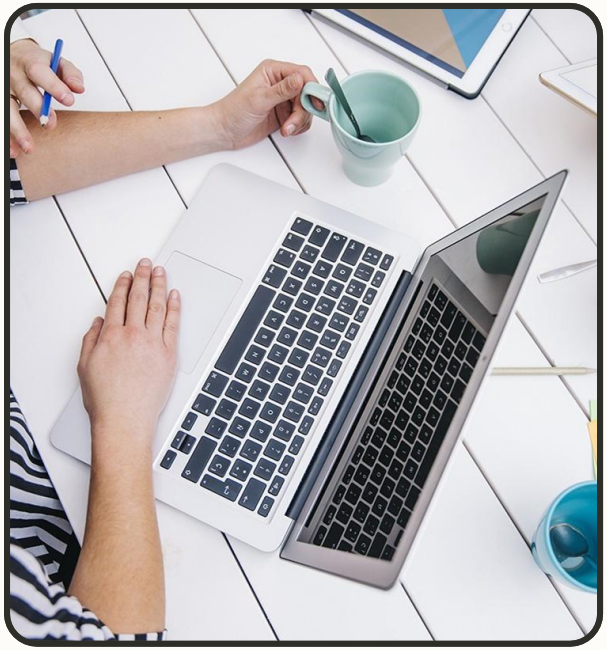
// Architecture Logicielle

ARCHITECTURE LOGICIELLE

Construisez des systèmes logiciels puissants avec une architecture intelligente

Térence FERUT



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)

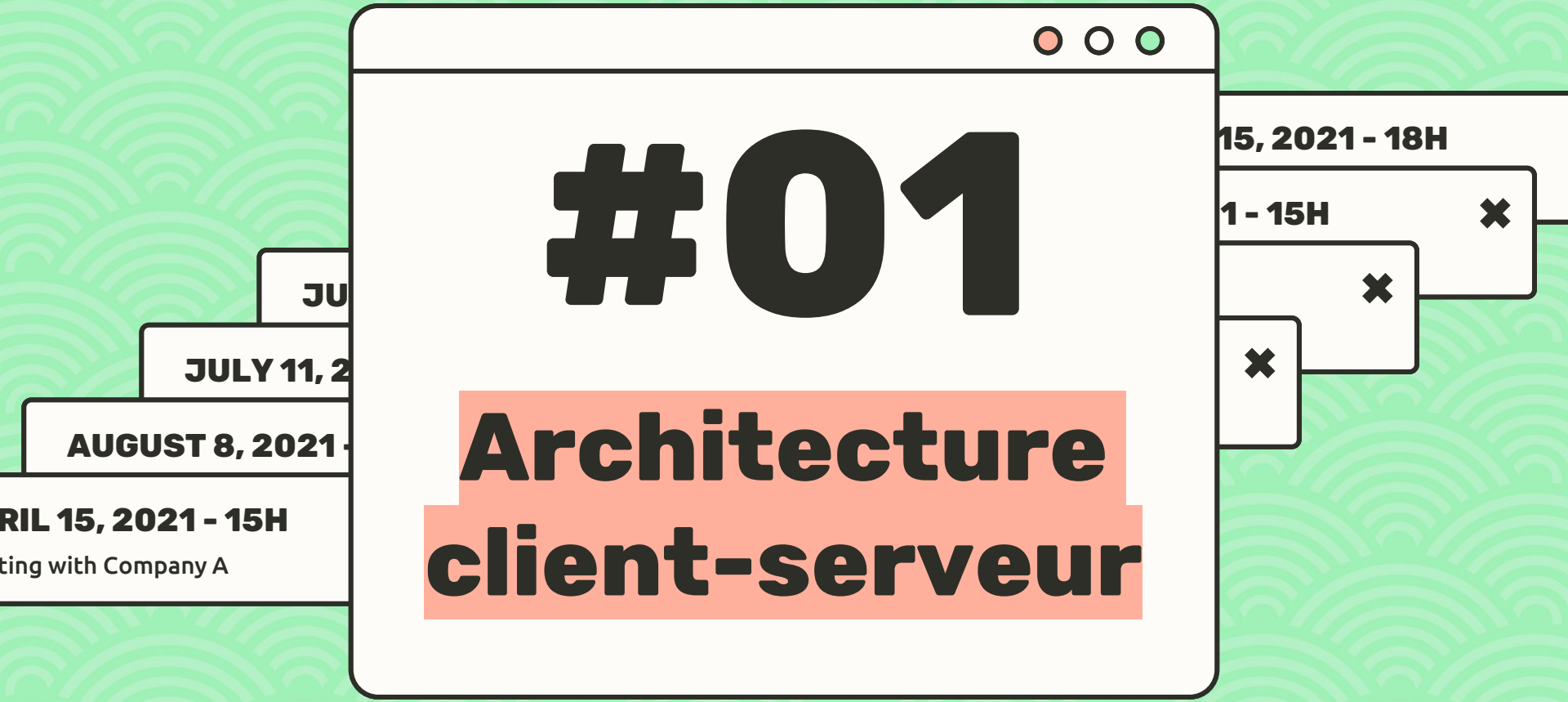
Plan du cours Styles d'Architecture Logicielle

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
15, 2021 - 18H
1 - 15H
JULY 11, 2
AUGUST 8, 2021 -
RIL 15, 2021 - 15H
ting with Company A

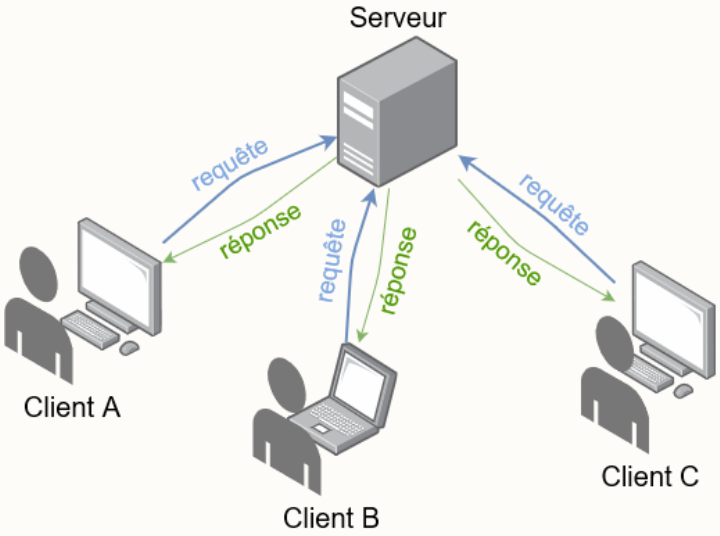
2.1.1 Introduction à l'architecture client-serveur

- 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.*



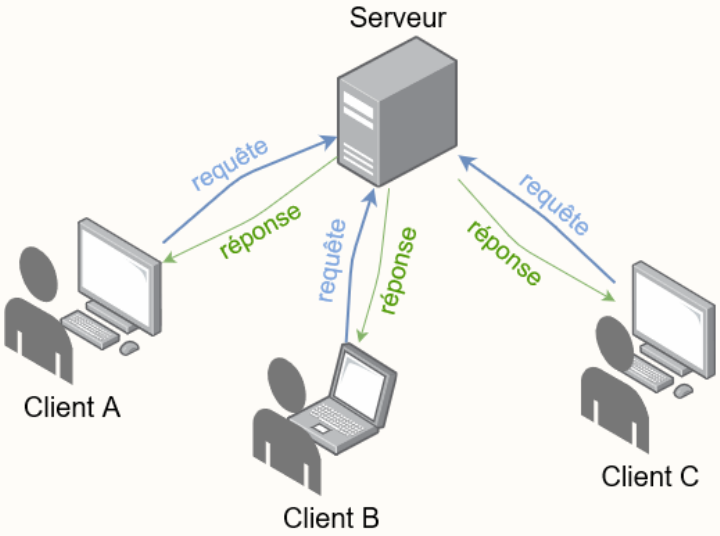
2.1.2 Caractéristiques de l'architecture client-serveur

- 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-serveur

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

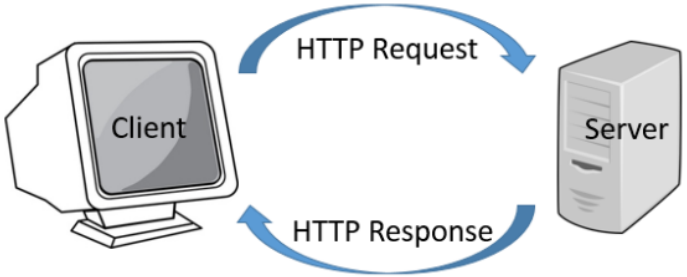
Exercice - Réflexion

- *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 ?*

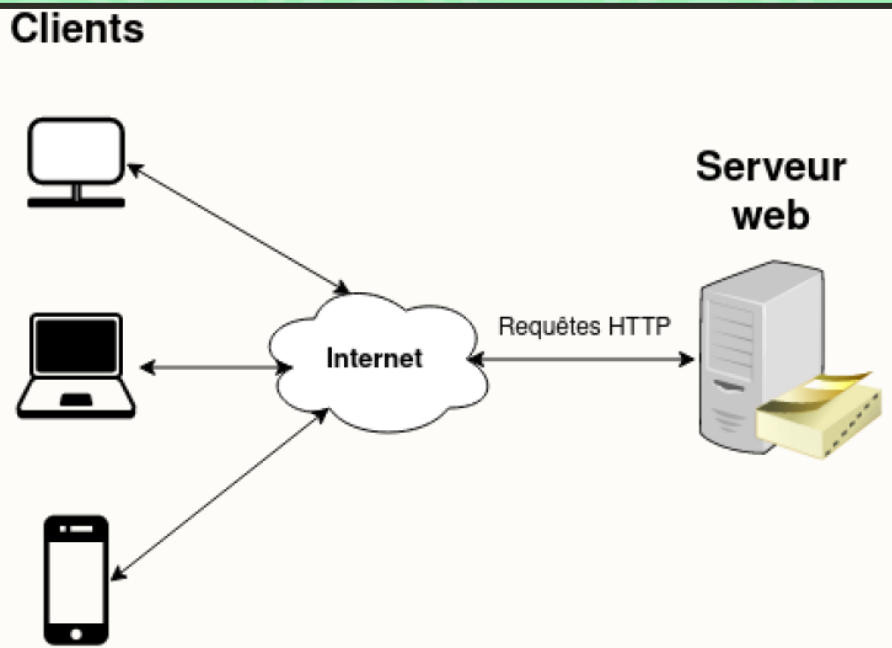
2.1.4 Communication entre le client et le serveur

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.*

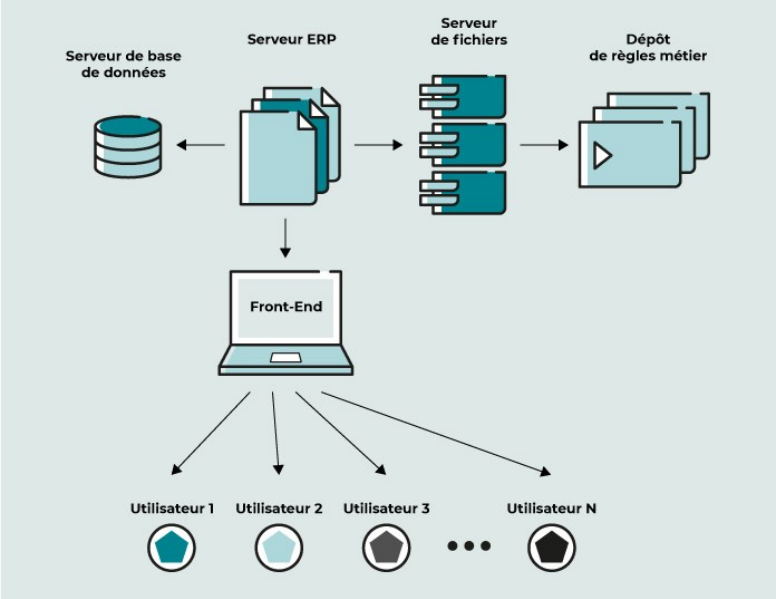


2.1.5 Déploiement et évolutivité



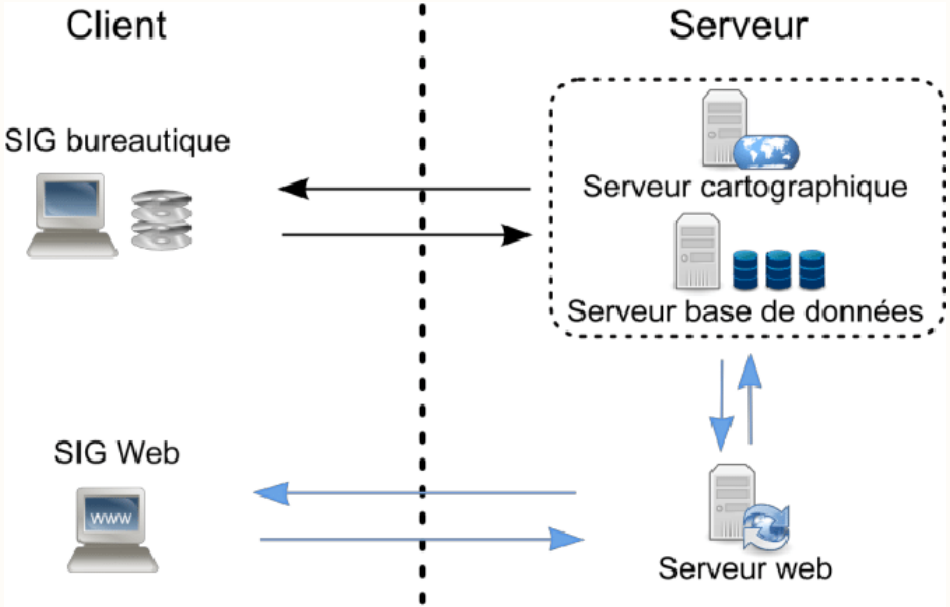
Exemple de diagramme d'architecture

2.1.5 Déploiement et évolutivité



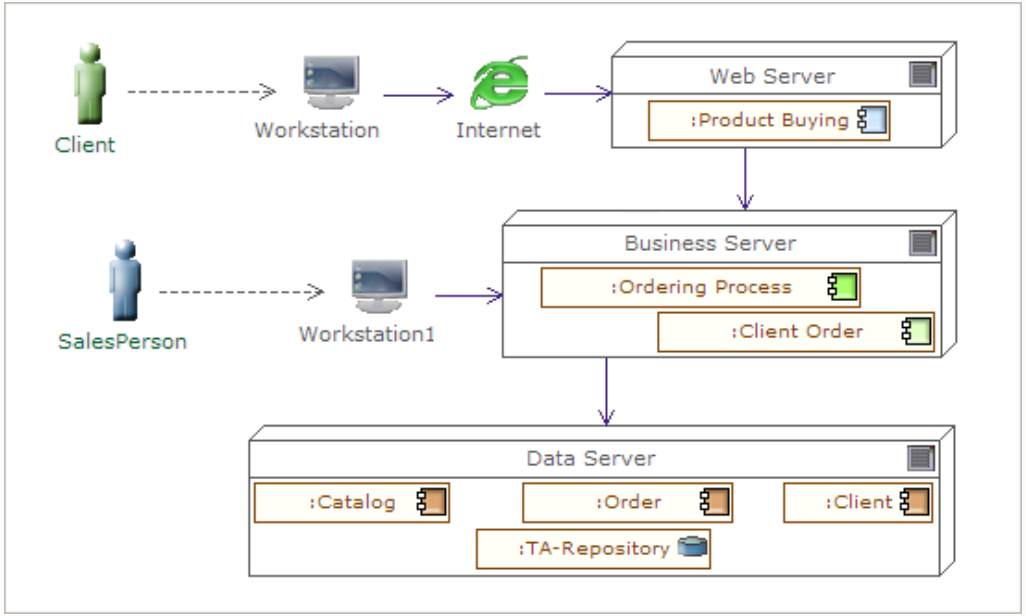
Exemple de diagramme d'architecture

2.1.5 Déploiement et évolutivité



Exemple de diagramme d'architecture

2.1.5 Déploiement et évolutivité

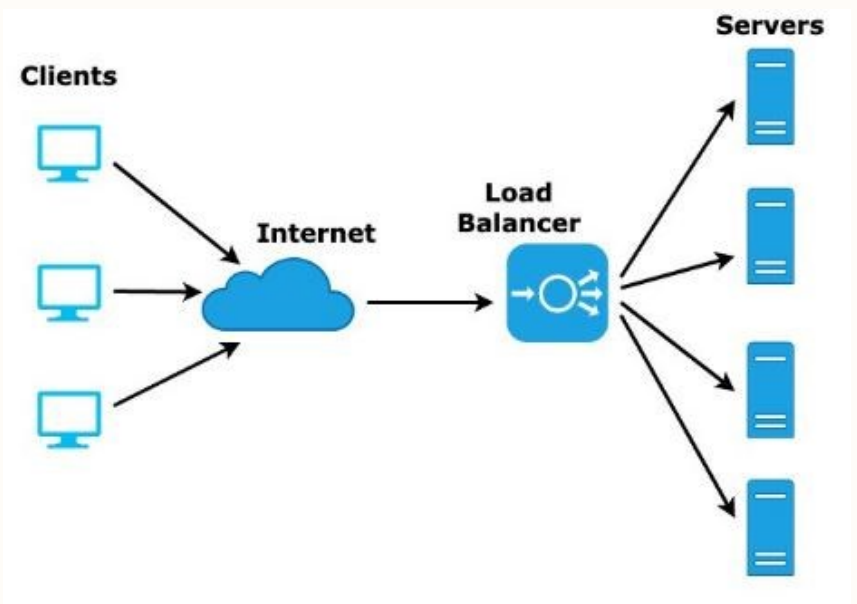


Exemple de diagramme d'architecture client-serveur

2.1.5 Déploiement et évolutivité

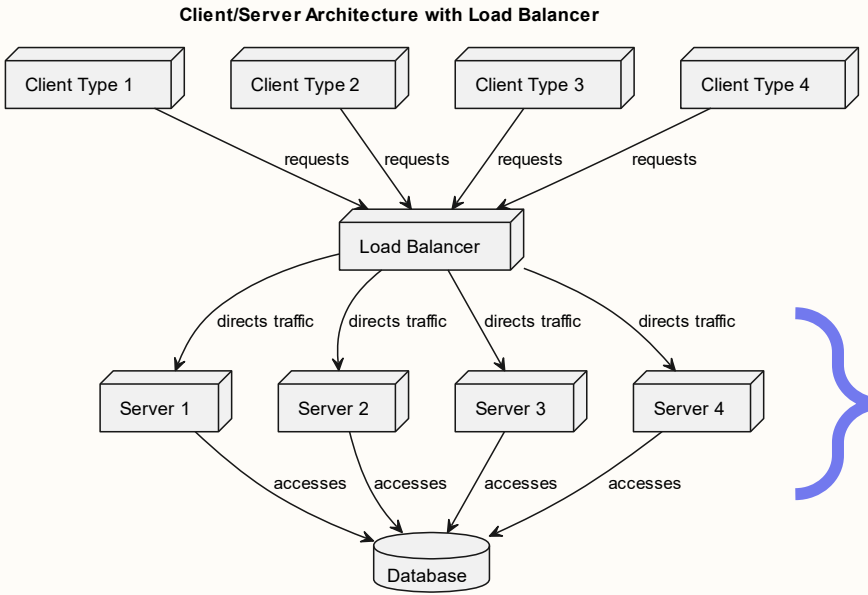
- L'architecture client-serveur permet d'augmenter la capacité du système en ajoutant des serveurs.
- Cela est connu sous le nom de mise à l'échelle horizontale.
- Contrairement à la mise à l'échelle verticale qui correspondrait à augmenter la puissance des serveurs

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

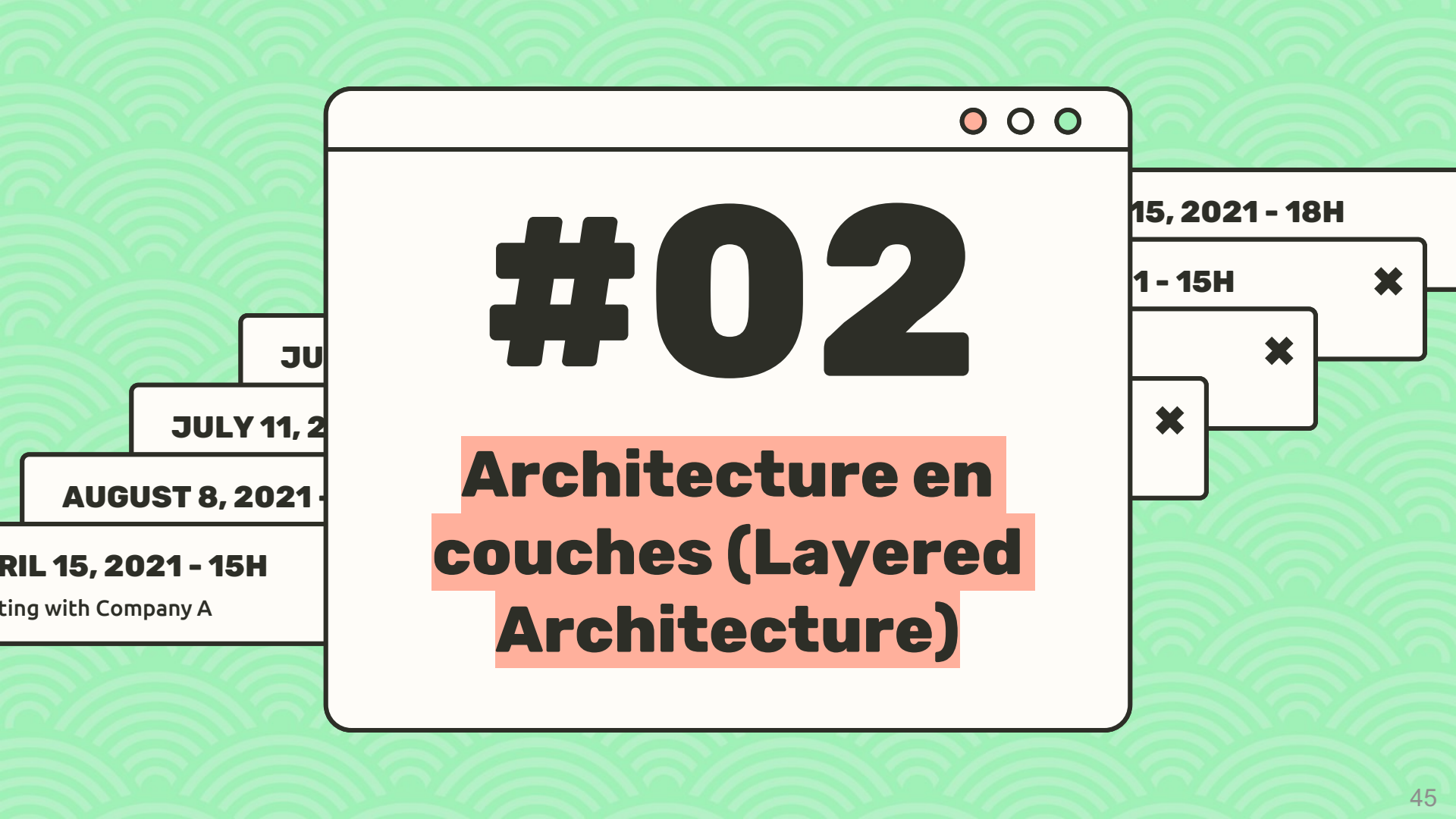
Exercice - Réflexion

- *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
JULY 11, 2
AUGUST 8, 2021 -
RIL 15, 2021 - 15H
ting with Company A

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.**

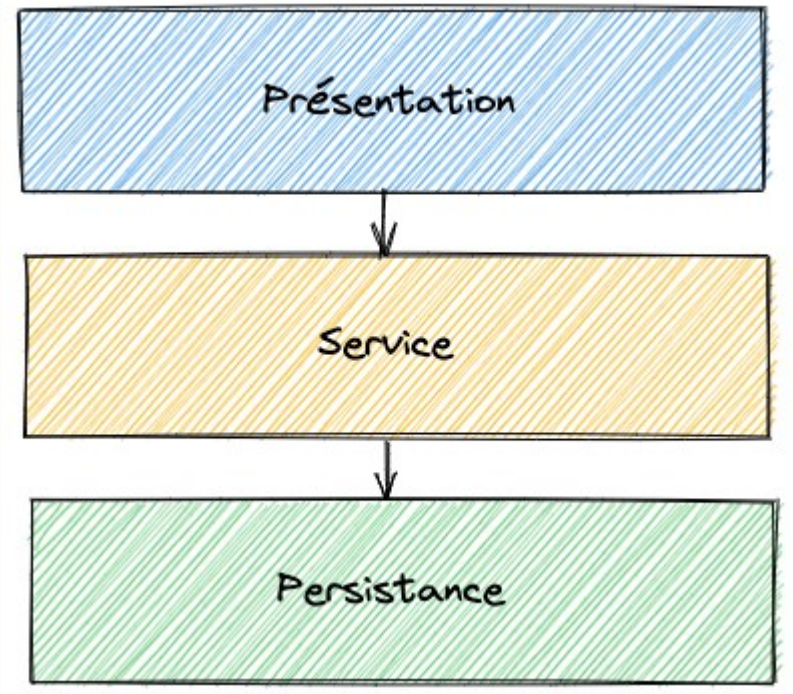
2.2.1 Introduction à l'architecture en couches

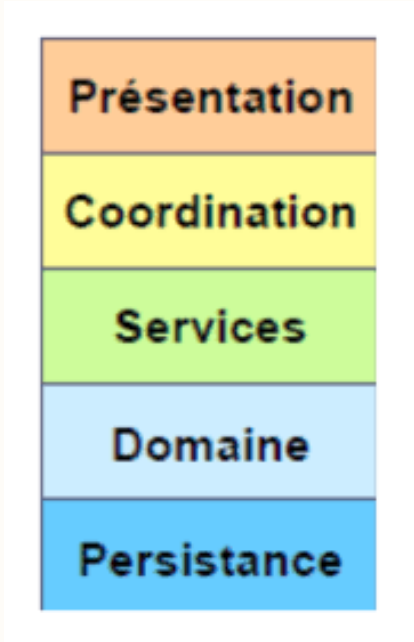
- L'architecture en couches repose sur le principe de séparation des responsabilités.
- Chaque couche est **responsable** d'une partie spécifique du système, ce qui facilite la gestion des fonctionnalités, des règles métier et de l'interaction avec les données.

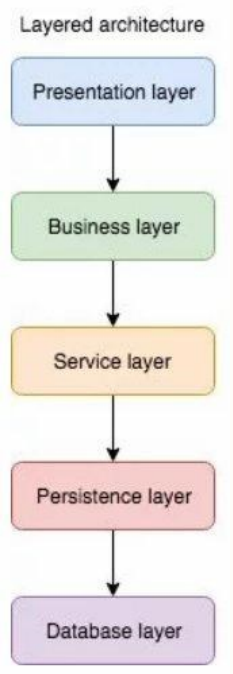
2.2.1 Introduction à l'architecture en couches

- **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 sûre 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.*

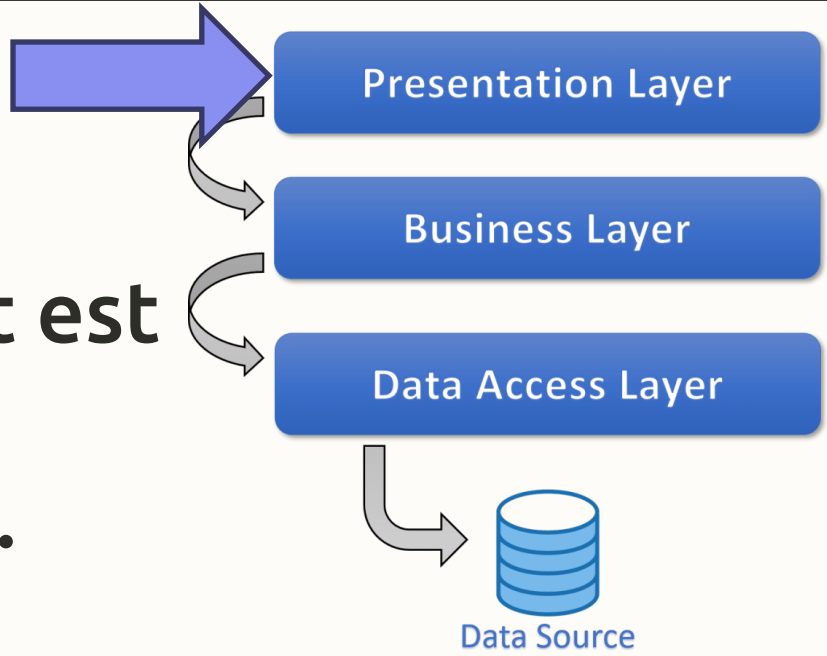
2.2.2 Couches de l'architecture en couches

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.*

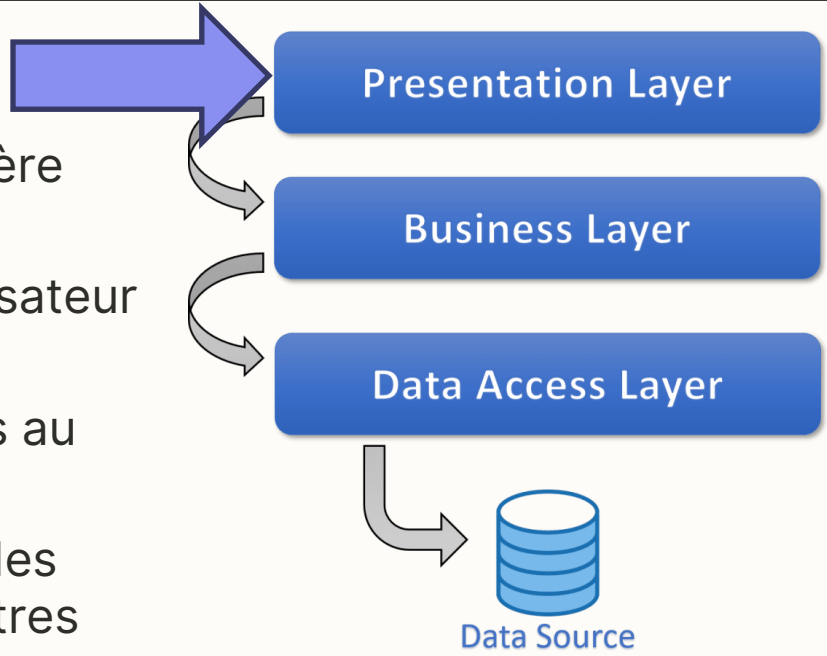
2.2.3 Responsabilités de chaque couche

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



2.2.3 Responsabilités de chaque couche

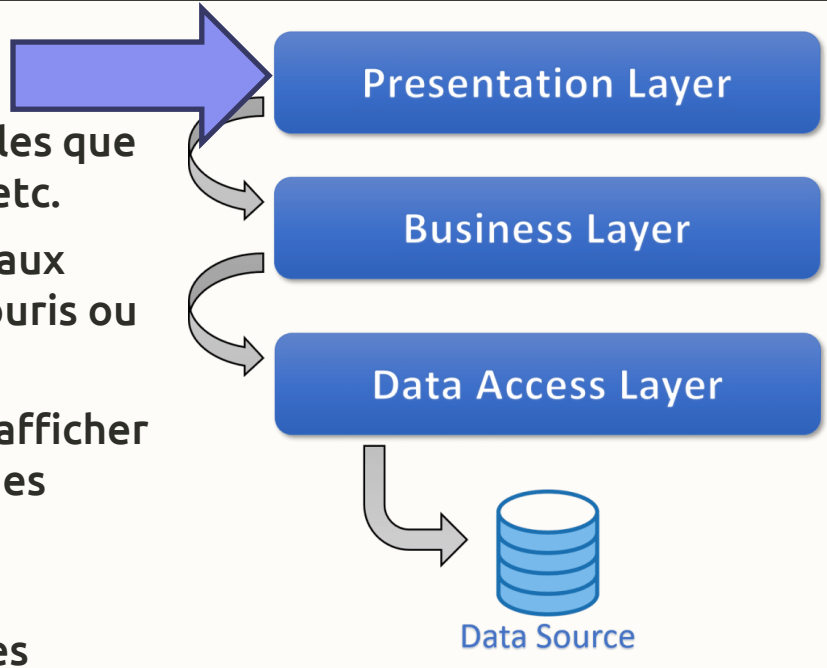
- **Cela comprend :**
  - l'affichage des données de manière claire et conviviale,
  - la réception des entrées de l'utilisateur (par exemple,
  - des clics de souris ou des saisies au clavier)
  - et la présentation des résultats des opérations effectuées par les autres couches.



2.2.3 Responsabilités de chaque couche

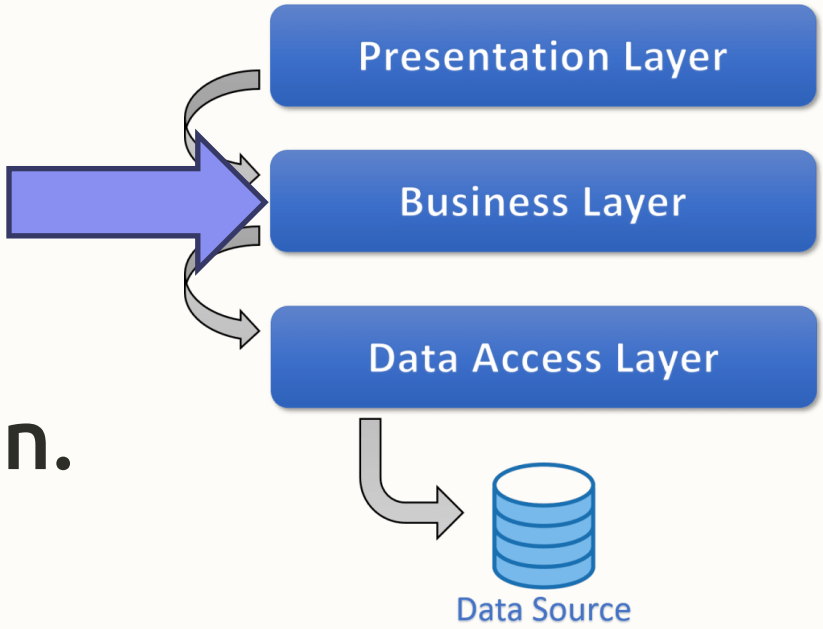
En termes de code :

- 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.



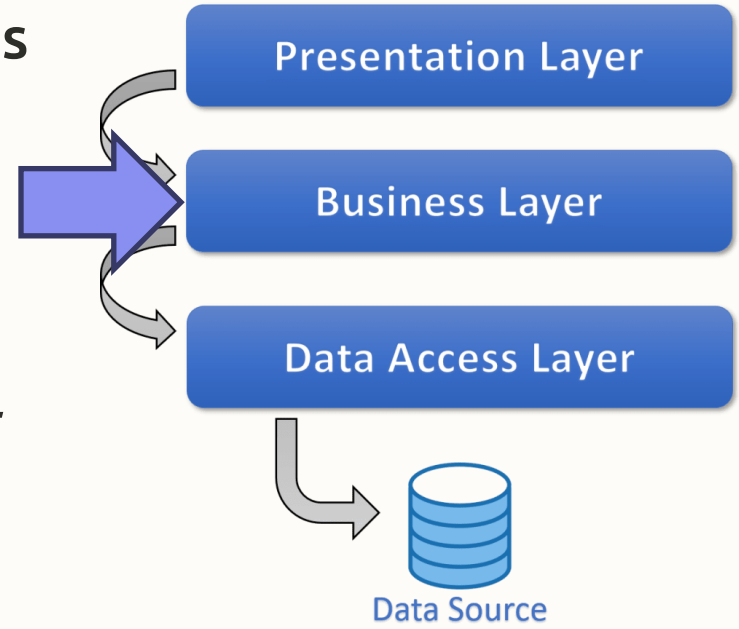
2.2.3 Responsabilités de chaque couche

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



2.2.3 Responsabilités de chaque couche

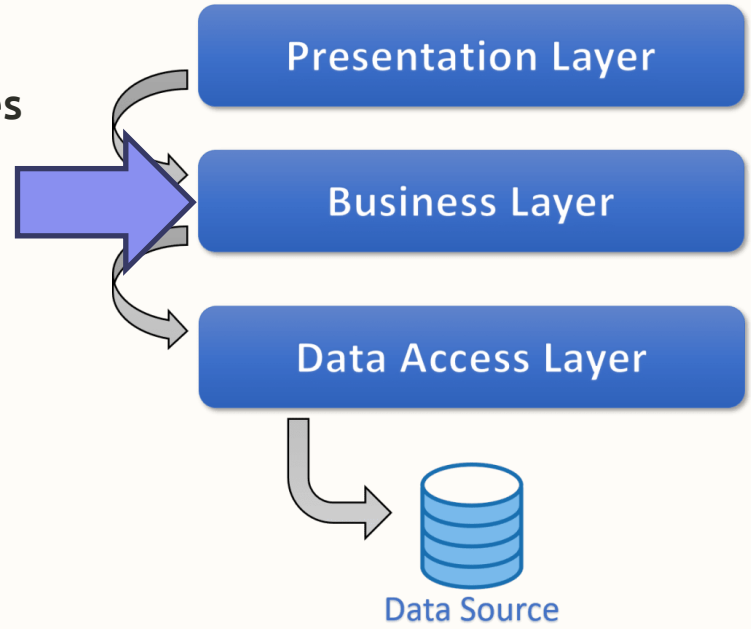
- Cela implique la mise en œuvre des **règles** et des **algorithmes** nécessaires pour traiter les données et prendre des décisions.
- *Par exemple, dans une application bancaire, la couche métier pourrait vérifier les conditions d'éligibilité d'un client pour un prêt et calculer les taux d'intérêt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



2.2.3 Responsabilités de chaque couche

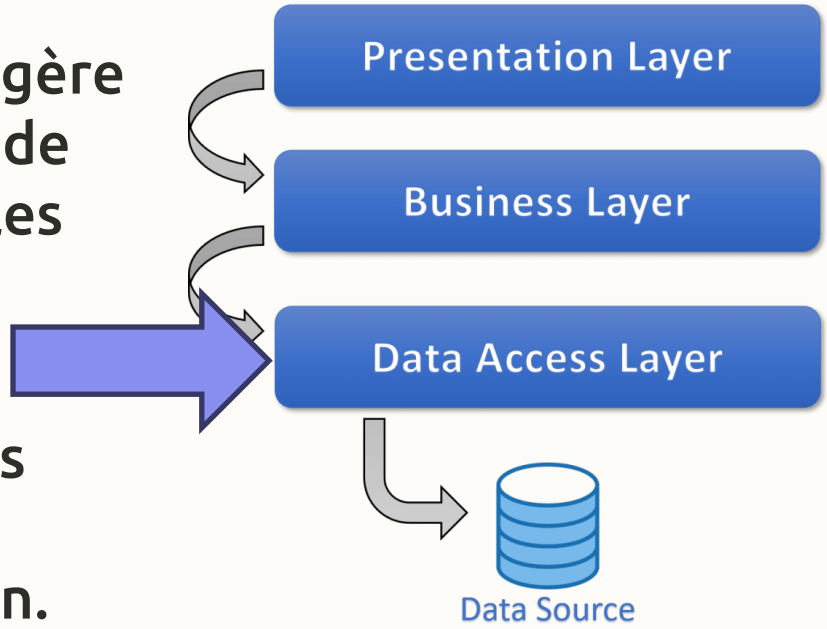
En termes de code :

- 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.



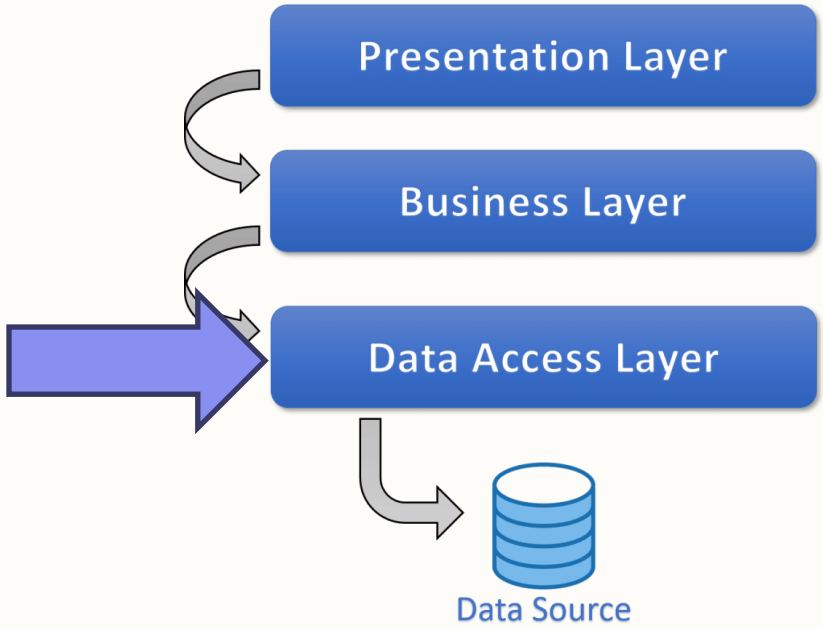
2.2.3 Responsabilités de chaque couche

- 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

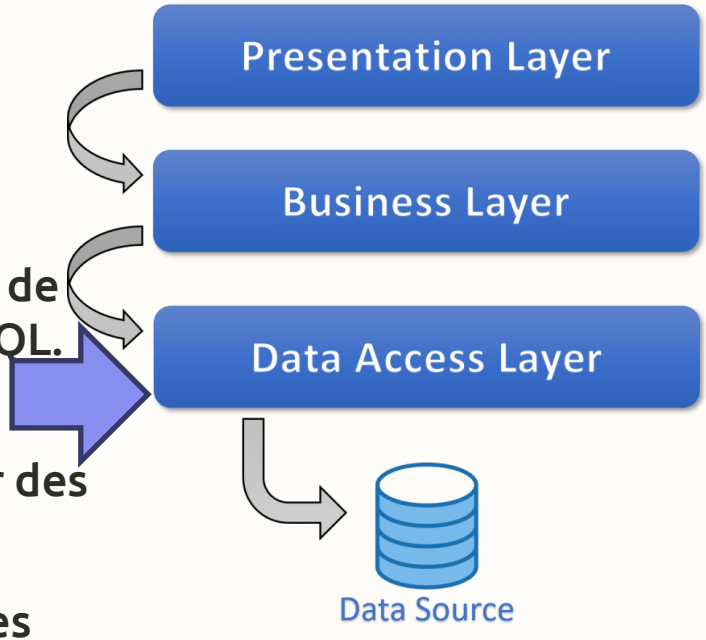
- *Par exemple, dans une application de commerce électronique, cette couche serait chargée de récupérer les informations sur les produits à partir de la base de données et de les rendre disponibles aux autres couches.*



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

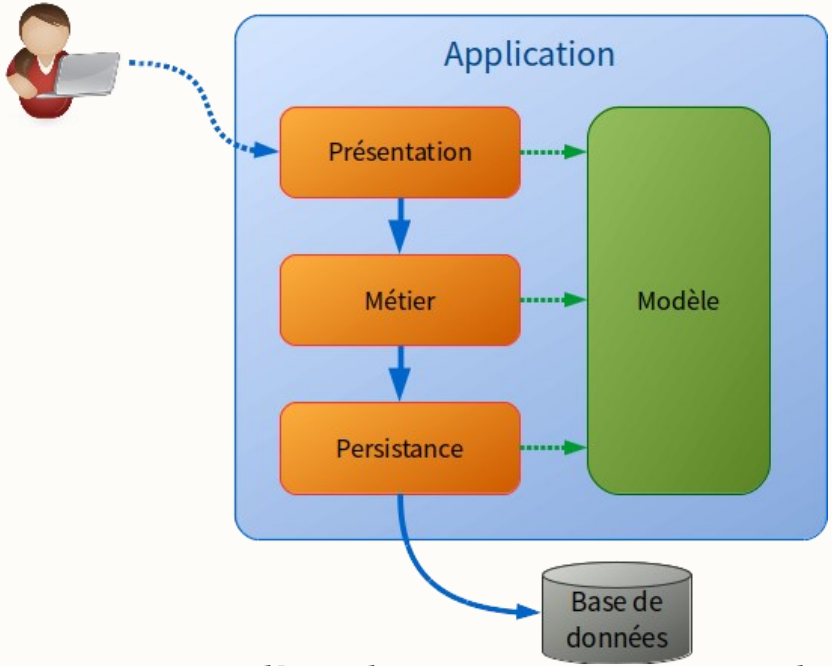
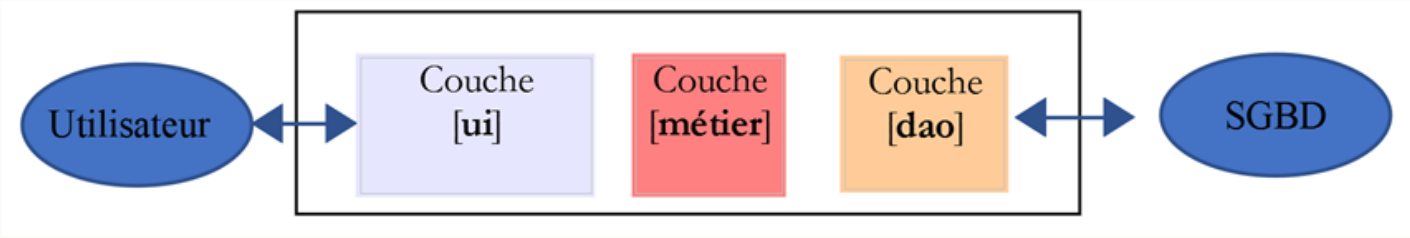
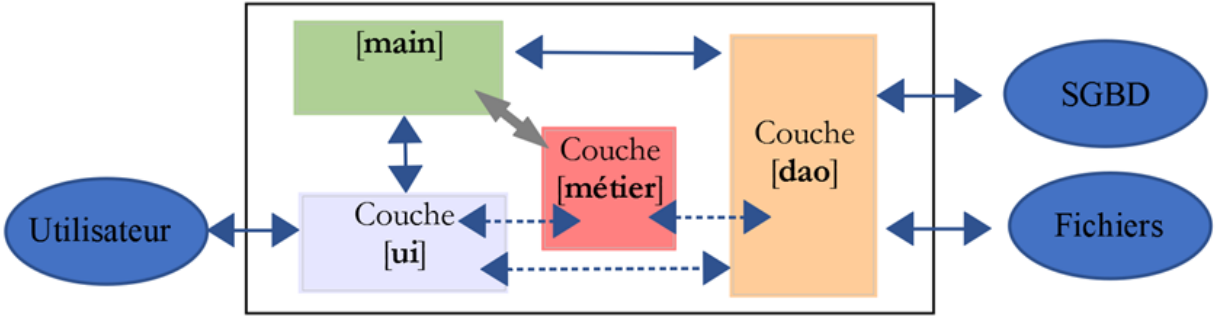


Diagramme d'architecture en couches

2.2.4 Communication entre les couches





Diagrammes d'architectures en couches

2.2.4 Communication entre les couches

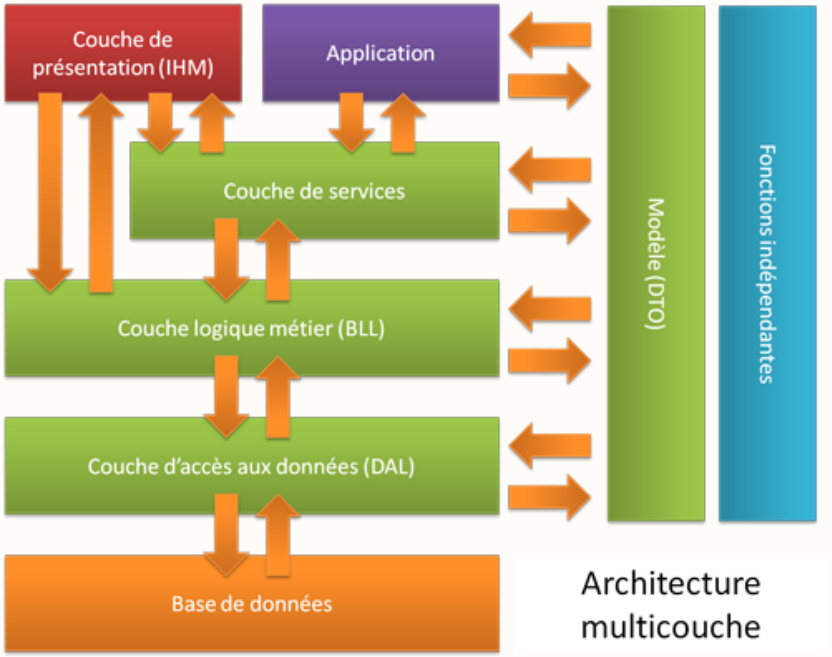


Diagramme d'architecture en couches

2.2.4 Communication entre les couches

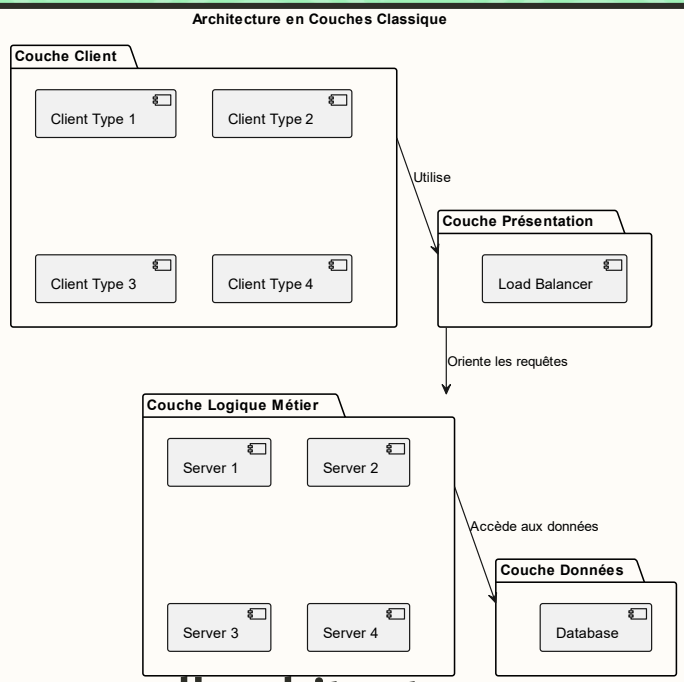


Diagramme d'architecture en couches

2.2.4 Communication entre les couches

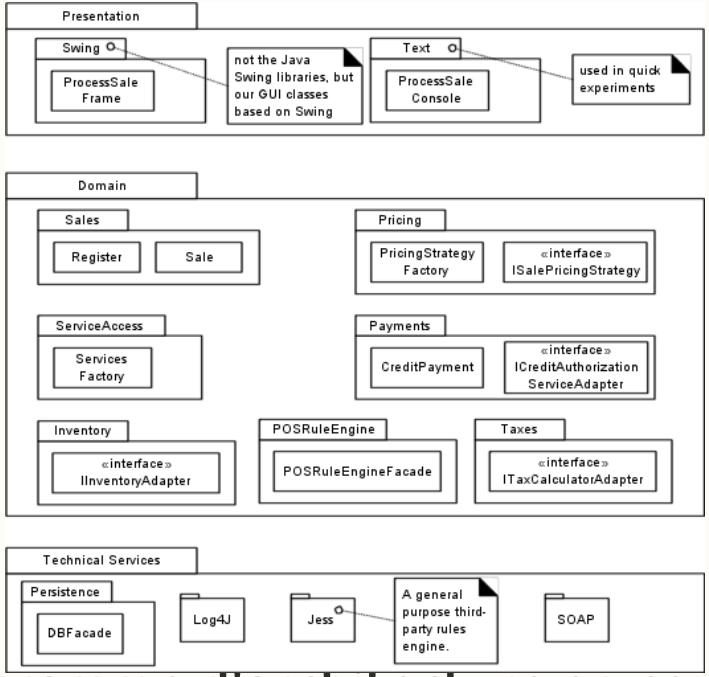


Diagramme d'architecture en couches

2.2.4 Communication entre les couches

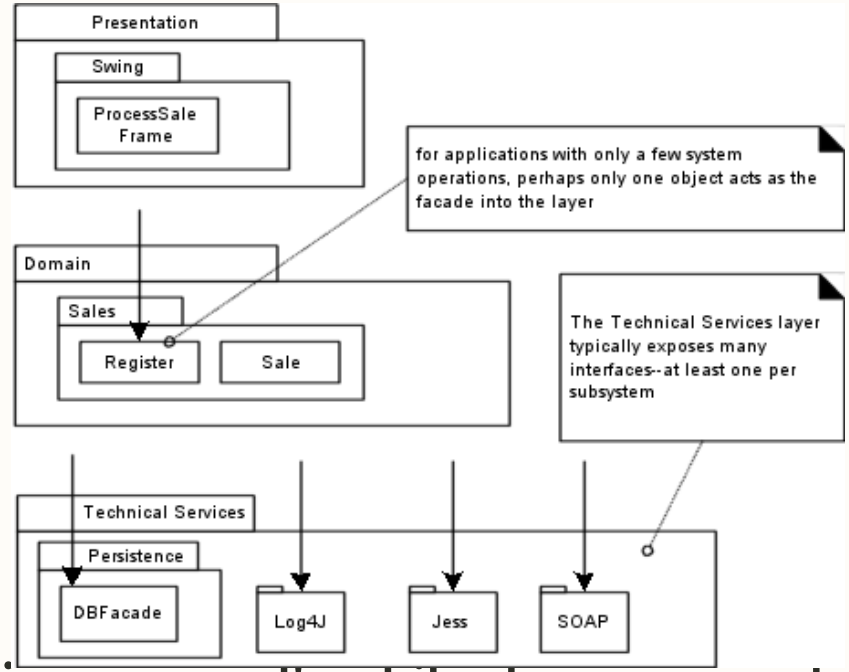


Diagramme d'architecture en couches

2.2.4 Communication entre les couches

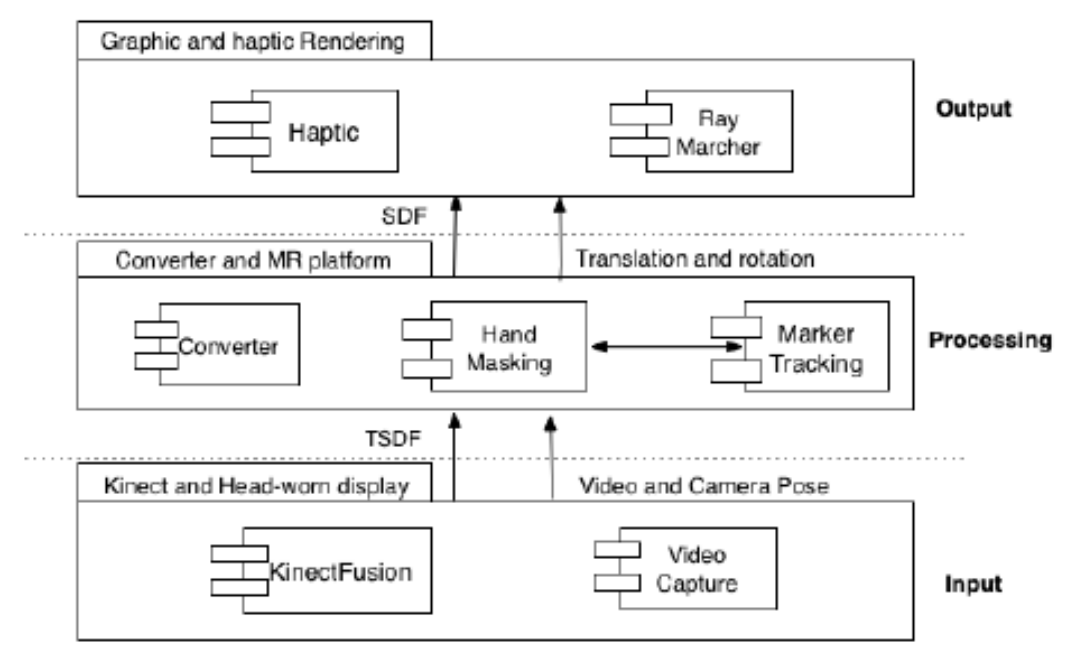


Diagramme de composants en couches

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

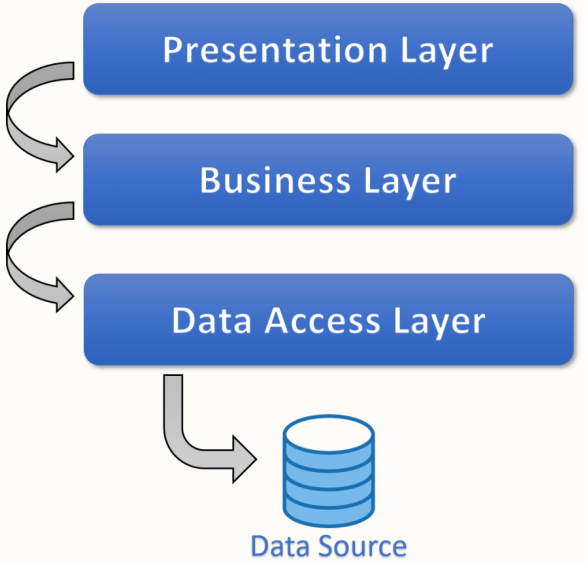
Exemple : Difficulté d'ajout d'une nouvelle couche

- *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.*



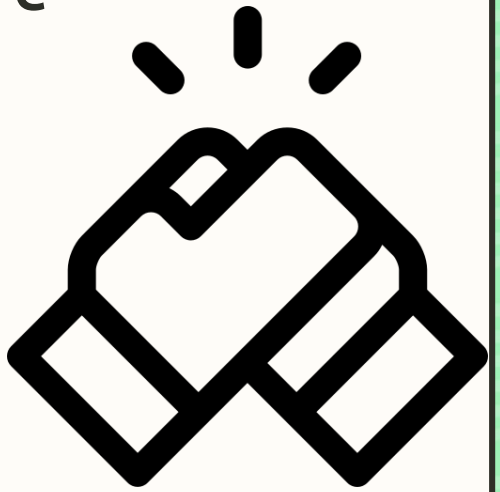
2.2.7 Exemples d'utilisation de l'archi. en couches

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.

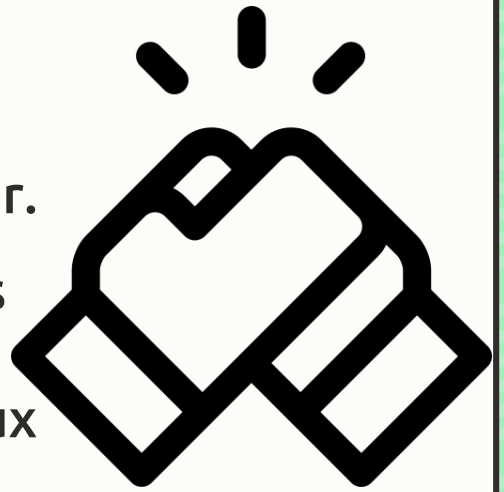
2.2.8 Architecture en couches vs Client-Serveur

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.

Merci pour votre attention !

Avez-vous des questions ?

contact@astroware-conception.com

www.astroware-conception.com

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