

«Une bonne implémentation technique est une conséquence d'une bonne architecture»

Bloc B1: De la vision métier à l'architecture logicielle

Les Principes de Conception

1. **Principes d'architecture (niveau macro)**
 - Séparation des préoccupations (Separation of Concerns, SoC) — découpage).
 - Modularité / Couverture
 - Cohésion & Couplage
 - Isolation des dépendances
2. **Principes de conception (classe / composant)**
 - S — Single Responsibility Principle (SRP)
 - O — Open/Closed Principle (OCP)
 - L — Liskov Substitution (LSP)
 - I — Interface Segregation (ISP)
 - D — Dependency Inversion (DIP)



Objectifs

1. Comprendre les principes architecturaux de base et de savoir répartir les responsabilités dans une application d'entreprise à l'échelle conceptuelle.
2. Différencier les grands principes d'architecture et de conception.
3. Appliquer ces principes sur un projet JEE-Spring Boot.
4. Être capable de diagnostiquer violations simples (couplage excessif, responsabilité floue).

À la fin du bloc, chaque étudiant doit pouvoir justifier une répartition simple (Interface / Métier / Données) et repérer 2 anti-patterns courants.

Principes de Conception Principes d'architecture (niveau macro)



	Définition	Pourquoi	Ex. MoroccoTravel	
Séparation des préoccupations (SoC: Separation of Concerns)	Diviser/découper le système selon des responsabilités fonctionnelles (UI , métier , données)	Facilite la compréhension, le développement parallèle et la maintenance.	Interface client (Recherche) séparée du module de réservation et du module paiement.	Citer 2 fonctions qui doivent rester séparées
Modularité / Couverture	Construire le en modules autonomes (indépendants), réutilisables et remplaçables.	Permet d'isoler les changements, réutiliser des composants et tester séparément.	Module "gestion des vols" réutilisable	Proposez un module réutilisable pour MoroccoTravel
Cohésion & Couplage	<ul style="list-style-type: none"> • Cohésion (intérieur) = les éléments d'un module travaillent pour un même but. • Couplage (entre)= degré de dépendance entre modules (faible = bon). 	Haute cohésion + faible couplage → maintenance facile, tests fiables.	<ul style="list-style-type: none"> • Cohésion: Tous les services de paiement dans un même module ; • faible couplage: le module paiement ne doit pas connaître la logique UI . 	Identifier un couplage dangereux entre 2 fonctions de MoroccoTravel
Isolation des dépendances	<ul style="list-style-type: none"> • Réduire les dépendances directes ; • accéder aux ressources via interfaces ou adaptateurs. 	Évolution/tests/Protège le système des changements externes (APIs partenaires, DB)	Un API Vols adapters entre notre service (ex.vols) et l'API d'une compagnie	Donnez un exemple d'adaptateur nécessaire pour MoroccoTravel.

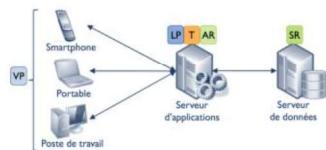
Principes de Conception

Principes d'architecture (niveau macro): JEE



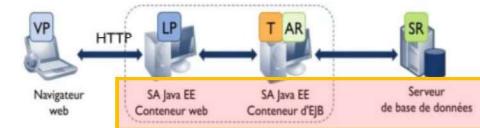
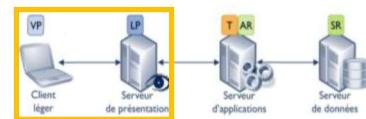
1- Où placer les couches applicatives dans une architecture 3-tiers?

- Le cœur de l'application est sur un serveur
- Les données sont sur un autre serveur
- Le client est une application « légère » de visualisation (ex : navigateur web)



2- Où placer les couches applicatives dans une architecture n-tiers?

- Distribution des responsabilités en 4 ou + tiers
 - Client léger
 - Serveur de présentation
 - Serveur d'application
 - Serveur de données



Exemple simplifié d'une Architecture jee:

- Client léger: navigateur web
- Serveur de présentation : ex. tomcat
- Serveur d'application:** conteneur web+ conteneur EJB

Pr. Hind Lamharhar

42

Principes de Conception

Exemple: Architecture JEE



Principe	Comment JEE aide	Exemple JEE
SoC – Séparation des préoccupations Séparer présentation, métier, accès aux données	JAX-RS + EJB + JPA	Resource REST → Service EJB → Repository JPA
Modularité Diviser en modules autonomes	WAR / EAR / EJB	Un module EJB peut être indépendant du web
Cohésion / Couplage Classes cohérentes, couplage faible	Interfaces EJB, JPA pour BD	DAO JPA découple SQL
Isolation des dépendances Modifier une partie sans casser	CDI pour injection abstraite	@Inject sans dépendance concrète
SRP Une classe = une responsabilité	?	?
OCP Ouvert extension, fermé modification	?	?
LSP Substitution sans casser	?	?
ISP Interfaces petites	?	?
DIP Dépendre d'abstractions	?	?

Pr. Hind Lamharhar

Principes de conception (classe / composant) — SOLID



Principe	Définition	Références
S: Single Responsibility Principle	« Une classe, un module ou une fonction ne doit avoir qu'une seule responsabilité et donc une seule raison de changer . »	Robert C. Martin, <i>Clean Architecture</i> (2017); Robert C. Martin, <i>Agile Software Development: Principles, Patterns, and Practices</i> (2002)
O: Open / Closed Principle	« Les entités logicielles doivent être ouvertes à l'extension mais fermées à la modification . »	Bertrand Meyer, <i>Object-Oriented Software Construction</i> (1988); Robert C. Martin, <i>Agile Software Development</i>
L: Liskov Substitution Principle	« Une classe dérivée doit pouvoir se substituer à sa classe de base sans modifier la correction du programme . »	Barbara Liskov, <i>Data Abstraction and Hierarchy</i> (1987)
I: Interface Segregation Principle	« Les clients ne doivent pas dépendre d'interfaces qu'ils n'utilisent pas ; il vaut mieux plusieurs interfaces spécifiques qu'une seule interface générale. »	Robert C. Martin, <i>Agile Software Development</i> (2002)
D: Dependency Inversion Principle	« Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau ; les deux doivent dépendre d'abstractions . »	Robert C. Martin, <i>Agile Software Development</i> ; Robert C. Martin, <i>Clean Architecture</i> (2017)

Pr. Hind Lamharhar

44

Principes de conception (classe / composant) — SOLID



Principe	Définition	Pourquoi ?	Exemple MoroccoTravel Hub	Question pédagogique
S — Single Responsibility Principle	Une classe ou un composant doit avoir une seule raison de changer	Code lisible, testable, maintenance simplifiée	BookingService gère uniquement la création/modification de réservations , pas l'envoie d'e-mails ni le paiement.	Quel composant ne devrait PAS gérer l'envoi d'e-mail ?
O — Open/Closed Principle	Les composants sont ouverts à l'extension , mais fermés à la modification .	Permet d'ajouter des fonctionnalités sans casser les existantes.	Ajouter une nouvelle stratégie de paiement via PaymentStrategy sans modifier PaymentService.	Donnez une fonctionnalité ajoutable par extension (sans modifier du code existant).
L — Liskov Substitution Principle	Toute sous-classe doit pouvoir remplacer sa superclasse sans changer le comportement attendu .	Evite les implémentations incompatibles, garantit la robustesse des modules.	AirlinePremium doit pouvoir remplacer AirlineBase dans un moteur de recherche de vols sans casser les appels existants.	Donnez un exemple où remplacer un composant pourrait casser le système.
I — Interface Segregation Principle	Mieux vaut plusieurs interfaces spécifiques qu'une seule interface lourde.	Réduit le couplage, évite de forcer des méthodes inutiles, meilleure modularité.	Au lieu de TravelService avec 20 méthodes, on crée SearchFlights et ManageBookings.	Proposez 2 interfaces claires pour la plateforme MoroccoTravel Hub.
D — Dependency Inversion Principle	Les modules haut niveau dépendent d'abstractions, pas d'implémentations concrètes.	Facilite le changement de fournisseur technique / implémentation.	BookingService dépend de PaymentGateway (interface), pas d'une classe VisaPayment.	Pourquoi préférer une interface PaymentGateway ?

Pr. Hind Lamharhar

45

Principes de Conception

Exemple: Architecture JEE



Principe	Comment JEE aide	Exemple JEE
SoC – Séparation des préoccupations Séparer présentation, métier, accès aux données	JAX RS + EJB + JPA	Resource REST → Service EJB → Repository JPA
Modularité Diviser en modules autonomes	WAR / EAR / EJB	Un module EJB peut être indépendant du web
Cohésion / Couplage Classes cohérentes, couplage faible	Interfaces EJB, JPA pour BD	DAO JPA découpe SQL
Isolation des dépendances Modifier une partie sans casser	CDI pour injection abstraite	@Inject sans dépendance concrète
SRP Une classe = une responsabilité	Services spécialisés	Un EJB = une fonction métier
OCP Ouvert extension, fermé modification	Alternatives CDI	changer implémentation via config
LSP Substitution sans casser	Héritage entités JPA	@Inheritance
ISP Interfaces petites	JAX-RS segment API	/flights, /users séparés
DIP Dépendre d'abstractions	CDI injecte interfaces	@Inject PaymentProcessor

Pr. Hind Lamharhar

46

Principes de conception (classe / composant)

Application des SOLID (MoroccoTravel)



- Dans le **cahier des charges de MoroccoTravel Hub** (analyse métier), les **fonctionnalités de réservation** couvrent plusieurs types de services touristiques (vols, hôtels, véhicules, activités, guides). Bien que ces services partagent un objectif commun — permettre une réservation — leur **fonctionnement métier réel** diffère fortement. **L'analyse de ces différences est une étape indispensable avant toute modélisation ou structuration de l'architecture logicielle.**
 - Les **vols** et les **hôtels** sont caractérisés par une logique entièrement pilotée par le SI. La disponibilité est connue à l'avance, le stock est quantifiable et géré automatiquement (places, chambres, dates), et les règles de réservation sont stables. Lorsqu'un utilisateur effectue une réservation, le système est capable de décider immédiatement si l'opération est possible, de déclencher le paiement et de fournir une confirmation instantanée. Cette prévisibilité permet de qualifier ces services de **déterministes**.
 - La **location de véhicule** présente une logique similaire, bien qu'elle introduise des règles complémentaires telles que la gestion des cautions, des états de véhicules ou des catégories. Toutefois, ces règles restent formalisables et automatisables. La disponibilité est également gérée par le système, et la réservation peut être confirmée immédiatement ou quasi immédiatement. Du point de vue du cahier des charges, ce type de service reste donc dans une logique **déterministe**, même si le processus métier est légèrement enrichi.
 - Les **activités touristiques de groupe** constituent un cas intermédiaire. La réservation repose sur des données systèmes (capacité maximale, dates, créneaux), mais peut être soumise à des conditions supplémentaires comme un nombre minimum de participants ou une validation organisationnelle légère. Le système peut initier la réservation, mais la confirmation finale dépend parfois d'un contexte collectif. Cette combinaison entre automatisation et conditions opérationnelles justifie une **logique semi-déterministe**, qui se situe entre le traitement entièrement automatique et le traitement humain.
 - À l'opposé, la réservation d'un **guide touristique privé** repose sur une logique fondamentalement différente. La disponibilité n'est pas uniquement une donnée système, mais dépend d'une personne réelle, avec des contraintes humaines non totalement prévisibles (langue, spécialité, agenda personnel, acceptation de la mission). Le processus décrit dans le cahier des charges est donc orienté **demande**, puis **attente**, puis **validation manuelle**, avec une confirmation qui n'est ni immédiate ni garantie. Le paiement peut être différé ou conditionné à cette validation. Cette nature rend le processus **asynchrone et conditionnel**, et incompatible avec un modèle de réservation automatique classique.

Principes de conception (classe / composant)

Application des SOLID (MoroccoTravel)



1. **Réservation de vols**
 - La disponibilité est connue à l'avance.
 - Le nombre de places est géré par le système.
 - La confirmation est immédiate après paiement.
 - Le système décide seul si la réservation est possible.
2. **Réservation d'hôtels**
 - La disponibilité des chambres est gérée par le système.
 - Les règles de réservation sont stables.
 - La confirmation est immédiate.
3. **Réservation de guides**
 - La disponibilité dépend du guide (personne réelle).
 - Une demande est envoyée au guide.
 - La confirmation n'est pas immédiate.
 - Le guide peut refuser la demande.
 - Le paiement peut être différé.
4. **Paiement**
 - Plusieurs moyens de paiement doivent être supportés.
 - De nouveaux moyens pourront être ajoutés ultérieurement.
 - Le système métier ne doit pas dépendre d'un moyen de paiement spécifique.

Contraintes d'évolution

- De nouveaux services touristiques pourront être ajoutés.
- De nouveaux moyens de paiement pourront être intégrés.
- Les règles métier de chaque service pourront évoluer indépendamment.

À partir de cette partie du cahier des charges fonctionnel,

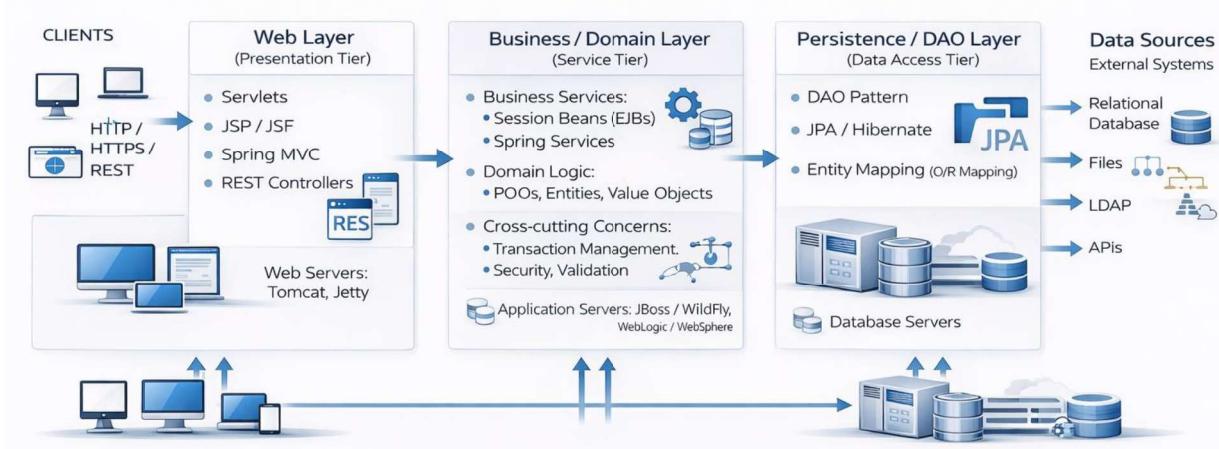
1. **déduire les responsabilités**,
2. **identifier les différences métier**,
3. **et proposer une conception respectant SOLID**.

48

Application des Principes de Conception

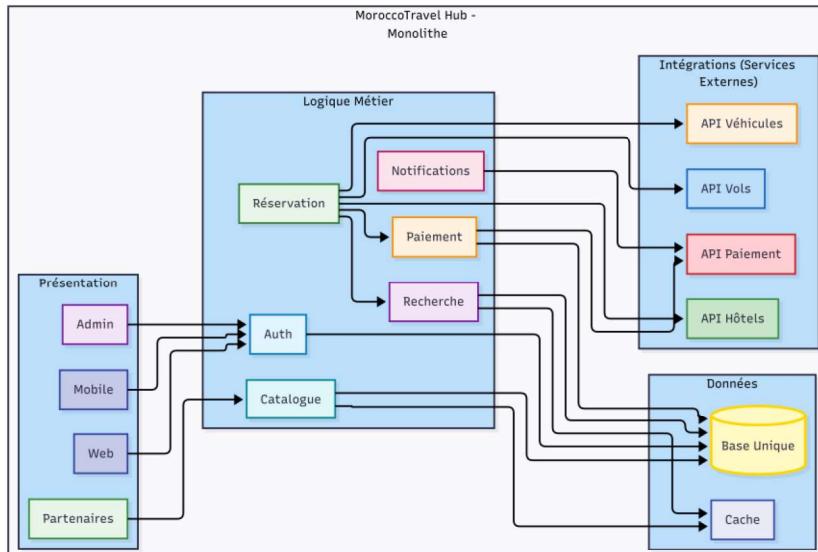
Principes d'architecture (niveau macro): JEE

Separation of Concerns (SoC)
Modularité / Couverture
Cohésion & Couplage
Isolation des dépendances



Application des Principes de Conception Principes d'architecture (niveau macro): JEE

Separation of Concerns (SoC)
Modularité / Couverture
Cohésion & Couplage
Isolation des dépendances

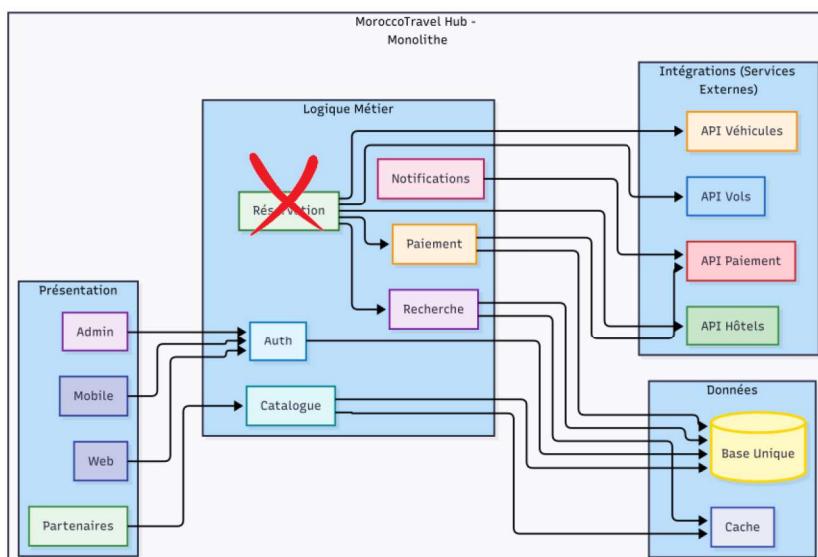


Pr. Hind Lamharhar

50

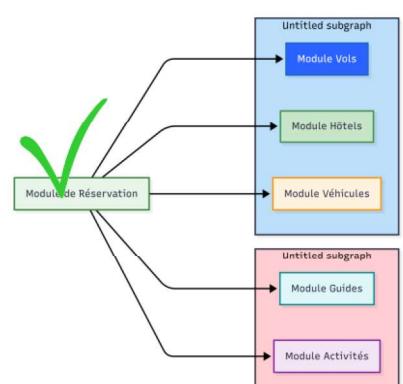
Application des Principes de Conception Principes d'architecture (niveau macro): JEE

Separation of Concerns (SoC)
Modularité / Couverture
Cohésion & Couplage
Isolation des dépendances



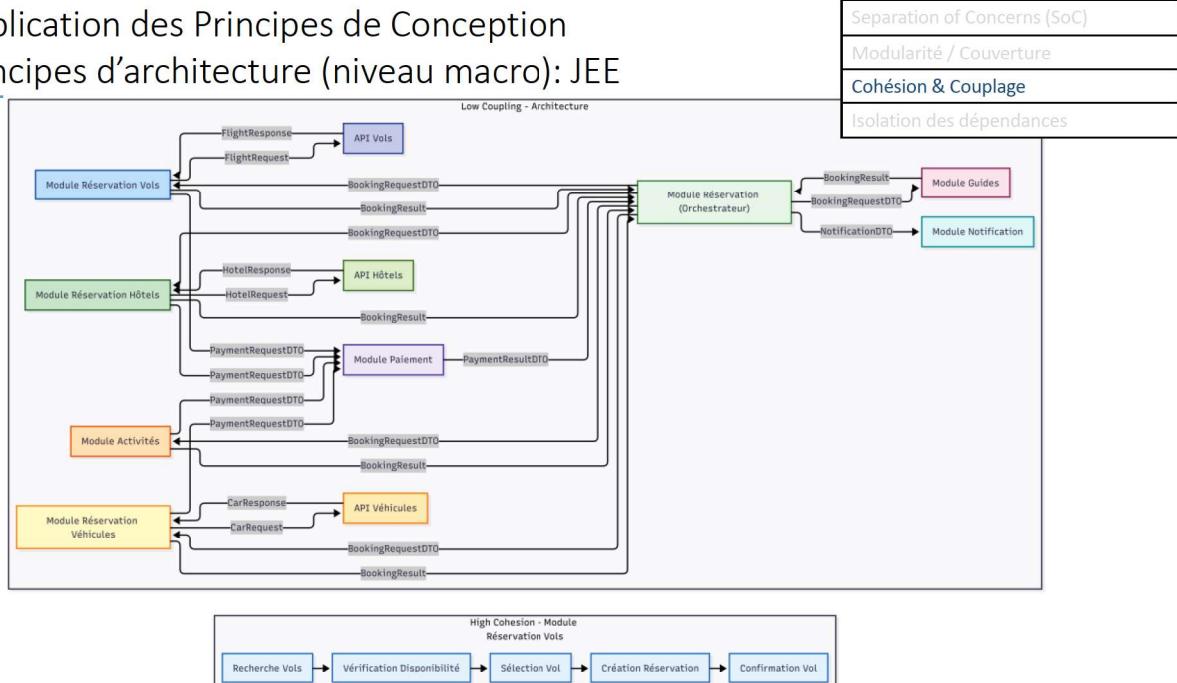
Pr. Hind Lamharhar

51



Application des Principes de Conception

Principes d'architecture (niveau macro): JEE

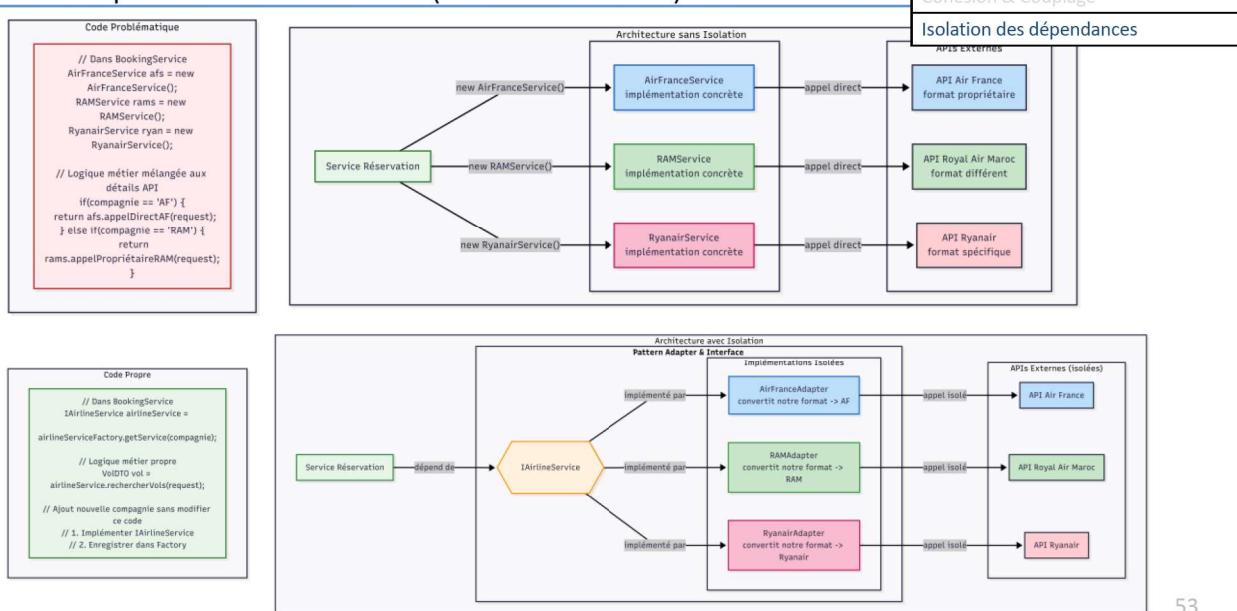


Pr. Hind Lamine

52

Application des Principes de Conception

Principes d'architecture (niveau macro): JEE



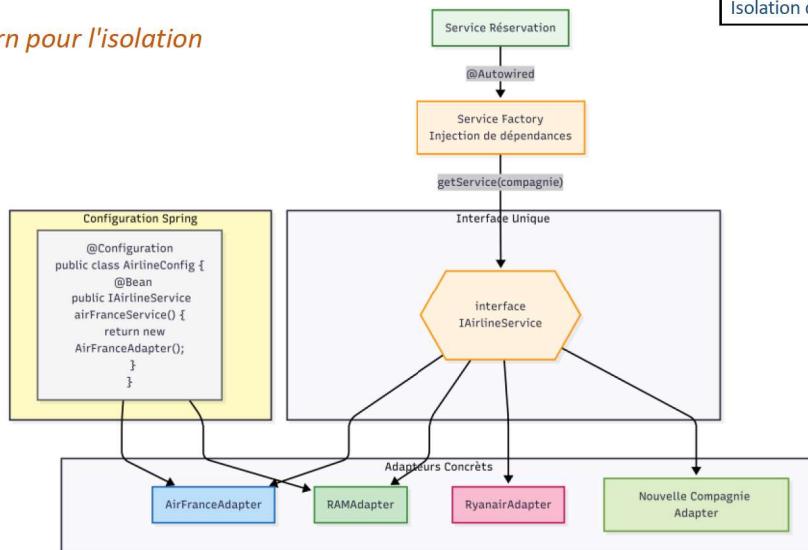
53

Application des Principes de Conception

Principes d'architecture (niveau macro): JEE

Separation of Concerns (SoC)
Modularité / Couverture
Cohésion & Couplage
Isolation des dépendances

Factory pattern pour l'isolation



Pr. Hind Lamharhar

54

Application des Principes de Conception

SOLID avec JEE



SOLID

✗ Mauvaise pratique

✓ Bonne pratique



Pr. Hind Lamharhar

55



	Mauvaise pratique	Bonne pratique
SRP	Une seule classe gère tout : réservation de vol + hôtel + guide + paiement + email <pre>class TravelManager { void bookFlight(){ void bookHotel(){ void bookGuide(){ void processPayment(){ void sendEmail(){ }</pre> X	Une responsabilité par service <pre>class FlightBookingService { void bookFlight(){ } } class HotelBookingService { void bookHotel(){ } } class GuideBookingService { void bookGuide(){ } } class PaymentService { void pay(){ } } class NotificationService { void sendEmail(){ } }</pre> ✓
OCP	Modifier le service de paiement pour chaque nouveau moyen <pre>class PaymentService { void payByCard(){ void payByPaypal(){ } }</pre> X	Étendre sans modifier <pre>interface PaymentStrategy { void pay(); } class CardPayment implements PaymentStrategy { public void pay(){ } } class PaypalPayment implements PaymentStrategy { public void pay(){ } } class PaymentService { void process(PaymentStrategy p){p.pay(); } }</pre> ✓
LSP	Une implémentation casse le comportement attendu <pre>class TravelService { void book(){ } } class GuideService extends TravelService { void book(){ throw new UnsupportedOperationException(); } }</pre> X	Substitution correcte <pre>interface BookableService { void book(); } class FlightService implements BookableService { public void book(){ } } class HotelService implements BookableService { public void book(){ } } class GuideService implements BookableService { public void book(){ } }</pre> ✓
ISP	Interface trop large pour tous les services <pre>interface TravelPlatformService { void bookFlight(); void bookHotel(); void bookGuide(); void pay(); }</pre> X	Interfaces spécifiques par besoin <pre>interface FlightService { void bookFlight(); } interface HotelService { void bookHotel(); } interface GuideService { void bookGuide(); } interface PaymentGateway { void pay(); }</pre> ✓
DIP	Dépendance directe à une implémentation externe <pre>class BookingService { private PaypalPayment payment = new PaypalPayment(); }</pre> X	Dépendre d'une abstraction <pre>interface PaymentGateway { void pay(); } class PaypalPayment implements PaymentGateway { public void pay(){ } } class BookingService { private PaymentGateway gateway; BookingService(PaymentGateway g){this.gateway = g; } }</pre> ✓

Application des Principes de Conception SOLID avec JEE

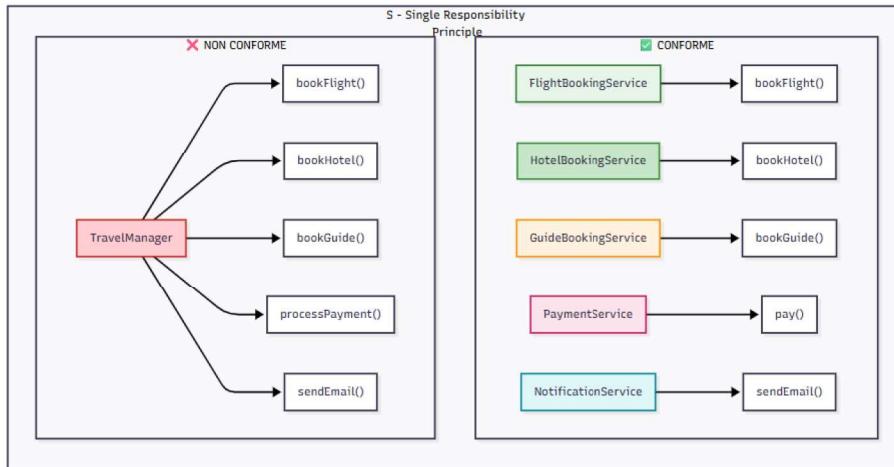
SRP: Une classe = une responsabilité

OCP: Ouvert extension, fermé modification

LSP: Substitution

ISP: petites Interfaces

DIP: Dépendances/injection



Application des Principes de Conception SOLID avec JEE

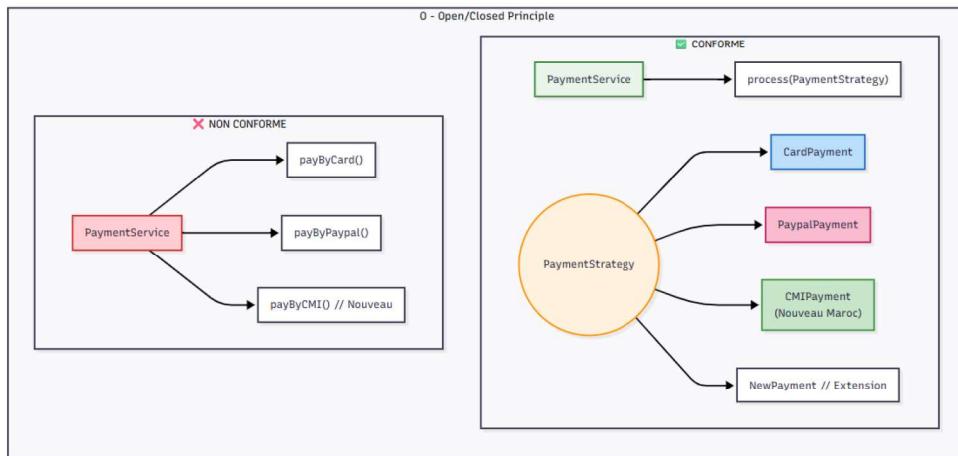
SRP: Une classe = une responsabilité

OCP: Ouvert extension, fermé modification

LSP: Substitution

ISP: petites Interfaces

DIP: Dépendances/injection



Pr. Hind Lamharhar

58

Application des Principes de Conception SOLID avec JEE

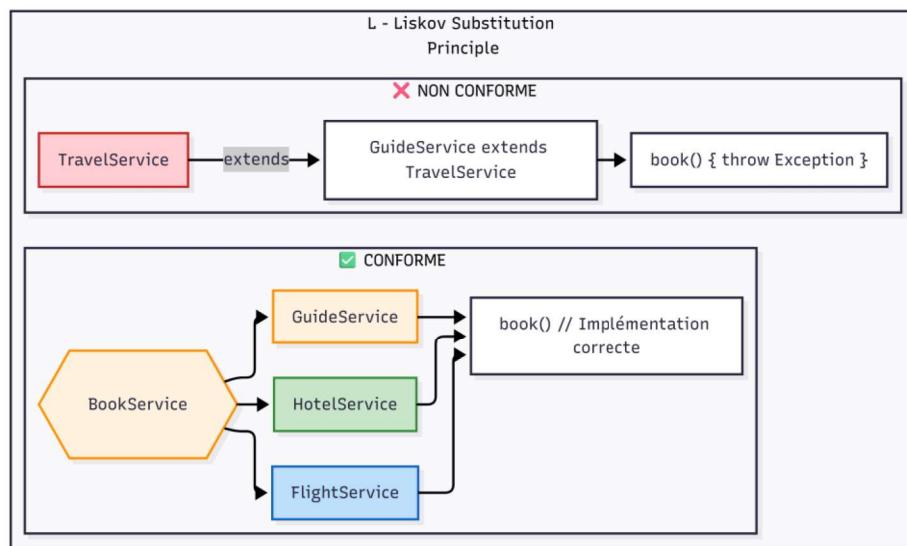
SRP: Une classe = une responsabilité

OCP: Ouvert extension, fermé modification

LSP: Substitution

ISP: petites Interfaces

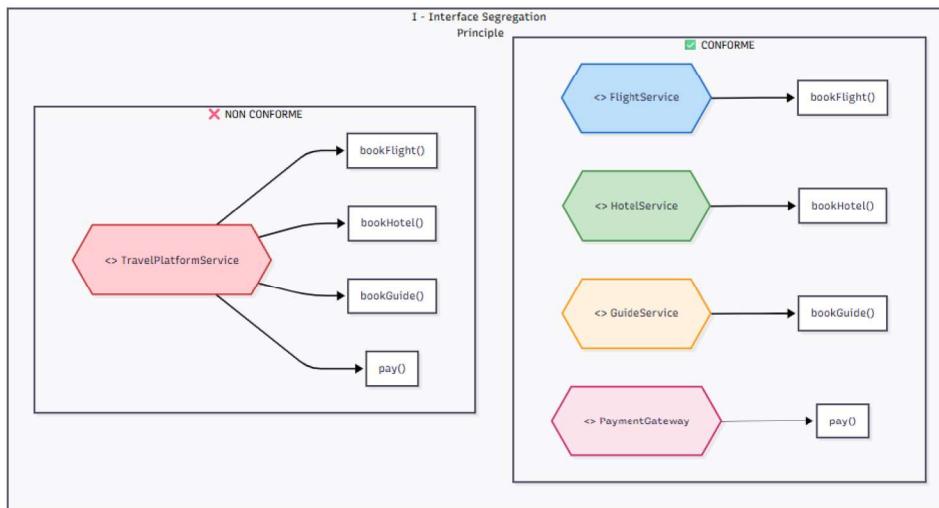
DIP: Dépendances/injection



Pr. Hind Lamharhar

59

Application des Principes de Conception SOLID avec JEE



SRP: Une classe = une responsabilité

OCP: Ouvert extension, fermé modification

LSP: Substitution

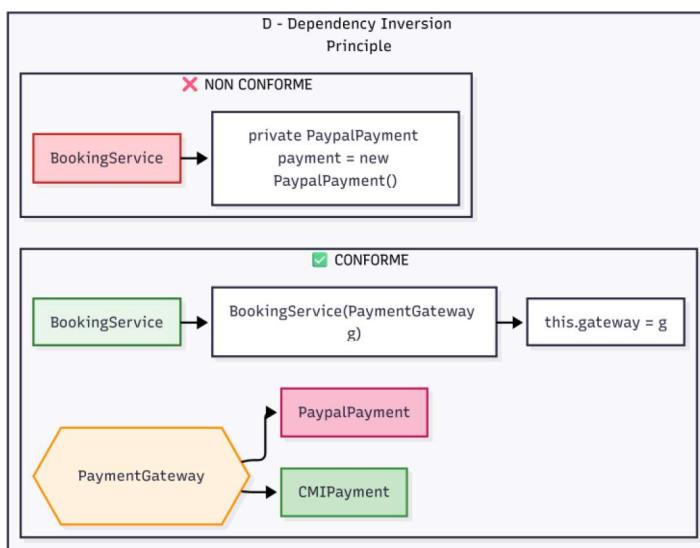
ISP: petites Interfaces

DIP: Dépendances/injection

Pr. Hind Lamharhar

60

Application des Principes de Conception SOLID avec JEE



SRP: Une classe = une responsabilité

OCP: Ouvert extension, fermé modification

LSP: Substitution

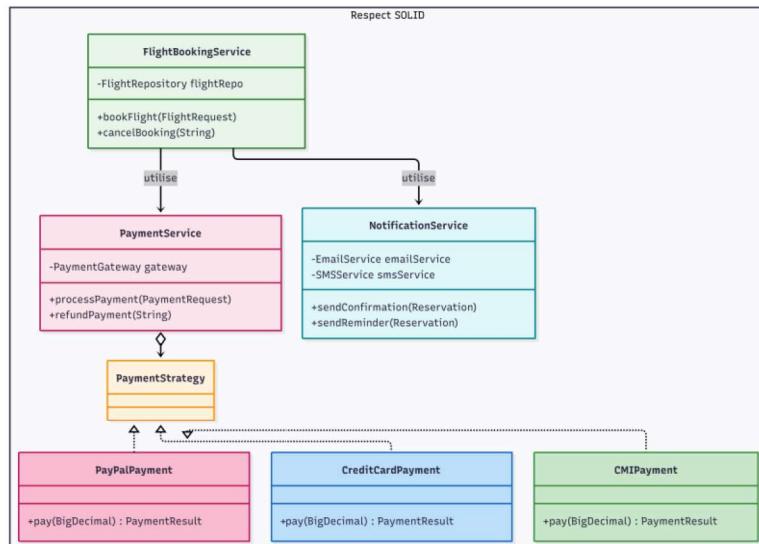
ISP: petites Interfaces

DIP: Dépendances/injection

Pr. Hind Lamharhar

61

Application des Principes de Conception SOLID avec JEE



Pr. Hind Lamharhar

62



Quiz Prinipes de Conception

1.SRP : Une classe doit avoir...

- a)Toutes les fonctionnalités
- b)Plusieurs responsabilités
- c)Une seule responsabilité

2.Quel exemple viole SRP ?

- a)BookingService pour réservation
- b)BookingManager : réservation + email + paiement
- c)PaymentService pour paiements

3.OCP signifie :

- a)Ouvert à l'extension, fermé à la modification
- b)Modifiable à volonté
- c)Réécriture obligatoire

5.Exemple qui respecte OCP :

- a)Modifier PaymentService
- b)Ajouter ApplePay implémentant PaymentStrategy
- c)Modifier tous les services

6.LSP : une sous-classe doit...

- a)Remplacer la classe mère sans casser
- b)Ne pas remplacer
- c)Pouvoir casser le comportement

7.Exemple violent LSP :

- a)PremiumFlight remplace Flight
- b)PaypalPayment remplace Payment
- c>Classe substituée sans modification

8.ISP : le principe dit :

- a) Une interface doit tout contenir
- b)Plusieurs interfaces spécifiques
- c)Une interface pour tout

9.Exemple respectant ISP :

- a)TravelService avec 20 méthodes
- b)SearchFlights + ManageBookings
- c>Interface imposant refund à tout paiement

10.DIP :

- a)Dépendre d'abstractions
- b)Dépendre d'implémentations
- c>Pas d'interfaces

11.Exemple correct DIP :

- a)BookingService dépend de PaymentGateway injectée
- b)BookingService crée VisaPayment
- c>Dépendre de classes concrètes

13.SoC consiste à...

- a)Mélanger interface et logique
- b>Séparer les couches
- c>Répéter le code

14.Exemple SoC :

- a)Controller → Service → Repository
- b>Service faisant SQL direct
- c>Module unique

15.Modularité signifie...

- a)Diviser en modules autonomes
- b>Tout mettre ensemble
- c>Faire des classes géantes

16.Exemple Modularité :

- a)Module Vols, Module Hôtels
- b>ServiceGénéral
- c>Mélange de 50 classes

17.Cohésion signifie...

- a)Module pour un objectif unique
- b>Module global
- c>Classe multi-usage

18.Exemple Cohésion :

- a)Module Paiement pour tout paiement
- b>Service qui gère vols+météo+paiements
- c>Service qui gère logs et réservations

20.Couplage fort →

- a>Un changement casse les autres modules
- b>Code plus clair
- c>Tests faciles

21.Exemple couplage fort :

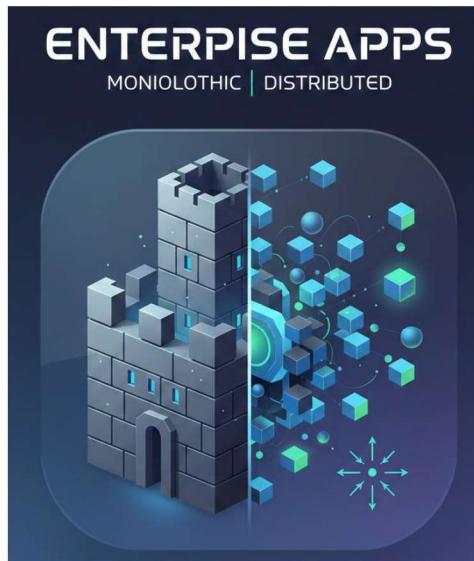
- a>Dépendance par interface
- b>Modules via interface
- c new PaypalPayment() dans BookingService

22.Isolation dépendances signifie...

- a>Gérer soi même
- b>Dépendre des implémentations
- c>Utiliser adaptateurs/interfaces

23.Exemple isolation dépendances :

- a)BookingService appelle API AirlineX directement
- b>Mélanger DB + métier
- c>BookingService passe par AirlineAdapter



Bloc B2: Vers une architecture logicielle Monolithique vs distribuées

1. Architecture monolithique

- Pourquoi elle existe
- Avantages / limites

2. Architecture distribuée

- Communication réseau
- Latence
- Défaillance partielle
- Données partagées

3. Comment résout les problèmes des Exigences

- Scalabilité
- Maintenabilité
- Disponibilité
- Cohérence
- Évolution
- Sécurité

Vers des applications d'entreprise

organisation (RPA, compliance, gestion de stock, réservation...). Elles automatisent les processus métier et optimisent les opérations.

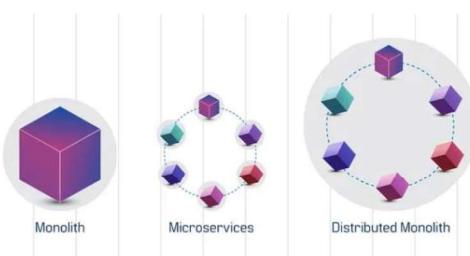


Une application d'entreprise est un programme exécutable sur une machine ou plusieurs machines qui représente la logique de traitement des données manipulées

Un grand nombre d'applications ne s'exécutent pas intégralement sur un seul nœud de calcul.

Application repartie = traitements coopérants sur des données ref

Applications Centralisées vs. Distribuées



Application Centralisée



Application Distribuée



Les choix d'architecture doivent répondre aux besoins stratégiques et non-fonctionnels

Vers des applications d'entreprise: (Pourquoi)



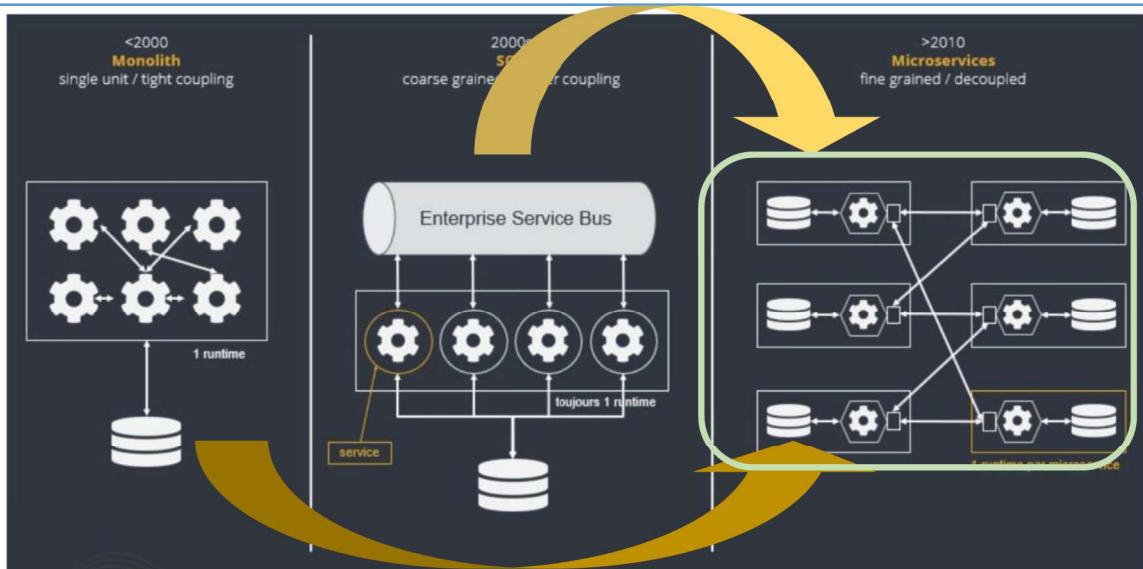
Exigence	Définition & Pourquoi	Critère mesurable / test d'acceptation	Priorité
<i>Scalabilité</i>	Capacité à gérer plus d'utilisateurs/charge sans dégrader le service.		
<i>Disponibilité</i>	Pourcentage du temps où le service est utilisable (SLA).		
<i>Résilience / Tolérance aux pannes</i>	Continuer à fonctionner malgré pannes partielles.		
<i>Performance (latence)</i>	Temps de réponse perçu par l'utilisateur.		
<i>Maintenabilité</i>	Facilité d'évolution et correction du code.		
<i>Sécurité</i>	Confidentialité, intégrité, authentification, autorisation.		
<i>Observabilité (monitoring & logs)</i>	Mesurer et diagnostiquer comportement et incidents.		
<i>Interoperabilité / APIs</i>	Capacité à échanger avec partenaires/tiers.		
<i>Fiabilité</i>	Probabilité de fonctionnement sans erreur sur une période.		
<i>Sauvegarde & Restauration</i>	Récupérer données après incident.		
<i>Conformité & Protection des données</i>	Respect lois (RGPD), contrats partenaires.		
<i>Accessibilité & Usabilité</i>	Facilité d'usage pour tous (y compris handicap).		
<i>Déployabilité / CI-CD</i>	Facilité et fréquence des déploiements sûrs.		
<i>Testabilité</i>	Possibilité d'automatiser tests unitaires/intégration.		
<i>Portabilité</i>	Pouvoir déployer sur cloud / on-prem sans gros changement.		

Vers des applications d'entreprise: (Pourquoi)



Exigence	Exemple Morocco Hub	Critère mesurable / test d'acceptation	Priorité
<i>Scalabilité</i>	Supporter pics de recherche pendant les vacances.	Monter de 1→10k requêtes/jour sans >2s de latence ; tests de charge.	<i>Must</i>
<i>Disponibilité</i>	Site accessible 24/7 pour réservations.	SLA = 99.9% mensuel ; monitoring uptime.	<i>Must</i>
<i>Résilience / Tolérance aux pannes</i>	Une panne d'un service partenaire n'arrête pas tout.	Scénario de panne simulée → système degrade gracieusement (test chaos). Mesure temps médian/p99 pour APIs sous charge.	<i>Must</i>
<i>Performance (latence)</i>	Temps de recherche < 2s.	Temps moyen pour corriger bug critique < 1 jour ; couverture tests.	<i>Must</i>
<i>Maintenabilité</i>	Modules clairs pour recherche, paiement.	Tests d'intrusion, chiffrement TLS, conformité RGPD.	<i>Must</i>
<i>Sécurité</i>	Paiement sécurisé, protection des données clients.	Dashboards, alertes (<5min), logs centralisés.	<i>Must</i>
<i>Observabilité (monitoring & logs)</i>	Logs requêtes, alertes paiement échoué.	Contrats OpenAPI, tests d'intégration automatisés.	<i>Should</i>
<i>Interoperabilité / APIs</i>	Connexion API avec hôtels et compagnies aériennes.	Taux d'erreur < 0.1% sur transactions critiques.	<i>Must</i>
<i>Fiabilité</i>	Réservations enregistrées sans perte.	RTO < 1h, RPO < 24h ; procédure testée trimestriellement.	<i>Must</i>
<i>Sauvegarde & Restauration</i>	Restore BDD à T-24h sans perte majeure.	Politique RGPD, registre traitement, audits.	<i>Must</i>
<i>Conformité & Protection des données</i>	Consentement pour stockage carte, conservation données.	Tests UX, score d'accessibilité, tests utilisateurs.	<i>Should</i>
<i>Accessibilité & Usabilité</i>	Site conforme WCAG minimal, parcours de réservation simple.	Pipeline CI, déploiement automatisé, rollback < 10min	<i>Should</i>
<i>Déployabilité / CI-CD</i>	Déploiement automatisé du backend monolithique.		

Vers des applications d'entreprise: Quelle architecture ?



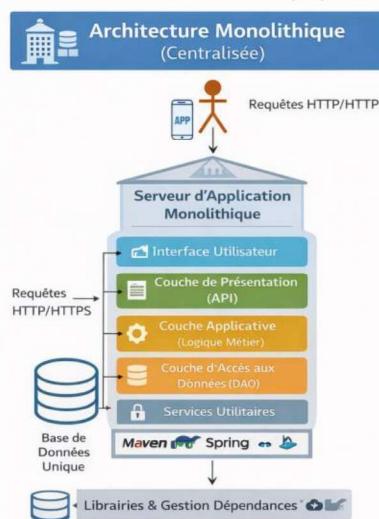
Pr. Hind Lamharhar

68

Vers des applications d'entreprise: Quelle architecture ?



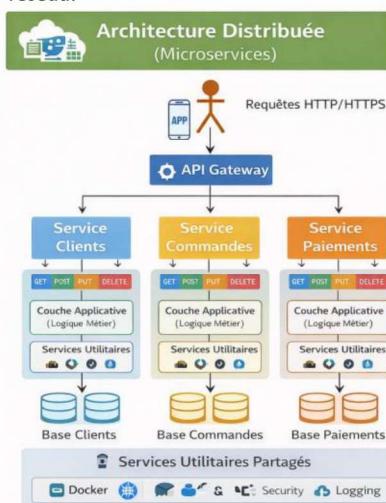
Architecture monolithique regroupe toutes les fonctions (interface, logique métier, données) dans une seule unité déployable.



Caractéristique

- Simplicité de conception et de déploiement
- Facilité de testing en local (tout est dans le même process)
- Moindre besoin d'infrastructure avancée au départ

Architecture distribuée = ensemble de services/applications autonomes (microservices ou services) communiquant par réseau.

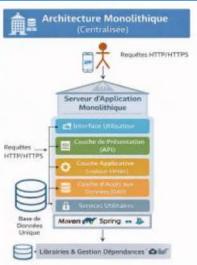
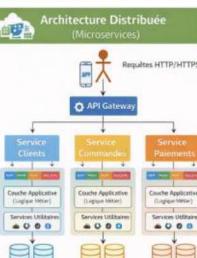


Pr. Hind Lamharhar

69

Architecture Monolithique vs Architecture distribuée



  <p>Pr. Hind Lamharhar</p>	<p>Une application monolithique regroupe :</p> <ul style="list-style-type: none"> le code métier les interfaces les traitements la logique métier l'accès aux données <p>...dans un seul bloc.</p> <p>Une architecture microservices décompose l'application en plusieurs services :</p> <ul style="list-style-type: none"> indépendants autonomes focalisés sur un domaine métier communiquant via des API (REST, etc.) <p>Chaque microservice possède :</p> <ul style="list-style-type: none"> sa propre base de données (ou schéma) son propre cycle de développement son propre déploiement <p>Communication réseau : appels synchrones (HTTP/REST) ou asynchrones (messages / events).</p>	<p>✓ Avantages</p> <ul style="list-style-type: none"> Simple à développer pour les petites équipes Une seule base de code Déploiement facile-unique (un seul fichier ; un jar / un war à mettre en production) Performances internes élevées 	<p>✗ Inconvénients</p> <ul style="list-style-type: none"> Difficile à maintenir si le projet devient grand Chaque changement nécessite un déploiement complet Risques d'effets de bord entre modules Scalabilité limitée Une seule panne peut arrêter tout le système
--	--	---	---

70

Architecture Monolithique vs Architecture distribuée : MoroccoTravel Hub



 <p>L'application MoroccoHub sera être construite comme</p> <ul style="list-style-type: none"> un seul backend qui gère tout : <ul style="list-style-type: none"> Authentification Recherche Réservation Paiement Recommandations une seule base de données (MoroccoHubDB) un seul code source un seul déploiement (MoroccoWAR) 	 <p>MoroccoHub peut être découpé en microservices :</p> <ul style="list-style-type: none"> Service Client Service Hôtel Service Vol Service Réservation Service Paiement Service Recommandation
--	---

Pr. Hind Lamharhar

71

Architecture Monolithique: Exemple



◆ de la couche Métier → Modules métier

- les processus deviennent des **modules internes** :
- Module "Recherche"
- Module "Réservation"
- Module "Paiement"
- Module "Administration"

◆ de la couche Application → Interfaces différentes

- Même si on a 3 interfaces :
 - Web
 - Mobile
 - Back-office
- elles utilisent le **même backend monolithique**.

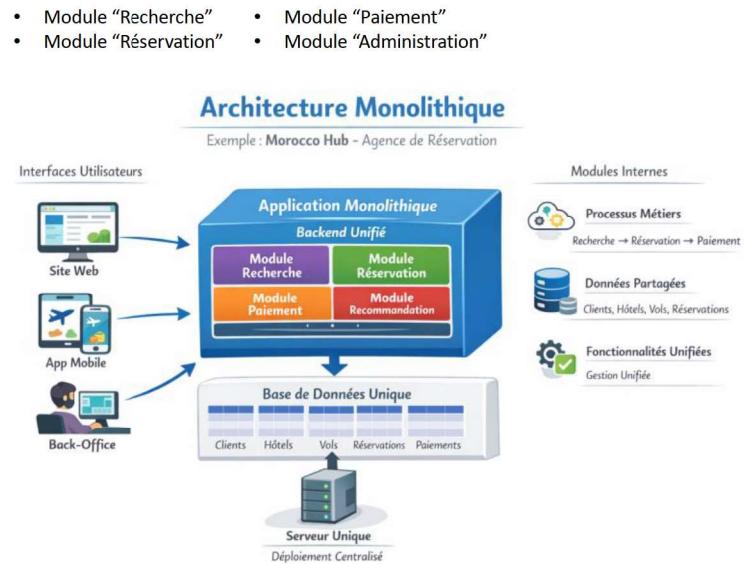
◆ de la couche Données → Tables de la base de données

Les entités deviennent des **tables dans une seule BD**:

- Client
- Hôtel
- Vol
- Réservation
- Paiement

◆ de la couche Technique → Un seul déploiement

- tourne sur un seul serveur
- utilise un seul serveur de BD
- déploiement en un seul bloc (WAR/JAR)



Pr. Hind Lamharhar

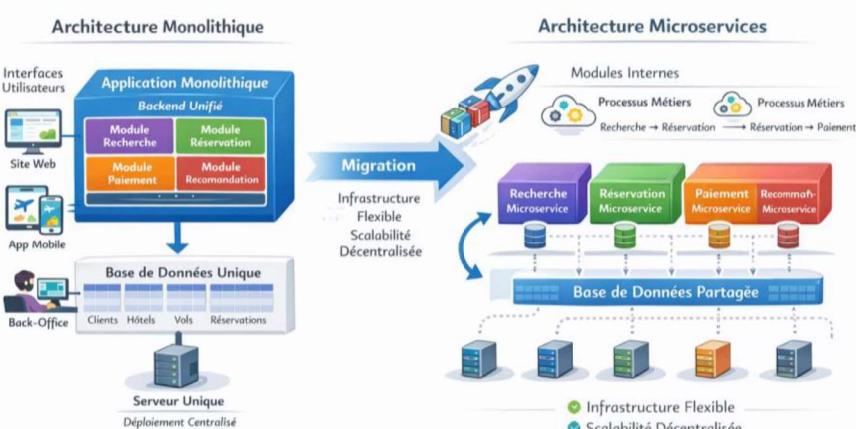
72

Vers les microservices



Architecture Microservices

Exemple : Morocco Hub - Agence de Réservation



Pr. Hind Lamharhar

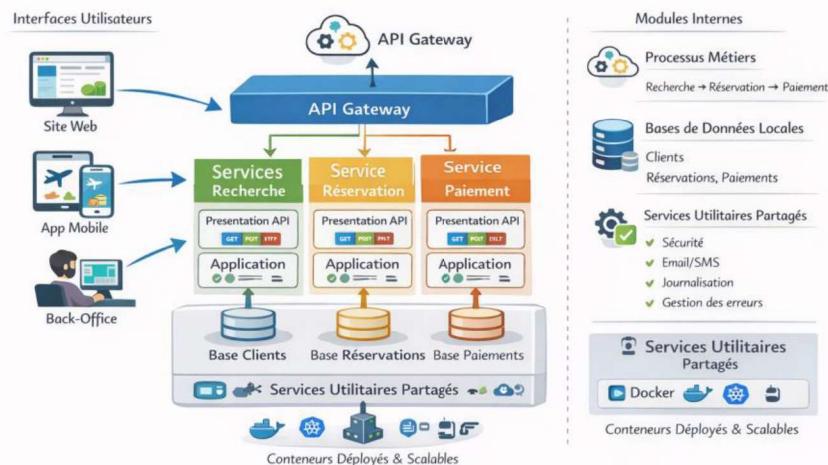
73

Architecture distribuée: Microservices



Architecture Microservices

Exemple : Morocco Hub - Agence de Réservation



Pr. Hind Lamharhar

74

Choix de l'architeccture: (Pourquoi)



Exigence Architecture Monolithique

Scalabilité Scalabilité horizontale possible par réplication de l'instance complète, mais coûteuse car toute l'application est répliquée même si une seule partie nécessite plus de ressources.

Disponibilité SLA global : panne d'un composant = panne totale. Redondance via réplication complète de l'application.

Résilience / Tolérance Point de défaillance unique. Une panne dans un module impacte tout le système.

Performance (latence) Appels internes *in-process* très rapides, aucune latence réseau entre modules.

Maintenabilité Codebase unique avec complexité croissante. Refactoring risqué et coûteux.

Sécurité Authentification et autorisation centralisées. Audit facilité sur une base unique.

Observabilité (Monitoring & Logs) Logs centralisés, supervision simple d'une instance unique.

Interopérabilité / APIs APIs externes unifiées. Interfaces internes non exposées.

Fiabilité Fiabilité binaire : tout fonctionne ou tout échoue.

Sauvegarde & Sauvegarde Sauvegarde unique de la base de données.

Restauration Restauration simple mais globale.

Conformité & Protection des données Audit centralisé. Gestion globale des données sensibles.

Accessibilité & Usabilité Interface utilisateur unifiée, cohérence fonctionnelle et visuelle garantie.

Déployabilité / CI-CD Pipeline unique. Déploiement global (*all or nothing*). Rollback simple.

Testabilité Tests d'intégration simples dans un environnement unique. Tests E2E directs.

Portabilité Déploiement simple (JAR/WAR). Peu de dépendances d'infrastructure.

Architecture Distribuée (Microservices)

Scalabilité Scalabilité granulaire : possibilité de scaler uniquement les services critiques (ex. Payment Service pendant les pics).

Disponibilité Disponibilité par service. Isolation des pannes via *circuit breakers* et stratégies de *retry/backoff*.

Résilience Résilience par isolation. Déploiement multi-zones, *circuit breaker*, *bulkhead*, *fallback*.

Latence Latence réseau entre services. Nécessite optimisation des APIs, *caching*, *gRPC*, asynchrone.

Codebases Codebases plus petites et indépendantes. Équipes autonomes par service, dépendances explicites via APIs.

Sécurité Sécurité distribuée : JWT, OAuth2, API Gateway, mTLS. Surface d'attaque plus large à sécuriser.

Observabilité Observabilité distribuée complexe : *distributed tracing*, logs agrégés, métriques par service.

APIs APIs natives par conception. Intégration naturelle avec partenaires et systèmes externes.

Fiabilité Fiabilité partielle : un service défaillant n'arrête pas l'ensemble du système.

Sauvegardes Sauvegardes par service/base. Restauration complexe nécessitant coordination inter-services.

Conformité Conformité par service. Complexité accrue pour garantir RGPD et traçabilité des données distribuées.

Risque Risque d'incohérence UI si services développés indépendamment sans gouvernance.

CI/CD CI/CD par service. Déploiements indépendants. Orchestration des versions plus complexe.

Tests Tests plus complexes : mocks, stubs, contrats, E2E sur système distribué.

Conteneurisation Conteneurisation (Docker) + orchestration (Kubernetes) réduisant la portabilité "simple".



Quiz

- | | | |
|--|---|---|
| 1. Qu'est-ce qu'une architecture monolithique ? | 7. En cas de panne d'un module dans un monolithe : | 13. Quelle architecture nécessite des outils comme Kubernetes ? |
| a) Une application composée de services indépendants
b) Une application déployée comme un seul bloc
c) Une application uniquement basée sur des APIs | a) Seul le module tombe
b) Toute l'application peut être impactée
c) Les autres modules continuent toujours à fonctionner | a) Monolithique
b) Microservices
c) Client-Server simple |
| 2. Le principal avantage du monolithe est : | 8. Le déploiement d'un monolithe est : | 14. Quelle architecture facilite l'autonomie des équipes ? |
| a) Sa complexité
b) Sa simplicité de développement initial
c) Sa forte scalabilité granulaire | a) Indépendant par fonctionnalité
b) Global (tout ou rien)
c) Automatique sans tests | a) Monolithique
b) Microservices |
| 3. Quelle architecture est la plus simple à déployer au début ? | 9. Une architecture microservices est composée de : | 15. Une application JEE monolithique est généralement : |
| a) Microservices
b) Hybride
c) Monolithique | a) Un seul déploiement
b) Plusieurs services indépendants
c) Un seul service avec plusieurs bases de données | a) Déployée en plusieurs conteneurs indépendant
b) Déployée comme une seule application (WAR / EAR)
c) Composée uniquement de microservices |
| 4. La scalabilité en microservices est : | 10. Le rôle de l'API Gateway est : | 16. Dans JEE, la logique métier se trouve principalement dans : |
| a) Impossible
b) Globale uniquement
c) Granulaire par service | a) Stocker les données
b) Gérer les requêtes clients et la sécurité
c) Remplacer les microservices | a) Servlets
b) EJB / Services métiers
c) JSP |
| 5. En microservices, chaque service : | 11. Dans une architecture monolithique JEE, les couches classiques sont : | 17. Quel format est typiquement utilisé pour déployer un monolithe JEE ? |
| a) Doit partager la même base de données
b) Peut être déployé indépendamment
c) Ne peut pas être mis à jour seul | a) UI – Business – Data
b) Client – Serveur – Cloud
c) API – Gateway – Services | a) Docker Image uniquement
b) WAR ou EAR
c) YAML |
| 6. Quelle architecture permet de scalar uniquement une partie du système ? | 12. La communication entre microservices se fait généralement via : | |
| a) Monolithique
b) Microservices
c) Aucune | a) Appels internes Java
b) APIs REST / Messaging
c) Variables globales | |



Q1. Un monolithe peut être scalé horizontalement.	Q6. Les microservices sont plus simples à tester que les monolithes.
<input type="checkbox"/> Vrai <input type="checkbox"/> Faux	<input type="checkbox"/> Vrai <input type="checkbox"/> Faux
Q2. En microservices, une panne d'un service entraîne toujours la panne totale.	Q7. Chaque microservice peut avoir sa propre base de données.
<input type="checkbox"/> Vrai <input type="checkbox"/> Faux	<input type="checkbox"/> Vrai <input type="checkbox"/> Faux
Q3. Les microservices communiquent principalement via le réseau.	Q8. Le monolithe est inadapté pour les petites applications.
<input type="checkbox"/> Vrai <input type="checkbox"/> Faux	<input type="checkbox"/> Vrai <input type="checkbox"/> Faux
Q4. Le monolithe est toujours plus performant que les microservices.	Q9. Les microservices augmentent la complexité opérationnelle.
<input type="checkbox"/> Vrai <input type="checkbox"/> Faux	<input type="checkbox"/> Vrai <input type="checkbox"/> Faux
Q5. Les EJB facilitent la séparation de la logique métier.	Q10. Le choix d'architecture dépend du contexte et des exigences.
<input type="checkbox"/> Vrai <input type="checkbox"/> Faux	<input type="checkbox"/> Vrai <input type="checkbox"/> Faux