

Meeting with Company A

JULY

AUGUST 8, 2021 - 16H

// Architecture Logicielle

JUNE 15, 2021 - 15H

ARCHITECTURE LOGICIELLE

Construisez des systèmes logiciels puissants avec une architecture intelligente

TÉRENCE FERUT

Support de cours réalisé pour Ynov © 2024

JUNE 15, 2021 - 15H

MARCH 22, 2021 - 15H



Styles d'Architecture Logicielle

**Explorez les multiples styles
d'architecture pour concevoir des
logiciels innovants et évolutifs**

PLAN DU COURS STYLES D'ARCHITECTURE LOGICIELLE



01

Architecture client-serveur



02

Architecture en couches (Layered Architecture)

03

Model-Vue-Contrôleur et ses Variantes

04

Architecture orientée services (SOA)

PLAN DU COURS STYLES D'ARCHITECTURE LOGICIELLE



05

Architecture microservices



06

Architecture monolithique

07

Clean Architecture

08

Architecture orientée événements

PLAN DU COURS STYLES D'ARCHITECTURE LOGICIELLE



09

Architecture peer-to-peer



APERÇU DU COURS

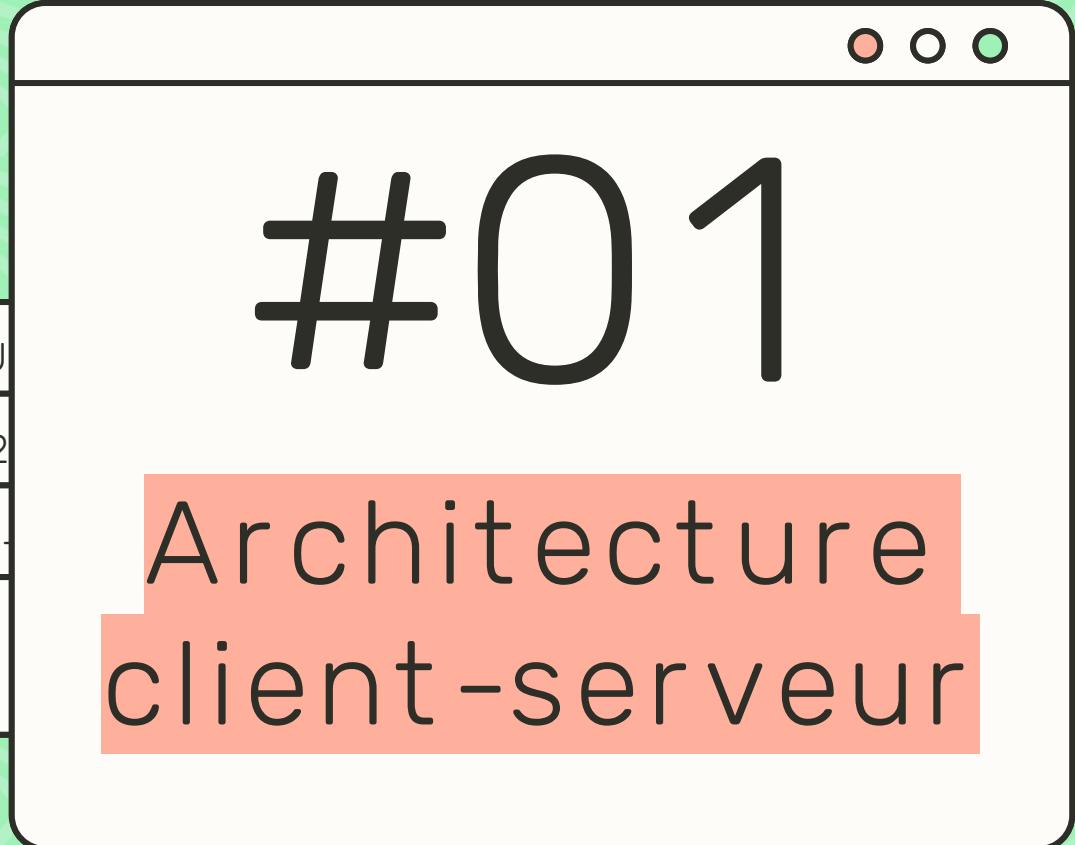


- A travers ce second cours, nous allons explorer les principales familles d'architectures logicielles, ainsi que ce qui les caractérise et fait leurs forces.
- Vous apprendrez à choisir une architecture adaptée à un besoin grâce à de nombreux exemples historiques

#01

Architecture client-serveur

JU
JULY 11, 2021
AUGUST 8, 2021
APRIL 15, 2021 - 15H Meeting with Company A



15, 2021 - 18H

1 - 15H

✗

✗

✗

2.1.1 INTRODUCTION À L'ARCHITECTURE CLIENT-SERVEUR



- L'architecture client-serveur est un modèle fondamental dans le monde du réseau.
- Un serveur fournit des services et des clients consomment ces services.
- *C'est le modèle utilisé pour les interactions sur le web, dans le courrier électronique, les bases de données, et bien d'autres applications.*



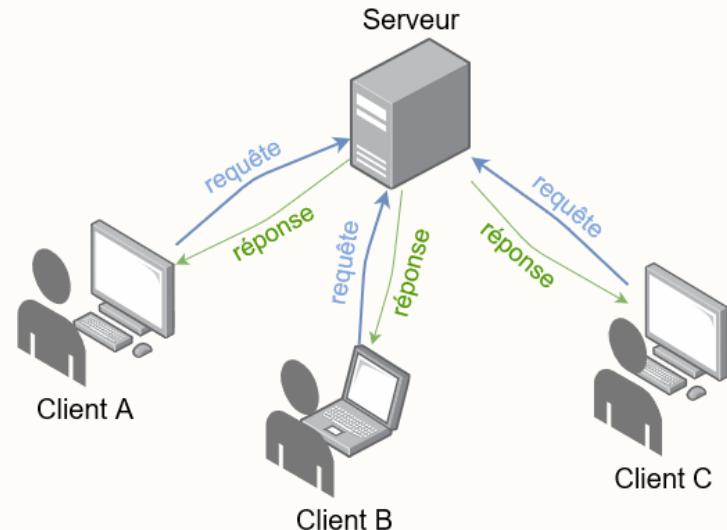
2.1.1 INTRODUCTION À L'ARCHITECTURE CLIENT-SERVEUR



EXEMPLE D'APPLICATION DE L'ARCHITECTURE



- Lorsque vous visitez un site web, votre navigateur (le client) envoie une requête au serveur du site.*
- Le serveur traite la requête et renvoie les informations, qui sont affichées sur votre navigateur.*



2.1.2 CARACTÉRISTIQUES DE L'ARCHITECTURE CLIENT-SERVEUR



- L'architecture client-serveur se caractérise par une **répartition des tâches**.
- Le **serveur** est généralement un **ordinateur puissant** qui héberge **les données** et exécute **les tâches complexes**.
- Les **clients**, qui peuvent être moins puissants, demandent **des services au serveur**.

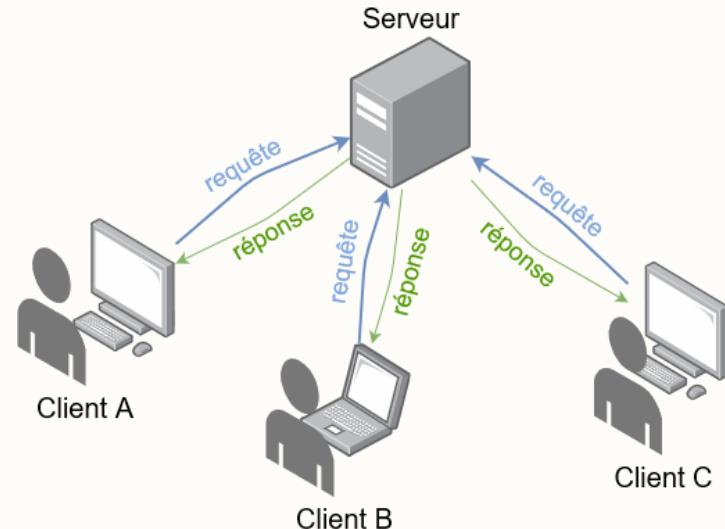


2.1.2 CARACTÉRISTIQUES DE L'ARCHITECTURE CLIENT-SERV~~E~~UR

EXEMPLE D'APPLICATION DE L'ARCHITECTURE



- *Un exemple historique est l'utilisation de terminaux légers dans les années 1970 et 1980.*
- *Ces terminaux, qui étaient des clients, se connectaient à des ordinateurs centraux (serveurs) pour exécuter des programmes et accéder à des données.*



2.1.4 COMMUNICATION ENTRE LE CLIENT ET LE SERVEUR

*

- La **communication** entre le client et le serveur est généralement basée sur la suite de protocoles **TCP/IP**.
- Les clients envoient des **requêtes** et les serveurs répondent.

*

2.1.4 CONCEPTION ET STRUCTURE D'UNE ARCHITECTURE CLIENT-SERVEUR

EXERCICE - RÉFLEXION



- *Comment le protocole TCP facilite-t-il la communication entre le client et le serveur dans une architecture client-serveur ?*
- *Quels problèmes pourrait-il y avoir si un autre protocole (comme UDP) était utilisé ?*



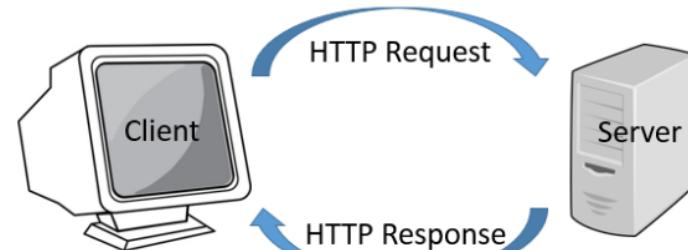
2.1.4 COMMUNICATION ENTRE LE CLIENT ET LE SERVEUR



EXEMPLE D'APPLICATION DE L'ARCHITECTURE



- Dans le protocole HTTP (surcouche de TCP), le client envoie une requête "GET" pour demander un document web, et le serveur envoie une réponse contenant le document HTML.*

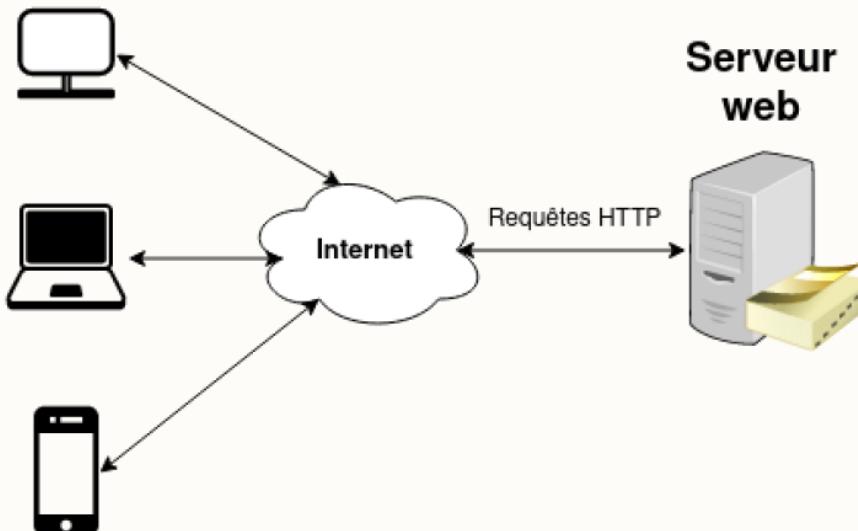


2.1.5 DÉPLOIEMENT ET ÉVOLUTIVITÉ

×

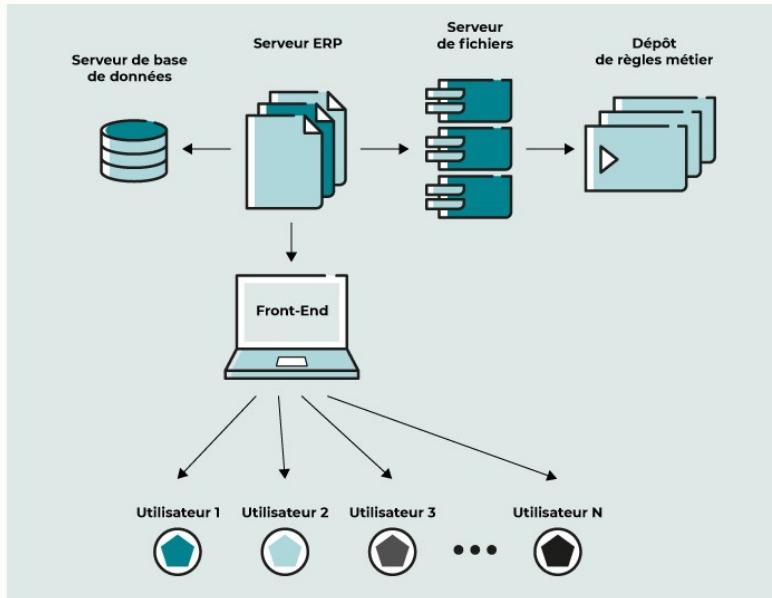
Clients

×



Exemple de diagramme d'architecture

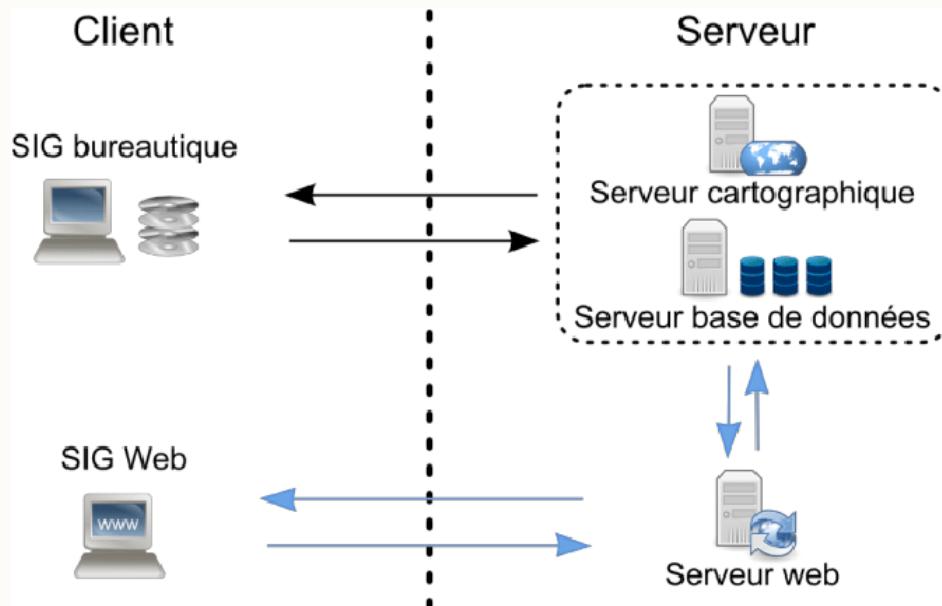
2.1.5 DÉPLOIEMENT ET ÉVOLUTIVITÉ



Exemple de diagramme d'architecture

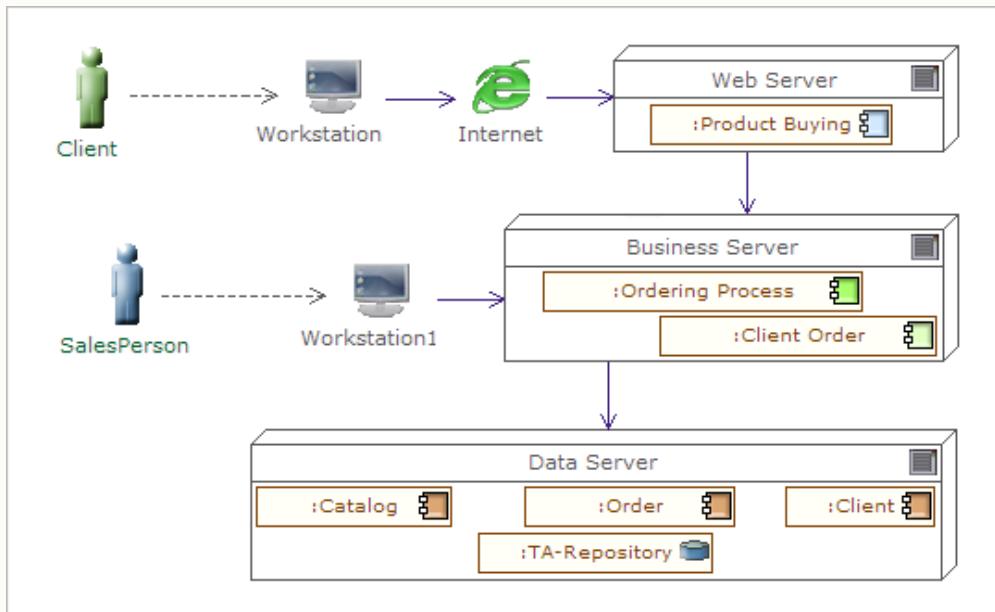
2.1.5 DÉPLOIEMENT ET ÉVOLUTIVITÉ

×



Exemple de diagramme d'architecture

2.1.5 DÉPLOIEMENT ET ÉVOLUTIVITÉ



Exemple de diagramme d'architecture client-serveur

2.1.5 DÉPLOIEMENT ET ÉVOLUTIVITÉ

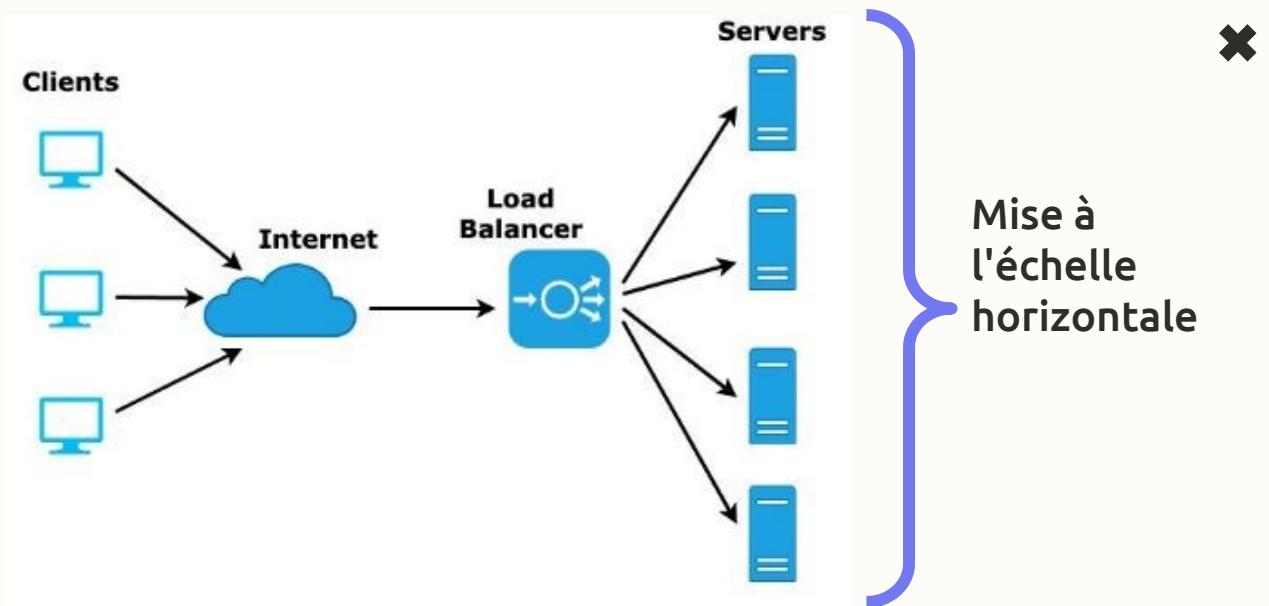


- L'architecture client-serveur permet d'augmenter la capacité du système en ajoutant des serveurs.
- Cela est connu sous le nom de mise à l'échelle horizontale.
- Contrairement à la mise à l'échelle verticale qui correspondrait à augmenter la puissance des serveurs



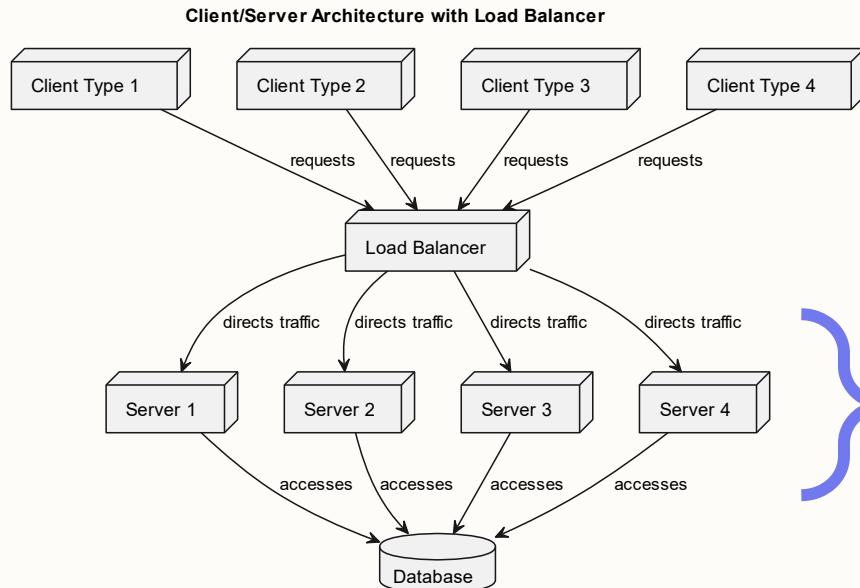
2.1.5 DÉPLOIEMENT ET ÉVOLUTIVITÉ

✗



Exemple de diagramme d'architecture avec répartition de charge

2.1.5 DÉPLOIEMENT ET ÉVOLUTIVITÉ



Mise à l'échelle horizontale

Exemple de diagramme d'architecture avec répartition de charge

2.1.5 DÉPLOIEMENT ET ÉVOLUTIVITÉ



EXEMPLE D'APPLICATION DE L'ARCHITECTURE



- Des entreprises comme Google et Facebook utilisent des milliers de serveurs pour gérer le grand nombre de requêtes des clients.*



2.1.5 DÉPLOIEMENT ET ÉVOLUTIVITÉ



EXEMPLE D'APPLICATION DE L'ARCHITECTURE



- *Exemple d'une application en python :*
<https://github.com/katmfoo/python-client-server>
- *Application en Go-Protobuf :*
<https://github.com/minaandrawos/Go-Protobuf-Examples>
- *En C++ :* <https://github.com/brkho/client-server-webrtc-example>

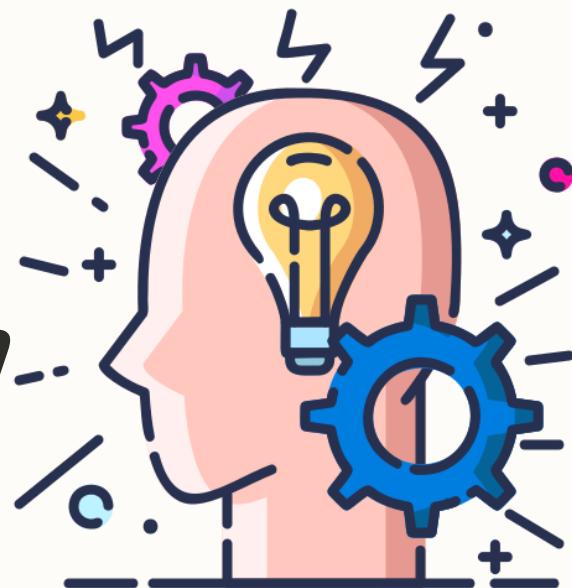
2.1.5 DÉPLOIEMENT ET ÉVOLUTIVITÉ

*

EXERCICE - RÉFLEXION



- Identifiez plusieurs scénarios où une entreprise aurait besoin de mettre à l'échelle son architecture client-serveur ?*



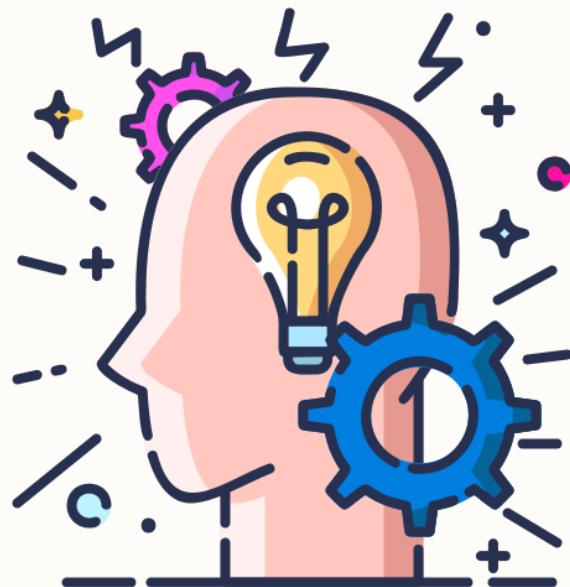
2.1.5 DÉPLOIEMENT ET ÉVOLUTIVITÉ



EXERCICE - RÉFLEXION



- Dans quels cas choisirait-on plutôt une mise à l'échelle horizontale (nombre de serveurs) par rapport à la mise à l'échelle verticale (puissance des serveurs) ?*



2.1.6 GESTION ET MAINTENANCE

*

- La **gestion** de l'architecture client-serveur * peut nécessiter des **administrateurs systèmes** pour **superviser** les serveurs, assurer la **sécurité** et effectuer des **mises à jour**.
- Ce qui nécessite des compétences supplémentaires

2.1.7 AVANTAGES ET INCONVÉNIENTS

*

Centralisation

*

- Dans l'architecture client-serveur, les données et les services sont centralisés sur le serveur.
- *Cela facilite la gestion, le contrôle et la mise à jour des données et des services.*

2.1.7 AVANTAGES ET INCONVÉNIENTS

*

Contrôle d'accès

*

- . Avec un **serveur central**, il est plus facile de **gérer les droits d'accès** aux ressources et de garantir que seules les **personnes autorisées** peuvent accéder aux données pertinentes.

2.1.7 AVANTAGES ET INCONVÉNIENTS

*

Efficacité

*

- Les serveurs sont généralement plus puissants que les clients et sont conçus pour gérer de nombreuses requêtes simultanément.
- Ils peuvent effectuer des tâches complexes plus rapidement et plus efficacement que si ces tâches étaient effectuées sur des clients individuels.

2.1.7 AVANTAGES ET INCONVÉNIENTS

*

Flexibilité

*

- L'architecture client-serveur peut supporter une variété de clients avec différents systèmes d'exploitation et configurations matérielles.
- Les clients peuvent également être mis à niveau ou remplacés sans affecter le fonctionnement du serveur.

2.1.7 AVANTAGES ET INCONVÉNIENTS

*

Scalabilité

*

- L'architecture client-serveur est conçue pour être scalable.
- On peut ajouter de nouveaux clients sans perturber le fonctionnement du serveur.
- De même, de nouveaux serveurs peuvent être ajoutés ou les serveurs existants peuvent être mis à niveau pour gérer une demande accrue.

2.1.7 AVANTAGES ET INCONVÉNIENTS

*

Inconvénients

*

L'architecture client-serveur peut également souffrir, du fait de son architecture, de plusieurs vrais problèmes qu'il faut connaître avant de choisir cette architecture

2.1.7 AVANTAGES ET INCONVÉNIENTS

*

Point de défaillance unique

*

- Dans l'architecture client-serveur, **si le serveur tombe en panne, tous les clients sont affectés.**
- *Cela peut entraîner des interruptions de service et inciter les attaques.*

2.1.7 AVANTAGES ET INCONVÉNIENTS

*

Goulots d'étranglement (*bottleneck*)

*

- Un serveur très sollicité peut devenir un **goulot d'étranglement**, ralentissant les performances du système.
- Les **ressources du serveur** (*bande passante, mémoire, puissance de calcul*) peuvent être saturées par une demande trop importante.

2.1.7 AVANTAGES ET INCONVÉNIENTS

*

Sécurité

*

- Les serveurs centralisent les données et les services, ce qui en fait des cibles attrayantes pour les attaques.
- *Il est donc crucial de maintenir une sécurité robuste.*

2.1.7 AVANTAGES ET INCONVÉNIENTS

*

Coûts de maintenance et de gestion

- La mise en place, la maintenance et la mise à niveau des serveurs nécessitent des ressources et des compétences techniques, ce qui peut engendrer des coûts importants.

*

2.1.7 AVANTAGES ET INCONVÉNIENTS

*

Scalabilité

*

- Bien que les architectures client-serveur puissent être mises à l'échelle pour répondre à une demande accrue, cela peut nécessiter **l'ajout de serveurs supplémentaires ou plus puissants**, ce qui peut être **coûteux et complexe**.

2.1.7 AVANTAGES ET INCONVÉNIENTS

*

Scalabilité

*

- Une **mise à l'échelle** d'une telle architecture sera adaptée à une augmentation globale/permanente de la charge
- *En revanche, pour faire face à des pics rares mais très importants de charge (exemple : mise en vente de PS5 en 2020), ce style architectural peut vite montrer ses limites*

2.1.7 AVANTAGES ET INCONVÉNIENTS



EXERCICE - RÉFLEXION



- *Considérant les **inconvénients** de l'architecture client-serveur, comment les studios de développement de jeux vidéo peuvent-ils garantir une expérience de jeu en ligne fluide et sans interruption pour les joueurs du monde entier ?*



2.1.8 EXEMPLES D'UTILISATION



EXEMPLE D'APPLICATION DE L'ARCHITECTURE



- L'architecture client-serveur est utilisée dans de nombreux domaines, des applications web aux jeux en ligne, en passant par les systèmes de messagerie électronique et les bases de données.*



2.1.8 EXEMPLES D'UTILISATION

*

EXEMPLE D'APPLICATION DE L'ARCHITECTURE



- *Lorsque vous jouez à un jeu en ligne, votre ordinateur (client) se connecte à un serveur de jeu.*
- *Le serveur gère la logique du jeu et envoie les mises à jour à tous les clients connectés.*



2.1.9 EXERCICE DE RÉFLEXION

*

EXEMPLE D'APPLICATION DE L'ARCHITECTURE



- *Un avantage est la centralisation des données, facilitant leur gestion.*
- *Cependant, cela rend le serveur une cible pour les attaques.*



2.1.9 EXERCICE DE RÉFLEXION



EXERCICE - RÉFLEXION



- Quels seraient les avantages et inconvénients d'une architecture client-serveur pour une petite entreprise qui cherche à mettre en place un système de partage de fichiers interne ?*



2.1.9 EXERCICE DE RÉFLEXION



EXERCICE - RÉFLEXION

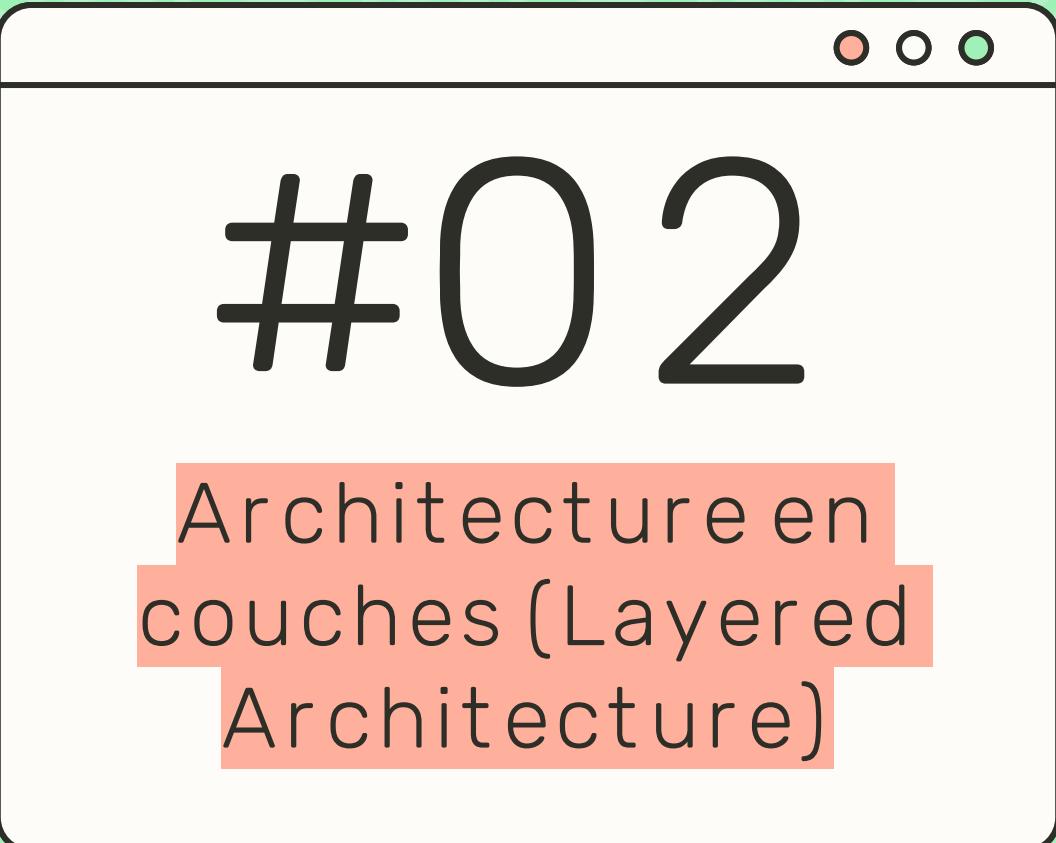


- *Pouvez-vous penser à une situation où l'utilisation d'une architecture client-serveur ne serait pas la meilleure solution ?*
- *Quelle alternative pourrait être plus appropriée dans ce cas ?*



#02

Architecture en couches (Layered Architecture)



15, 2021 - 18H

1 - 15H

✗

✗

✗

2.2.1 INTRODUCTION À L'ARCHITECTURE EN COUCHES

- L'architecture en couches est un style d'architecture logicielle largement utilisé dans le développement d'applications. ✖
- Elle consiste à organiser les différents composants d'un système en couches distinctes, chaque couche ayant des responsabilités spécifiques.
- Cette approche permet de séparer les préoccupations et de faciliter la maintenance, la réutilisabilité et le test du système.

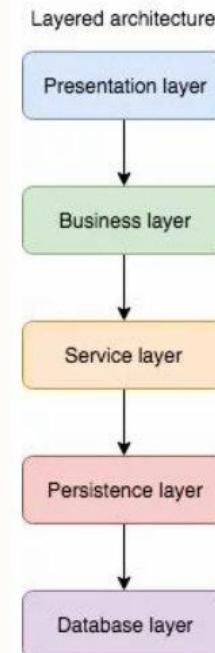
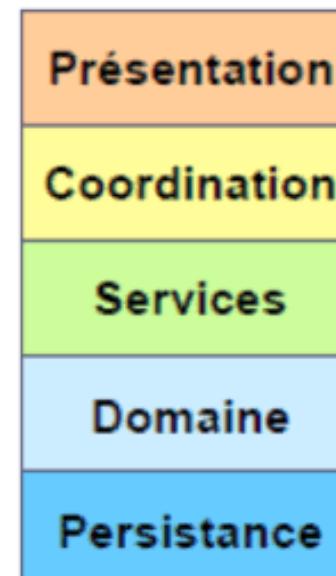
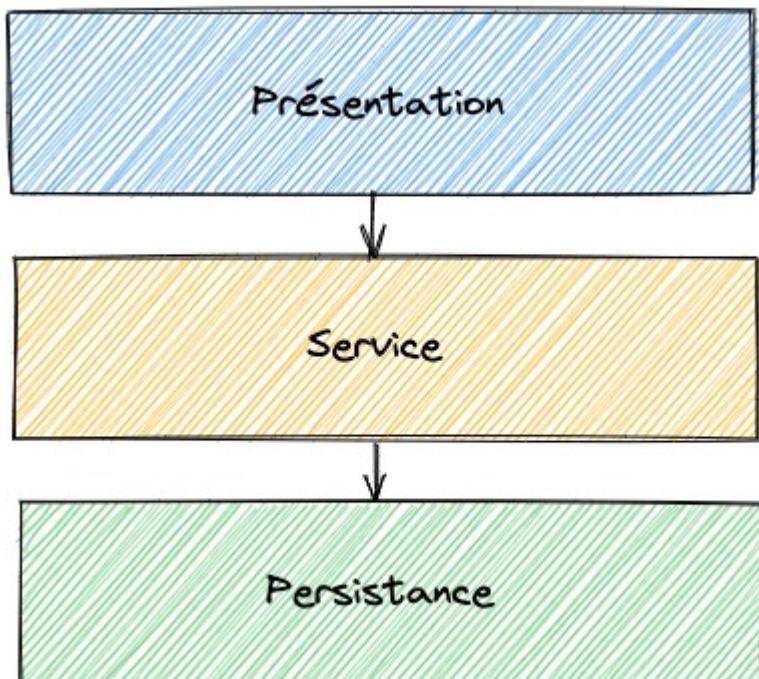
2.2.1 INTRODUCTION À L'ARCHITECTURE EN COUCHES

- L'architecture en couches repose sur le principe de **séparation des responsabilités**.
- Chaque couche est **responsable** d'une partie spécifique du système, ce qui facilite la gestion des fonctionnalités, des règles métier et de l'interaction avec les données.

2.2.1 INTRODUCTION À L'ARCHITECTURE EN COUCHES

- Par exemple, une application en couches peut avoir :
 - ❖ une couche de présentation pour gérer l'interface utilisateur,
 - ❖ une couche métier pour traiter la logique métier
 - ❖ et une couche d'accès aux données pour interagir avec les bases de données.

2.2.1 INTRODUCTION À L'ARCHITECTURE EN COUCHES



2.2.1 INTRODUCTION À L'ARCHITECTURE EN COUCHES

- L'architecture en couches se caractérise par une structure **hiérarchique et modulaire**.
- Chaque couche est indépendante des autres et communique avec les couches adjacentes par le biais d'**interfaces**, d'**API** ou de **services**.
- Cette modularité permet de **développer**, **tester** et **maintenir** chaque couche séparément, ce qui facilite également la **réutilisation** des composants.

2.2.2 COUCHES DE L'ARCHITECTURE EN COUCHES *

- Une architecture en couches typique comprend plusieurs niveaux d'abstraction. *
- Les couches les plus courantes sont :
 - ❖ la couche de présentation (interface utilisateur),
 - ❖ la couche métier (logique métier)
 - ❖ et la couche d'accès aux données.
- Cette liste n'est bien sûr pas exhaustive !

2.2.2 COUCHES DE L'ARCHITECTURE EN COUCHES *

- **En fonction des besoins spécifiques de l'application, d'autres couches peuvent être ajoutées pour traiter des aspects tels que :**
 - ❖ la sécurité,
 - ❖ la gestion des transactions,
 - ❖ La synchronisation,
 - ❖ etc.

2.2.2 COUCHES DE L'ARCHITECTURE EN COUCHES *

- La couche de **présentation** est responsable de l'**interaction** avec l'utilisateur.
- Elle affiche les informations à l'écran, collecte les entrées de l'utilisateur et les transmet à la couche métier pour traitement.

2.2.2 COUCHES DE L'ARCHITECTURE EN COUCHES *

- La couche **métier** contient la logique métier de l'application, c'est-à-dire les règles et les algorithmes qui permettent de traiter les données et de prendre des décisions.
- Enfin, la couche **d'accès aux données** gère l'interaction avec les bases de données ou d'autres systèmes de stockage.

2.2.2 COUCHES DE L'ARCHITECTURE EN COUCHES



EXEMPLE D'APPLICATION DE L'ARCHITECTURE



- *Prenons l'exemple d'une application de gestion des tâches (to-do list) basée sur une architecture en couches.*
- *La couche de présentation serait responsable de l'affichage des tâches à l'utilisateur et de la collecte des nouvelles tâches.*



2.2.2 COUCHES DE L'ARCHITECTURE EN COUCHES *

EXEMPLE D'APPLICATION DE L'ARCHITECTURE



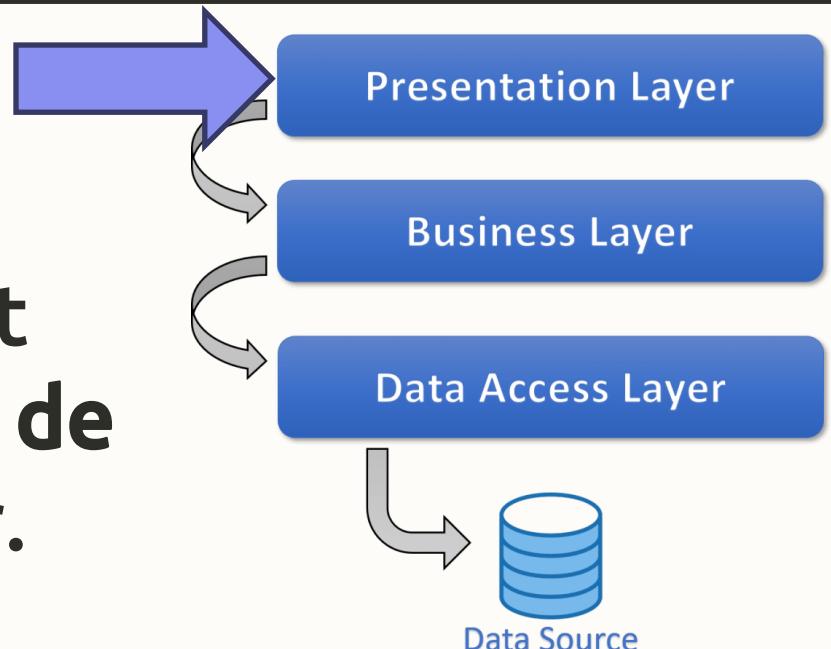
- *Ces tâches seraient ensuite transmises à la couche métier, qui vérifierait leur validité, effectuerait des opérations telles que la suppression ou la mise à jour.*
- *Et les transmettrait à la couche d'accès aux données pour les stocker dans une base de données.*



2.2.3 RESPONSABILITÉS DE CHAQUE COUCHE



- Dans la couche de **présentation**, l'accent est mis sur la gestion de l'interface utilisateur.

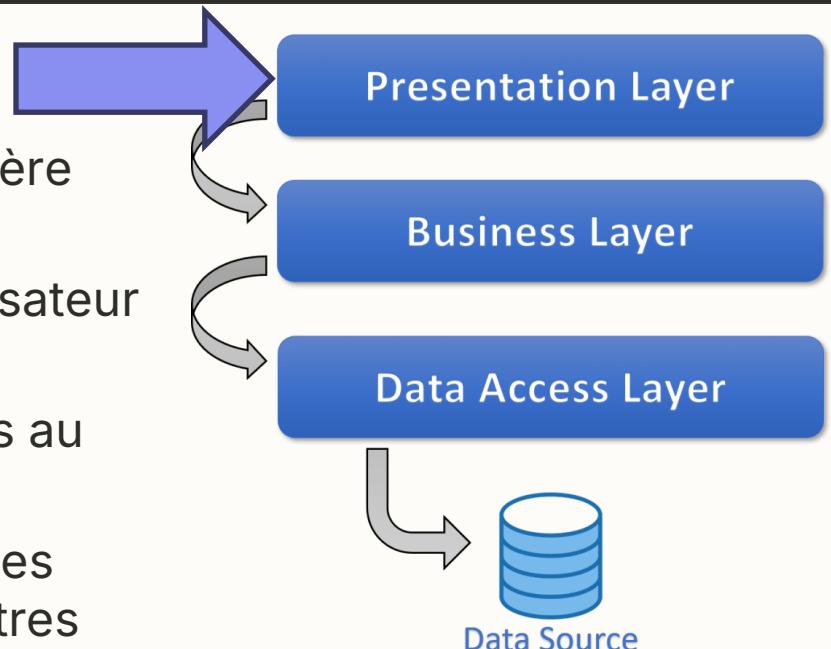


2.2.3 RESPONSABILITÉS DE CHAQUE COUCHE



- Cela comprend :**

- l'affichage des données de manière claire et conviviale,
- la réception des entrées de l'utilisateur (par exemple, des clics de souris ou des saisies au clavier)
- et la présentation des résultats des opérations effectuées par les autres couches.

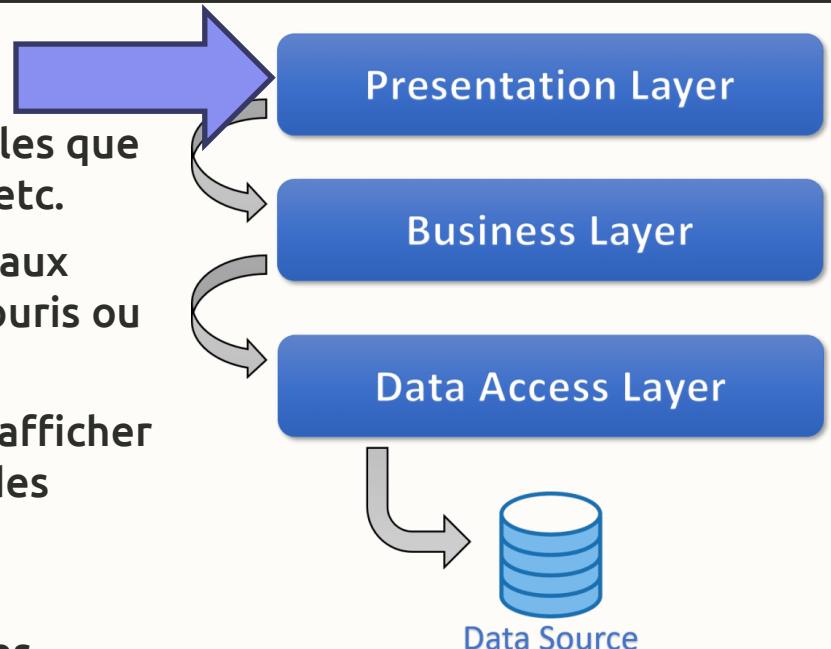


2.2.3 RESPONSABILITÉS DE CHAQUE COUCHE



En termes de code :

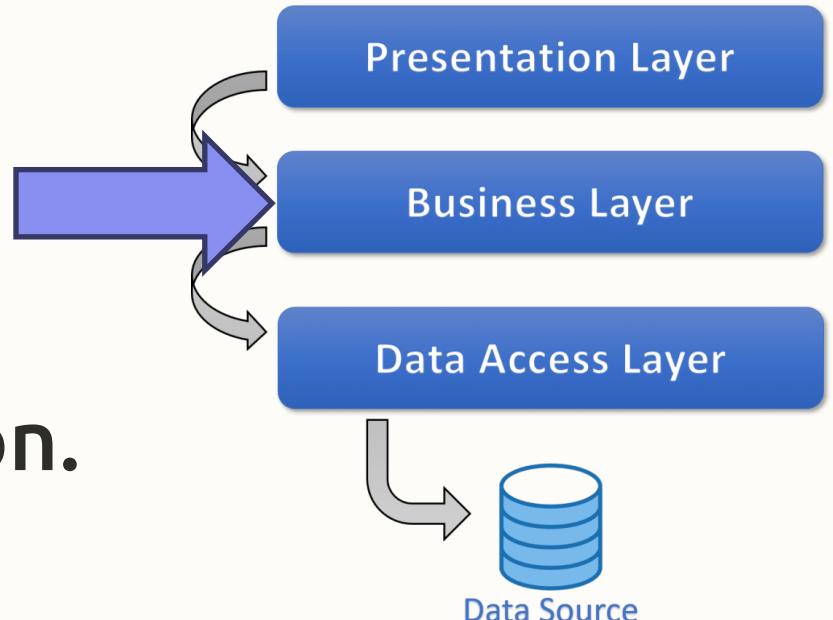
- **Classes** pour gérer l'interface utilisateur, telles que des fenêtres, des formulaires, des boutons, etc.
- **Gestionnaires d'événements** pour répondre aux actions de l'utilisateur, comme les clics de souris ou les saisies au clavier.
- **Interfaces graphiques utilisateur (GUI)** pour afficher des informations à l'utilisateur et collecter des données.
- **Contrôleurs ou façades** qui coordonnent les interactions entre l'interface utilisateur et les couches inférieures.



2.2.3 RESPONSABILITÉS DE CHAQUE COUCHE



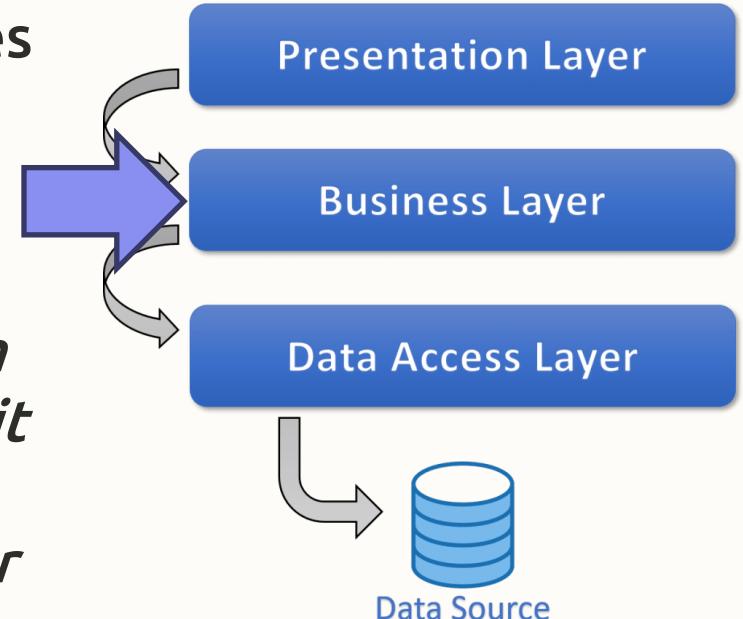
- La couche **métier** contient la logique métier de l'application.



2.2.3 RESPONSABILITÉS DE CHAQUE COUCHE



- Cela implique la mise en œuvre des règles et des algorithmes nécessaires pour traiter les données et prendre des décisions.
- *Par exemple, dans une application bancaire, la couche métier pourrait vérifier les conditions d'éligibilité d'un client pour un prêt et calculer les taux d'intérêt appropriés.*

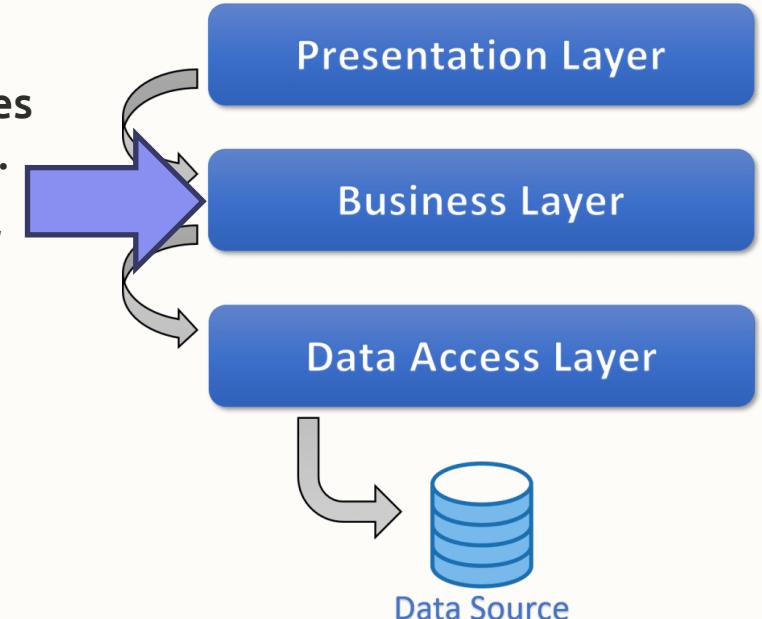


2.2.3 RESPONSABILITÉS DE CHAQUE COUCHE



En termes de code :

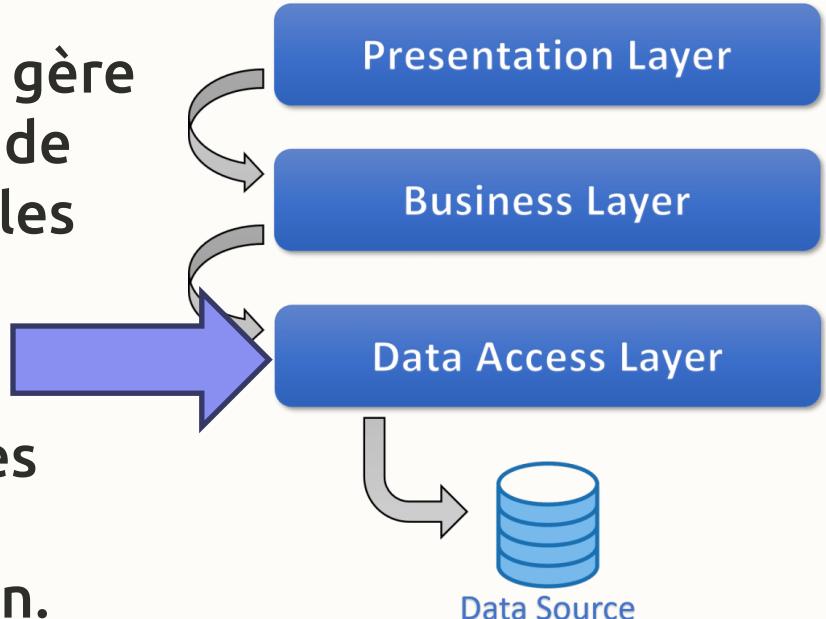
- **Classes** de logique métier qui implémentent les règles et les processus métier de l'application.
- **Modèles de données ou classes d'entités** pour représenter les concepts métier, tels que des utilisateurs, des produits ou des commandes.
- **Services ou gestionnaires de domaine** pour effectuer des opérations complexes sur les données métier.
- **Classes de validation** pour vérifier la validité des données entrantes.



2.2.3 RESPONSABILITÉS DE CHAQUE COUCHE



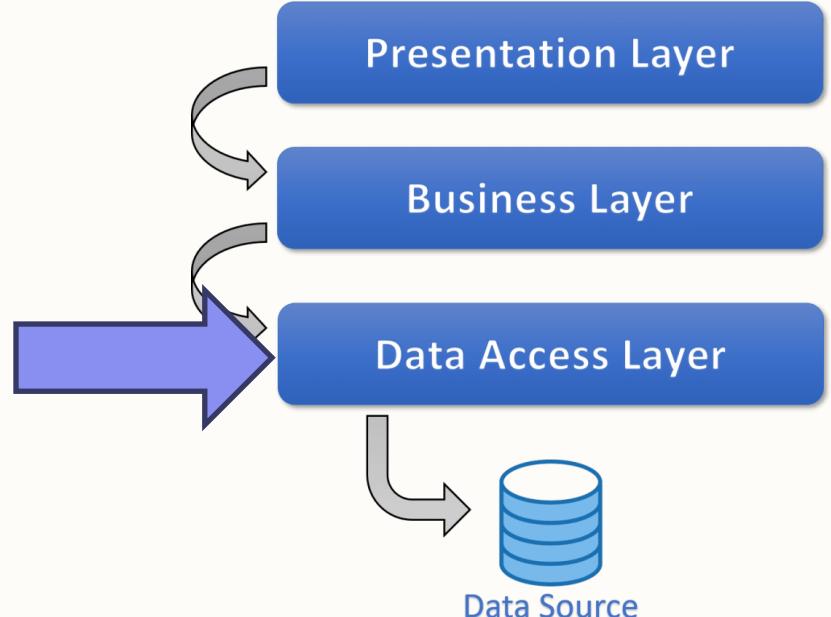
- La couche **d'accès aux données** gère l'interaction avec les systèmes de **stockage de données**, tels que les bases de données.
- Elle est responsable de la **récupération et du stockage** des données nécessaires au fonctionnement de l'application.



2.2.3 RESPONSABILITÉS DE CHAQUE COUCHE



- Par exemple, dans une application de commerce électronique, cette couche serait chargée de récupérer les informations sur les produits à partir de la base de données et de les rendre disponibles aux autres couches.*

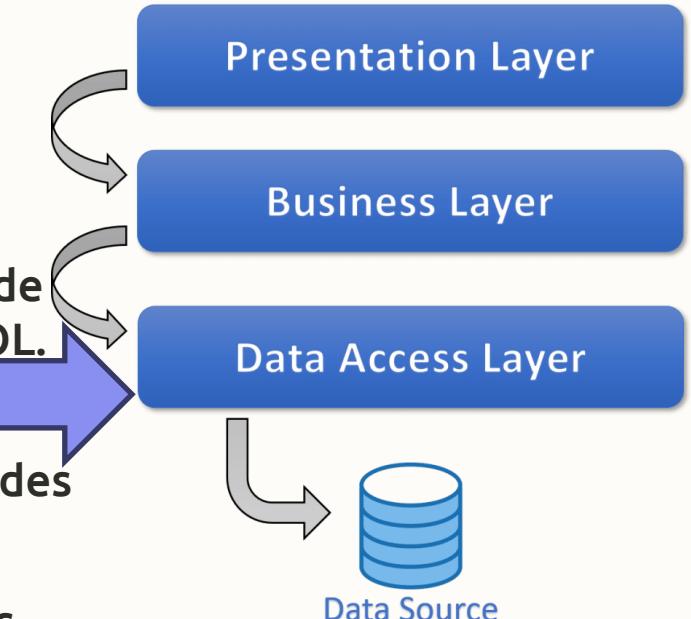


2.2.3 RESPONSABILITÉS DE CHAQUE COUCHE



En termes de code :

- **Classes pour interagir avec les sources de données, comme les bases de données, les fichiers, les services web, etc.**
- **Classes de connexion et de gestion de la base de données, telles que les objets de connexion SQL.**
- **Modèles de données ou classes d'accès aux données pour la récupération et la mise à jour des données.**
- **Mapper de données pour convertir les données entre les formats de stockage et les objets métier.**



2.2.3 RESPONSABILITÉS DE CHAQUE COUCHE

*

EXERCICE - RÉFLEXION



- Identifiez les différentes couches d'une application de banque en ligne et décrivez les responsabilités de chaque couche.*



2.2.3 RESPONSABILITÉS DE CHAQUE COUCHE



EXEMPLE : SYSTÈME DE BANQUE EN LIGNE



Couche de présentation (Interface Utilisateur) :

- **Responsabilités :**

- Fournir une interface utilisateur conviviale pour les clients de la banque en ligne.
- Afficher les comptes, les soldes, les transactions et les fonctionnalités de gestion de compte.
- Recueillir les entrées des utilisateurs, telles que les informations de connexion, les détails de transaction et les demandes de service.
- *Exemple : La page d'accueil du site web de la banque où les clients se connectent, consultent leurs comptes et effectuent des opérations.*



2.2.3 RESPONSABILITÉS DE CHAQUE COUCHE



EXEMPLE : SYSTÈME DE BANQUE EN LIGNE



Couche de Logique Métier (Couche Métier) :

- **Responsabilités :**
 - Gérer la logique métier de l'application bancaire, notamment les règles de traitement des transactions, les calculs de solde, les vérifications de sécurité, etc.
 - Autoriser ou refuser les demandes de transaction en fonction de divers critères (par exemple, solde disponible, plafonds de retrait, etc.).
 - Assurer la sécurité des transactions et la protection des données des clients.
- *Exemple : Les classes et les composants qui gèrent les opérations de dépôt, de retrait, de virement, de paiement de factures, etc.*



2.2.3 RESPONSABILITÉS DE CHAQUE COUCHE



EXEMPLE : SYSTÈME DE BANQUE EN LIGNE



- Couche d'Accès aux Données (Couche d'Accès aux Données) :
- Responsabilités :
 - Interagir avec les systèmes de stockage de données, tels que les bases de données relationnelles, pour récupérer et mettre à jour les informations sur les comptes, les transactions, les utilisateurs, etc.
 - Gérer les opérations de lecture, d'écriture et de modification de données en garantissant la cohérence et l'intégrité des données.
 - Gérer la persistance des données pour les transactions, les journaux, etc.
- ***Exemple : Les classes et les composants qui exécutent des requêtes SQL pour récupérer des informations sur les comptes et enregistrer des transactions.***



2.2.3 RESPONSABILITÉS DE CHAQUE COUCHE

*

EXEMPLE : SYSTÈME DE BANQUE EN LIGNE



Couche de Sécurité :

- **Responsabilités :**
 - Garantir la sécurité des transactions et des données des clients, en incluant l'authentification et l'autorisation.
 - Surveiller les activités suspectes ou non autorisées et déclencher des alertes en cas de violation de sécurité.
 - Gérer l'infrastructure de sécurité, telle que les pare-feu, les mécanismes de chiffrement, les protocoles d'authentification, etc.
- ***Exemple : Les mécanismes d'authentification par mot de passe, les protocoles de cryptage pour sécuriser les communications, et les règles de gestion des mots de passe.***



2.2.4 COMMUNICATION ENTRE LES COUCHES

*

- La communication entre les couches d'une architecture en couches peut être réalisée à l'aide de mécanismes tels que les **interfaces**, les **services** ou les **API** (*Application Programming Interfaces*).

*

2.2.4 COMMUNICATION ENTRE LES COUCHES

*

- Les couches supérieures communiquent avec les couches inférieures en utilisant ces mécanismes pour transmettre des **données** et des **instructions**.

*

2.2.4 COMMUNICATION ENTRE LES COUCHES

*

- Les **interfaces**, les **services** ou les **API** permettent d'établir des **interactions contrôlées** entre les couches.
- Ils définissent les **méthodes** et les **protocoles** à suivre pour échanger des informations.

*

2.2.4 COMMUNICATION ENTRE LES COUCHES



- *Par exemple, une interface entre la couche de présentation et la couche métier pourrait définir une méthode "soumettreTache()" qui permet à la couche de présentation de transmettre une nouvelle tâche à la couche métier.*



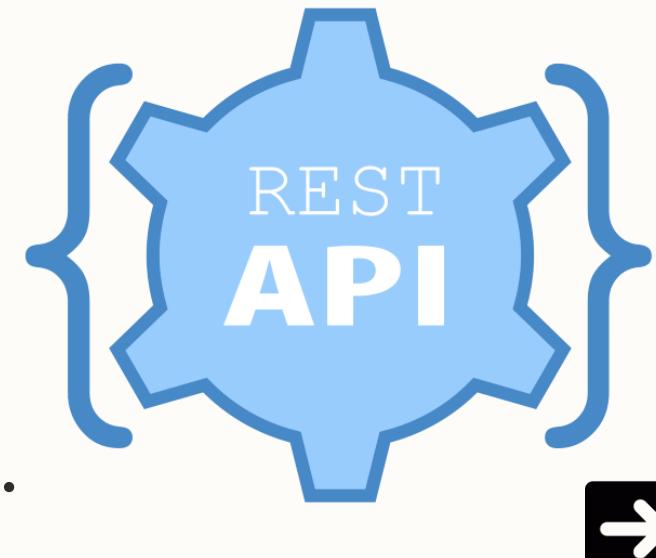
2.2.4 COMMUNICATION ENTRE LES COUCHES



EXEMPLE : PASSAGE DE DONNÉES VIA API



- *Dans une application web, le passage de données entre la couche de présentation et la couche métier peut se faire à l'aide d'une API REST (REpresentational State Transfer).*



2.2.4 COMMUNICATION ENTRE LES COUCHES



EXEMPLE : PASSAGE DE DONNÉES VIA API



- *La couche de présentation enverrait une requête HTTP contenant les données nécessaires à la couche métier, qui les traiterait et renverrait une réponse appropriée.*



2.2.4 COMMUNICATION ENTRE LES COUCHES



EXEMPLE : PASSAGE DE DONNÉES VIA API

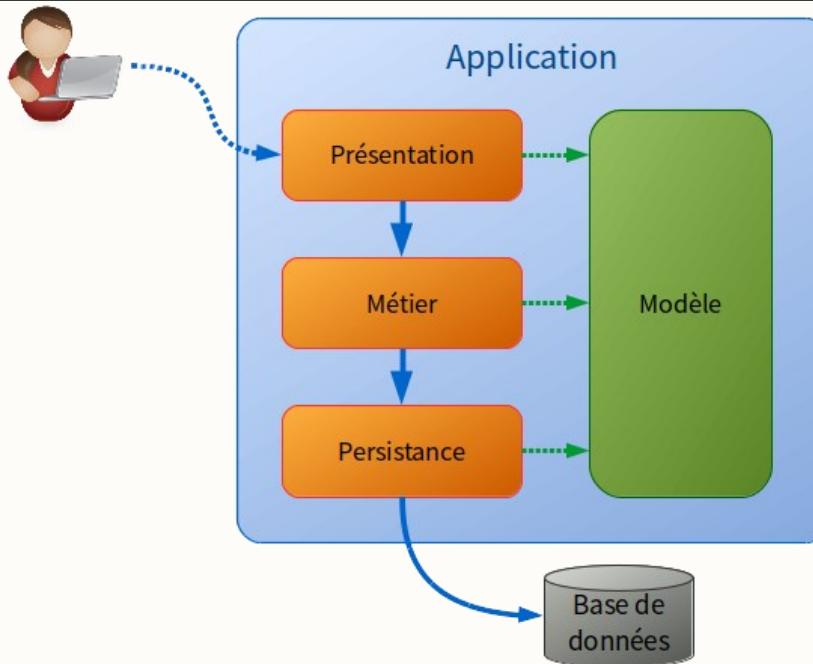


- Par exemple, une requête **POST** contenant les détails d'une nouvelle tâche serait envoyée à une URL spécifique, et la couche métier récupérerait ces données (en faisant elle-même appel à la couche d'accès aux données) pour les traiter.*



2.2.4 COMMUNICATION ENTRE LES COUCHES

✗

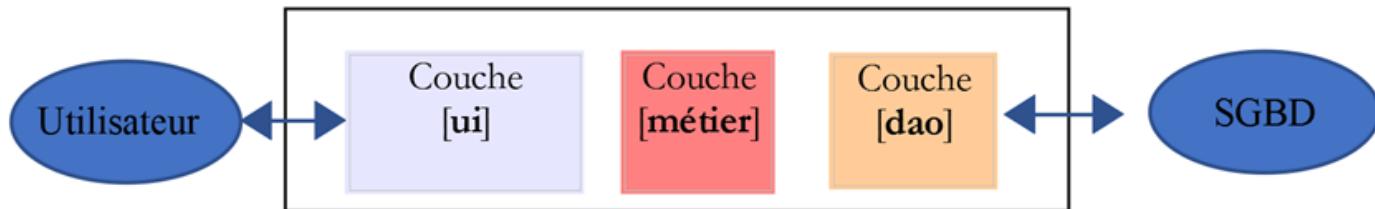


✗

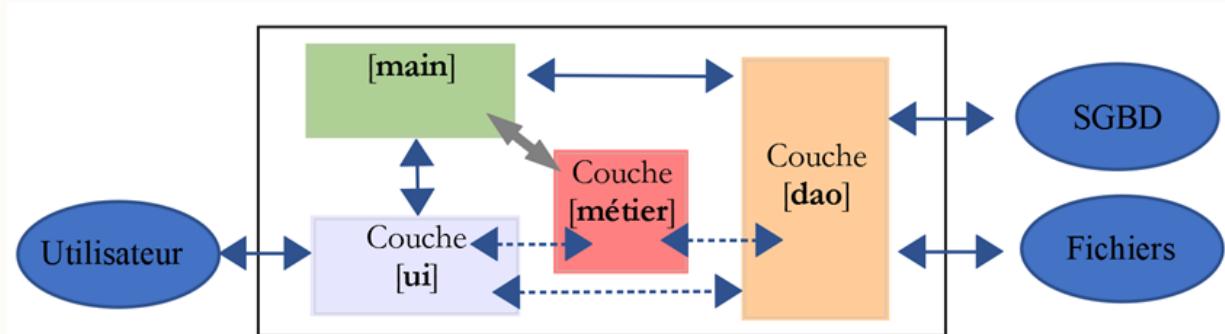
Diagramme d'architecture en couches

2.2.4 COMMUNICATION ENTRE LES COUCHES

✗



✗



Diagrammes d'architectures en couches

2.2.4 COMMUNICATION ENTRE LES COUCHES

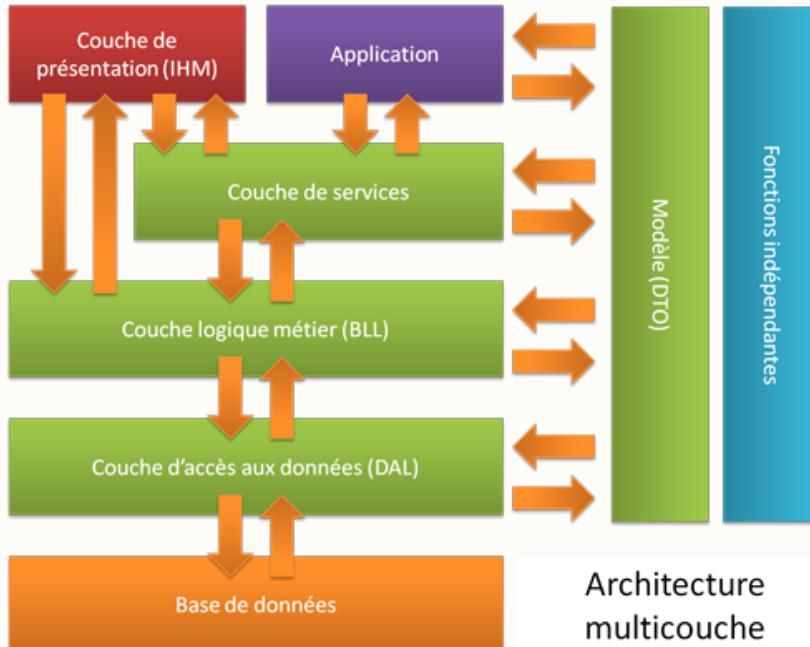


Diagramme d'architecture en couches

2.2.4 COMMUNICATION ENTRE LES COUCHES

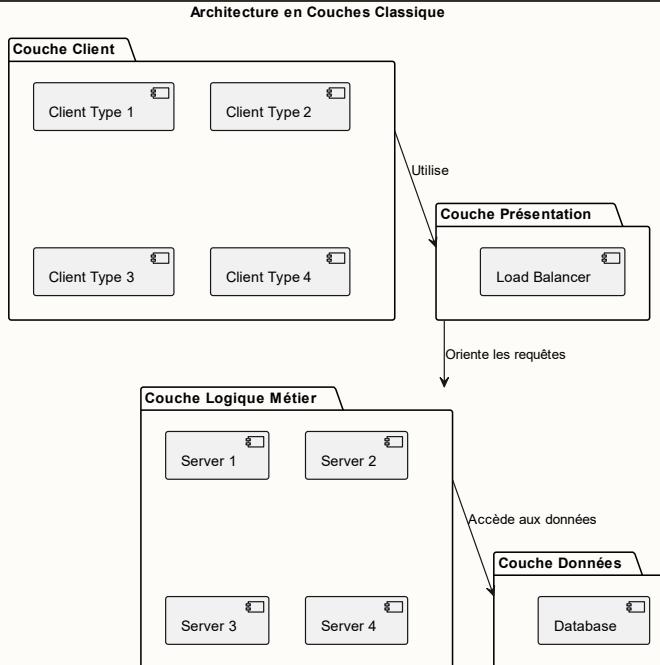


Diagramme d'architecture en couches

2.2.4 COMMUNICATION ENTRE LES COUCHES

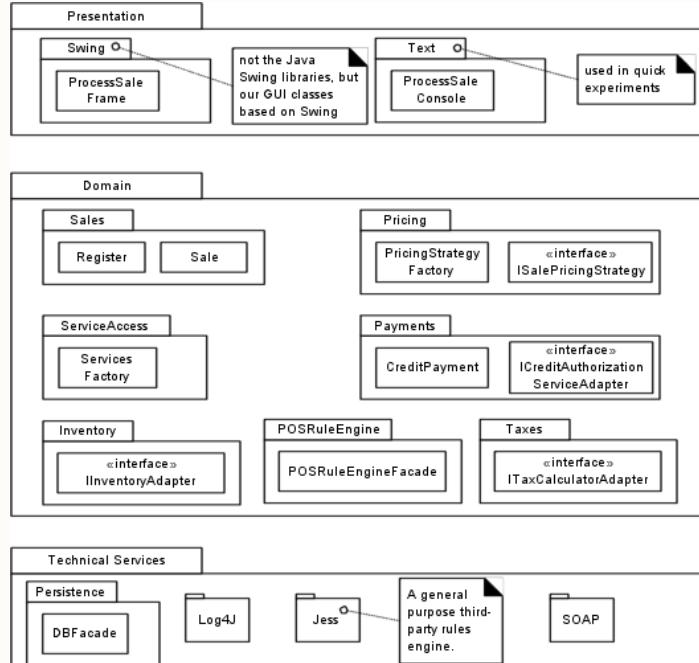


Diagramme d'architecture en couches

2.2.4 COMMUNICATION ENTRE LES COUCHES

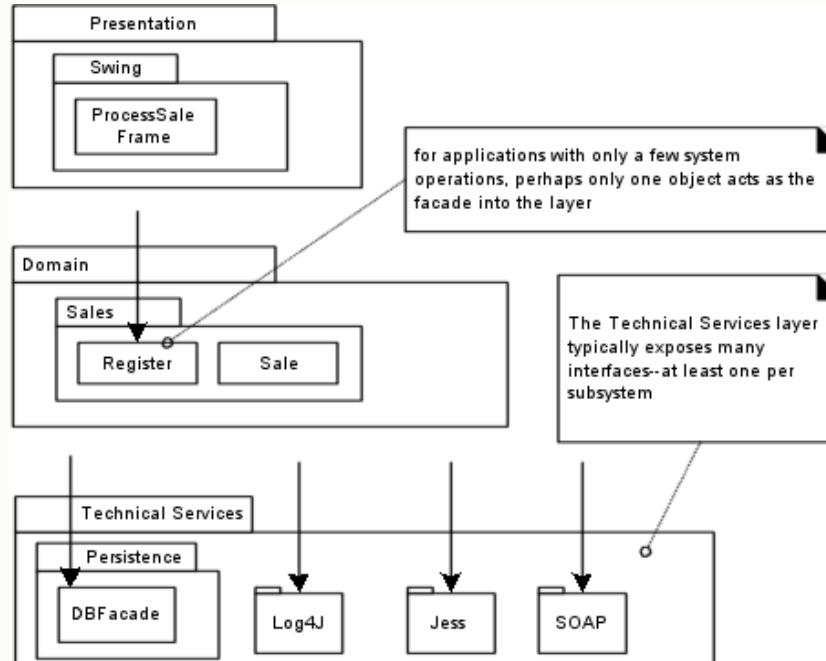
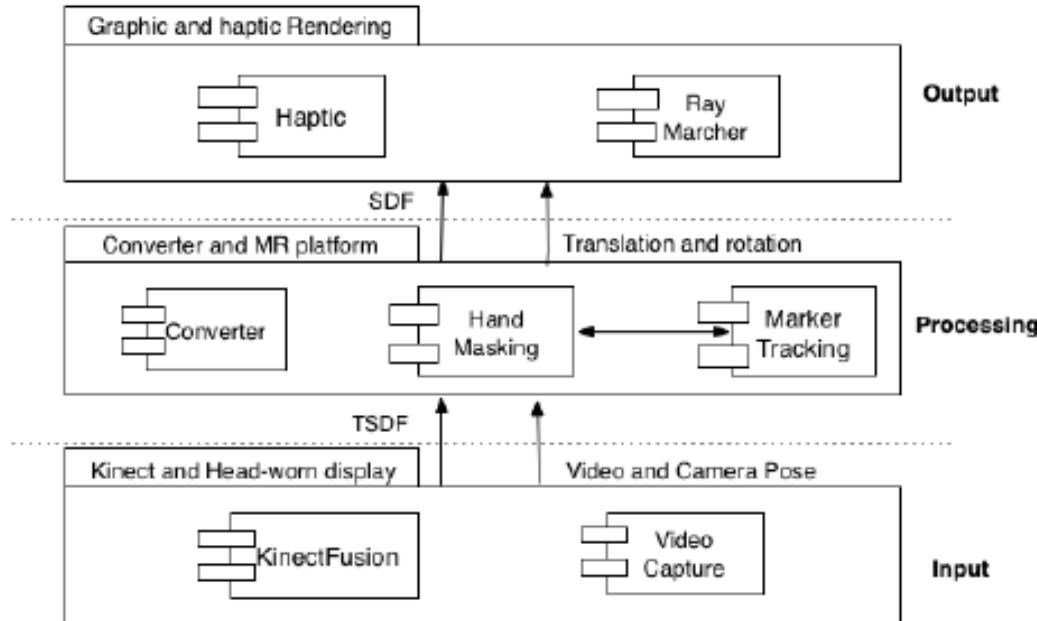


Diagramme d'architecture en couches

2.2.4 COMMUNICATION ENTRE LES COUCHES



2.2.5 AVANTAGES DE L'ARCHITECTURE EN COUCHES

- L'architecture en couches offre plusieurs avantages clés.
- Tout d'abord, la **séparation des responsabilités** facilite la gestion des fonctionnalités et des règles métier, ce qui rend le système plus **modulaire** et facile à **maintenir**.
- De plus, chaque couche peut être **développée et testée de manière indépendante**, ce qui améliore la **réutilisabilité** des composants et facilite les **tests unitaires**

2.2.5 AVANTAGES DE L'ARCHITECTURE EN COUCHES

- La **séparation des préoccupations** permet à chaque couche de se concentrer sur des aspects spécifiques du système, ce qui améliore la **modularité**.
- Les composants développés pour une couche peuvent être **réutilisés** dans d'autres applications, ce qui accélère le processus de développement.
- De plus, la **maintenance** et les **tests** sont simplifiés car chaque couche peut être traitée de manière isolée.

2.2.5 AVANTAGES DE L'ARCHITECTURE EN COUCHES

EXEMPLE : BOUTIQUE DE E-COMMERCE



- *Prenons l'exemple d'une application de commerce électronique basée sur une architecture en couches.*
- *Si une nouvelle fonctionnalité, telle que le paiement par crypto-monnaie, doit être ajoutée, cela peut être réalisé en développant une nouvelle couche dédiée au traitement des transactions en crypto-monnaie.*
- *Cela n'impliquerait pas de modifier les autres couches (ou peu), ce qui faciliterait la maintenance du système et réduirait le risque d'effets indésirables sur les fonctionnalités existantes.*

2.2.6 LIMITATIONS DE L'ARCHITECTURE EN COUCHES

- Malgré ses avantages, l'architecture en couches présente également certaines limitations.
- L'une d'entre elles est la **surcharge des communications intercouches**.
- *Étant donné que les couches doivent communiquer entre elles, il peut y avoir une surcharge de communication si les données échangées sont volumineuses et donc une perte de performances.*



2.2.6 LIMITATIONS DE L'ARCHITECTURE EN COUCHES

- De plus, l'architecture en couches peut être plus difficile à adapter à des changements majeurs.
- *L'ajout ou la suppression d'une couche peut nécessiter des modifications substantielles dans tout le système pour maintenir la cohérence et la compatibilité*



2.2.6 LIMITATIONS DE L'ARCHITECTURE EN COUCHES

EXEMPLE : DIFFICULTÉ D'AJOUT D'UNE NOUVELLE COUCHE   

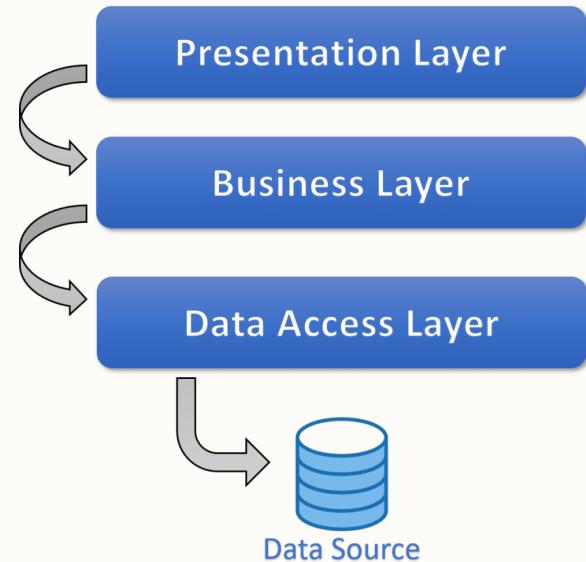
- *Imaginons l'architecture en couches d'un système de gestion de contenu.*
- *Si l'on décide d'ajouter une couche de recommandation de contenu basée sur l'apprentissage automatique, cela nécessiterait des ajustements significatifs.*
- *Il faudrait modifier la communication entre la couche de présentation et la couche métier pour inclure les recommandations, ainsi que mettre à jour la couche d'accès aux données pour intégrer les modèles de recommandation.*
- *Cette modification peut être complexe et coûteuse en termes de temps et de ressources.*

2.2.7 EXEMPLES D'UTILISATION DE L'ARCHI. EN COUCHES

EXEMPLES DANS DES CAS D'UTILISATION RÉELS



- *L'architecture en couches est couramment utilisée dans de nombreux domaines.*
- *Par exemple, les applications web, on l'a vu, utilisent souvent une architecture en couches pour séparer la présentation, la logique métier et l'accès aux données.*



2.2.7 EXEMPLES D'UTILISATION DE L'ARCHI. EN COUCHES

EXEMPLES DANS DES CAS D'UTILISATION RÉELS

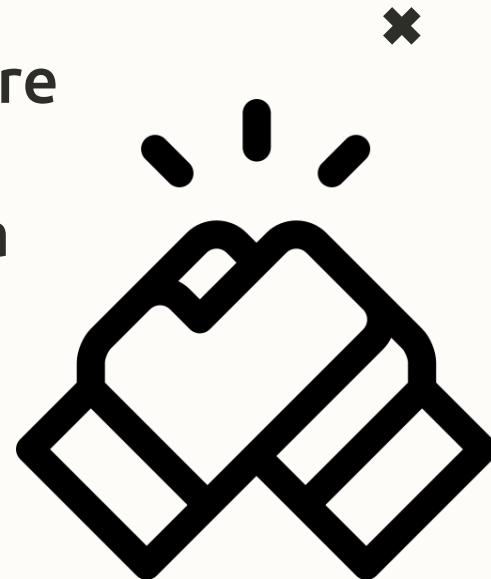


- *Exemple de template en couches pour FastAPI (Python) :*
https://github.com/fastapi-practices/fastapi_best_architecture
- *En .NET :* <https://github.com/joaosouzaaa/LayeredArchitecture>
- *En Java :* <https://github.com/inna-khachikyan/cs322-layered-architecture>

2.2.8 ARCHITECTURE EN COUCHES VS CLIENT-SERVEUR



- L'architecture en couches et l'architecture client-serveur sont deux concepts qui peuvent être **complémentaires** lors de la conception d'une application ou d'un système logiciel.
- Voici comment elles peuvent travailler ensemble de manière harmonieuse



2.2.8 ARCHITECTURE EN COUCHES VS CLIENT-SERVEUR



Séparation des Responsabilités :



- L'architecture en couches favorise la séparation des responsabilités au sein d'une application.
- Chaque couche a un ensemble spécifique de tâches et de responsabilités, ce qui rend le code plus modulaire et plus facile à gérer.
- D'un autre côté, l'architecture client-serveur divise l'application en deux parties : le client (interface utilisateur) et le serveur (logique métier et accès aux données).
- En combinant les deux, vous obtenez une séparation claire entre les responsabilités du client et du serveur, ce qui simplifie la maintenance et l'évolutivité.

2.2.8 ARCHITECTURE EN COUCHES VS CLIENT-SERVEUR



Répartition des Charges :



- L'architecture client-serveur permet de répartir les charges de travail entre le client et le serveur.
- Le client gère l'interface utilisateur et les interactions directes avec l'utilisateur, tandis que le serveur gère la logique métier, la gestion des données et les opérations qui nécessitent une sécurité accrue.
- En ajoutant une architecture en couches du côté du serveur, vous pouvez organiser la logique métier en plusieurs couches, ce qui simplifie la gestion des fonctionnalités complexes tout en conservant la simplicité du client.

2.2.8 ARCHITECTURE EN COUCHES VS CLIENT-SERVEUR



Évolutivité et Maintenance Facilitées :



- En combinant ces deux architectures, vous obtenez une structure qui facilite l'évolutivité et la maintenance.
- Si vous devez ajouter de nouvelles fonctionnalités, vous pouvez le faire dans la couche de serveur sans toucher au client, tant que l'interface client-serveur reste inchangée.
- De plus, la maintenance du serveur est simplifiée car vous pouvez mettre à jour chaque couche de manière indépendante sans affecter les autres.
- Cela permet une évolutivité verticale et horizontale plus efficace.

2.2.8 ARCHITECTURE EN COUCHES VS CLIENT-SERVEUR



Interopérabilité :

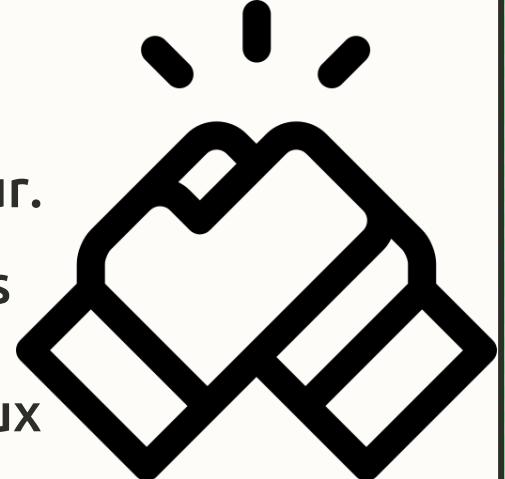
- L'architecture client-serveur permet également une interopérabilité plus facile, car différentes parties du système peuvent être développées dans des langages de programmation ou des environnements différents.
- L'architecture en couches côté serveur permet de structurer la logique métier de manière indépendante des détails d'implémentation, ce qui facilite l'intégration de nouvelles technologies ou la mise à jour de composants spécifiques sans affecter l'ensemble de l'application.



2.2.8 ARCHITECTURE EN COUCHES VS CLIENT-SERVEUR



- En résumé, l'architecture en couches complète l'architecture client-serveur en ajoutant une **structure de conception modulaire** et une **séparation claire des responsabilités** côté serveur.
- Cette combinaison offre de nombreux avantages en termes de **maintenance**, **d'évolutivité** et **d'interopérabilité**, ce qui en fait un choix judicieux pour de nombreuses applications et systèmes logiciels.



TRAVAUX PRATIQUES #1

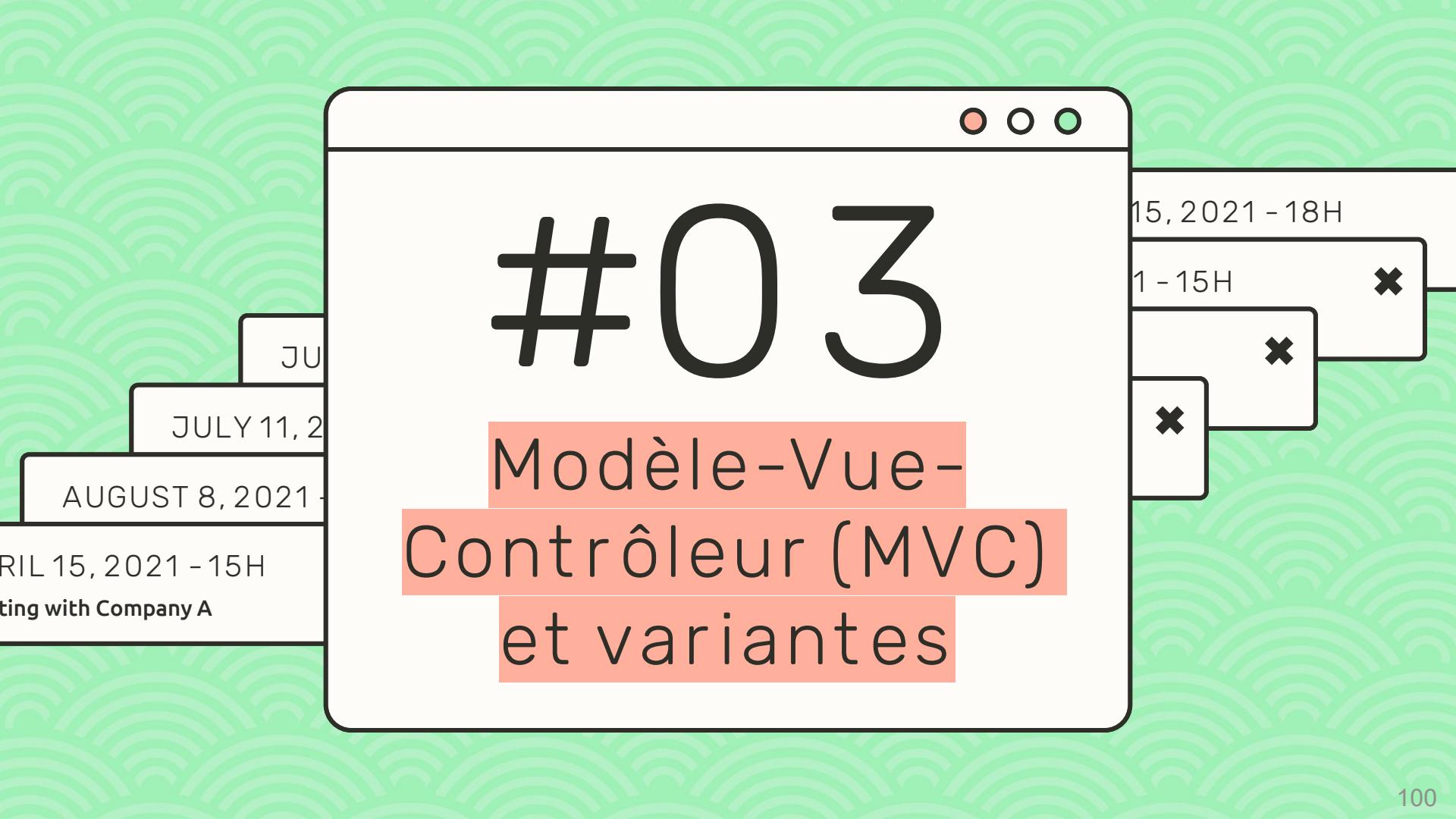
Vous pouvez faire le premier TP de ce cours qui porte particulièrement sur les styles architecturaux que nous venons de voir.



15, 2021 - 18H

1 - 15H





#03

Modèle-Vue-

Contrôleur (MVC)

et variantes

JU

JULY 11, 2

AUGUST 8, 2021

IRL 15, 2021 -15H

ting with Company A

15, 2021 -18H

1 - 15H



2.3.1 INTRODUCTION À L'ARCHITECTURE MVC



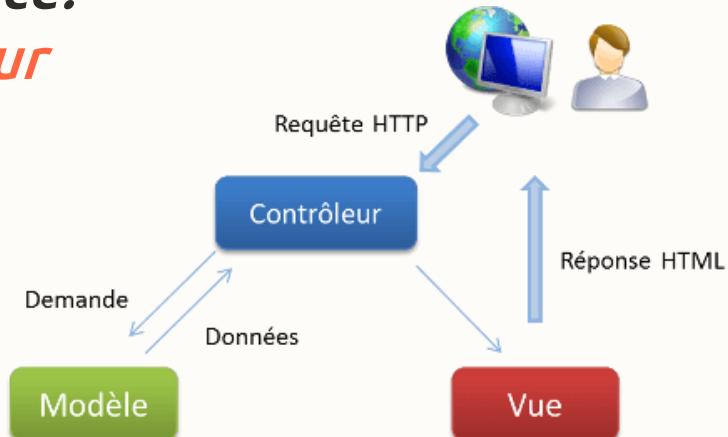
- L'architecture **MVC**, pour Modèle-Vue-Contrôleur, est un **modèle d'architecture logicielle** couramment utilisé dans le développement de logiciels.
- Il divise une application **en trois composants** interagissant entre eux :
 - le **modèle** (la logique de données),
 - la **vue** (la présentation des données)
 - et le **contrôleur** (la gestion des interactions).

2.3.1 INTRODUCTION À L'ARCHITECTURE MVC



EXEMPLES D'UTILISATION

- *Pensez à un site web de e-commerce.*
- *Le modèle gère les informations sur les produits, la vue affiche ces produits sur votre écran et le contrôleur gère ce qui se passe lorsque vous ajoutez un produit à votre panier et que vous passez commande.*



2.3.2 CARACTÉRISTIQUES DE L'ARCHITECTURE MVC

- L'architecture MVC favorise l'**organisation** du code, sa **réutilisabilité** et sa **maintenance**.
- Chaque composant a une **responsabilité** spécifique.
- Le **modèle gère les données**, la **vue affiche ces données** et le **contrôleur gère la logique** d'interaction entre le modèle et la vue.

2.3.3 CONCEPTION ET STRUCTURE D'UNE ARCHITECTURE MVC

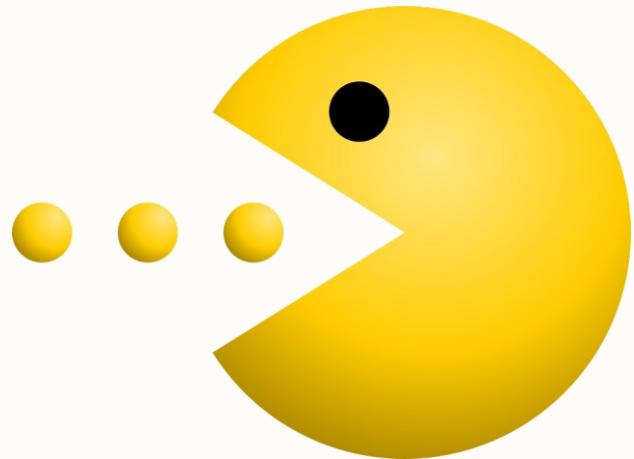
- La conception d'une **architecture MVC** implique de diviser le logiciel en trois composants principaux.
 - Le **modèle** contient les données et la logique liée aux données.
 - La **vue** affiche les informations à l'utilisateur.
 - Et le **contrôleur** gère les interactions entre le modèle et la vue.

2.3.3 CONCEPTION ET STRUCTURE D'UNE ARCHITECTURE MV*

EXERCICE - RÉFLEXION



- *Imaginez que vous développez un jeu vidéo simple, du genre jeu d'arcade comme Pac Man.*
- *Quelles seraient les responsabilités du modèle, de la vue et du contrôleur ?*



2.3.5 DÉPLOIEMENT ET ÉVOLUTIVITÉ D'UNE ARCHITECTURE MVC



- Une application MVC peut être déployée et évoluer facilement.
- Les modifications apportées à un composant n'affectent généralement pas les autres.
- De plus, plusieurs vues peuvent utiliser le même modèle, ce qui favorise le développement d'applications à plusieurs interfaces.



2.3.6 GESTION ET MAINTENANCE D'UNE ARCHITECTURE MVC

*

- La **maintenance** d'une application MVC est facilitée par sa **structure**.
- Comme chaque composant a des **responsabilités** bien définies, il est plus facile de **localiser** et de **corriger** les problèmes.

*

2.3.7 AVANTAGES ET INCONVÉNIENTS DE L'ARCHITECTURE MVC

- Parmi les avantages de l'architecture MVC, * citons la **modularité**, la **réutilisabilité** du code et la facilité de **maintenance**.
- Cependant, elle peut aussi entraîner une **complexité accrue** et n'est pas toujours la meilleure option pour des **applications simples ou de petite taille**.

2.3.7 AVANTAGES ET INCONVÉNIENTS DE L'ARCHITECTURE MVC

EXERCICE - RÉFLEXION



- *Identifiez une situation où l'utilisation de l'architecture MVC pourrait être **contre-productive** ?*



2.3.8 EXEMPLES D'UTILISATION DE L'ARCHITECTURE MVC ✖

EXEMPLES D'UTILISATION

- *L'architecture MVC est utilisée dans de nombreux domaines, notamment dans le développement web.*
- *Par exemple, des frameworks populaires comme Ruby on Rails, Django (Python) et Laravel (PHP) utilisent tous une forme d'architecture MVC.*



Laravel



django



2.3.8 EXEMPLES D'UTILISATION DE L'ARCHITECTURE MVC

EXEMPLES D'UTILISATION



Quelques exemples de boilerplates MVC :

- *En Python/Flask :* <https://github.com/salimane/flask-mvc>
- *En Python/Bottle :* <https://github.com/salimane/bottle-mvc>
- *En PHP :* <https://github.com/mmilanovic4/mvc>
- *En Node.js/Express :*
<https://github.com/oguzhanoya/express-mvc-boilerplate>
- *En Java/Swing :* <https://github.com/ashiishme/java-swing-mvc>

2.3.9 INTRODUCTION AUX VARIANTES DE L'ARCHITECTURE M~~V~~C

- En plus du modèle MVC classique, il existe d'autres variantes, comme le modèle-vue-présentateur (MVP) et le modèle-vue-vue-modèle (MVVM).
- Ces variantes respectent la philosophie MVC mais ajustent les responsabilités et les interactions des trois composants pour s'adapter à différentes situations.

2.3.10 VARIANTE MVP

*

- L'architecture **MVP** (Modèle-Vue-Présentateur) est une variante de MVC, utilisée pour structurer le code d'une application en séparant les **préoccupations logiques**.
- Le **Modèle** s'occupe de la gestion des données et de la logique métier.
- La **Vue** gère l'interface utilisateur et l'affichage des données.

2.3.10 VARIANTE MVP

*

- Le **Présentateur** fonctionne comme un **intermédiaire**, prenant les données du Modèle pour les afficher dans la Vue.
- Contrairement à MVC, où le Contrôleur manipule les données avant qu'elles n'atteignent la Vue, dans MVP, c'est la Vue qui appelle le Présentateur pour récupérer les données.

*

2.3.10 VARIANTE MVP



- La Vue est **plus passive**, se concentrant uniquement sur l'affichage, tandis que le Présentateur ne se soucie ni de la manière dont les données sont produites (*Modèle*) ni de la manière dont elles sont présentées (*Vue*). *
- MVP facilite le test de la logique de présentation et offre une séparation claire entre la logique d'affichage et la logique métier.

2.3.10 VARIANTE MVP



MVC

Model View Controller

VIEW

MODEL

CONTROLLER



MVP

Model View Presenter

VIEW

MODEL

PRESENTER



2.3.11 VARIANTE MVVM



- L'architecture **MVVM** (*Modèle-Vue-ViewModel*) est un schéma de conception populaire dans le développement d'applications avec des **interfaces utilisateur riches**.
- Le **Modèle** gère les **données et la logique métier**.
- La **Vue** s'occupe de l'**affichage et de l'interaction utilisateur**.
- Le **ViewModel** fait le **lien entre les deux**, transformant les **données du Modèle en une forme optimale pour la Vue**.

2.3.11 VARIANTE MVVM



- L'avantage réside dans la **séparation nette** entre la logique de l'interface utilisateur et la logique métier, favorisant ainsi la maintenance et les tests unitaires.
- La **liaison de données bidirectionnelle** est une caractéristique clé, permettant une communication fluide entre la **Vue** et le **ViewModel**.



2.3.11 VARIANTE MVVM

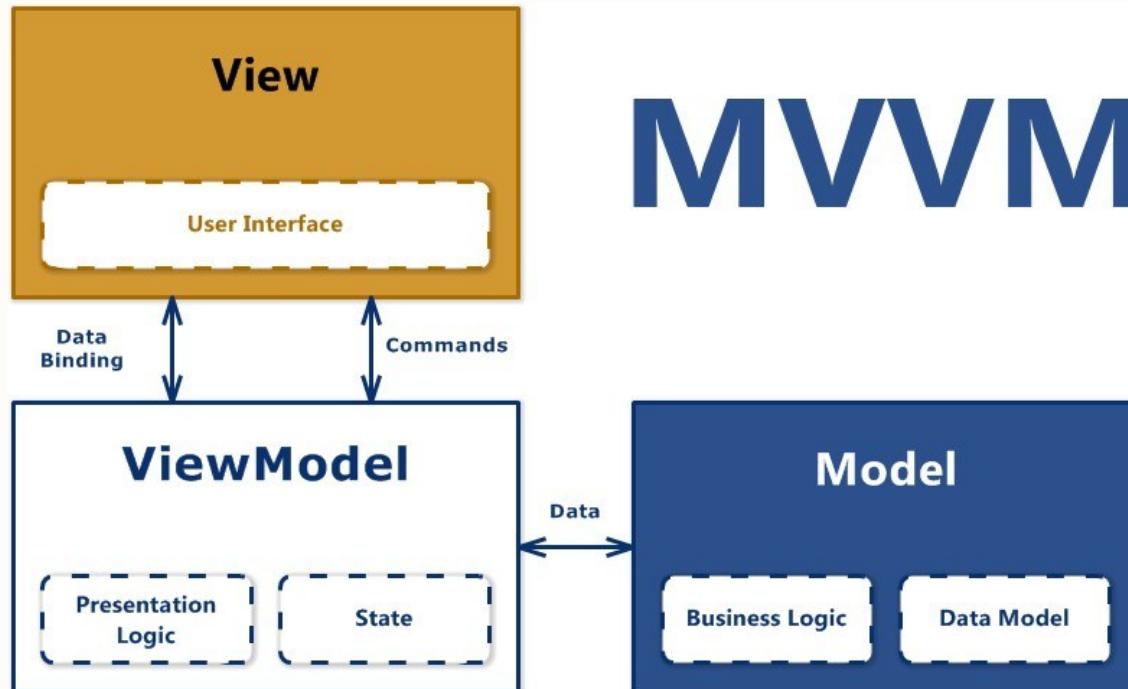


- MVVM est particulièrement efficace dans les environnements qui supportent le data-binding, comme les applications utilisant le framework WPF (*Windows Presentation Foundation*) ou des frameworks front-end tels qu'Angular ou Vue.js, facilitant ainsi la gestion des états et interactions au sein de l'application.



2.3.11 VARIANTE MVVM

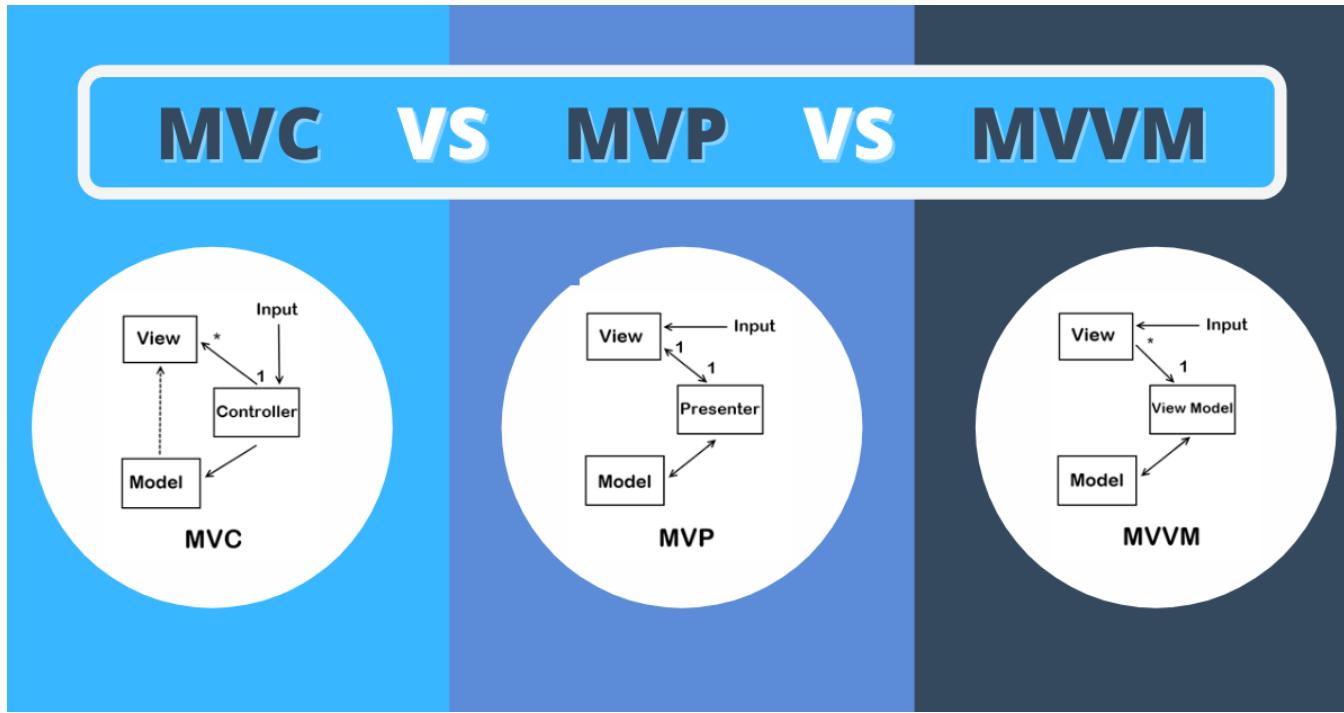
×



×

MVVM

2.3.12 COMPARAISON DES VARIANTES DE L'ARCHITECTURE MVC ✖



✖

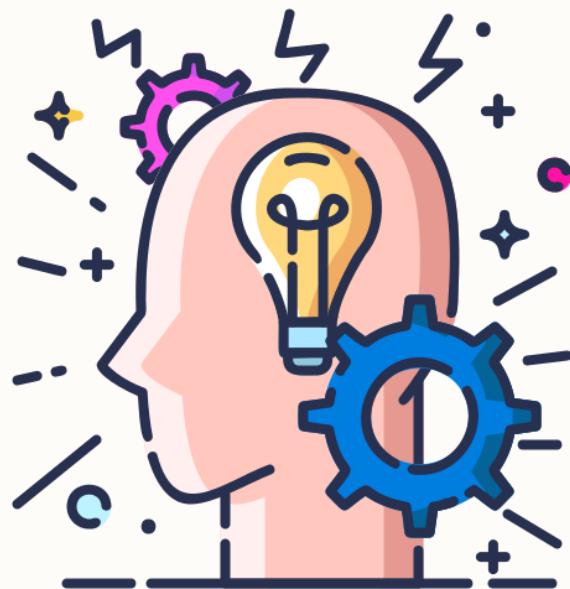
2.3.12 COMPARAISON DES VARIANTES DE L'ARCHITECTURE MVC



EXERCICE - RÉFLEXION



- Quelles pourraient être les raisons de choisir l'architecture **MVP** ou **MVVM** plutôt que l'architecture **MVC** traditionnelle ?*



2.3.13 CHOISIR ENTRE L'ARCHITECTURE MVC ET SES VARIANTES



- Chaque variante de l'architecture MVC a ses **avantages** et **inconvénients**.
- Le choix dépend des besoins spécifiques du projet.
- *Aucune solution ne peut être la meilleure dans tous les cas*



2.3.13 CHOISIR ENTRE L'ARCHITECTURE MVC ET SES VARIANTES



Choix de l'architecture MVC classique :



- **Web traditionnel** : MVC est souvent préféré pour les applications web traditionnelles où la **logique serveur** joue un rôle central.
- **Simplicité** : Optez pour MVC quand vous avez besoin d'une architecture **simple et directe**, particulièrement dans des petits à moyens projets.
- **Rapidité du développement** : Lorsque la **rapidité de développement** est cruciale, MVC avec son approche établie et sa vaste documentation peut être bénéfique.

2.3.13 CHOISIR ENTRE L'ARCHITECTURE MVC ET SES VARIANTES



Choix de l'architecture MVP:



Testabilité accrue : Si vous avez besoin d'une forte couverture de tests unitaires, particulièrement sur la logique de présentation, MVP est souvent un meilleur choix.

Séparation stricte : Lorsqu'une séparation claire entre la logique de présentation et l'UI est essentielle, pour des raisons de testabilité ou de clarté dans le code.

Technologies sans data-binding : Dans les environnements technologiques qui ne supportent pas le *data-binding bidirectionnel*, MVP pourrait être plus approprié.

2.3.13 CHOISIR ENTRE L'ARCHITECTURE MVC ET SES VARIANTES



Choix de l'architecture MVVM:



Data-binding : MVVM est particulièrement efficace dans les technologies qui supportent le **data-binding bidirectionnel**, comme WPF ou [Angular](#).

Applications riches : Pour les **applications riches client lourd** où une séparation nette des préoccupations et une **interaction UI dynamique** sont nécessaires.

Évolutions futures : Si votre application pourrait s'étendre ou se compliquer davantage, MVVM permet une **évolutivité plus aisée** avec une meilleure structuration.

2.3.13 CHOISIR ENTRE L'ARCHITECTURE MVC ET SES VARIANTES



CAS SPÉCIFIQUES



- *Applications mobiles* : MVVM est souvent favorisé dans le développement d'applications mobiles avec Xamarin ou en utilisant des frameworks comme ReactiveUI.
- *Applications web modernes (SPA)* : MVVM peut également être un choix judicieux pour les applications web monopage (SPA) utilisant des frameworks comme Angular ou Vue.js, qui tirent parti du data-binding bidirectionnel.
- *Applications d'entreprise* : MVP pourrait être privilégié pour des applications d'entreprise où la testabilité et la maintenance sont primordiales.

#04

Architecture
orientée services
*(Service-Oriented
Architecture)*

JU
JULY 11, 2021
AUGUST 8, 2021
AUGUST 15, 2021 - 15H
ing with Company A

15, 2021 - 18H

1 - 15H



2.4.1 INTRODUCTION À L'ARCHITECTURE ORIENTÉE SERVICES

- L'architecture orientée services (*SOA*) est un style d'architecture logicielle qui favorise des **systèmes modulaires** et **flexibles** basés sur des **services**.
- Elle émerge dans les années 1990 et gagne en popularité depuis.
- SOA repose sur des principes tels que la **modularité**, le **découplage**, la **réutilisabilité** et **l'interopérabilité**, permettant aux organisations de créer des systèmes agiles capables de s'adapter rapidement.

2.4.1 INTRODUCTION À L'ARCHITECTURE ORIENTÉE SERVICES

- Dans l'architecture orientée services, les services représentent des **fonctionnalités spécifiques** offertes par des composants logiciels.
- Ils interagissent en utilisant des **protocoles standardisés**.



2.4.1 INTRODUCTION À L'ARCHITECTURE ORIENTÉE SERVICES

- Par exemple, un *service de paiement* peut communiquer avec un *service de gestion des comptes* pour vérifier la disponibilité des fonds avant une transaction.
- Cette approche favorise la **réutilisabilité** et la **modularité**, permettant aux services d'être développés, testés et déployés indépendamment.



2.4.1 INTRODUCTION À L'ARCHITECTURE ORIENTÉE SERVICES ✖

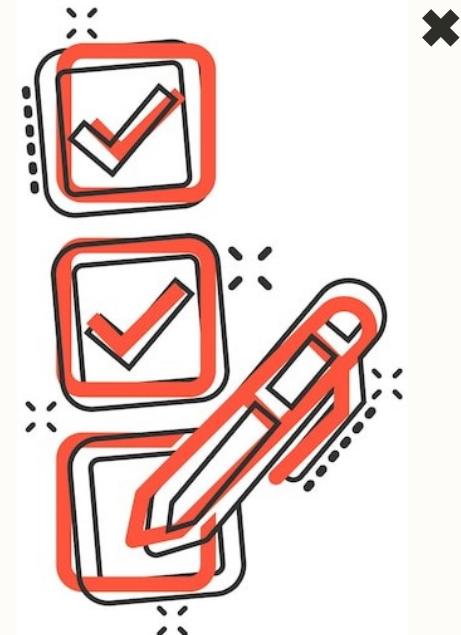
EXERCICE - RÉFLEXION

- *Identifiez trois fonctionnalités clés d'une application de réservation de voyages et imaginez comment elles pourraient être développées en tant que services autonomes.*



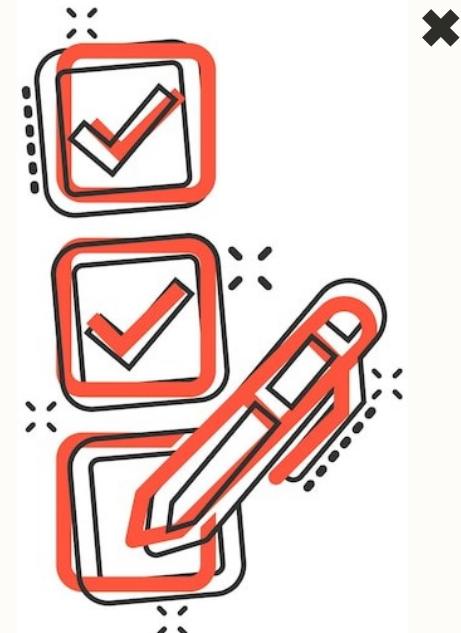
2.4.2 CARACTÉRISTIQUES CLÉS DE L'ARCHI. ORIENTÉE SERVICES ✖

- L'architecture orientée services se caractérise par le **découplage**, la **modularité**, la **réutilisabilité** et l'**interopérabilité**.
- Le **découplage** permet à chaque service de fonctionner de **manière indépendante**, facilitant les modifications ou les remplacements.



2.4.2 CARACTÉRISTIQUES CLÉS DE L'ARCHI. ORIENTÉE SERVICES ✖

- La modularité divise les fonctionnalités en services autonomes, simplifiant ainsi le développement, le test et le déploiement.
- La réutilisabilité permet l'utilisation de services dans différentes applications, réduisant les efforts de développement.



2.4.2 CARACTÉRISTIQUES CLÉS DE L'ARCHI. ORIENTÉE SERVICES ✖

- L'interopérabilité, enfin, assure la communication entre services, indépendamment des langages ou des plates-formes utilisés.



2.4.2 CARACTÉRISTIQUES CLÉS DE L'ARCHI. ORIENTÉE SERVICES ✕

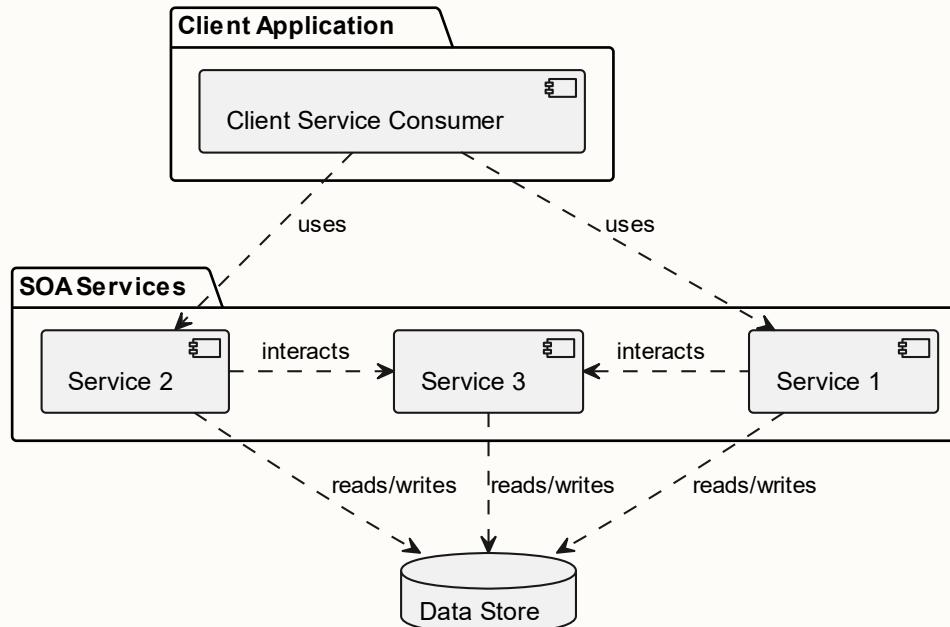


Diagramme de composants représentant une SOA en UML

2.4.2 CARACTÉRISTIQUES CLÉS DE L'ARCHI. ORIENTÉE SERVICES ✖

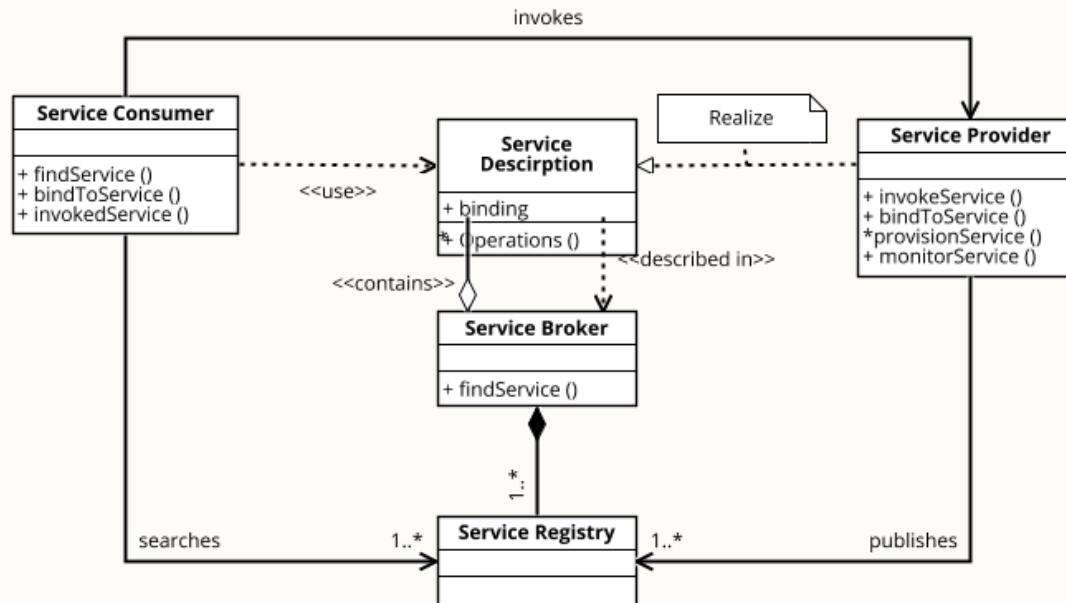


Diagramme de classes représentant une SOA en UML

2.4.2 CARACTÉRISTIQUES CLÉS DE L'ARCHI. ORIENTÉE SERVICES

EXERCICE - RÉFLEXION



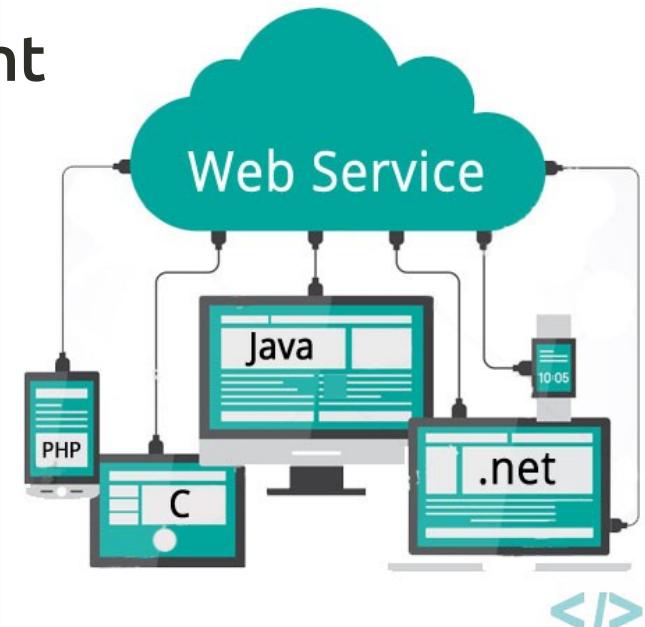
- Identifiez une autre caractéristique de l'architecture orientée services et expliquez comment elle peut bénéficier au développement de logiciels basés sur une SOA*



2.4.3 SERVICES WEB ET PROTOCOLES

*

- Les **services web** sont couramment utilisés pour mettre en œuvre l'architecture orientée services.
- Ils permettent aux services de communiquer via **Internet** en utilisant des **protocoles standardisés**.



2.4.3 SERVICES WEB ET PROTOCOLES



- Les **services web** utilisent généralement le protocole **HTTP** pour la communication et le langage **XML** pour représenter les données échangées.
- Cette approche facilite la **communication transparente** entre les services développés dans **différents langages** et exécutés sur **différentes plates-formes**.



2.4.3 SERVICES WEB ET PROTOCOLES

*

- Deux méthodes couramment utilisées pour les services web sont **SOAP** (*Simple Object Access Protocol*) et **REST** (*REpresentational State Transfer*).
- **SOAP** est un protocole basé sur **XML** qui offre un **formalisme strict** et **complexe**.



2.4.3 SERVICES WEB ET PROTOCOLES

*

- REST, quant à lui, repose sur les principes du web et utilise les méthodes HTTP (*GET, POST, PUT, DELETE*) pour interagir avec les services.
- REST est plus simple, léger et largement adopté pour les services web.



2.4.3 SERVICES WEB ET PROTOCOLES



EXEMPLE : CRÉATION D'UN SERVICE WEB RESTFUL



- *Supposons que vous deviez créer un service web **RESTful** pour récupérer des données d'un système de gestion des clients.*
- *Vous pouvez définir une **API REST** avec une **URI** spécifiant l'adresse du service et les ressources à récupérer.*
- *Par exemple, **/clients** pour récupérer tous les clients ou **/clients/{id}** pour récupérer un client spécifique en fonction de son identifiant.*
- *Les méthodes **HTTP** sont utilisées pour interagir avec le service, offrant une **interface** simple et intuitive pour récupérer les données du système de gestion des clients.*

2.4.3 SERVICES WEB ET PROTOCOLES

*

EXERCICE - RÉFLEXION



- *Identifiez plusieurs différences entre les approches **SOAP** et **REST**.*



2.4.4 COMPOSITION DE SERVICES



- La **composition de services** consiste à **combiner** plusieurs services existants pour créer des **fonctionnalités complexes**.
- Au lieu de développer une fonctionnalité complexe à partir de zéro, la **composition de services** permet de tirer parti des services existants et de les **orchestrer** pour répondre à un besoin spécifique.
- *Par exemple, pour créer un processus de réservation de billets en ligne, vous pouvez combiner des services tels que la recherche de vols, la réservation d'hôtels et le paiement.*

2.4.4 COMPOSITION DE SERVICES

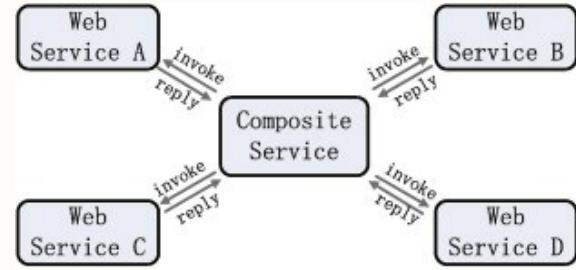


- Il existe deux approches principales pour la composition de services : **l'orchestration** et la **chorégraphie**.
- Dans **l'orchestration**, un service principal (**l'orchestrateur**) contrôle et coordonne l'exécution des autres services.
- L'**orchestrateur** définit un flux d'exécution et appelle séquentiellement les services nécessaires pour accomplir une tâche.

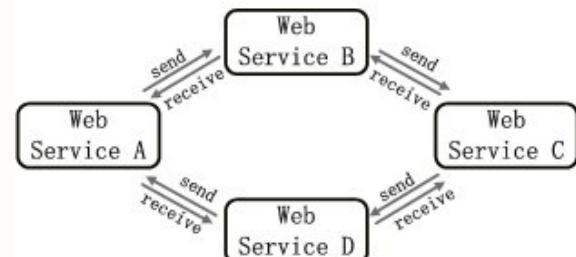
2.4.4 COMPOSITION DE SERVICES



- Dans la **chorégraphie**, chaque service connaît ses propres responsabilités et interagit directement avec les autres services pour accomplir une tâche.
- Les services communiquent entre eux en utilisant des messages échangés selon un **protocole** défini.



(a) Service Orchestration



(b) Service Choreography

2.4.4 COMPOSITION DE SERVICES



EXEMPLE : COMPOSITION DE SERVICES POUR UN SYSTÈME DE RÉSERVATION

- *Imaginons que vous souhaitez créer un flux de processus de réservation de billets en ligne.*
- *Vous pouvez utiliser l'orchestration pour contrôler les étapes du processus.*
- *L'orchestre appelle successivement les services de recherche de vols pour trouver des options, de réservation d'hôtels pour réserver un hébergement et de paiement pour effectuer la transaction.*



2.4.4 COMPOSITION DE SERVICES

*

EXEMPLE : COMPOSITION DE SERVICES POUR UN SYSTÈME DE RÉSERVATION

- *L'orchestrateur gère les erreurs éventuelles et garantit le bon déroulement du processus.*
- *En composant ces services existants, vous créez un système complet de réservation de billets en ligne.*



2.4.4 COMPOSITION DE SERVICES



EXERCICE - RÉFLEXION



- *Identifiez une **différence** majeure entre **l'orchestration** et la **chorégraphie** dans la composition de services et expliquez comment chaque approche peut être utilisée pour créer un flux de processus de réservation de billets en ligne.*



2.4.5 GESTION DES SERVICES



- La gestion des services dans une architecture orientée services englobe plusieurs aspects importants.
- La **découverte des services** permet aux développeurs de trouver rapidement les services nécessaires à leurs applications sans avoir à connaître tous les détails de chaque service existant.
- Les services peuvent être découverts à l'aide de **registres**, de **moteurs de recherche** ou **d'annuaires de services**.

2.4.5 GESTION DES SERVICES

*

- La **publication des services** consiste à les rendre accessibles à d'autres applications en exposant leurs interfaces et leurs fonctionnalités.
- Un **annuaire de services** agit comme une base de données centralisée où les services peuvent être enregistrés et recherchés.
- La **surveillance des services** implique de suivre leur disponibilité, leurs performances et de détecter d'éventuelles erreurs.

2.4.5 GESTION DES SERVICES



- *Supposons que vous utilisez un annuaire de services pour rechercher et accéder à des services disponibles.*
- *L'annuaire contient des informations sur les différents services, tels que leurs descriptions, leurs interfaces et leurs emplacements.*
- *Vous pouvez utiliser l'annuaire pour trouver un service spécifique, obtenir les détails nécessaires et établir une connexion pour l'utiliser dans votre application.*



2.4.5 GESTION DES SERVICES

*

EXEMPLE : UTILISATION D'UN ANNUAIRE DE SERVICES



- L'annuaire facilite la découverte et l'intégration des services, simplifiant ainsi le processus de développement et d'utilisation des services dans une architecture orientée services.*



2.4.6 SÉCURITÉ DANS UNE SOA



- La sécurité est un aspect essentiel de l'architecture orientée services.
- Les services doivent être protégés contre les accès non autorisés et les attaques potentielles.
- Les protocoles de sécurité tels que SSL/TLS (HTTPS) sont utilisés pour sécuriser les échanges de données entre les services.
- Les considérations de sécurité incluent :
 - l'authentification des utilisateurs et des services,
 - l'autorisation pour contrôler les accès,
 - la confidentialité des données échangées
 - et l'intégrité des messages.

2.4.6 SÉCURITÉ DANS UNE SOA



1. L'**authentification** garantit que seules les **entités légitimes** peuvent accéder aux services en vérifiant leur identité.
2. L'**autorisation** contrôle **les actions autorisées** pour chaque entité après l'authentification.
3. La **confidentialité** assure que les **données échangées entre les services sont protégées** contre les regards indiscrets.
4. L'**intégrité** garantit que les **messages ne sont pas modifiés** lors de leur transmission, en utilisant des mécanismes de vérification de l'intégrité des données.

2.4.6 SÉCURITÉ DANS UNE SOA



EXEMPLE : SÉCURISATION DES ÉCHANGES DE DONNÉES



- *Prenons l'exemple de la sécurisation des échanges de données entre les services à l'aide de protocoles de sécurité tels que SSL/TLS.*
- *Ces protocoles permettent d'établir des canaux de communication chiffrés entre les services, assurant ainsi la confidentialité des données.*



2.4.6 SÉCURITÉ DANS UNE SOA

*

EXEMPLE : SÉCURISATION DES ÉCHANGES DE DONNÉES



- *Ils utilisent des **certificats** pour vérifier l'identité des services et garantir l'authenticité.*
- *De plus, des mécanismes de vérification d'intégrité, tels que les fonctions de **hachage**, sont utilisés pour s'assurer que les données n'ont pas été altérées lors de leur transit.*



SSL/TLS

2.4.7 AVANTAGES ET DÉFIS DE LA SOA



- L'architecture orientée services offre de nombreux **avantages**, notamment la **réutilisabilité** des services qui permet de développer plus rapidement en tirant parti des services existants.
- Elle favorise également la **flexibilité** en permettant aux services d'être **modifiés ou remplacés indépendamment**, sans affecter l'ensemble du système.
- De plus, **l'interopérabilité** facilite l'intégration entre différentes applications et plates-formes.



2.4.7 AVANTAGES ET DÉFIS DE LA SOA



- Malgré ses avantages, l'architecture orientée services présente également des **défis potentiels**.
- La **complexité** peut augmenter en raison de l'orchestration ou de la **composition** de plusieurs services.
- La **gestion** des services, la **coordination** des flux de données et la **gestion des erreurs** peuvent également être des défis.
- De plus, la **performance** peut être affectée par la **communication** entre les services et le **traitement** des messages.



2.4.7 AVANTAGES ET DÉFIS DE LA SOA



EXEMPLE : UTILISATION D'UNE SOA POUR FACILITER L'INTÉGRATION DE SERVICES

- *Un exemple concret de l'utilisation de l'architecture orientée services est la facilitation de l'intégration entre différentes applications d'entreprise.*
- *En adoptant une approche orientée services, les entreprises peuvent exposer des services spécifiques qui peuvent être utilisés par d'autres applications internes ou externes.*

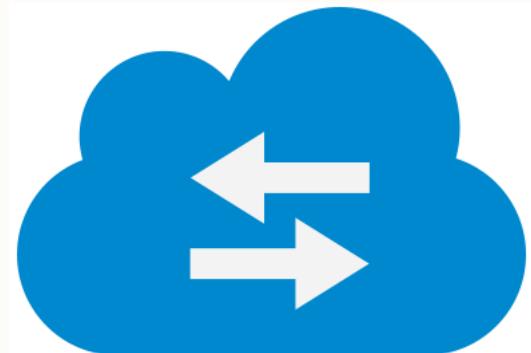


2.4.7 AVANTAGES ET DÉFIS DE LA SOA



EXEMPLE : UTILISATION D'UNE SOA POUR FACILITER L'INTÉGRATION DE SERVICES

- *Par exemple, un service de gestion des clients peut être partagé entre un système de facturation, un système de support client et une application mobile.*
- *Cela permet une meilleure collaboration et une meilleure évolutivité des systèmes d'entreprise.*



2.4.7 AVANTAGES ET DÉFIS DE LA SOA



EXEMPLE : UTILISATION D'UNE SOA



- *Exemple d'une architecture SOA sur AWS :*
<https://github.com/jcolemorrisson/foundational-soa>

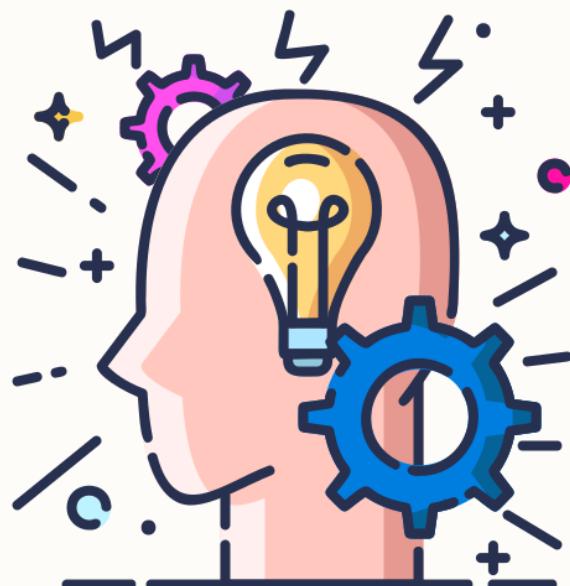
2.4.7 AVANTAGES ET DÉFIS DE LA SOA



EXERCICE - RÉFLEXION



- *Identifiez d'autres avantages de l'architecture orientée services et expliquez comment ils peuvent bénéficier au développement de systèmes complexes.*



TRAVAUX PRATIQUES #2

Vous pouvez faire le nouveau TP de ce cours qui porte particulièrement sur les styles architecturaux que nous venons de voir.



15, 2021 - 18H

1 - 15H





#05

Architecture microservices

JU
JULY 11, 2021
AUGUST 8, 2021
RIL 15, 2021 - 15H
ting with Company A

15, 2021 - 18H

1 - 15H



2.5.1 INTRODUCTION À L'ARCHITECTURE MICROSERVICES



- L'architecture **microservices** est une approche moderne de la conception logicielle qui se distingue de l'architecture orientée services (**SOA**) par sa **granularité**.
- Alors que SOA privilégie des **services plus vastes**, les microservices sont de **petits modules autonomes** responsables de tâches spécifiques.

2.5.1 INTRODUCTION À L'ARCHITECTURE MICROSERVICES



- Cette granularité favorise la **flexibilité**, l'**agilité** et **l'évolutivité**, permettant aux équipes de développement de travailler de manière **indépendante** sur des fonctionnalités précises.
- Les microservices sont souvent déployés dans des **conteneurs**, simplifiant ainsi la **gestion** et la **scalabilité**.

2.5.1 INTRODUCTION À L'ARCHITECTURE MICROSERVICES



- Cette approche est de plus en plus populaire dans la création d'applications modernes, offrant une meilleure adaptabilité aux besoins changeants.
- Par exemple, Netflix a migré vers cette architecture à partir de 2008 pour des mises à jour plus fréquentes et des déploiements continus.

2.5.1 INTRODUCTION À L'ARCHITECTURE MICROSERVICES



- Comparée à l'architecture monolithique, l'architecture microservices offre une meilleure flexibilité et évolutivité.
- Amazon.com a connu un succès notable après avoir migré vers cette architecture, améliorant ainsi l'agilité et la disponibilité de son site.

2.5.1 INTRODUCTION À L'ARCHITECTURE MICROSERVICES



- Twitter est un autre exemple d'entreprise qui a migré, dès 2010, d'une architecture monolithique vers une architecture microservices pour faire face à la complexité croissante.
- La transition leur a permis d'obtenir une évolutivité plus efficace et une gestion aisée des différents services.

2.5.1 INTRODUCTION À L'ARCHITECTURE MICROSERVICES



EXEMPLE CONCRET D'UTILISATION : EBAY



- *eBay aussi a effectué une migration vers les microservices.*
- *eBay a alors décomposé son application monolithique en plusieurs microservices autonomes.*
- *Cela a permis à eBay de gérer efficacement la complexité de sa plateforme tout en offrant des fonctionnalités et des services plus rapidement à ses utilisateurs.*



2.5.2 CARACTÉRISTIQUES DES MICROSERVICES

*

- Les microservices se caractérisent par leur autonomie et leur indépendance.
- Uber est un exemple historique où chaque module, tels que la gestion des utilisateurs et le suivi des courses, est développé et déployé en tant que microservice indépendant.

*

2.5.2 CARACTÉRISTIQUES DES MICROSERVICES

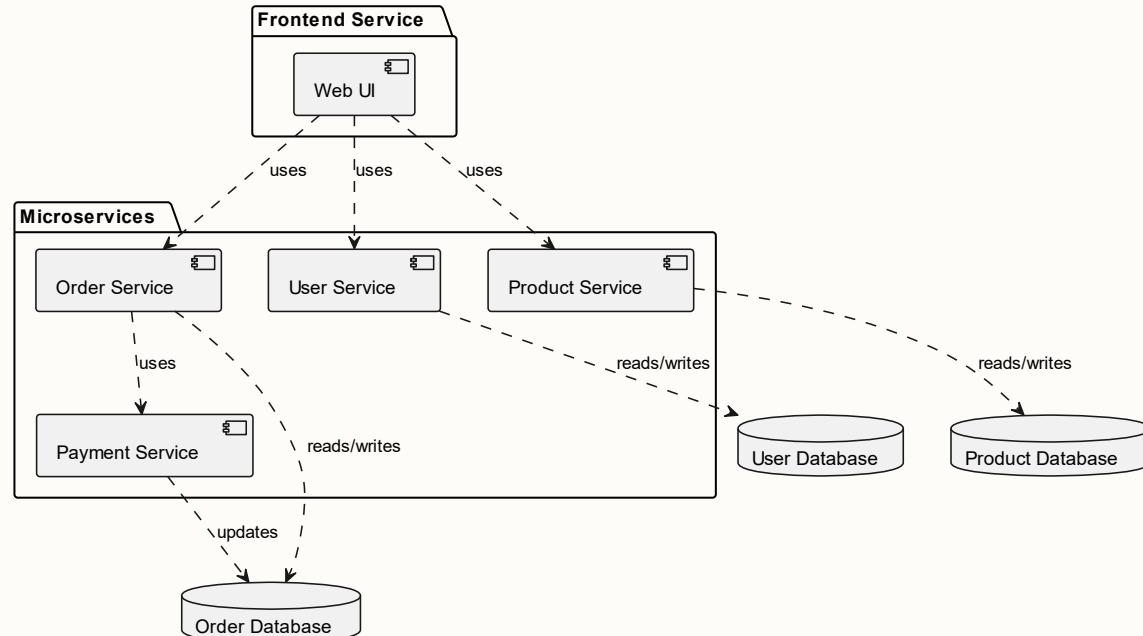


Diagramme de composants UML représentant une architecture micro-services

2.5.2 CARACTÉRISTIQUES DES MICROSERVICES



EXEMPLES CONCRETS D'UTILISATION : UBER



- *Uber utilise une architecture microservices pour son application de réservation de trajets.*
- *Chaque fonctionnalité clé, telle que la gestion des utilisateurs, la recherche de conducteurs et le suivi des courses, est gérée par un microservice distinct.*
- *Cela permet une indépendance et une évolutivité des services, facilitant ainsi l'adaptation à la demande croissante.*



2.5.2 CARACTÉRISTIQUES DES MICROSERVICES



EXERCICE - RÉFLEXION



- Choisissez une entreprise de votre choix et identifiez trois de ses services ou fonctionnalités clés.*
- Discutez de la manière dont ces services pourraient être développés et déployés en utilisant une approche microservices.*



2.5.3 CONCEPTION DES MICROSERVICES



- La conception des microservices repose sur la délimitation du domaine métier et la communication entre eux.
- Spotify est un exemple concret où chaque aspect de l'application, tels que la recherche de musique et les listes de lecture, est géré par des microservices distincts.

2.5.3 CONCEPTION DES MICROSERVICES



- La **délimitation du domaine métier** permet d'attribuer des **responsabilités claires** à chaque microservice.
- La **communication entre eux** peut être réalisée via des protocoles tels que **HTTP** ou **gRPC**.
- La **gestion des données** peut se faire en utilisant des **bases de données dédiées** à chaque microservice ou en partageant les données via un **bus de messages**.

2.5.3 CONCEPTION DES MICROSERVICES

*

EXEMPLES CONCRETS D'UTILISATION : SPOTIFY



- *Dans le domaine du divertissement, Spotify utilise des microservices pour son application de streaming musical.*
- *Ils ont des microservices spécifiques pour la recherche de musique, la gestion des listes de lecture et la recommandation de contenu, ce qui permet une conception modulaire et une évolutivité efficace.*



2.5.3 CONCEPTION DES MICROSERVICES

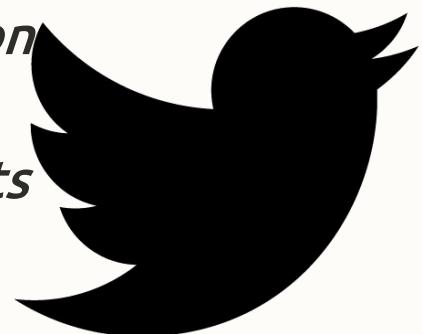
*

EXEMPLES CONCRETS D'UTILISATION : TWITTER



Liste (non exhaustive) des microservices de Twitter :

- *Service de publication de tweets* : Gère la création, la mise à jour et la suppression de tweets.
- *Service de timeline* : Agrège les tweets pertinents pour l'utilisateur et les affiche dans sa timeline.
- *Service de gestion des utilisateurs* : Gère les profils, l'authentification et l'autorisation des utilisateurs.



2.5.3 CONCEPTION DES MICROSERVICES



EXEMPLES CONCRETS D'UTILISATION : TWITTER



- *Service de notifications* : Envoie des notifications aux utilisateurs pour les interactions, les mentions et les messages directs.
- *Service de recherche* : Permet aux utilisateurs de rechercher des tweets, des utilisateurs et des hashtags.
- *Service de messagerie directe* : Gère les messages privés entre utilisateurs.
- *Service de gestion des médias* : Stocke et gère les images, vidéos et autres médias partagés dans les tweets.



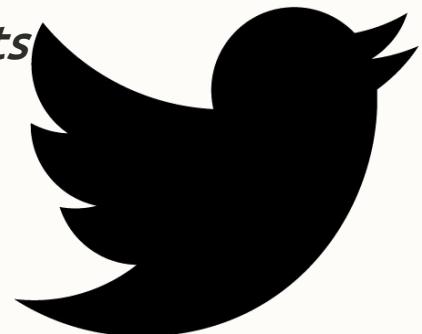
2.5.3 CONCEPTION DES MICROSERVICES



EXEMPLES CONCRETS D'UTILISATION : TWITTER



- *Service de gestion des hashtags : Gère les hashtags associés aux tweets pour faciliter la recherche.*
- *Service de suivi des utilisateurs : Gère les abonnements et les abonnés d'un utilisateur.*
- *Service de statistiques : Fournit des données analytiques sur l'engagement des utilisateurs et les performances des tweets.*
- *Service de gestion des tendances : Identifie les sujets tendance et les affiche sur la page d'accueil.*



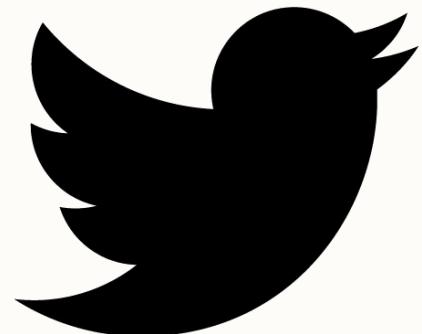
2.5.3 CONCEPTION DES MICROSERVICES



EXEMPLES CONCRETS D'UTILISATION : TWITTER



- *Service de géolocalisation* : Permet aux utilisateurs d'ajouter des informations de localisation à leurs tweets.
- *Service de signalement* : Permet aux utilisateurs de signaler des abus ou des contenus inappropriés.
- *Service de gestion des paramètres* : Gère les préférences et les paramètres de confidentialité des utilisateurs.
- *Service de gestion des publicités* : Gère les campagnes publicitaires et les annonces ciblées.



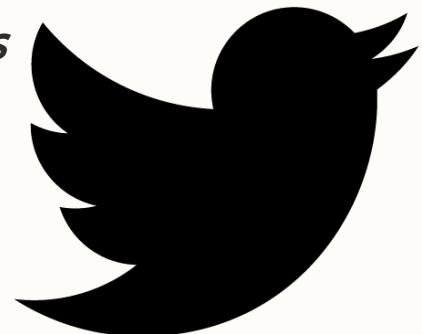
2.5.3 CONCEPTION DES MICROSERVICES



EXEMPLES CONCRETS D'UTILISATION : TWITTER



- *Service de traduction : Fournit une traduction automatique des tweets dans différentes langues.*
- *Service de recommandations : Fournit des suggestions de comptes à suivre et de tweets à consulter.*
- *Service de gestion des hashtags sponsorisés : Permet aux annonceurs de promouvoir des hashtags spécifiques.*
- *Service de sécurité : Surveille et protège le système contre les menaces et les attaques.*



2.5.4 COMMUNICATION ENTRE LES MICROSERVICES



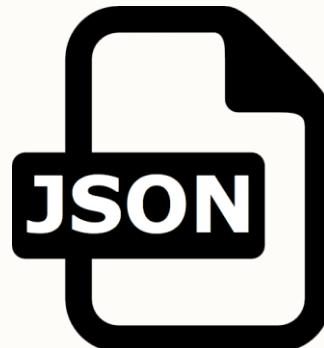
- Les microservices peuvent communiquer de manière **synchrone** via des protocoles tels que **HTTP** ou **gRPC**, ou de manière **asynchrone** en utilisant des **systèmes de messagerie** ou des **bus d'événements**.



2.5.4 COMMUNICATION ENTRE LES MICROSERVICES



- Les formats de données populaires incluent **JSON** et **protobuf**, le format binaire de sérialisation de données développé par Google



2.5.4 COMMUNICATION ENTRE LES MICROSERVICES *

EXEMPLES CONCRETS D'UTILISATION : AMAZON

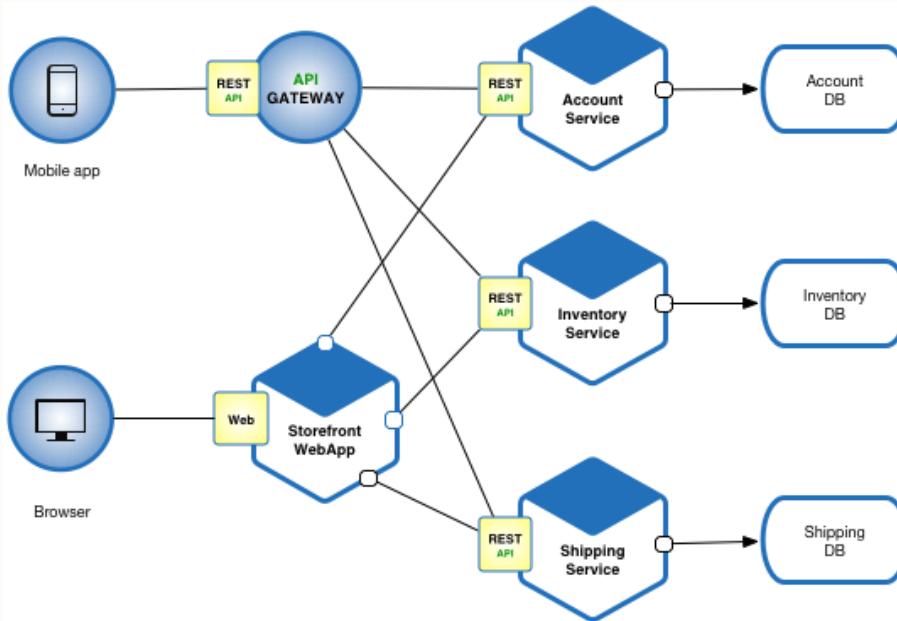


- *Dans le secteur de la vente en ligne, Amazon utilise une architecture microservices pour son système de commerce électronique.*
- *Par exemple, leur microservice de gestion des utilisateurs communique avec leur microservice de traitement des commandes via des appels d'API RESTful, permettant une coordination efficace entre les différentes parties du système.*



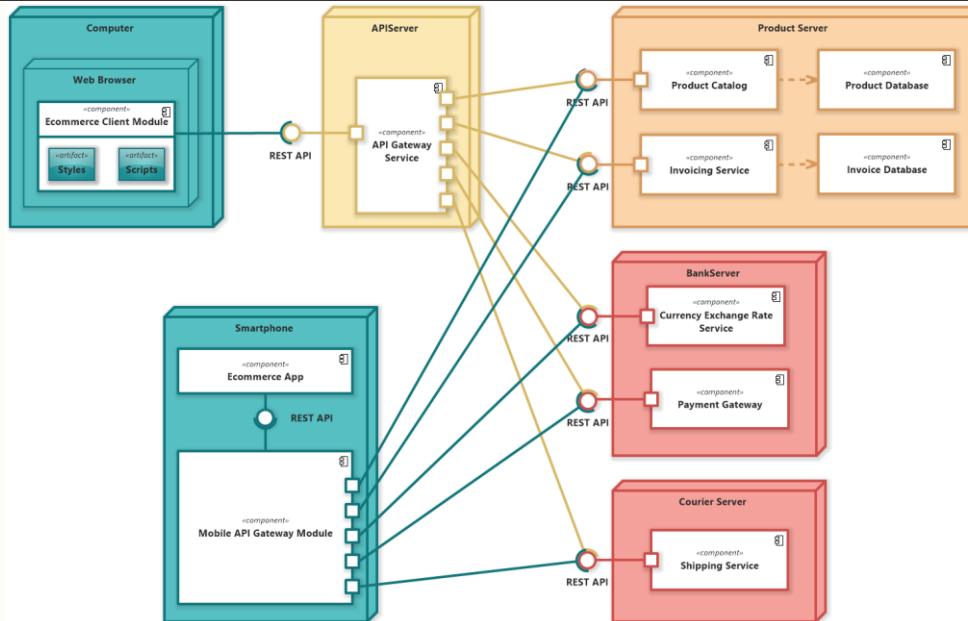
2.5.4 COMMUNICATION ENTRE LES MICROSERVICES ✖

EXEMPLES DE DIAGRAMMES



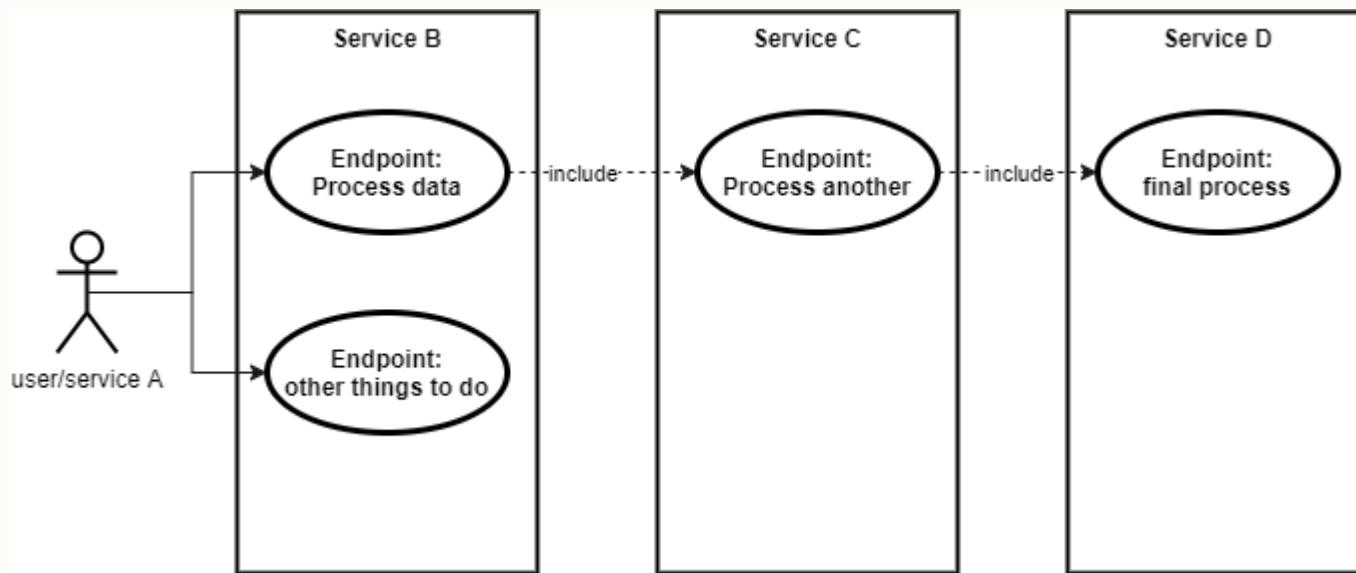
2.5.4 COMMUNICATION ENTRE LES MICROSERVICES ✖

EXEMPLES DE DIAGRAMMES UML



2.5.4 COMMUNICATION ENTRE LES MICROSERVICES ✖

EXEMPLES DE DIAGRAMMES UML



2.5.4 COMMUNICATION ENTRE LES MICROSERVICES *

EXERCICE - RÉFLEXION



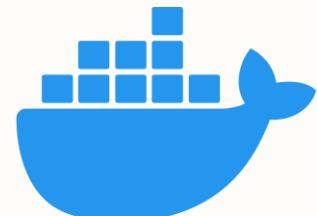
- *Comparez les protocoles de communication tels que **HTTP** et **gRPC** en mettant en évidence leurs **avantages** et leurs cas d'utilisation respectifs.*



2.5.5 DÉPLOIEMENT DES MICROSERVICES

*

- Les microservices peuvent être déployés sur des machines virtuelles, des conteneurs tels que Docker ou avec des outils d'orchestration comme Kubernetes.



docker®



kubernetes

2.5.5 DÉPLOIEMENT DES MICROSERVICES



- Les conteneurs, tels que Docker, offrent une isolation légère pour les microservices et facilitent le déploiement.
- Les outils d'orchestration, tels que Kubernetes, permettent de gérer efficacement le déploiement, la mise à l'échelle et la gestion des microservices.



2.5.5 DÉPLOIEMENT DES MICROSERVICES



EXEMPLES CONCRETS D'UTILISATION : AIRBNB



- *Airbnb utilise une architecture microservices pour sa plateforme de réservation d'hébergements.*
- *En utilisant des conteneurs Docker et l'outil d'orchestration Kubernetes, ils peuvent déployer et gérer facilement plusieurs microservices, tels que la réservation de logements, la gestion des paiements et la gestion des commentaires, assurant ainsi une évolutivité et une disponibilité élevées.*



2.5.6 GESTION DES MICROSERVICES



- La **gestion** des microservices comprend plusieurs aspects, notamment la **surveillance** des **performances**, la **tolérance aux pannes**, la **mise à l'échelle** et la **gestion des versions**.
- La **surveillance** des microservices est essentielle pour collecter et **analyser les métriques** liées aux **performances** et à la disponibilité.



2.5.6 GESTION DES MICROSERVICES

*

- La **tolérance aux pannes** peut être assurée en mettant en place des mécanismes de **redémarrage automatique**, de **basculement** ou de **réPLICATION** des microservices.
- La **mise à l'échelle** permet **d'ajuster les ressources** en fonction de la charge.
- La **gestion des versions** garantit une **transition en douceur** lors des mises à jour et des évolutions des microservices.

2.5.6 GESTION DES MICROSERVICES



EXEMPLES CONCRETS D'UTILISATION : PAYPAL



- *Dans le domaine de la finance, PayPal adopte une architecture microservices pour son système de paiement en ligne.*
- *Ils surveillent constamment les performances et la disponibilité de leur cluster de microservices à l'aide d'outils tels que Prometheus, leur permettant de détecter les problèmes et d'assurer un fonctionnement fluide de leur service.*



PayPal

2.5.7 AVANTAGES ET DÉFIS DE L'ARCHITECTURE MICROSERVICES



- Les avantages de l'architecture microservices incluent une plus grande **flexibilité**, une **évolutivité granulaire**, une **meilleure résilience** et une **facilité de déploiement**.
- Chaque microservice peut être **développé**, **testé** et **déployé** de manière **indépendante**, ce qui facilite les mises à jour et les évolutions.
- L'**évolutivité granulaire** permet de **mettre à l'échelle** uniquement les services nécessaires en fonction de la charge.



2.5.7 AVANTAGES ET DÉFIS DE L'ARCHITECTURE MICROSERVICES



- Parmi les défis de l'architecture microservices, on trouve la **complexité accrue de la gestion des nombreux services indépendants.**
- La **latence des communications** entre les microservices peut également être un défi, car ils communiquent généralement via le réseau.
- Il est important de concevoir et **d'optimiser la communication** entre les microservices pour minimiser les retards.

2.5.7 AVANTAGES ET DÉFIS DE L'ARCHITECTURE MICROSERVICES



EXEMPLES CONCRETS D'UTILISATION : EXPEDIA



- *Dans le secteur des voyages, Expedia utilise une architecture microservices pour son système de réservation en ligne.*
- *Les avantages incluent une flexibilité accrue pour intégrer de nouveaux services et une évolutivité granulaire pour faire face à la demande fluctuante.*
- *Cependant, la gestion de nombreux microservices distincts peut introduire une complexité supplémentaire dans le système.*



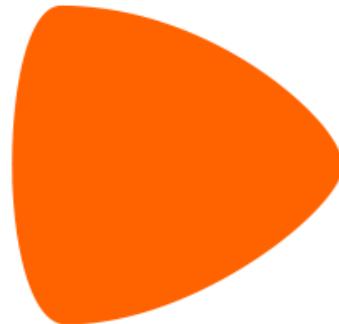
2.5.7 AVANTAGES ET DÉFIS DE L'ARCHITECTURE MICROSERVICES



EXEMPLES CONCRETS D'UTILISATION : ZALANDO



- *Zalando, une entreprise de vente de mode en ligne, utilise une architecture microservices pour son système de gestion des utilisateurs.*
- *Lorsque la charge augmente, ils mettent à l'échelle horizontalement leur microservice de gestion des utilisateurs en ajoutant automatiquement de nouvelles instances.*
- *Cela leur permet de maintenir des performances optimales et de répondre efficacement à la demande croissante des utilisateurs.*



zalando

2.5.7 AVANTAGES ET DÉFIS DE L'ARCHITECTURE MICROSERVICES



EXEMPLES CONCRETS D'APPLICATIONS



- *Exemple en Javascript/Scala* : <https://github.com/cer/microservices-examples>
- *Exemples en Java/Spring* : <https://github.com/eazybytes/microservices>
- *Exemple en Python* : <https://github.com/umermansoor/microservices>
- *Exemple en Go* : <https://github.com/raycad/go-microservices>

TRAVAUX PRATIQUES #3

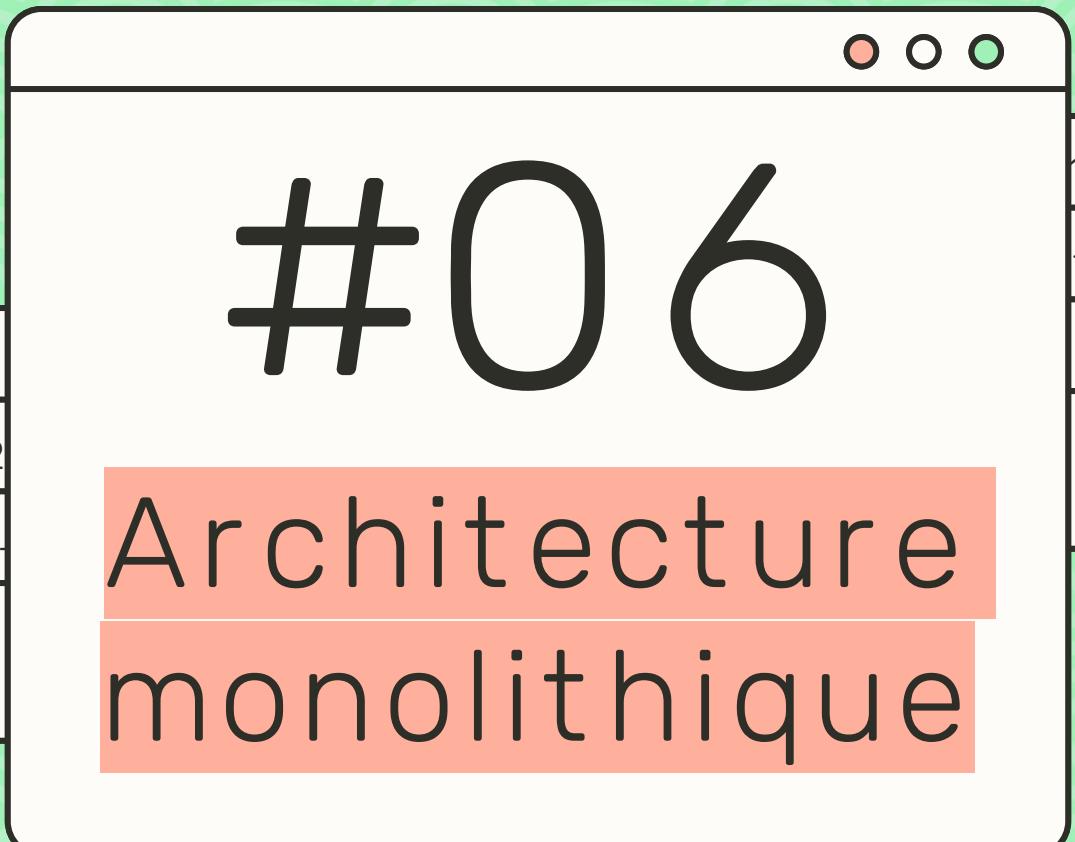
Vous pouvez faire le nouveau TP de ce cours qui porte particulièrement sur les styles architecturaux que nous venons de voir.



15, 2021 - 18H

1 - 15H





#06

Architecture monolithique

JU
JULY 11, 2
AUGUST 8, 2021
UR 15, 2021 - 15H ting with Company A

15, 2021 - 18H

1 - 15H



2.6.1 INTRODUCTION À L'ARCHITECTURE MONOLITHIQUE



- L'architecture monolithique désigne un style d'architecture logicielle où **une seule application** exécute toutes les tâches.
- Elle est comme un grand bloc unique, d'où son nom "**monolithique**".
- L'objectif est de garder toutes les fonctionnalités sous un même toit, facilitant la **coordination** et la **communication**.

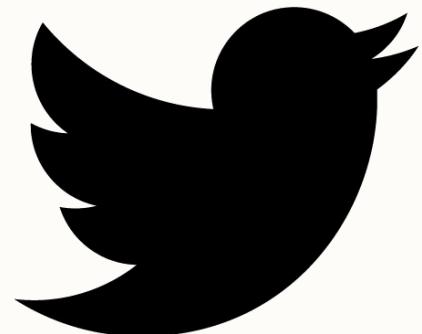
2.6.1 INTRODUCTION À L'ARCHITECTURE MONOLITHIQUE



EXEMPLES CONCRETS D'UTILISATION DE L'ARCHITECTURE



- *Prenons l'exemple de la première version de Twitter.*
- *C'était une application monolithique basée sur Ruby on Rails.*
- *Toutes les fonctionnalités, comme la publication de tweets, l'abonnement à des utilisateurs, et la visualisation de fils d'actualité, étaient gérées par une seule application avant de passer en microservices.*



2.6.2 CARACTÉRISTIQUES DE L'ARCHITECTURE MONOLITHIQUE

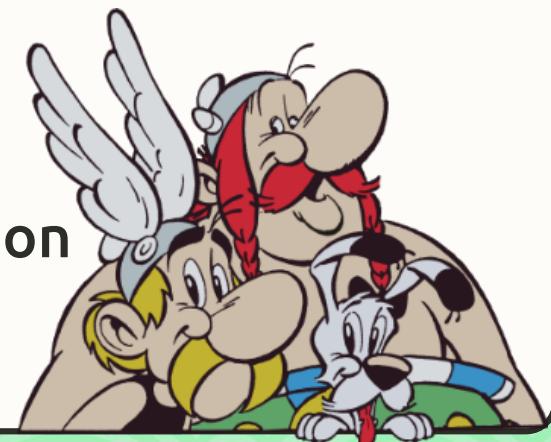
- Les architectures monolithiques sont caractérisées par leur base de code unique et l'interdépendance de leurs fonctionnalités.
- Cela signifie que toutes les tâches sont traitées en interne, sans la nécessité de communiquer avec d'autres applications ou services.

2.6.2 CARACTÉRISTIQUES DE L'ARCHITECTURE MONOLITHIQUE

MONOLITHE = UNE SEULE APPLICATION



- L'architecture monolithique est souvent la première architecture d'une application.
- Nous allons voir quelles en sont les raisons, et pourquoi c'est souvent un bon choix pour une première architecture.



2.6.2 CARACTÉRISTIQUES DE L'ARCHITECTURE MONOLITHIQUE

Simplicité et Rapidité de Développement :



- **Facilité de Démarrage** : Le démarrage avec une architecture monolithique est généralement simple et ne requiert pas une connaissance approfondie des microservices ou d'autres architectures distribuées.
- **Développement Rapide** : Tout étant en un seul endroit, il n'y a pas besoin de passer du temps à configurer les communications inter-services.
- **Déploiement Simplifié** : Il n'y a généralement pas besoin de systèmes complexes de gestion de conteneurs ou d'orchestration pour déployer un monolithe.

2.6.2 CARACTÉRISTIQUES DE L'ARCHITECTURE MONOLITHIQUE

Cohérence :

- **Unité de Code** : Les développeurs peuvent travailler sur une base de code unifiée sans avoir à jongler entre différents projets ou dépôts.
- **Cohérence de Données** : Les transactions de données sont souvent plus simples à gérer dans un système monolithique par rapport à un système distribué, où la cohérence des données peut être un défi.



2.6.2 CARACTÉRISTIQUES DE L'ARCHITECTURE MONOLITHIQUE

Gestion Facilitée :

- **Un Seul Point de Gestion** : La surveillance, le déploiement, et la gestion des monolithes est centralisée et souvent moins complexe que pour les microservices.
- **Pas de Latence de Communication** : Puisque toutes les fonctionnalités résident dans une seule *codebase*, il n'y a pas de problèmes de latence dus aux appels réseau entre services séparés.



2.6.2 CARACTÉRISTIQUES DE L'ARCHITECTURE MONOLITHIQUE

Testabilité :

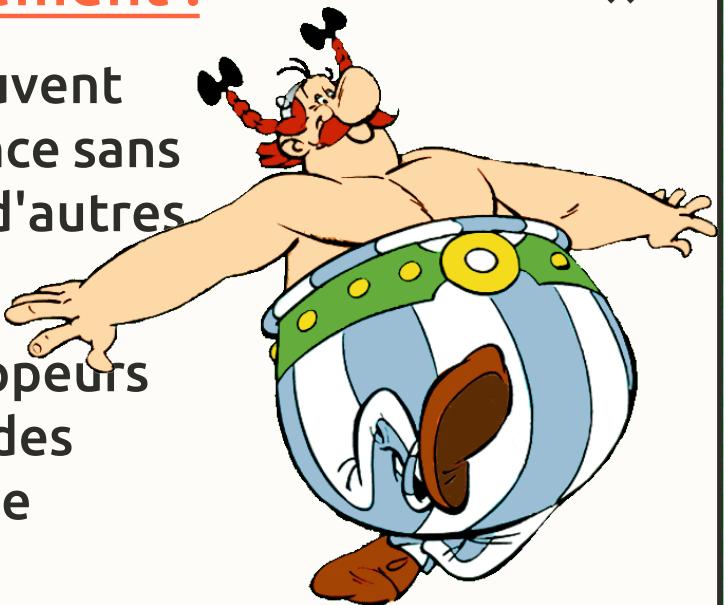
- **Tests Simplifiés** : Les tests peuvent être plus simples à réaliser dans une architecture monolithique car ils peuvent s'exécuter dans un seul environnement sans avoir à gérer les appels réseau ou les données distribuées.



2.6.2 CARACTÉRISTIQUES DE L'ARCHITECTURE MONOLITHIQUE

Efficacité de l'Équipe de Développement :

- **Moins de Coordination:** Les équipes peuvent travailler avec une certaine indépendance sans avoir à coordonner constamment avec d'autres équipes gérant d'autres services.
- **Compétences Généralistes:** Les développeurs n'ont pas besoin de se spécialiser dans des compétences de gestion de réseau ou de communications inter-services.



2.6.2 CARACTÉRISTIQUES DE L'ARCHITECTURE MONOLITHIQUE

Fiabilité :

- **Moins de Points de Défaillance :**
L'architecture monolithique présente moins de points de défaillance par rapport à une architecture microservices, où la défaillance d'un service peut avoir un effet domino.



2.6.2 CARACTÉRISTIQUES DE L'ARCHITECTURE MONOLITHIQUE

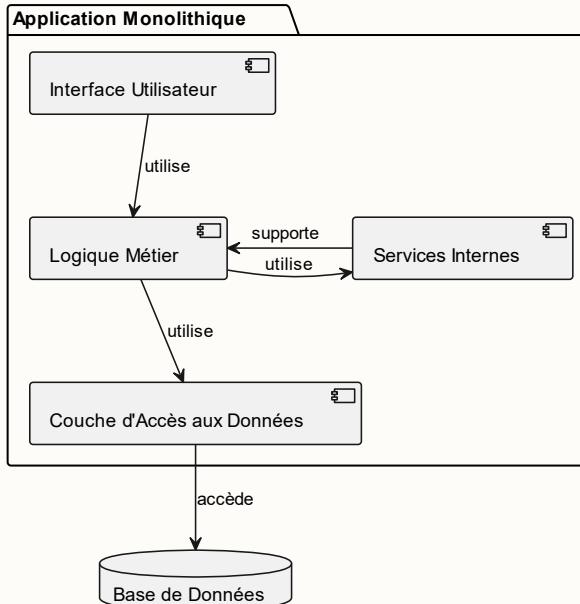


Diagramme UML représentant une architecture monolithique

2.6.2 CARACTÉRISTIQUES DE L'ARCHITECTURE MONOLITHIQUE

EXEMPLES CONCRETS D'UTILISATION DE L'ARCHITECTURE



- *Amazon, dans ses premières années, fonctionnait comme une application monolithique.*
- *Toutes les fonctionnalités, des recommandations de produits à la gestion des stocks et au traitement des paiements, étaient gérées par une seule application.*



2.6.3 CONCEPTION ET STRUCTURE D'UNE ARCHI. MONOLITHIQUE

✗

- La conception d'une architecture monolithique doit assurer la **cohésion forte** et le **couplage faible**.
- Chaque module doit être **bien organisé** et les **dépendances** entre les modules doivent être **minimisées**.

✗

2.6.3 CONCEPTION ET STRUCTURE D'UNE ARCHI. MONOLITHIQUE

✗

- Dans un monolithe, les modules partagent généralement des **ressources**, comme les **données**.
- La **communication est interne**, sans besoin de protocoles réseau complexes.
- Cela rend la **communication rapide et efficace**.

✗

2.6.4 DÉPLOIEMENT ET ÉVOLUTIVITÉ D'UNE ARCHI. MONO. ✘

- Pour déployer un monolithe, on crée généralement un seul artefact déployable, comme un fichier JAR pour une application Java. ✘
- Cet artefact est ensuite déployé sur un serveur ou une machine virtuelle.
- Si l'application doit être mise à l'échelle, on déploie généralement des copies de l'application sur plusieurs serveurs.

2.6.4 DÉPLOIEMENT ET ÉVOLUTIVITÉ D'UNE ARCHI. MONO.



EXEMPLES CONCRETS D'UTILISATION DE L'ARCHITECTURE



- *Netflix, avant de passer à une architecture de microservices, utilisait une architecture monolithique déployée sur des machines virtuelles dans le cloud d'Amazon.*



2.6.5 GESTION ET MAINTENANCE D'UNE ARCHI. MONOLITHIQUE

- La **maintenance** d'une application monolithique peut être **complexe**, en particulier pour les grandes applications.
- Les mises à jour du code et les corrections de bugs doivent être appliquées avec soin, car elles peuvent avoir des **effets sur l'ensemble de l'application**.

2.6.5 GESTION ET MAINTENANCE D'UNE ARCHI. MONOLITHIQUE

- Dans un monolithe, une modification du code nécessite généralement de **redéployer l'ensemble de l'application.**
- La **surveillance des performances** est cruciale pour s'assurer que l'application fonctionne correctement et pour identifier les éventuels **goulots d'étranglement.**

2.6.6 AVANTAGES ET INCONVÉNIENTS DE L'ARCHI. MONOLITHIQUE



- Les architectures monolithiques sont souvent plus simples à comprendre et à développer, car tout le code se trouve au même endroit.
- De plus, comme la communication entre les modules se fait en interne, les performances sont généralement meilleures que dans les architectures distribuées.



2.6.6 AVANTAGES ET INCONVÉNIENTS DE L'ARCHI. MONOLITHIQUE



- L'inconvénient majeur des architectures monolithiques est leur **complexité croissante** avec l'ajout de nouvelles fonctionnalités.
- De plus, la **mise à l'échelle** est souvent plus difficile, car elle nécessite de dupliquer l'application entière, ce qui peut être **coûteux en ressources**, et parfois difficile à développer.



2.6.6 AVANTAGES ET INCONVÉNIENTS DE L'ARCHI. MONOLITHIQUE



EXEMPLES CONCRETS D'UTILISATION DE L'ARCHITECTURE



- *Reprendons l'exemple d'un service de streaming comme Netflix.*
- *Dans une architecture monolithique, si le nombre d'utilisateurs augmente rapidement, il peut être difficile d'ajuster rapidement les ressources pour maintenir des performances élevées.*



2.6.7 EXEMPLES D'UTILISATION DE L'ARCHITECTURE MONOLITHIQUE



EXEMPLES CONCRETS D'UTILISATION DE L'ARCHITECTURE



- *L'architecture monolithique est souvent le choix par défaut pour les petites à moyennes applications, car elle est plus simple à mettre en œuvre.*
- *Cependant, pour les grandes applications avec de nombreux utilisateurs, une architecture plus flexible et évolutive peut devenir nécessaire.*

2.6.7 EXEMPLES D'UTILISATION DE L'ARCHITECTURE MONOLITHIQUE



EXEMPLES CONCRETS D'UTILISATION DE L'ARCHITECTURE



- *WordPress est un exemple d'un système de gestion de contenu basé sur une architecture monolithique.*
- *Toutes les fonctionnalités, comme la gestion des articles, des commentaires et des utilisateurs, sont gérées par une seule application.*



2.6.7 EXEMPLES D'UTILISATION DE L'ARCHITECTURE MONOLITHIQUE



EXEMPLES CONCRETS D'UTILISATION DE L'ARCHITECTURE



- *Etsy est un exemple d'un site de vente en ligne basé sur une architecture monolithique.*
- *Toutes les fonctionnalités, comme la liste des produits, la gestion des stocks et le traitement des commandes, sont également gérées par une seule application.*



2.6.7 EXEMPLES D'UTILISATION DE L'ARCHITECTURE MONOLITHIQUE



EXEMPLES D'APPLICATIONS MONOLITHIQUES



- *Application Java/Spring :*
<https://github.com/mzubal/spring-boot-monolith>
- *Application en PHP :*
<https://github.com/uitsmijter/example-todo-php-application>
- *Application en .NET :*
<https://github.com/valeravorobjev/monolith>

TRAVAUX PRATIQUES #4

Vous pouvez faire le nouveau TP de ce cours qui porte particulièrement sur les styles architecturaux que nous venons de voir.



15, 2021 - 18H

1 - 15H



#07

Clean Architecture

JU
JULY 11, 2021
AUGUST 8, 2021
RIL 15, 2021 - 15H
ting with Company A

15, 2021 - 18H

1 - 15H



×



2.7.1 NAISSANCE DE L'ARCHITECTURE CLEAN

*

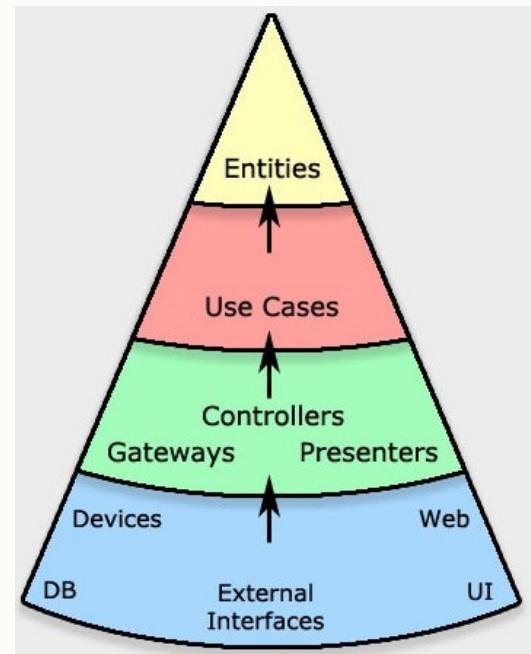
- L'architecture **Clean**, aussi appelée *architecture oignon*, vise à rendre les systèmes **plus lisibles, flexibles et indépendants** des **détails de l'infrastructure**.
- Le but est de rendre le **code source facile à comprendre, modifier et tester**.

*

2.7.1 NAISSANCE DE L'ARCHITECTURE CLEAN

*

- L'architecture monolithique contient tout le code en un seul endroit, alors que l'architecture de microservices divise le code en services distincts.
- L'architecture Clean quant à elle sépare les règles métier des détails de l'infrastructure.



2.7.1 NAISSANCE DE L'ARCHITECTURE CLEAN



- L'architecture Clean a été créée par **Robert C. Martin**, également connu sous le nom d'Uncle Bob.
- Martin est un **ingénieur logiciel**, **consultant** et **auteur renommé**, avec une longue carrière dans le secteur du logiciel.



2.7.1 NAISSANCE DE L'ARCHITECTURE CLEAN



- Années 1990 et 2000 :
 - Martin a commencé à parler de nombreux **principes de conception** et de développement logiciel qui allaient devenir la **base de l'architecture Clean**.
- *Par exemple, il a co-écrit le "Manifeste Agile" en 2001, qui met l'accent sur la simplicité, la flexibilité et la collaboration.*



2.7.1 NAISSANCE DE L'ARCHITECTURE CLEAN



- 2002 :
 - Martin a formulé les principes **SOLID**, un ensemble de cinq principes de conception orientée objet qui sont essentiels pour l'architecture Clean.
 - *Nous reviendrons sur ces principes plus tard dans le cours.*



2.7.1 NAISSANCE DE L'ARCHITECTURE CLEAN

*

- 2012 :

- Martin a commencé à **décrire et à promouvoir** explicitement l'architecture Clean.
- Il a écrit une série de **posts** de blog expliquant les **concepts** et les **avantages** de l'architecture Clean, et comment elle s'inscrit dans la conception de logiciels modernes.

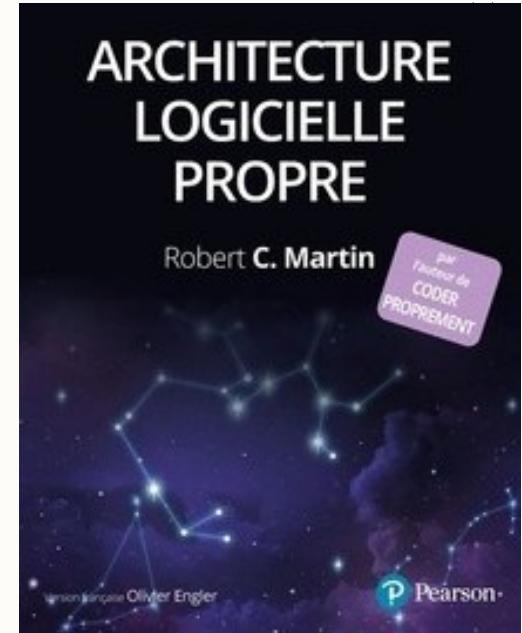


*

2.7.1 NAISSANCE DE L'ARCHITECTURE CLEAN



- Depuis 2012 :
 - L'architecture Clean a gagné en popularité, en particulier parmi les développeurs qui cherchent à créer des logiciels plus modulaires, testables et indépendants de l'infrastructure.
 - Martin a continué à promouvoir l'architecture Clean, notamment dans son livre "Clean Architecture: A Craftsman's Guide to Software Structure and Design", publié en 2017.



2.7.1 NAISSANCE DE L'ARCHITECTURE CLEAN



EXEMPLES CONCRETS D'UTILISATION DE L'ARCHITECTURE

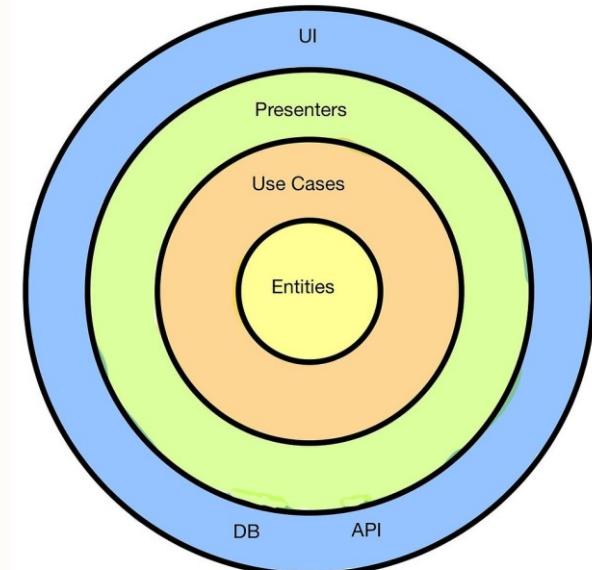


- *Prenons l'exemple d'une application mobile pour les commandes de restaurant.*
- *Dans une architecture Clean, le code qui gère les commandes (règles métier) est distinct du code qui interagit avec l'interface utilisateur ou la base de données.*
- *Cela permet de modifier l'interface utilisateur ou de changer de base de données sans affecter le code des règles métier.*

2.7.2 CARACTÉRISTIQUES DE L'ARCHITECTURE CLEAN



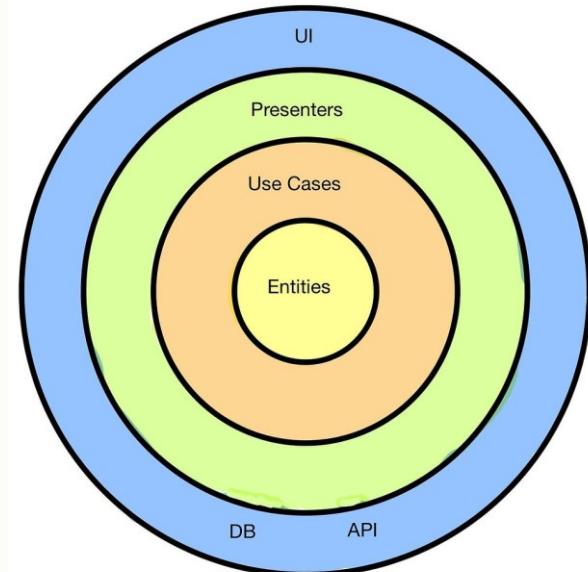
Les principales caractéristiques de l'architecture Clean sont la séparation des préoccupations et l'indépendance par rapport aux détails de l'infrastructure.



2.7.2 CARACTÉRISTIQUES DE L'ARCHITECTURE CLEAN



- Les composants clés de l'architecture Clean sont :
 - les entités
 - les cas d'utilisation
 - les adaptateurs d'interface
 - et les frameworks et pilotes



2.7.2 CARACTÉRISTIQUES DE L'ARCHITECTURE CLEAN



Entities (Entités) :



- Ce sont les objets métier fondamentaux de l'application.
- Ils contiennent la logique métier la plus pure et sont indépendants de toute technologie ou framework.
- Par exemple, une règle de gestion comme "un utilisateur ne peut pas commander si son compte est désactivé" serait ici.

2.7.2 CARACTÉRISTIQUES DE L'ARCHITECTURE CLEAN



Use Cases (Cas d'utilisation) :



- Ils orchestrent les entités pour répondre à des besoins spécifiques du domaine métier.
- Ils définissent ce que fait le système pour répondre à une action utilisateur.
- Cette couche ne dépend ni de l'interface utilisateur ni des détails techniques.

2.7.2 CARACTÉRISTIQUES DE L'ARCHITECTURE CLEAN



Interface Adapters (Adaptateurs d'interface) :

- Ces composants traduisent les données des cas d'utilisation en formats compatibles avec l'interface utilisateur (ou avec une base de données).
- Par exemple, un adaptateur pourrait convertir un objet métier en JSON pour une API REST.



2.7.2 CARACTÉRISTIQUES DE L'ARCHITECTURE CLEAN



Frameworks and Drivers (Frameworks et moteurs) :

- La couche la plus externe, elle contient tout ce qui est lié à des technologies spécifiques (frameworks web, bases de données, etc.).
- Ces éléments doivent rester **remplaçables et dépendants des abstractions des couches internes.**



2.7.2 CARACTÉRISTIQUES DE L'ARCHITECTURE CLEAN



- Dans l'architecture Clean, le code est organisé en **couches concentriques**.
- Les **règles métier sont au centre** et sont entourées par les **cas d'utilisation**, les **adaptateurs d'interface**, et enfin les **frameworks et pilotes**.
- *C'est comme les couches d'un oignon, d'où le nom d'architecture en oignon.*

2.7.2 CARACTÉRISTIQUES DE L'ARCHITECTURE CLEAN

EXEMPLES CONCRETS D'UTILISATION DE L'ARCHITECTURE



Prenons l'exemple d'une application de gestion de commandes :

- **Entities** : *Order, Product, Customer.*
- **Règles métier** : *"Un client doit avoir un solde positif pour commander".*
- **Use Cases** : *PlaceOrder, CancelOrder.*
- **Cas d'utilisation** : *Vérifie les règles métier et exécute la commande si elles sont validées.*
- **Interface Adapters** : *Contrôleurs API REST, transformateurs JSON, DTO.*
- **Frameworks and Drivers** : *Framework web (Spring Boot), ORM (Hibernate), base de données (PostgreSQL).*

2.7.2 CARACTÉRISTIQUES DE L'ARCHITECTURE CLEAN

EXEMPLES CONCRETS D'UTILISATION DE L'ARCHITECTURE



- *Dans le cadre de cette application, le code qui gère le panier d'achats et les commandes serait au centre de l'architecture.*
- *Le code qui gère l'interface utilisateur (comme l'affichage des produits), les frameworks et la base de données (via l'ORM) serait à l'extérieur.*



e-commerce

2.7.3 CONCEPTION ET STRUCTURE D'UNE ARCHITECTURE CLE~~X~~N

- Les **principes de conception** pour une architecture Clean incluent :
 - l'**encapsulation** (*cacher les détails de mise en œuvre*),
 - le **faible couplage** (*minimiser les dépendances entre les parties du code*),
 - la **cohésion forte** (*regrouper ensemble ce qui est lié fonctionnellement*),
 - et l'**abstraction** (*représenter les concepts de haut niveau sans détails spécifiques*).

2.7.3 CONCEPTION ET STRUCTURE D'UNE ARCHITECTURE CLEAN

- Les **principes SOLID** sont des principes de conception de logiciels orientés objets qui aident à rendre le code plus **compréhensible, flexible et maintenable**.
- Ils s'appliquent particulièrement bien à l'architecture Clean.
- **Nous verrons en détails ces principes dans la suite du cours**

2.7.3 CONCEPTION ET STRUCTURE D'UNE ARCHITECTURE CLE~~X~~N



SOLID Principles

- S** Single responsibility principle
- O** Open/closed principle
- L** Liskov substitution principle
- I** Interface segregation principle
- D** Dependency inversion principle

2.7.3 CONCEPTION ET STRUCTURE D'UNE ARCHITECTURE CLEAN

- Dans une architecture Clean, le code est organisé en fonction de son niveau d'abstraction.
- Les entités et les cas d'utilisation, qui représentent les concepts de haut niveau et les actions métier, sont au centre de l'architecture.
- Les adaptateurs d'interface et les frameworks et pilotes, qui gèrent les détails de l'infrastructure, sont à l'extérieur.

2.7.3 CONCEPTION ET STRUCTURE D'UNE ARCHITECTURE CLE~~X~~N

La règle fondamentale de la Clean Architecture est que les dépendances pointent toujours vers l'intérieur : ✖

- Les couches internes (*entités, use cases*) ne connaissent rien des couches externes (frameworks, adaptateurs).
- Les couches externes dépendent des abstractions définies dans les couches internes, et non l'inverse.

Cela garantit que les parties critiques du système (la *logique métier*) ne sont pas affectées par les changements de détails techniques.

2.7.3 CONCEPTION ET STRUCTURE D'UNE ARCHITECTURE CLEAN

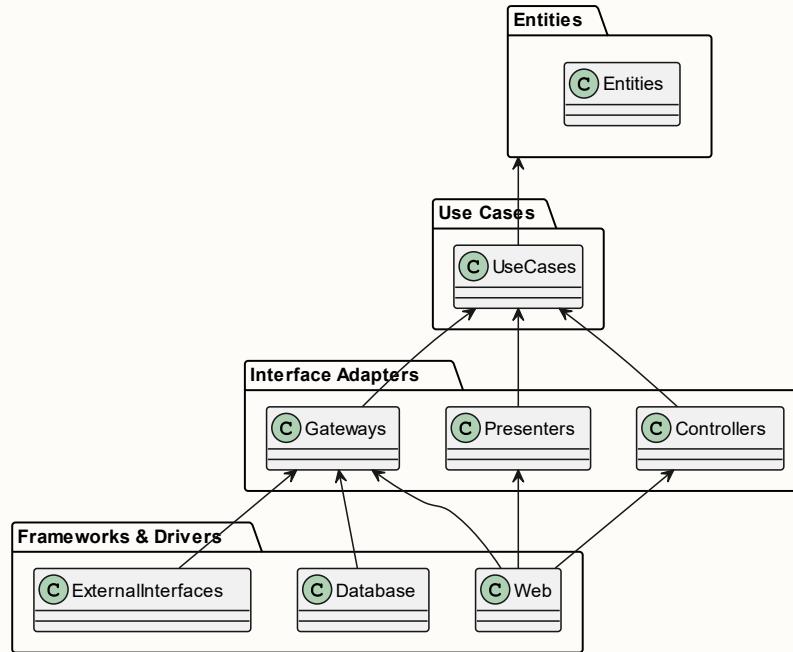


Diagramme représentant une architecture Clean en UML

2.7.3 CONCEPTION ET STRUCTURE D'UNE ARCHITECTURE CLE~~N~~N

EXEMPLES CONCRETS D'UTILISATION DE L'ARCHITECTURE



- *Dans une application de gestion de projet, le code qui gère les projets et les tâches serait au centre de l'architecture.*
- *Le code qui gère l'interface utilisateur (comme l'affichage des projets et des tâches) et la base de données (comme le stockage des projets et des tâches) serait à l'extérieur.*

2.7.4 ARCHITECTURE CLEAN & AUTRES STYLES



- Combiner l'architecture Clean avec divers styles architecturaux tels que SOA (*Service-Oriented Architecture*), microservices, client/serveur et l'architecture monolithique peut nécessiter une approche stratégique qui allie les principes fondamentaux de chaque style.
- Voici une perspective générale sur la manière dont cela pourrait être réalisé

2.7.4 ARCHITECTURE CLEAN & AUTRES STYLES



Clean Architecture & Client/Serveur:



- **Séparation des préoccupations** : Le modèle client/serveur est naturellement aligné avec l'architecture Clean en ce sens que les préoccupations client et serveur peuvent être nettement séparées et communiquer via des API ou des protocoles clairs.
- **Cohérence des données** : Les opérations du serveur peuvent être construites en utilisant l'architecture Clean pour assurer la cohérence, la sécurité et l'intégrité des données.

2.7.4 ARCHITECTURE CLEAN & AUTRES STYLES



Clean Architecture & SOA :



- **Services Indépendants** : Les services dans SOA peuvent être conçus en suivant les principes de l'architecture Clean, où chaque service a une structure interne propre, divisée en couches.
- **Responsabilité Unique** : Chaque service devrait avoir une seule responsabilité, ce qui est aligné avec le principe SOLID de la **responsabilité unique**.

2.7.4 ARCHITECTURE CLEAN & AUTRES STYLES



Clean Architecture & Microservices:



- **Modularité** : Les microservices encouragent une forte modularité, ce qui se marie bien avec l'architecture Clean où différentes parties du système (entités, use-cases, UI, etc.) sont séparées en couches distinctes.
- **Isolation** : Les microservices peuvent être développés de manière à suivre les principes de l'architecture Clean au niveau micro, isolant au niveau de chaque microservice la logique métier de l'infrastructure et du framework.

2.7.4 ARCHITECTURE CLEAN & AUTRES STYLES



Clean Architecture & Architecture monolithique :



- **Couplage faible** : Bien que les architectures monolithiques soient souvent associées à un **couplage fort**, l'architecture Clean peut aider à définir des frontières claires entre différents composants, même dans un *codebase* uniifié.
- **Testabilité** : Les principes de l'architecture Clean peuvent augmenter la testabilité d'une application monolithique en isolant la logique métier des dépendances externes.

2.7.5 AVANTAGES ET INCONVÉNIENTS DE L'ARCHITECTURE CLEAN ✖

- L'architecture Clean offre plusieurs avantages, notamment :
 - la modularité (*le code est divisé en parties indépendantes qui peuvent être modifiées séparément*),
 - la testabilité (*le code peut être testé facilement*),
 - et l'indépendance par rapport aux détails de l'infrastructure (*le code des règles métier n'est pas affecté par les changements dans l'infrastructure*).

2.7.5 AVANTAGES ET INCONVÉNIENTS DE L'ARCHITECTURE CLEAN ✖

- Les frameworks (comme Spring, Django, Angular, etc.) sont considérés comme des **outils** et non des piliers structurels.
- Une application Clean Architecture peut fonctionner sans dépendre d'un framework particulier.
- Les frameworks peuvent être intégrés dans la couche la plus externe, mais ils ne doivent pas imposer leur organisation au reste de l'application.

2.7.5 AVANTAGES ET INCONVÉNIENTS DE L'ARCHITECTURE CLEAN ✖

- En isolant les responsabilités et en réduisant les dépendances directes entre les couches, il devient beaucoup plus facile de tester chaque partie du système.
- Par exemple : *Les entités et cas d'utilisation peuvent être testés sans avoir besoin de base de données ni de serveur.*
- Les tests unitaires et d'intégration sont donc plus simples et plus fiables.

2.7.5 AVANTAGES ET INCONVÉNIENTS DE L'ARCHITECTURE CLEAN ✖

- **Un des objectifs principaux de la Clean Architecture est de minimiser l'impact des changements.**
- **Elle protège contre :**
 - Les changements dans l'interface utilisateur : *Par exemple, remplacer un frontend React par Angular.*
 - Les changements dans les technologies de persistance : *Par exemple, passer d'une base MySQL à MongoDB.*
 - Les nouvelles règles métier : Les entités et cas d'utilisation sont facilement adaptables car découplés des détails techniques.

2.7.5 AVANTAGES ET INCONVÉNIENTS DE L'ARCHITECTURE CLEAN ✖

• Les inconvénients potentiels de l'architecture Clean comprennent :

- la complexité initiale (*la conception et l'implémentation de l'architecture peuvent être complexes*)
- et la nécessité de discipline (*il faut respecter les principes de l'architecture pour en tirer les bénéfices*).

2.7.5 AVANTAGES ET INCONVÉNIENTS DE L'ARCHITECTURE CLEAN ✖

- Enfin, comme on l'a vu, l'architecture Clean repose sur des principes rigoureux qui se combinent très bien avec les principaux autres styles architecturaux.
- Quand vous choisissez un style architectural pour votre application, vous pourrez souvent lui ajouter cette dimension "*d'artisanat*" (craftsmanship) provenant du respect des principes de l'architecture Clean.



2.7.6 PHILOSOPHIE DE L'ARCHITECTURE CLEAN



- **La structure prime sur les détails** : Les détails techniques (frameworks, bases de données) sont secondaires et adaptables.
- **Les règles métier sont au centre** : Tout tourne autour de la logique métier, qui est protégée de l'influence des détails externes.
- **Code indépendant et remplaçable** : Chaque couche peut évoluer sans affecter les autres.



2.7.7 EXEMPLES D'UTILISATION DE L'ARCHITECTURE CLEAN



EXEMPLES D'APPLICATIONS CLEAN



- *Boilerplate ASP.NET :* <https://github.com/iammukeshm/CleanArchitecture.WebApi>
- *Exemple en .NET :* <https://github.com/matthewrenze/clean-architecture-demo>
- *Exemple en Go :* <https://github.com/eminetto/clean-architecture-go-v2>
- *Boilerplate en Go :* <https://github.com/bmf-san/go-clean-architecture-web-application-boilerplate>
- *Exemple Node.js :* <https://github.com/jbuget/nodejs-clean-architecture-app>

TRAVAUX PRATIQUES #5

Vous pouvez faire le nouveau TP de ce cours qui porte particulièrement sur le style architectural que nous venons de voir.



15, 2021 - 18H

1 - 15H



TRAVAUX PRATIQUES #7

Vous pouvez faire le nouveau TP de ce cours qui porte particulièrement sur le style architectural que nous venons de voir.



15, 2021 - 18H

1 - 15H



JUNE 15, 2021 - 15H

JUNE 15, 2021 - 15H

// Architecture Logicielle



Merci pour votre attention !

Avez-vous des questions ?

contact@astroware-conception.com

www.astroware-conception.com



www.linkedin.com/in/terence-ferut/

1 - 18H

JULY 11, 2021 - 11H



AUGUST 8, 2021 - 16H



JUNE 15, 2021 - 15H

JUNE 15, 2021 - 15H

