Chapitre 2
L’architecture en couches

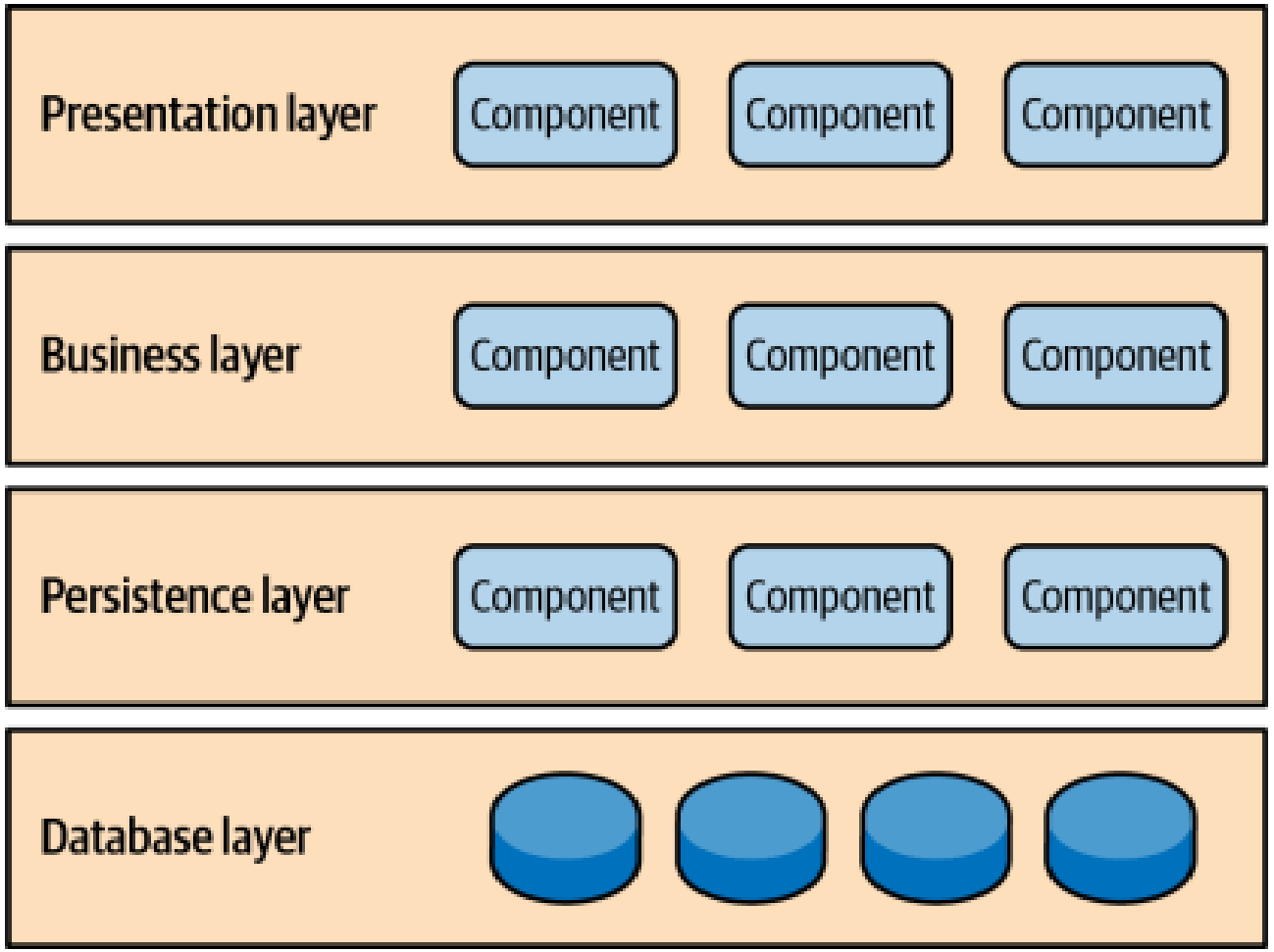
Architecture en couches

- Le style d'architecture le plus courant est **l'architecture en couches**, également appelée **architecture n-tiers**.
- Ce style constitue le standard de facto pour la plupart des applications, car il correspond à l'organisation traditionnelle des équipes informatiques, où les équipes sont structurées par domaines techniques (comme les équipes de présentation, les équipes de développement back-end, les équipes base de données, etc.).
- Étant largement connu de la plupart des architectes, concepteurs et développeurs, l'architecture en couches représente un choix naturel pour la majorité des projets de développement d'applications métier.
- Comme tout style architectural, elle possède ses forces et ses faiblesses et ne convient pas toujours à certains systèmes.

Description

- Les composants sont organisés en **couches horizontales**, chacune remplissant un rôle spécifique au sein de l'application (comme la logique de présentation, la logique métier, la logique de persistance, etc.).
- Bien que le nombre de couches puisse varier, la plupart des architectures en couches se composent de **quatre couches standard** :
  - la couche de présentation,
  - la couche métier,
  - la couche de persistance,
  - la couche base de données.
- Dans certains cas, la couche métier et la couche de persistance sont combinées en une seule couche métier, notamment lorsque la logique de persistance (comme le SQL) est intégrée directement dans les composants de la couche métier.
- Ainsi, les applications de petite taille peuvent ne comporter que **trois couches**, tandis que les applications métier plus grandes et plus complexes peuvent contenir **cinq couches ou davantage**.

Le style d'architecture en couches est une architecture partitionnée techniquement.



- Chaque couche possède un rôle et une responsabilité spécifiques au sein de l'application.
- Par exemple, la **couche de présentation** est responsable de la gestion de toute la logique liée à l'interface utilisateur et à la communication avec le navigateur, tandis que la **couche métier** est chargée d'exécuter les règles métier spécifiques associées à la requête.
- Chaque couche de l'architecture constitue une **abstraction** autour du travail nécessaire pour satisfaire une demande métier donnée.
- Par exemple, la couche de présentation n'a pas besoin de savoir comment récupérer les données client ; elle doit uniquement afficher ces informations à l'écran dans un format particulier.
- De la même manière, la couche métier n'a pas à se préoccuper de la manière dont les données client sont formatées pour l'affichage, ni même de leur provenance ; elle doit simplement récupérer les données depuis la couche de persistance, appliquer la logique métier sur ces données (par exemple, effectuer des calculs ou agréger des informations), puis transmettre le résultat à la couche de présentation.

- Les couches se manifestent généralement par un espace de noms, une structure de packages ou une structure de répertoires (selon le langage de programmation utilisé).
- Par exemple, les fonctionnalités liées aux clients dans une couche métier pourraient être représentées par *app.business.customer*, tandis que dans la couche de présentation, la logique client serait représentée par *app.presentation.customer*.
- Dans cet exemple :
  - le deuxième élément de l'espace de noms représente la couche,
  - le troisième élément représente le composant du domaine.
- Le troisième élément de l'espace de noms (customer) est dupliqué dans toutes les couches — cela illustre une architecture partitionnée techniquement, où le domaine est réparti sur toutes les couches de l'architecture.

- La séparation entre les différentes couches se reflète dans la façon dont le code est organisé. Selon le langage de programmation utilisé, cette organisation peut se faire de différentes manières :
- Espace de noms (namespace) :
  - Dans certains langages comme C# ou C++, on utilise des namespaces pour grouper des classes et fonctions par couche ou module.
  - Exemple : *App.Business.Customer* → ici Business indique la couche métier.
- Structure de packages :
  - Dans des langages comme Java ou Python, on organise le code en packages ou modules.
  - Exemple : *app.presentation.customer* → ici presentation indique la couche de présentation.
- Structure de répertoires (dossiers) :
  - Le code peut être simplement organisé dans des dossiers sur le disque, chaque dossier représentant une couche.
  - Exemple : un dossier business pour la couche métier, un dossier presentation pour la couche de présentation.

- L’une des caractéristiques majeures du style d’architecture en couches est la séparation des préoccupations (separation of concerns) entre les composants.
- Les composants d’une couche donnée ne traitent que la logique propre à cette couche.
- Par exemple, les composants de la couche de présentation ne gèrent que la logique de présentation,
- tandis que les composants de la couche métier ne traitent que la logique métier.
- Ce type de classification des composants facilite la définition des rôles et responsabilités au sein de l’architecture, et rend plus simple le développement, les tests et la maintenance des applications utilisant ce style architectural, à condition que des interfaces et contrats de composants bien définis soient utilisés entre les couches.

Exemple

Dans une architecture multi-couches, on sépare les responsabilités en couches distinctes :

- Couche Présentation (Frontend)
- Couche Métier / Service (Backend)
- Couche Persistance / Base de données

- **Couche 1 : Présentation**
  - Technologies : AngularJS, React.js
  - Responsabilité : interface utilisateur, interactions, affichage dynamique
  - Communication avec le backend via **API REST / GraphQL**
- **Couche 2 : Métier / Services**
  - Technologies : Java Spring Boot, Node.js, .NET, Python Flask
  - Responsabilité : logique métier, sécurité, validation
  - Expose des endpoints pour le frontend
- **Couche 3 : Persistance**
  - Technologies : PostgreSQL, MySQL, MongoDB
  - Responsabilité : stockage et récupération des données via DAO/Repository

Remarques

- Une API (Application Programming Interface) est un ensemble de règles et d'interfaces qui permet à deux logiciels ou composants de communiquer entre eux.
  - c'est le "pont" ou "contrat" qui permet à une application de demander des services ou des données à une autre.
- API REST (Representational State Transfer) : pour créer des services web. Une API REST expose des endpoints HTTP qui permettent de récupérer ou modifier des ressources.
- GraphQL est un langage de requête pour les APIs créé par Meta (Facebook). Il permet au client de demander exactement les données dont il a besoin, et rien de plus.

Concepts Clés de l'architecture en couches

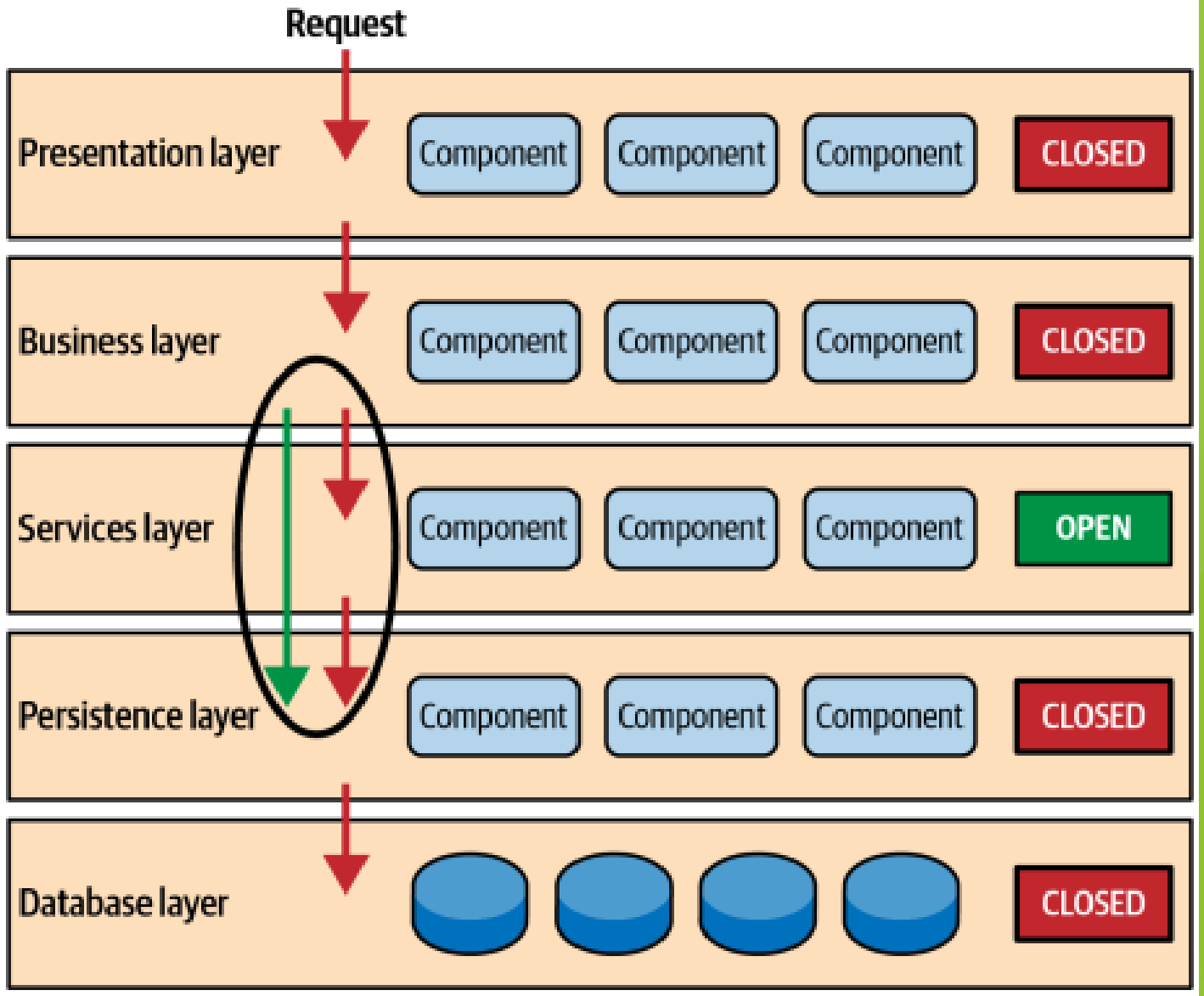
- Dans ce style d'architecture, les couches peuvent être **ouvertes** ou **fermées**.
- Une **couche fermée** signifie que lorsqu'une requête passe d'une couche à une autre, elle doit obligatoirement **passer par la couche immédiatement inférieure** pour atteindre la couche suivante en dessous.
- Par exemple, une requête provenant de la **couche de présentation** doit d'abord passer par la **couche métier**, puis par la **couche de persistance**, avant d'atteindre enfin la **couche base de données**.
- Alors pourquoi ne pas permettre à la **couche de présentation** d'accéder directement à la **couche de persistance** ou à la **couche base de données** ?
- Après tout, un accès direct à la base de données depuis la couche de présentation serait beaucoup plus rapide que de passer par plusieurs couches inutiles juste pour récupérer ou enregistrer des informations en base de données.
- La réponse à cette question se trouve dans un concept clé appelé **les couches d'isolation**.

- Le concept de couches d'isolation signifie que les modifications apportées à une couche de l'architecture n'affectent généralement pas les composants des autres couches : la modification reste isolée aux composants de cette couche, et éventuellement à une autre couche associée (comme une couche de persistance contenant du SQL).
- Si l'on permet à la couche de présentation d'accéder directement à la couche de persistance, alors toute modification du SQL dans la couche de persistance aurait un impact à la fois sur la couche métier et sur la couche de présentation, ce qui créerait une application fortement couplée avec de nombreuses interdépendances entre composants.
- Ce type d'architecture devient alors fragile et très difficile, voire coûteux, à modifier.

- Pour comprendre la puissance et l'importance de ce concept, considérons un important travail de refactoring visant à remplacer le framework de présentation Angular.js par React.js.
- En supposant que les contrats utilisés entre la couche de présentation et la couche métier restent les mêmes, la couche métier n'est pas affectée par ce refactoring et demeure complètement indépendante du type de framework d'interface utilisateur utilisé par la couche de présentation.
- Il en va de même pour la couche de persistance : si elle est correctement conçue, le remplacement d'une base de données relationnelle par une base NoSQL ne devrait impacter que la couche de persistance, et non la couche de présentation ni la couche métier.

- Alors que les couches fermées facilitent les couches d'isolation et permettent ainsi de limiter l'impact des changements au sein de l'architecture, il arrive que certaines couches puissent être ouvertes.
- Par exemple, supposons que vous souhaitiez ajouter une couche de services partagés à une architecture, contenant des fonctionnalités communes accessibles par les composants de la couche métier (par exemple, des classes utilitaires pour les données et les chaînes de caractères, ou des classes pour l'audit et le journal de bord).
- Créer une couche de services est généralement une bonne idée dans ce cas, car sur le plan architectural, cela limite l'accès aux services partagés à la couche métier (et non à la couche de présentation).
- Sans cette couche séparée, rien n'empêche architecturalement la couche de présentation d'accéder à ces services communs, ce qui rendrait difficile le contrôle de cet accès.

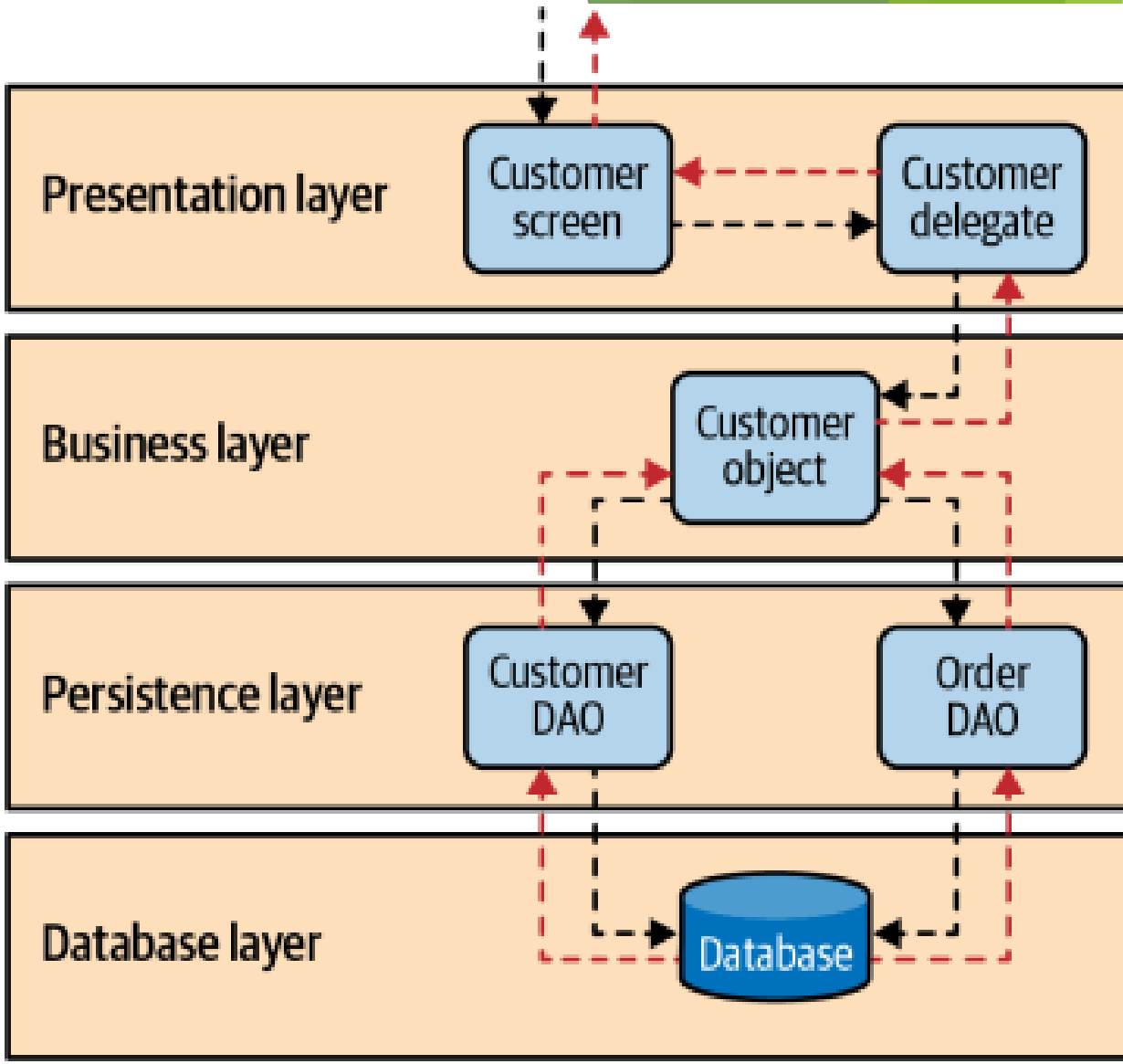
- Comme illustré sur la Figure, la couche de services devrait être marquée comme ouverte, ce qui signifie que les requêtes peuvent contourner cette couche et aller directement vers la couche située en dessous.
- Dans l'exemple suivant, puisque la couche de services est ouverte, la couche métier peut la contourner et accéder directement à la couche de persistance, ce qui est parfaitement logique.
- Avec des couches ouvertes, la requête peut contourner la couche située en dessous.



- Exploiter le concept de couches ouvertes et fermées permet de définir la relation entre les couches de l'architecture et le flux des requêtes, et fournit aux concepteurs et développeurs les informations nécessaires pour comprendre les différentes restrictions d'accès entre les couches de l'architecture.
- Ne pas documenter ou ne pas communiquer correctement quelles couches de l'architecture sont ouvertes ou fermées (et pourquoi) conduit généralement à des architectures fortement couplées et fragiles, très difficiles à tester, maintenir et déployer.

Exemple

- Dans cet exemple, les informations client comprennent à la fois les données du client et les données de commande (commandes passées par le client).
- Ici, l'écran client est responsable de recevoir la requête et d'afficher les informations client. Il ne sait pas où se trouvent les données, comment elles sont récupérées, ni combien de tables de la base de données doivent être interrogées pour obtenir ces informations.
- Une fois que l'écran client reçoit une requête pour obtenir les informations d'un client particulier, il transmet cette requête au module délégué client dans la couche de présentation.
- Ce module est responsable de savoir quels modules de la couche métier peuvent traiter cette requête, ainsi que de comment accéder à ces modules et quelles données ils nécessitent (le contrat).
- L'objet client dans la couche métier est responsable de regrouper toutes les informations nécessaires pour satisfaire la requête métier (dans ce cas, obtenir les informations du client).



- Ensuite, le module objet client appelle le module DAO client (Data Access Object) dans la couche de persistance pour obtenir les données client, ainsi que le module DAO commande pour récupérer les informations sur les commandes.
- Ces modules exécutent ensuite les instructions SQL pour récupérer les données correspondantes et les renvoient au module objet client dans la couche métier.
- Une fois que l'objet client reçoit les données, il les agrège et transmet ces informations au délégué client, qui les envoie ensuite à l'écran client pour les afficher à l'utilisateur.

Utilités principales du DAO

- **Séparation des responsabilités (Separation of Concerns)**
  - Le DAO :
    - Contient uniquement le code d'accès aux données
    - Evite de mettre du SQL ou du code de base de données dans la couche métier
      - *La logique métier ne sait pas comment les données sont stockées, seulement qu'elle peut les obtenir.*

Utilités principales du DAO

- **Abstraction de la base de données**
  - Le DAO masque :
    - Le type de base de données (MySQL, PostgreSQL, Oracle…)
    - Le langage de requête (SQL, JPQL…)
    - Le framework utilisé (JPA-Java Persistence API, Hibernate*…)

```
public interface CustomerDAO {
    Customer findById(Long id);
    void save(Customer customer);
}
```

      - La couche métier utilise simplement cette interface, sans connaître l'implémentation.

- * Hibernate est un framework ORM (Object Relational Mapping) qui implémente JPA.

Utilités principales du DAO

- **Faciliter la maintenance et l'évolution**
  - Si on change :
    - Le SGBD
    - Le framework ORM
    - La stratégie de persistance (BDR, NoSQL, Fichiers JSON, XML, CSV, …
  - Seule l'implémentation du DAO change
  - La couche métier reste intacte

Exemple d’entité JPA

- Depuis les versions récentes: Java EE devient **Jakarta EE**
- @Entity indique que la classe Java représente une table dans une base de données lorsqu’on utilise Java Persistence API ou un framework comme Hibernate.
- @Id : indique la clé primaire de la table.

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;

@Entity
public class Task {

    @Id
    private Long id;
    private String title;

}
```

Annotation	Rôle
@Entity	transforme une classe Java en entité persistante
@Id	définit la clé primaire
@Table	définit le nom de la table
@Column	définit les propriétés d'une colonne
@GeneratedValue	génère automatiquement la clé primaire

```java
import jakarta.persistence.*;

@Entity
@Table(name="tasks")
public class Task {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name="task_title")
    private String title;
}
```

```java
Task task = new Task();
task.setTitle("Préparer le cours");

entityManager.persist(task);
```

- JPA génère automatiquement :

```sql
INSERT INTO task (title) VALUES ('Préparer le cours');
```

Remarques

- L'architecture en couches constitue un style architectural largement éprouvé et à vocation générale,
- Ce qui en fait un point de départ pertinent pour la plupart des applications, en particulier lorsque le choix du style architectural le plus adapté au contexte du projet demeure incertain.
- Toute architecture en couches comporte inévitablement certains scénarios relevant de l'anti-pattern du *sinkhole* architectural. L'élément clé consiste toutefois à analyser la proportion de requêtes qui entrent dans cette catégorie.
- L'anti-pattern du *sinkhole* architectural survient lorsque les requêtes traversent plusieurs couches d'une architecture sans qu'aucune logique significative ne soit exécutée dans les couches intermédiaires.
- Chaque couche se contente alors de transmettre la requête à la suivante, puis de renvoyer le résultat, sans traitement, transformation ou enrichissement des données, ce qui entraîne une complexité structurelle inutile.

- La règle empirique des 80–20 constitue généralement un bon repère pour évaluer la situation.
- Il est courant d'observer qu'environ 20 % des requêtes correspondent à un simple transit (pass-through processing), tandis que 80 % impliquent une certaine logique métier.
- En revanche, si ce ratio s'inverse et que la majorité des requêtes se limite à un simple passage d'une couche à l'autre, il peut être pertinent d'envisager l'ouverture de certaines couches de l'architecture.
- Il faut toutefois garder à l'esprit que, bien qu'une telle ouverture puisse améliorer la rapidité, elle complique le contrôle des évolutions en raison de la réduction de l'isolation entre les couches.

Quand envisager une architecture multi-couches?

- Elle est pertinente lorsque le projet présente des contraintes importantes de budget ou de temps.
- Étant généralement considérée comme un style d'architecture monolithique, elle ne comporte pas les complexités liées à une architecture distribuée, telles que l'accès à distance, la gestion des contrats, …
- De plus, la plupart des développeurs et architectes sont familiarisés avec l'architecture en couches, ce qui facilite sa compréhension et sa mise en œuvre.

- Une autre raison de choisir l'architecture en couches est lorsque la majorité des modifications sont confinées à des couches spécifiques de l'application.
- Par exemple :
  - Des modifications portant uniquement sur les règles métier sans impact sur l'interface utilisateur.
  - Des changements relatifs à l'apparence ou à l'ergonomie de l'interface utilisateur.
  - La migration vers un nouveau framework d'interface utilisateur.
  - La migration vers un nouveau type de base de données.
- Dans tous ces cas, les changements restent isolés à une couche particulière de l'architecture, ce qui facilite l'identification et la gestion des composants affectés.

- Étant donné que l'architecture en couches est une architecture **techniquement partitionnée**, elle convient particulièrement si la structure de l'équipe l'est également.
- Autrement dit, si l'organisation globale de l'équipe comprend des groupes distincts de développeurs d'interface utilisateur (UI), de développeurs back-end, d'équipes de services partagés, d'équipes base de données, etc., cette répartition correspond bien à la structuration de l'architecture (couche présentation, couche métier, couche persistance, etc.).
- Cette correspondance est connue sous le nom de **loi de Conway**.
  - La structure d'un logiciel tend à reproduire la structure des équipes qui l'ont conçu.

Quand ne pas choisir ce style d'architecture?

- Elle est peu adaptée aux applications exigeant scalabilité, tolérance aux pannes ou haute performance, car elle tend à être monolithique : faire évoluer ses fonctionnalités est coûteux.
  - Bien qu'il soit parfois possible de scaler une architecture en couches en répartissant les différentes couches sur des déploiements physiques séparés ou en créant plusieurs instances de l'application sur différentes machines virtuelles, cette approche s'avère coûteuse et inefficace, car 100 % des fonctionnalités de l'application doivent être mises à l'échelle.
- De plus, l'architecture en couches est peu tolérante aux pannes : un crash critique dans n'importe quelle partie de l'application entraîne l'arrêt complet de l'ensemble des fonctionnalités.

Quand ne pas choisir ce style d'architecture?

- Une autre raison d'éviter l'architecture en couches est lorsque la majorité des changements concernent le domaine métier plutôt que des aspects techniques.
- Par exemple, si l'on souhaite ajouter une date d'expiration à la liste « Mes films » d'un utilisateur dans une application de streaming, cette nouvelle fonctionnalité nécessiterait :
  - Une modification du schéma de la base de données.
  - Une adaptation du SQL dans la couche de persistance.
  - Une mise à jour des règles métier et des contrats dans la couche métier (durée avant expiration, actions à effectuer à l'expiration, etc.).
  - Une modification de la couche présentation pour afficher la date d'expiration à côté de chaque film.
- Ainsi, un simple changement métier entraîne des modifications dans toutes les couches, ce qui rend l'architecture plus rigide et difficile à maintenir.

Quand ne pas choisir ce style d'architecture?

- En analysant ce changement relativement simple dans la fonctionnalité « Mes films », on constate que toutes les couches de l'architecture sont affectées et nécessitent des modifications.
- Dans de grands systèmes avec des équipes techniquement partitionnées, cela peut même impliquer la coordination de plusieurs équipes (UI, back-end, base de données, etc.).
- Cette situation réduit l'agilité globale (capacité à répondre rapidement aux changements) et augmente le temps et l'effort nécessaires pour mettre en œuvre la modification.

Quand ne pas choisir ce style d'architecture?

- Si la structure globale de votre équipe est organisée en équipes transversales basées sur le domaine (chaque équipe regroupant des compétences UI, back-end et base de données pour un domaine spécifique de l'application), l'architecture en couches n'est pas adaptée.
- En effet, la structure d'architecture techniquement partitionnée ne s'aligne pas avec l'organisation des équipes partitionnées par domaine.

Anti-pattern

- Un anti-pattern désigne une solution couramment adoptée face à un problème récurrent, mais dont l'application conduit, de manière systématique ou fréquente, à des conséquences négatives sur la qualité du système.
- Contrairement à un simple « mauvais choix », l'anti-pattern présente généralement trois caractéristiques :
  - Il répond à un problème réel et récurrent.
  - Il est fréquemment utilisé en pratique, souvent parce qu'il semble intuitif ou rapide à mettre en œuvre.
  - Il produit des effets contre-productifs, tels qu'une augmentation de la complexité, une dégradation de la maintenabilité ou une dette technique accrue.

Anti-pattern	Définition synthétique	Causes fréquentes	Conséquences	Stratégies de remédiation
Architecture Sinkhole	Transit des requêtes à travers plusieurs couches sans logique significative	Sur-application stricte de l'architecture en couches	Complexité inutile, baisse de performance	Ouvrir certaines couches, simplifier la structure
Big Ball of Mud	Absence d'architecture claire, accumulation désorganisée de code	Croissance rapide, manque de gouvernance architecturale	Dette technique élevée, faible maintenabilité	Refactoring progressif, modularisation, documentation
God Object (Blob)	Une classe/composant concentre trop de responsabilités	Mauvaise répartition des responsabilités, absence de design initial	Couplage fort, difficulté de test	Appliquer le principe SRP, découpage fonctionnel
Spaghetti Code	Flux de contrôle complexe et peu structuré	Manque de discipline de conception, évolution non maîtrisée	Code difficile à comprendre et maintenir	Refactoring, introduction de patterns structurants
Golden Hammer	Utilisation systématique d'une même solution technique	Expertise limitée à une technologie, biais organisationnel	Mauvais alignement avec les besoins	Analyse contextuelle, évaluation multicritère
Lava Flow	Conservation de code obsolète ou inutilisé	Peur de supprimer du code, absence de tests	Complexité accrue, maintenance coûteuse	Tests automatisés, suppression contrôlée
Vendor Lock-In	Dépendance excessive à un fournisseur	Choix technologique fermé, absence d'abstraction	Difficulté de migration, coûts élevés	Introduction de couches d'abstraction, standards ouverts
Chatty Interface	Multiplication excessive d'appels inter-composants	Mauvaise granularité des services	Latence et surcharge réseau	Regrouper les appels, revoir la conception API
Distributed Monolith	Système distribué mais fortement couplé	Mauvaise découpe des microservices	Déploiement complexe, perte d'indépendance	Découpage basé sur les domaines, DDD
Architecture by Implication	Architecture émergente sans formalisation explicite	Absence de documentation et de décisions formalisées	Difficulté d'évolution,[38] incohérences	ADR (Architecture Decision Records), documentation continue

Résumons

- **Type de partitionnement : Technique**
  L'architecture en couches est organisée selon des responsabilités techniques (couche présentation, couche métier, couche persistance) plutôt que par domaines fonctionnels.
- **Coût global :** Indique que l'architecture est peu coûteuse à mettre en place, surtout pour des projets standards ou des équipes familières avec ce style.
- **Agilité:** L'agilité est faible : les changements affectant le domaine métier nécessitent souvent des modifications dans toutes les couches, ce qui ralentit la capacité à réagir rapidement aux évolutions.
- **Simplicité:** La simplicité est élevée : la structure en couches est claire, compréhensible et bien connue des développeurs, ce qui facilite la compréhension et la maintenance.
- **Scalabilité:** La scalabilité est faible : les architectures en couches sont souvent monolithiques, rendant difficile la montée en charge sans redéploiement complet ou duplication des instances.
- **Tolérance aux pannes:** Faible tolérance aux pannes : un crash dans une couche peut entraîner l'arrêt de l'ensemble de l'application.
- **Performance:** Performance moyenne : la structure en couches ajoute une certaine latence due au passage des données entre les différentes couches, mais pour des applications standards, elle reste acceptable.
- **Extensibilité:** L'extensibilité est limitée : ajouter de nouvelles fonctionnalités impliquant plusieurs couches peut être coûteux et complexe.

Characteristic	Star rating
Partitioning type	Technical
Overall cost	$
Agility	★
Simplicity	★★★★★
Scalability	★
Fault tolerance	★
Performance	★★★
Extensibility	★