **attributs de qualité du système** ou « -ilities »).
- Offrent un **gain de temps** pour définir l'architecture d'un système donné,
- Facilitent la **communication** entre les développeurs, les architectes, les testeurs qualité et, dans certains cas, les parties prenantes métier.

29

Classification des architectures

- Les styles architecturaux sont classés en deux grandes structures principales:
  - **Monolithique** : une seule unité de déploiement
  - **Distribuée** : plusieurs unités de déploiement, généralement constituées de services
- Classification importante: les architectures distribuées offrent des caractéristiques très différentes des architectures monolithiques.
- Savoir quel type de classification utiliser constitue la première étape pour choisir l'architecture adaptée à votre problème métier.

30

Unité de déploiement

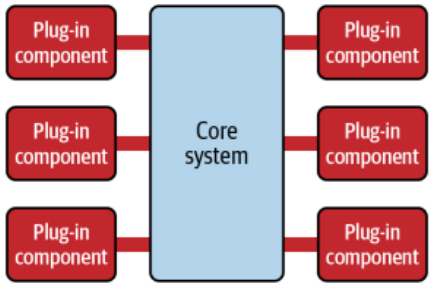
- Une unité de déploiement (*deployment unit*) est un ensemble de composants logiciels qui peuvent être déployés ensemble sur un serveur ou un environnement d'exécution.
- C'est l'élément que l'on installe, démarre, met à jour ou retire dans un système logiciel.
- **Dans une architecture monolithique** :
  - L'unité de déploiement est souvent **l'application entière**.
  - Tous les modules (interface, logique métier, base de données, etc.) sont regroupés dans un seul package.
  - Exemple : un fichier .jar ou .war Java, ou un exécutable unique.
- **Dans une architecture distribuée** :
  - Chaque service ou composant peut être une **unité de déploiement indépendante**.
  - On peut déployer, mettre à jour ou redémarrer un service sans affecter les autres.
  - Exemple : un microservice exécuté dans un conteneur Docker ou sur un serveur cloud.

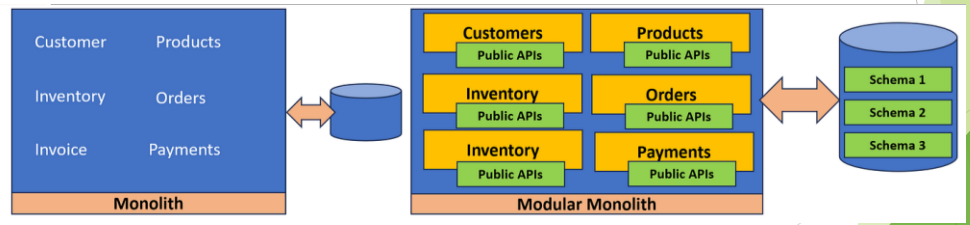
31

Architectures monolithiques

- Beaucoup plus simples que les architectures distribuées et, de ce fait, plus faciles à concevoir et à mettre en œuvre.
- Constituées d'une seule unité de déploiement, sont relativement peu coûteuses dans l'ensemble.
- La plupart des applications conçues selon un style d'architecture monolithique peuvent être développées et déployées beaucoup plus rapidement que les applications distribuées.
- Des exemples de styles d'architecture monolithique incluent:
  - l'architecture en couches
  - le monolithe modulaire: application **déployée en une seule unité** (un seul .jar ou .war) mais le code est structuré en modules séparés
  - l'architecture en pipeline
  - l'architecture micro-noyau

32

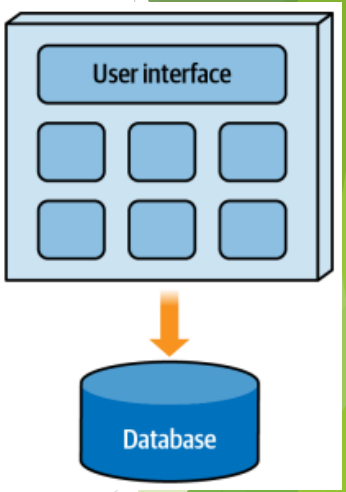




33

Points faibles d'une architecture monolithique

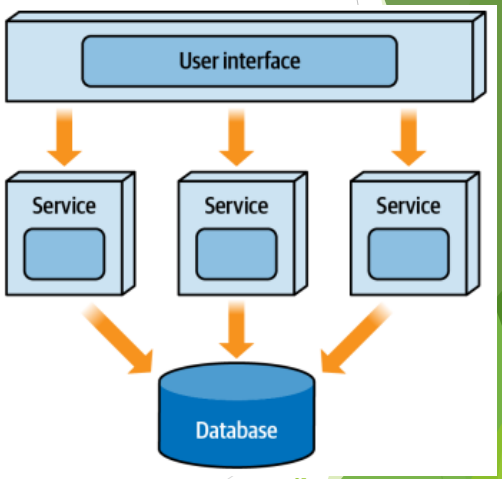
- Scalabilité, tolérance aux pannes et élasticité limitées.
- Une erreur critique (ex. dépassement de mémoire) bloque toute l'application.
- Redémarrage long : MTTR(Mean Time To Recovery) et MTTS (Mean Time to Start) mesurés en minutes.
- Scalabilité inefficace : toute l'application doit s'adapter, même si seule une partie en a besoin, ce qui est coûteux et peu efficace.



34

Architectures distribuées

- Se composent de plusieurs unités de déploiement qui travaillent ensemble pour réaliser une fonction métier cohérente.
- Aujourd'hui, la plupart des architectures distribuées sont basées sur des services, bien que chaque style d'architecture distribuée ait son propre nom spécifique pour un service.



35

Scalabilité et Elasticité

- Scalabilité (scalability): capacité d'un système à supporter une augmentation de charge (plus d'utilisateurs, plus de requêtes, plus de données) sans dégrader ses performances.
- Est-ce que mon système peut grandir ?
- Deux types principaux :
  - Scalabilité verticale (scale up): on augmente la puissance d'une machine (CPU, RAM).
  - Scalabilité horizontale (scale out): on ajoute plusieurs instances (serveurs ou conteneurs).
- Elasticité (elasticity): capacité d'un système à s'adapter automatiquement aux variations de charge, en augmentant ou réduisant les ressources en temps réel.
- Est-ce que mon système peut s'agrandir ET se réduire automatiquement selon le besoin ?
- Exemple :
  - À 10h : forte charge → le système ajoute 5 instances.
  - À 23h : faible charge → le système supprime 4 instances.

36

Architectures distribuées

- Les super-pouvoirs des architectures distribuées relèvent généralement des **caractéristiques opérationnelles** — des aspects tels que la **scalabilité**, l'**élasticité**, la **tolérance aux pannes**, et dans certains cas, la **performance**.
- Dans ces styles d'architecture, la scalabilité s'effectue généralement **au niveau de chaque service individuel**, tout comme l'élasticité : le **MTTS (Mean Time To Start)** et le **MTTR (Mean Time To Recover)** sont beaucoup plus courts que dans une application monolithique (ils se mesurent généralement en **secondes** et dans certains cas en **millisecondes** plutôt qu'en minutes).
- Dans ces styles d'architecture, la scalabilité s'effectue généralement **au niveau de chaque service individuel**, tout comme l'élasticité.

37

Architectures distribuées

- Bien adaptées pour offrir **des niveaux élevés de tolérance aux pannes**.
- Si un service tombe en panne, les autres services peuvent continuer à traiter les requêtes **comme si aucune défaillance ne s'était produite**.
- Les services qui échouent peuvent se rétablir très rapidement — parfois si rapidement qu'un utilisateur final ne se rend même pas compte qu'une erreur critique s'est produite.
- L'**agilité** (la capacité à réagir rapidement au changement) constitue souvent un autre avantage des architectures distribuées.
- Étant donné que les fonctionnalités de l'application sont réparties en **unités logicielles déployées séparément**, il est plus facile de localiser et d'appliquer une modification.
- Le périmètre des tests est limité uniquement au service concerné, et le risque lié au déploiement est considérablement réduit, puisque seul le service impacté est généralement déployé.

38

Architectures distribuées

- Les architectures distribuées sont confrontées à ce que l'on appelle les **fallacies of distributed computing** (les erreurs ou illusions du calcul distribué), un ensemble de huit idées que l'on croit vraies à propos des réseaux et du calcul distribué, mais qui sont en réalité fausses.
- Des affirmations comme « *le réseau est fiable* », « *la bande passante est infinie* » ou encore « *la latence est nulle* » rendent les architectures distribuées non seulement difficiles à maintenir déterministes, mais aussi difficiles à rendre totalement fiables.
- Les réseaux tombent en panne, la bande passante n'est pas infinie et la latence n'est pas nulle. Ces réalités sont toujours d'actualité aujourd'hui, tout comme elles l'étaient à la fin des années 1990, lorsque ces principes ont été formulés.

39

Architectures distribuées

- Les **transactions distribuées**, la **cohérence éventuelle**, la **gestion des workflows**, la **gestion des erreurs**, la **synchronisation des données**, la **gestion des contrats**, ainsi qu'une multitude d'autres complexités font partie intégrante du monde des architectures distribuées.
- Toute cette complexité entraîne généralement des **coûts beaucoup plus élevés**, tant au niveau de l'implémentation initiale que de la maintenance continue, comparativement aux architectures monolithiques.
- Tous les avantages semblent moins impressionnants lorsqu'on prend en compte l'ensemble des compromis liés aux architectures distribuées.

40

Architectures distribuées

- Parmi les exemples d'architectures distribuées, on trouve:
  - l'architecture orientée événements,
  - la très populaire architecture microservices,
  - l'architecture orientée services (SOA)
  - l'architecture basée sur l'espace (space-based architecture).

41

Architecture monolithique ou distribuée ?

- Lors du choix entre une architecture **monolithique** et une architecture **distribuée**, la première question à se poser est de savoir si le système que vous créez présente **différents ensembles de caractéristiques architecturales** qui doivent être prises en charge.
- L'ensemble du système doit-il monter en charge et garantir une haute disponibilité, ou seulement certaines parties ?
- Les systèmes qui comportent plusieurs ensembles de caractéristiques architecturales différentes nécessitent généralement une architecture distribuée.
- Un bon exemple est celui d'une fonctionnalité orientée client qui exige la **scalabilité**, la **réactivité**, la **disponibilité** et l'**agilité**, alors qu'une fonctionnalité administrative ou de traitement en arrière-plan (backend) ne nécessite aucune de ces caractéristiques.

42

Architecture monolithique ou distribuée ?

- Les systèmes ou sites web simples justifient généralement l'adoption d'un style d'architecture monolithique, plus simple et plus économique, tandis que les systèmes plus complexes, qui exécutent plusieurs fonctions métier, nécessitent généralement des architectures distribuées plus sophistiquées.
- De même, le besoin de rapidité, le besoin d'une grande scalabilité et le besoin d'une haute tolérance aux pannes sont autant de caractéristiques qui orientent naturellement vers des architectures distribuées.

43

Partitionnement architectural

- En plus d'être classées comme monolithiques ou distribuées, les architectures peuvent également être classées selon la manière dont la structure globale du système est partitionnée.
- Les architectures, qu'elles soient monolithiques ou distribuées, peuvent être partitionnées **techniquement** ou partitionnées **par domaine**.

44

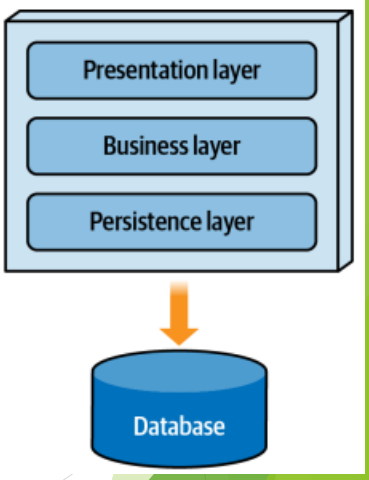
Partitionnement technique

- Les architectures techniquement partitionnées ont leurs composants organisés selon leur usage technique.
- L'exemple classique d'une architecture techniquement partitionnée est le style d'architecture en couches (n-tier).
- Dans ce style d'architecture, les composants sont organisés par couches techniques :
  - les composants de présentation, liés à l'interface utilisateur ;
  - les composants de la couche métier, liés aux règles métier et au traitement central ;
  - les composants de la couche de persistance, qui interagissent avec la base de données ;
  - la couche base de données, qui contient les données du système.

45

Partitionnement technique

- Les composants d'un domaine donné sont répartis sur toutes ces couches techniques.
- Par exemple, la fonctionnalité du domaine client se trouve :
  - dans la **couche de présentation** sous forme d'écrans clients ;
  - dans la **couche métier** sous forme de logique client ;
  - dans la **couche de persistance** sous forme de requêtes clients ;
  - et dans la **couche base de données** sous forme de tables clients.
- Lorsqu'ils sont représentés sous forme de **namespaces**, ces composants seraient organisés ainsi : *app.presentation.customer, app.business.customer, app.persistence.customer,* etc.
- Le deuxième nœud du namespace précise la couche technique, tandis que le nœud customer est réparti sur toutes ces couches.



46

Partitionnement technique

- Les architectures techniquement partitionnées sont utiles si la majorité des modifications se limitent à une zone technique spécifique de l'application.
- Exemple: si vous modifiez constamment l'apparence et l'ergonomie de l'interface utilisateur sans changer les règles métier correspondantes, la modification est limitée à une seule partie de l'architecture (la couche de présentation).
- De même, si vos règles métier changent fréquemment mais n'ont aucun impact sur la couche de données ou la couche de présentation, les modifications se limitent à la couche métier de l'architecture, sans affecter les autres parties du système.

47

Partitionnement technique

- Cependant, si vous deviez implémenter une nouvelle exigence consistant à ajouter une date d'expiration aux éléments des listes de souhaits des clients dans une architecture techniquement partitionnée.
- Ce type de modification est considéré comme une modification basée sur le domaine (et non sur l'usage technique): impacte toutes les couches de l'architecture.
- Pour réaliser ce changement, il faudrait :
  - ajouter une nouvelle colonne à la table des listes de souhaits dans la couche base de données ;
  - modifier le SQL correspondant dans la couche de persistance ;
  - ajouter les règles métier correspondantes dans les composants de la couche métier ;
  - modifier les contrats entre la couche métier et la couche de présentation ;
  - modifier les écrans dans la couche de présentation.
- Selon la taille du système et la structure de l'équipe, cette simple modification pourrait nécessiter la coordination de trois à quatre équipes différentes.

48

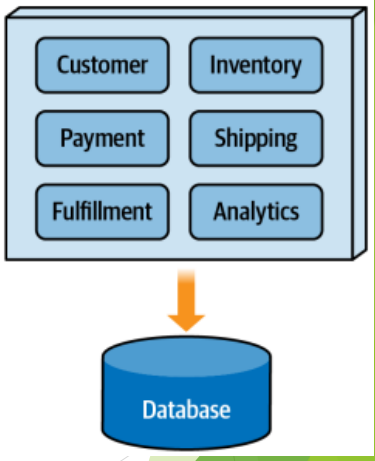
Partitionnement technique

- Parmi les exemples d'architectures techniquement partitionnées, on trouve:
  - l'architecture en couches,
  - l'architecture micro-noyau,
  - l'architecture pipeline,
  - l'architecture orientée événements
  - l'architecture basée sur l'espace
- L'architecture micro-noyau est particulièrement intéressante: le seul style d'architecture qui peut être soit techniquement partitionné, soit partitionné par domaine, selon la manière dont les composants plug-in sont utilisés.
  - Lorsque les composants plug-in sont utilisés comme adaptateurs ou paramètres de configuration spécifiques, l'architecture est considérée comme techniquement partitionnée.

49

Partitionnement par domaine

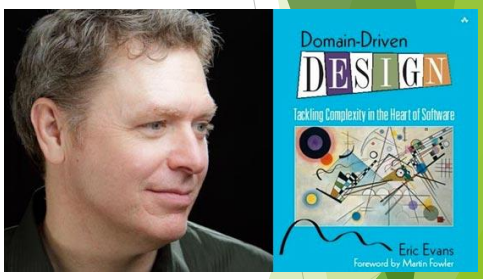
- Contrairement aux architectures techniquement partitionnées, les composants des architectures partitionnées par domaine sont organisés selon les domaines fonctionnels, et non selon l'usage technique.
- Cela signifie que toute la fonctionnalité (présentation, logique métier et logique de persistance) est regroupée pour chaque domaine et sous-domaine dans des zones séparées de l'application.
- Pour les architectures partitionnées par domaine, les composants peuvent être représentés via une structure de namespace telle que :
  - *app.customer, app.shipping, app.payment*, etc.
  - Le deuxième nœud représente le domaine plutôt qu'une couche technique.



50

Partitionnement par domaine

- Les architectures partitionnées par domaine ont gagné en popularité au fil des années, en partie grâce à l'utilisation accrue et à l'acceptation du Domain-Driven Design (DDD), une technique de modélisation et d'analyse logicielle introduite par Eric Evans.
- Le Domain-Driven Design met l'accent sur la conception d'un domaine plutôt que sur des workflows complexes ou des composants techniques.
- Cette approche permet aux équipes de collaborer étroitement avec les experts du domaine et de se concentrer sur une partie clé du système, développant ainsi un logiciel qui reflète fidèlement la fonctionnalité du domaine.



51

Partitionnement par domaine

- Principal avantage: les modifications apportées à un domaine ou sous-domaine particulier sont autonomes et limitées à une zone spécifique du système, permettant ainsi aux équipes de cibler précisément la partie du système nécessitant le changement.
- Dans l'exemple de l'implémentation d'une date d'expiration pour les éléments de la liste de souhaits d'un client : avec le partitionnement par domaine, les modifications dans le code sont limitées à une seule petite partie du système, rendant ce type de changement beaucoup plus efficace que dans le cas du partitionnement technique.
- Ici, par exemple, les changements seraient isolés dans le namespace *app.customer.wishlist*, ce qui signifie que la logique de présentation, la logique métier et la logique de persistance se trouvent toutes dans la même zone du système.
- La maintenance devient plus facile, les tests sont simplifiés, et le déploiement est beaucoup moins risqué lorsque ce type de modification est réalisé.

52

Partitionnement par domaine

- Parmi les exemples d'architectures partitionnées par domaine:
  - l'architecture micro-noyau,
  - l'architecture microservices,
  - l'architecture monolithique modulaire
  - l'architecture basée sur les services.
- Comme indiqué précédemment, l'architecture micro-noyau peut être soit techniquement partitionnée, soit partitionnée par domaine.
- Si les composants plug-in sont utilisés pour étendre l'application en ajoutant des fonctionnalités, alors l'architecture est considérée comme partitionnée par domaine.

53

Partitionnement technique ou par domaine?

- La structure globale de l'architecture doit être alignée non seulement avec la structure des équipes, mais aussi avec la nature des types de changements attendus dans le système, afin d'être réussie et efficace.
- Les architectures **partitionnées techniquement** (qu'elles soient monolithiques ou distribuées) sont bien adaptées lorsque la structure globale de l'équipe de développement est organisée selon ces mêmes domaines techniques.
- Par exemple, si les équipes de développement sont organisées en équipes de développeurs d'interfaces utilisateur, de développeurs backend et de développeurs bases de données, les architectures partitionnées techniquement constituent un bon choix, car la structure des équipes correspond aux couches techniques de l'architecture.
- Les architectures partitionnées techniquement sont également un choix naturel lorsque la plupart des changements prévus sont alignés sur des couches techniques (par exemple, plusieurs interfaces utilisateurs, des modifications de l'apparence et de l'ergonomie du système, le remplacement d'une base de données par une autre, etc.).

54

Partitionnement technique ou par domaine?

- Envisager une **architecture partitionnée par domaine si** une approche basée sur le **Domain-Driven Design (DDD) est adoptée**.
- Les architectures partitionnées par domaine sont particulièrement adaptées lorsque les équipes sont organisées en **équipes transversales spécialisées,** càd des équipes uniques alignées sur une fonctionnalité métier spécifique et regroupant des développeurs interface utilisateur (front-end), des développeurs back-end ainsi que des développeurs bases de données au sein d'une même équipe physique
- Les architectures partitionnées par domaine constituent également un bon choix lorsque l'on prévoit que la plupart des modifications seront liées au domaine fonctionnel plutôt qu'à des aspects techniques d'utilisation: meilleure agilité.
- Il faut être prudent lors du choix d'une architecture partitionnée par domaine si vous anticipez de nombreux changements au niveau des couches techniques.
  - Par exemple, remplacer un type de base de données par un autre ou modifier entièrement le framework d'interface utilisateur serait une tâche difficile et chronophage dans une architecture partitionnée par domaine.

55