Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université de Jijel
Faculté des Sciences exactes et de l'informatique
Département d'informatique

– Module –
Environnements et Programmation Dédiés

Master 1 : IA

Enseignant du module : Dr. Hemza FICEL

Contact: hemza.ficel@univ-jijel.dz

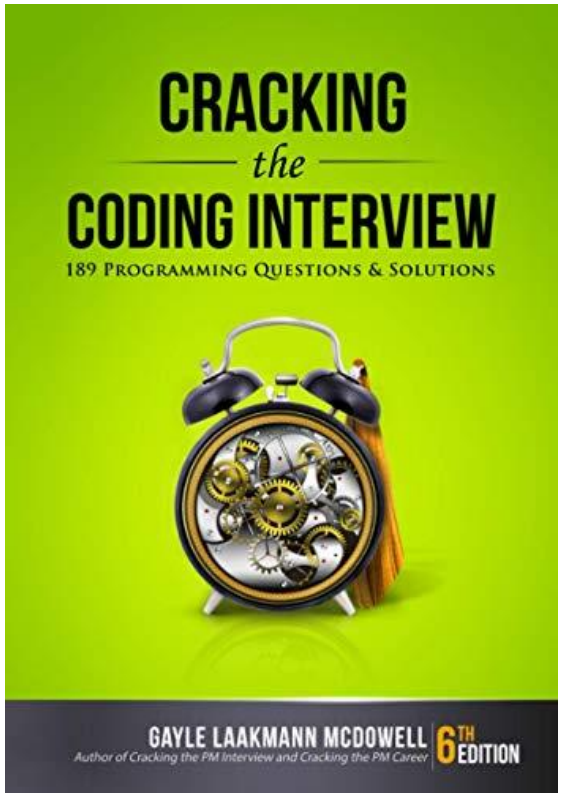
amazon

Google
INTERVIEW
QUESTIONS

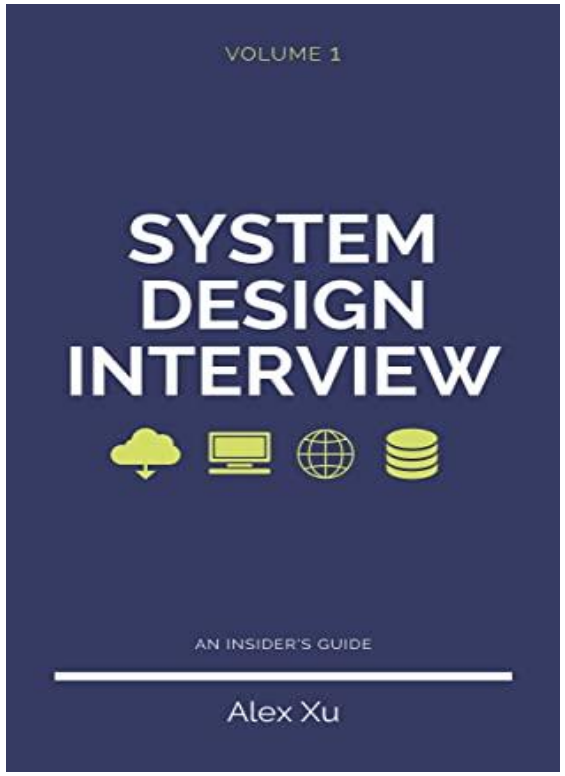
Microsoft

LinkedIn

2 types de compétences …



La résolution de problèmes (algorithmes, structures de données, paradigmes de programmation, …).



La conception architecturale (Ingénierie logicielle, architectures réseau, architectures logicielles, …)

Rappel …

Si vous êtes sollicité pour développer une application WEB

Quelle architecture allez-vous adopter ?

Votre application

Client

Serveur

Architecture client-serveur

Rappel

- **Architecture client-serveur :** est une **architecture logicielle** qui désigne une façon de communication entre plusieurs applications. Cette architecture divise une application en deux parties distinctes : la partie **client** qui émet des requêtes et la partie **serveur** qui traite et répond à ces requêtes.

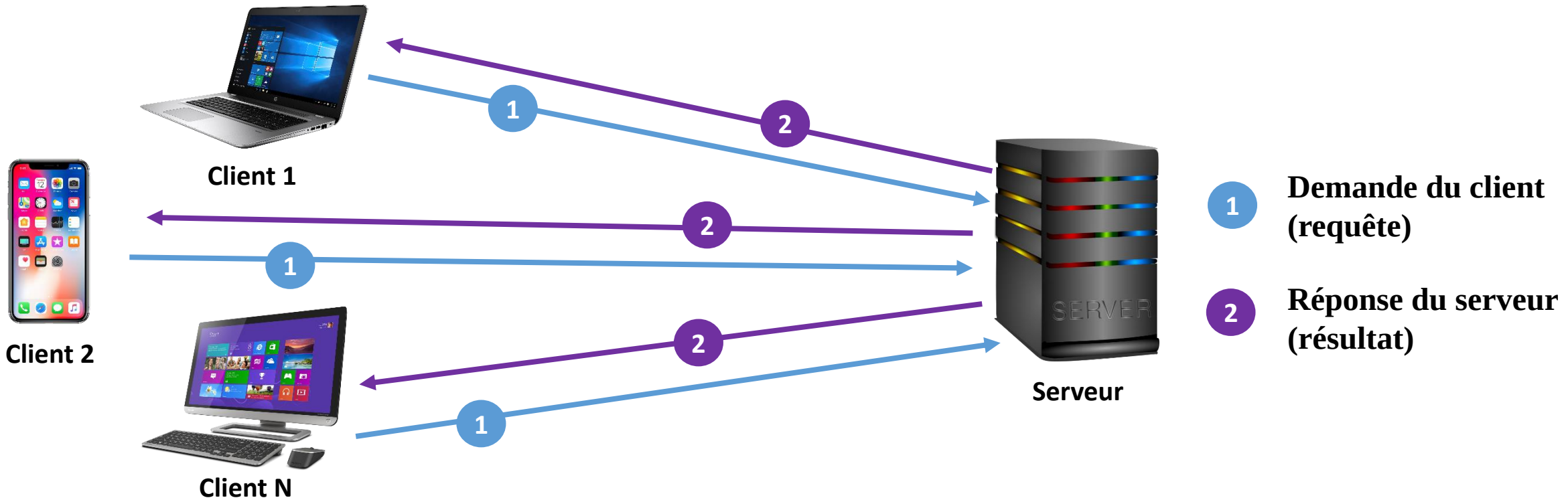
Rappel

- **Serveur** : une machine généralement très puissante (stockage, calcul, mémoire, …) qui **offre des services** sur un réseau (des programmes qui traitent des demandes et fournissent des résultats). Généralement, le serveur est identifié par une adresse et chaque service est désigné par un port.

Les types de l’architecture client-serveur ?

Rappel

Architecture à 2 niveaux (2 tiers) : une architecture client-serveur est dite **à 2 niveaux** lorsque toutes les ressources nécessaires pour traiter les requêtes des clients sont présentes sur un seul serveur.



Exemple d'architecture à 2 niveaux : la consultation des pages web

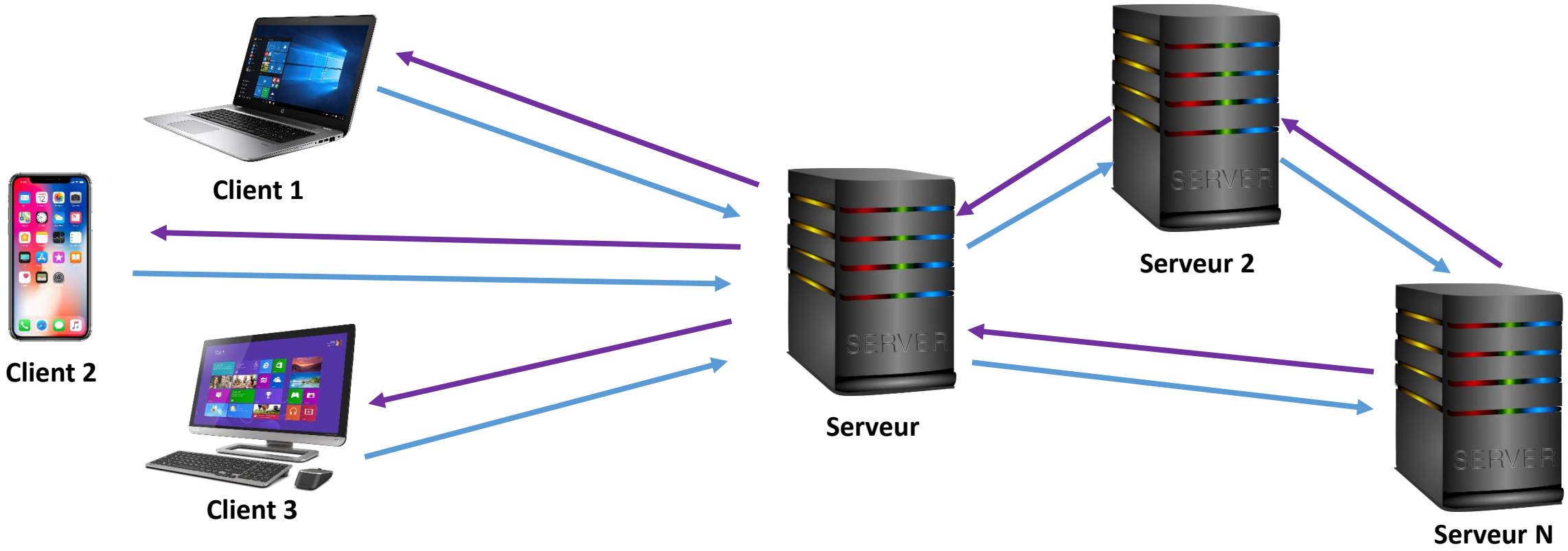
Navigateur web
du client

Réseau : internet

Serveur web
Adresse : www.site.com
port du service : 80

Rappel

- **Architectures à N niveaux (N tiers) :** une architecture client-serveur est dite **à N niveaux** lorsque les ressources nécessaires pour traiter les requêtes des clients sont présentes sur plusieurs serveurs.

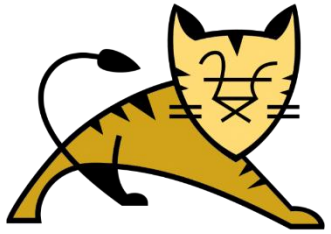


Rappel

- **Un serveur d'applications:** est un logiciel qui offre un environnement d'exécution pour des applications Web.

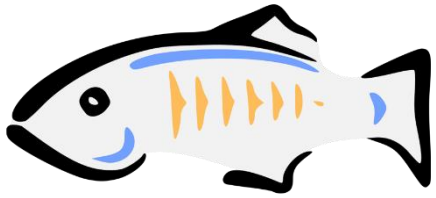
Exemples de serveur d'applications

Serveurs d'applications compatibles JAVA (JEE)



Tomcat

Apache Software Foundation

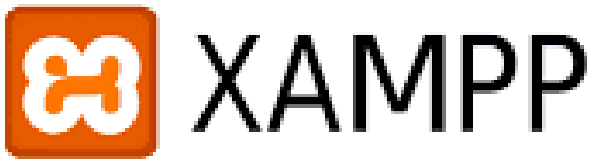


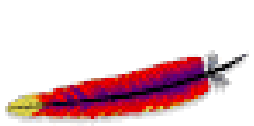
GlassFish

Oracle/ Eclipse Foundation

Attention

- Java est un langage intermédiaire qui nécessite un environnement d'exécution spécialisé (un serveur d'application), tandis que PHP est **un langage interprété** qui peut être exécuté directement par **un serveur web qui intègre un Interpréteur PHP** (module PHP).





Ils ne sont pas des serveurs d'applications, mais des environnements de travail (ensemble de logiciels) qui intègrent un module PHP permettant d'interpréter le code PHP **!!**

Chapitre 5
Architecture orientée services

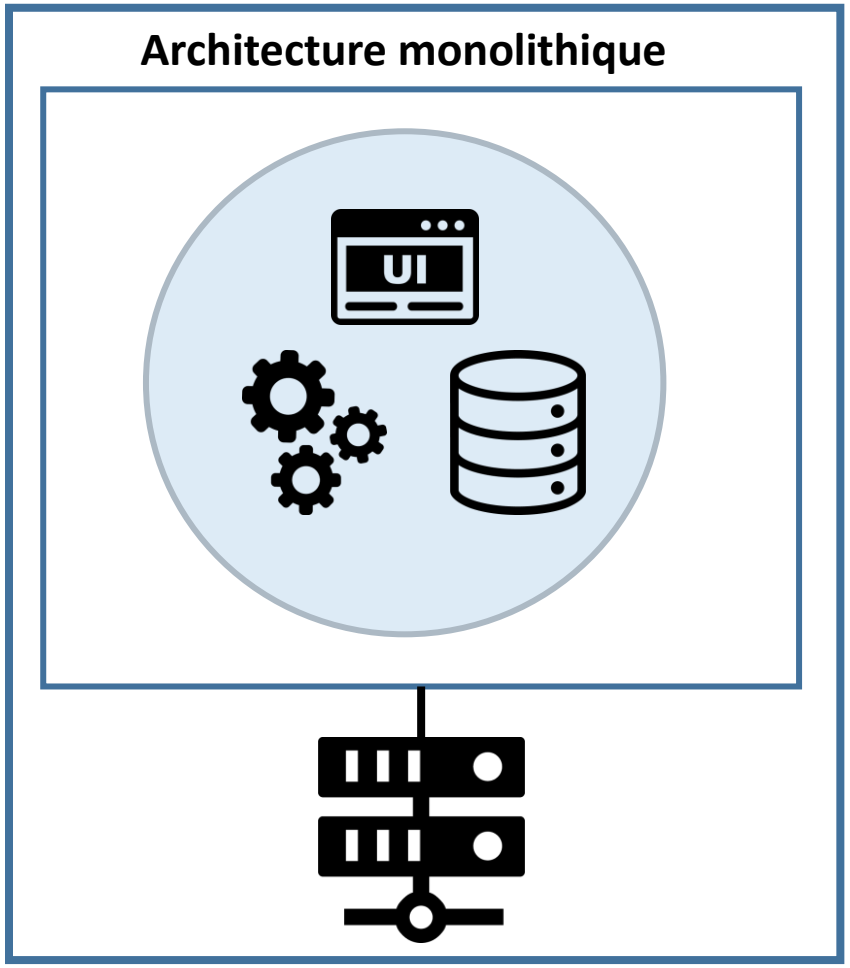
Architecture monolithique vers SOA

Les besoins croissants en termes de scalabilité, de flexibilité et d'efficacité dans le développement logiciel

Architecture monolithique vers SOA

Les premières applications étaient conçues comme des **blocs monolithiques simple**, où toutes les fonctionnalités étaient contenues dans une seule base de code …

Architecture monolithique vers SOA



Monolithe simple : Adapté aux petites applications autonomes.

Architecture monolithique vers SOA

Limites

- **Faible modularité** : Les modifications dans une partie du programme nécessitaient souvent des révisions importantes.
- **Couplage fort** : Tous les composants étaient interdépendants.
- **Maintenance difficile** : Toute mise à jour ou correction de bug impliquait un déploiement complet du système.

Architecture monolithique vers SOA

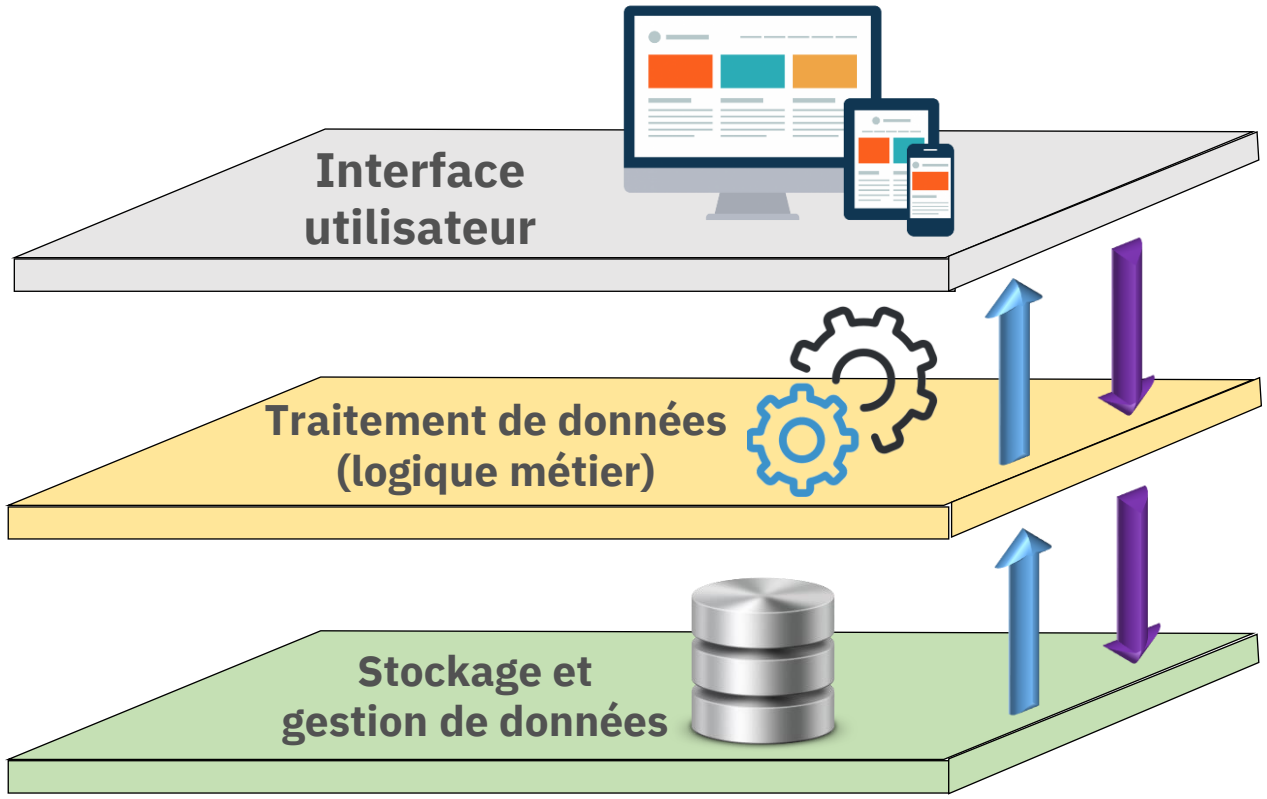
Architecture Monolithique à Plusieurs Couches : séparer la logique en couches distinctes (Présentatio, Logique métier, Accès aux données)

Architecture monolithique vers SOA

Couche de présentation (partie visible)

Couche de traitement (partie fonctionnelle)

Couche d'accès aux données



Poste de travail du client

Serveur d'applications

Serveur de base de données

Monolithe multicouche : Réponse à des applications plus complexes et au besoin de modularité.

Architecture monolithique vers SOA

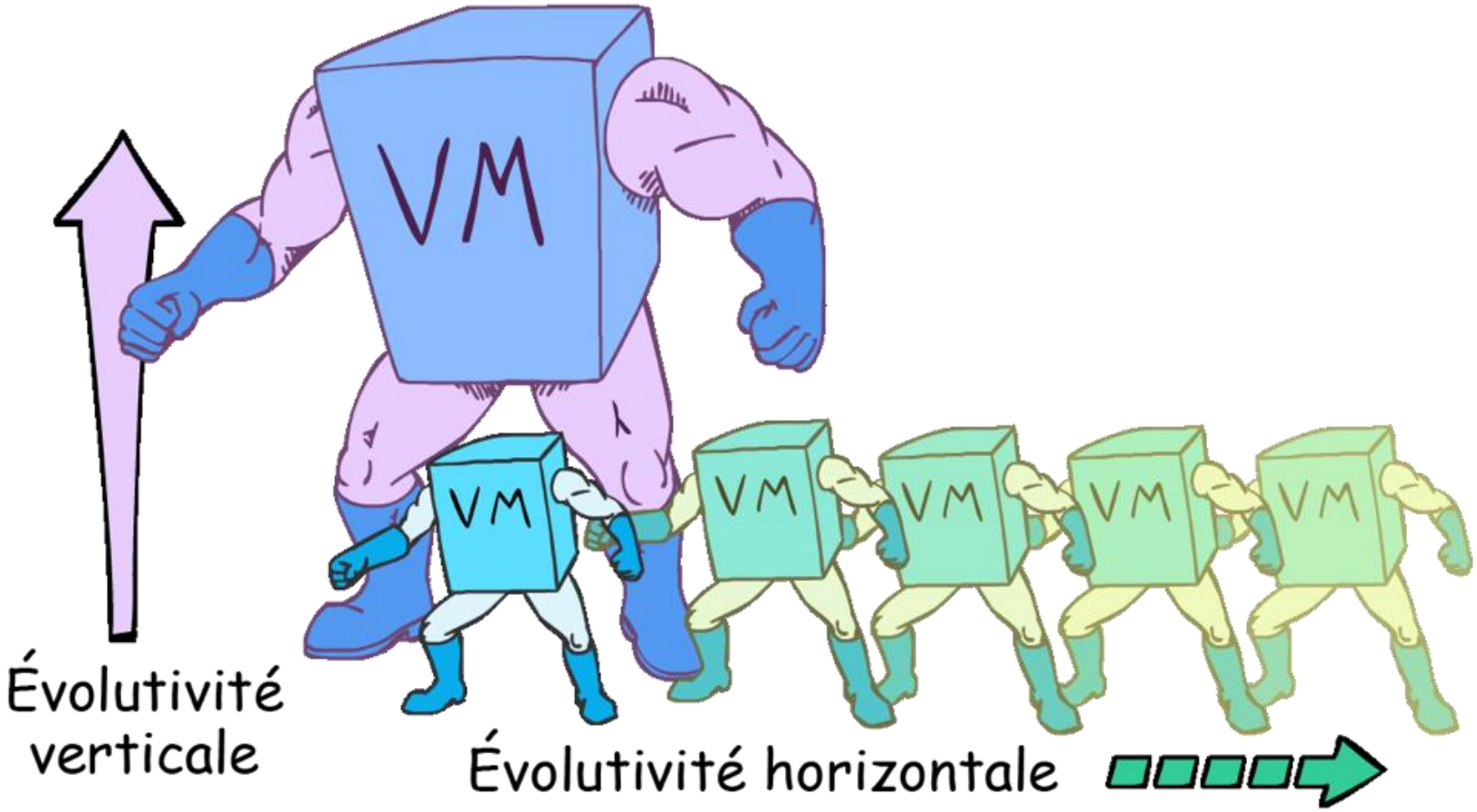
Avantages

- Modularité accrue : Les couches pouvaient être développées et testées indépendamment.
- Séparation des responsabilités : Amélioration de la maintenabilité.

Limites

- Toujours un déploiement monolithique : Une seule base de code pour l'ensemble des couches.
- **Scalabilité verticale** : Nécessite des machines plus puissantes pour supporter la charge croissante.

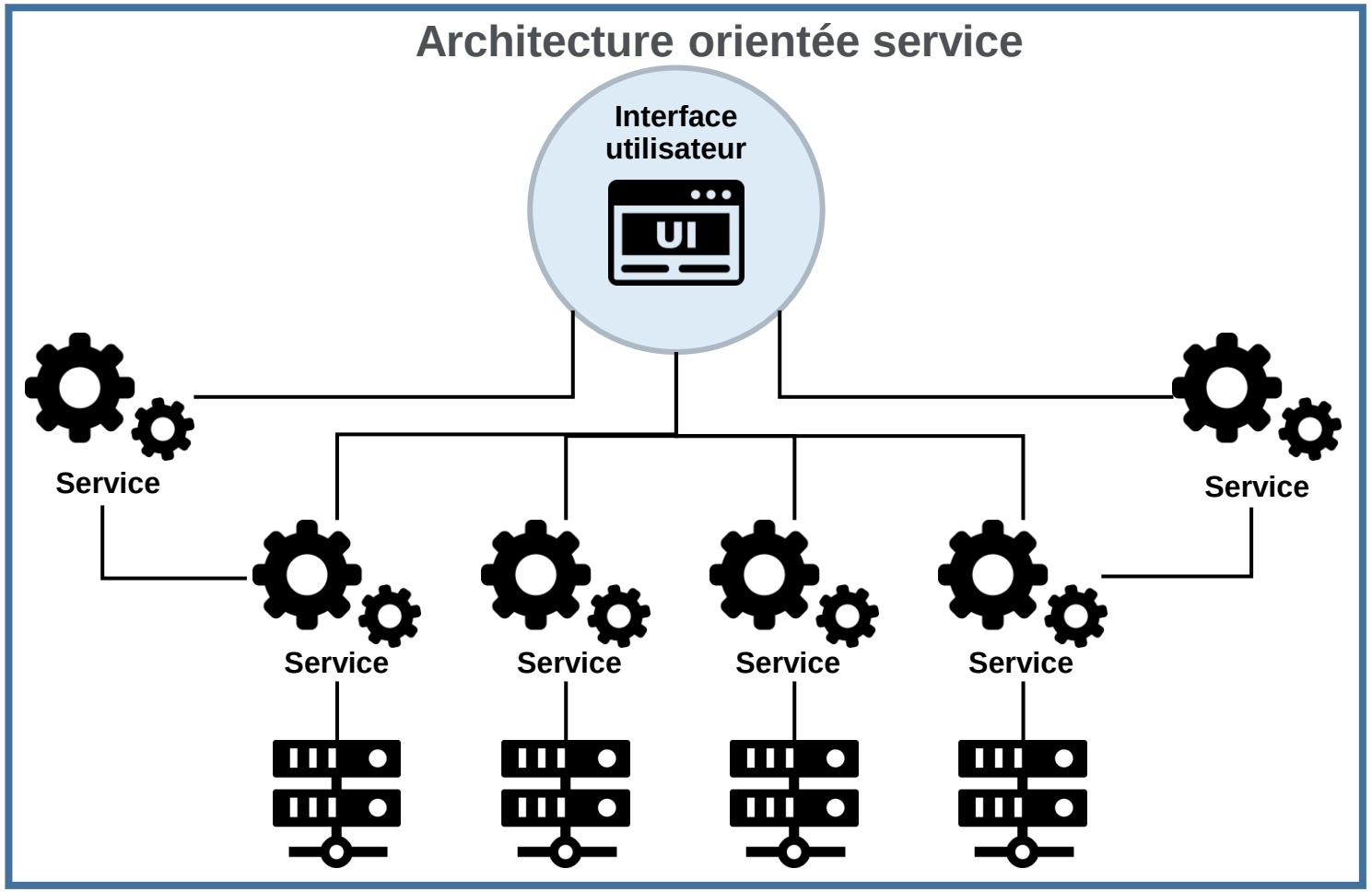
Architecture monolithique vers SOA



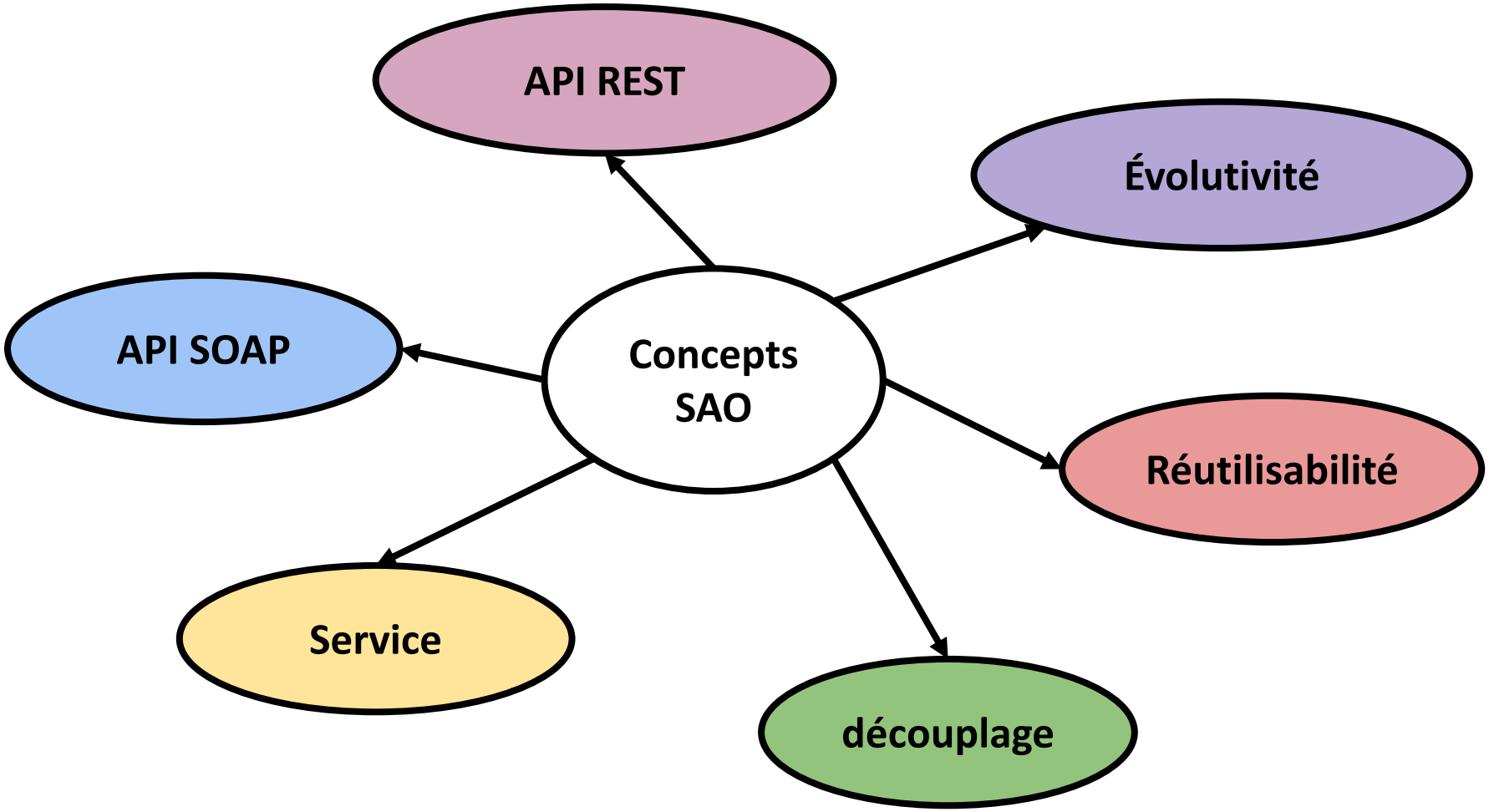
Architecture monolithique vers SOA

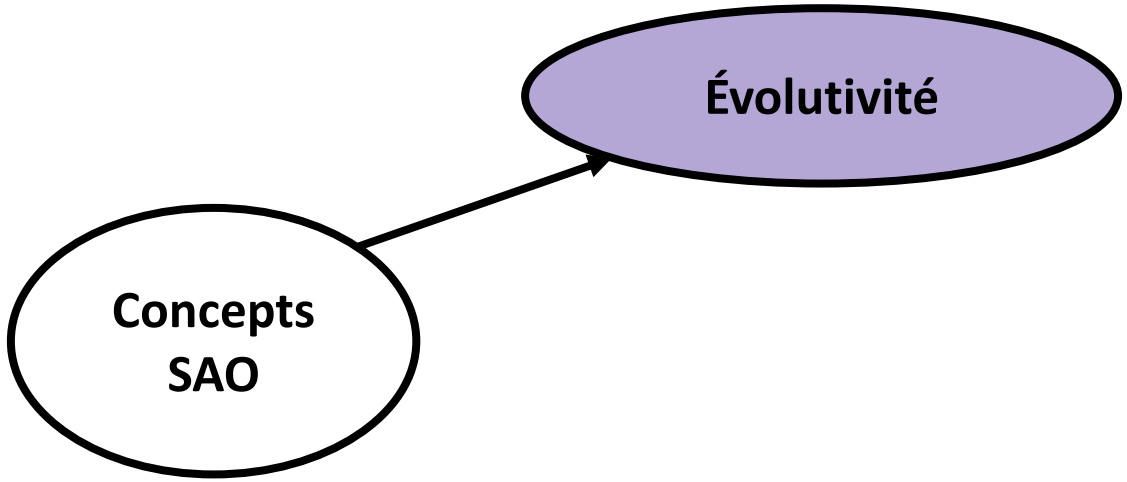
L’architecture orientée services (Architecture Distribuée) :
Orientation métier et flexibilité pour répondre aux besoins d’interopérabilité.

Architecture monolithique vers SOA



Concepts de l'architecture orientée services




Évolutivité
Concepts
SAO

Principe d’évolutivité

Fondamental

Le principe **« d’évolutivité »** se réfère à la capacité d’une application à s’adapter facilement pour prendre en charge de nouvelles fonctionnalités ou mises à jour **sans perturber le fonctionnement des fonctionnalité existantes**. Ce principe est un facteur crucial à prendre en compte lors du processus de développement des applications, car il permet de répondre **aux besoins changeants des entreprises.**

Principe d’évolutivité

Imaginez le scénario suivant …

En tant que développeur, vous êtes sollicité pour développer une application WEB AlphaDZ qui doit fournir deux fonctionnalités principales : A et B. Lors de votre réunion de recueil de besoin, le client vous informe que d'autres fonctionnalités pourraient être ajoutées à l'application dans le futur.

Principe d’évolutivité

L'application doit être conçue pour être évolutive, afin de pouvoir facilement ajouter de nouvelles fonctionnalités. C'est pourquoi votre code doit être modulaire, réutilisable et facile à maintenir.

Le langage Java offre une structure idéale pour ces exigences, grâce à sa nature orientée objet qui favorise la modularité, la réutilisabilité et la facilité de maintenance du code.

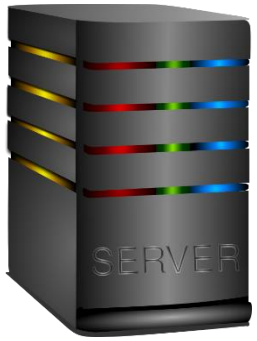
Principe d’évolutivité

Besoins :

- Une application **WEB AlphaDZ.**
- Elle doit fournir **deux fonctionnalités : A et B.**
- D'autres fonctionnalités pourraient être ajoutées dans le futur **: principe d’évolutivité (modularité)**

Principe d'évolutivité

L'application AlphaDZ



www.AlphaDZ.com/doA
www.AlphaDZ.com/doB

```
class LogicAlphaDZ {
    ....
    void doA(){};
    void doB(){};
}
```

Principe d’évolutivité

Maintenant, si nous sommes amené à étendre notre application par l'ajout de nouvelles fonctionnalités C et D …

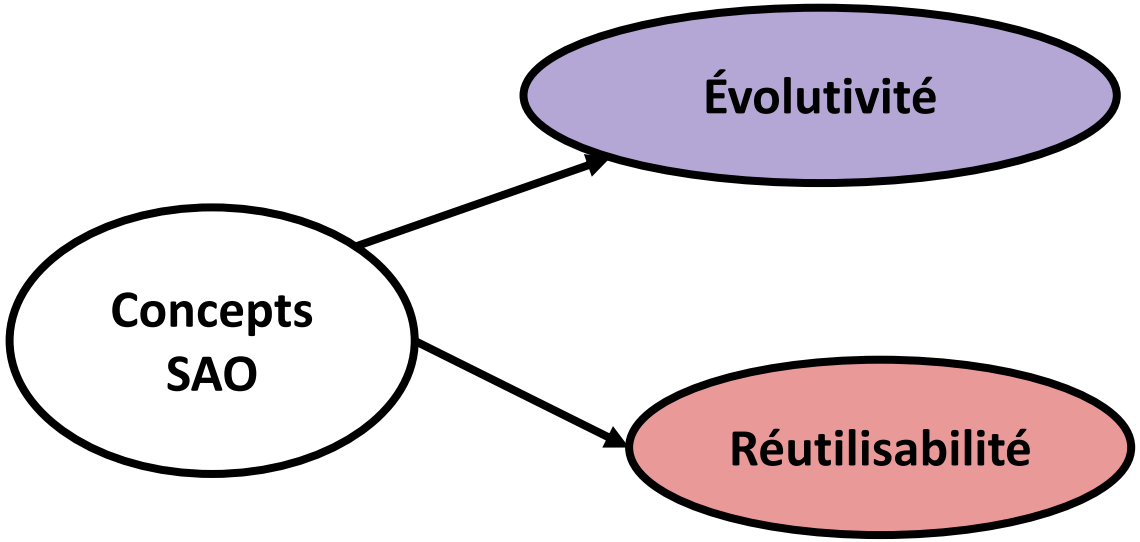
Il suffit juste d’ajouter deux nouvelles méthodes (« C » et « D ») dans notre logique métier, tout en garantissant que l'évolution de l'application n’affecte pas les fonctionnalités existantes.

Principe d'évolutivité

Java nous simplifie la tâche !

www.AlphaDZ.com/doA
www.AlphaDZ.com/doB
www.AlphaDZ.com/doC
www.AlphaDZ.com/doD

```
class LogicAlphaDZ {
    ....
    void doA(){};
    void doB(){};
    void doC(){};
    void doD(){};
}
```


Évolutivité
Concepts
SAO
Réutilisabilité

Principe de réutilisabilité

Fondamental

- Le concept de « **réutilisabilité** » consiste à concevoir vos composants logiciels de manière à ce qu'ils puisse être **utilisés dans différents contextes**. Ce principe joue un rôle crucial dans le processus de développement des applications d'entreprise, car il permet **d'économiser du temps et des efforts de développement.**

Principe de réutilisabilité

Maintenant, supposons que nous somme sollicité une autre fois pour développer une autre application BetaDZ qui doit fournir les fonctionnalités B, C, D, et E

L'application existante AlphaDZ

```
class LogicAlphaDZ {
    ....
    void doA(){};
    void doB(){};
    void doC(){};
    void doD(){};
}
```

Principe de réutilisabilité

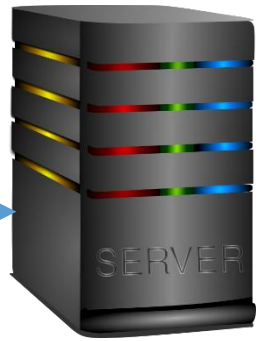
Vu que les fonctionnalités B, C et D sont déjà fournies par l'application AlphaDZ, il serait judicieux de les réutiliser et de ne pas les développer à nouveau (principe de la réutilisabilité).

Principe de réutilisabilité

L'application BetaDZ

BetaDZ

AlphaDZ



1

2

3

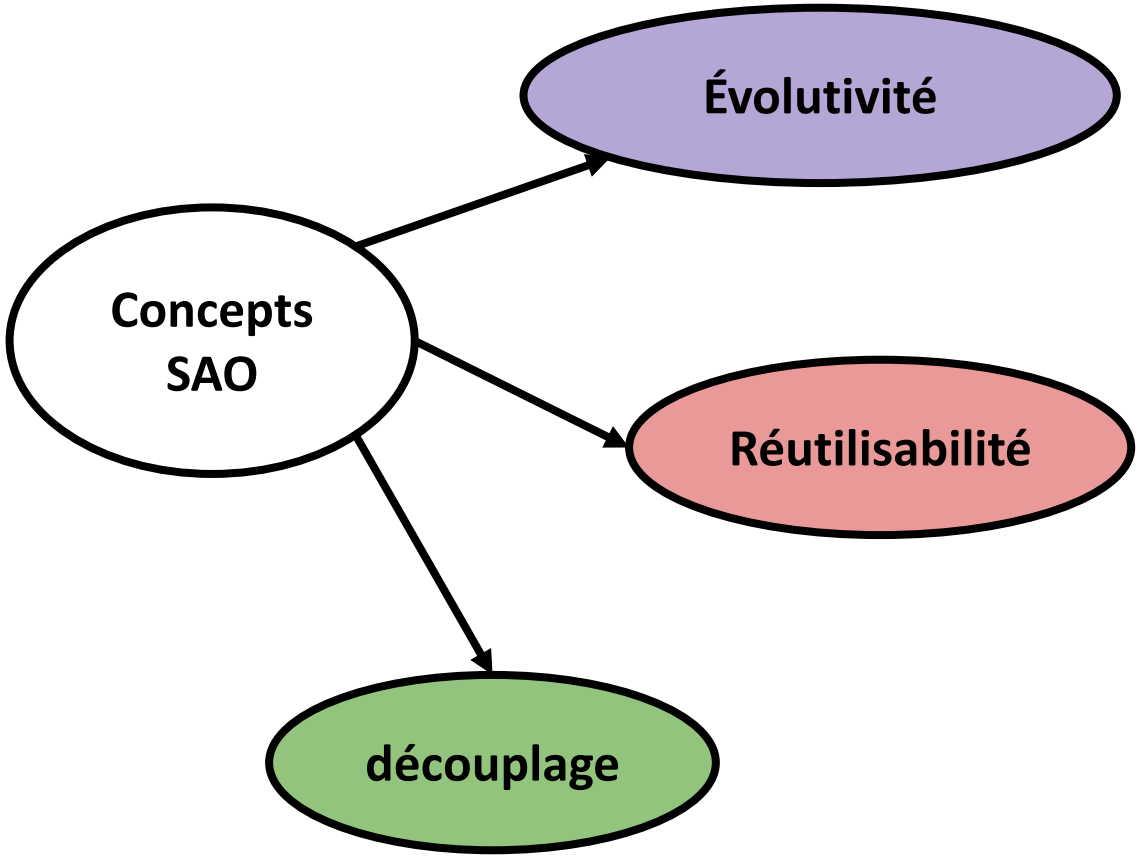
4

www.AlphaDZ.com/doA
www.AlphaDZ.com/doB
www.AlphaDZ.com/doC

www.BetaDZ.com/doA
www.BetaDZ.com/doB
www.BetaDZ.com/doC
www.BetaDZ.com/doE

```
class LogicBetaDZ {
    ....
  void doE(){};
}
```

```
class LogicAlphaDZ {
    void doA(){};
    void doB(){};
    void doC(){};
    void doD(){};
}
```


Évolutivité
Concepts
SAO
Réutilisabilité
découplage

Principe de découplage

Fondamental

Le principe de « découpage » implique la division du code en modules indépendants, créant ainsi des éléments qui peuvent opérer de manière **autonome.** Cette approche vise à **faciliter la flexibilité et la maintenance** des applications en réduisant les interdépendances entre les différents modules.

Principe de découplage

Un scénario réel : L'achat d'un produit sur une plateforme de commerce électronique (p. ex. Amazon, Ebay, Jumia, …)

Principe de découplage

Exemples des achats sur Internet : la collaboration de plusieurs applications appartenant à différentes organisations

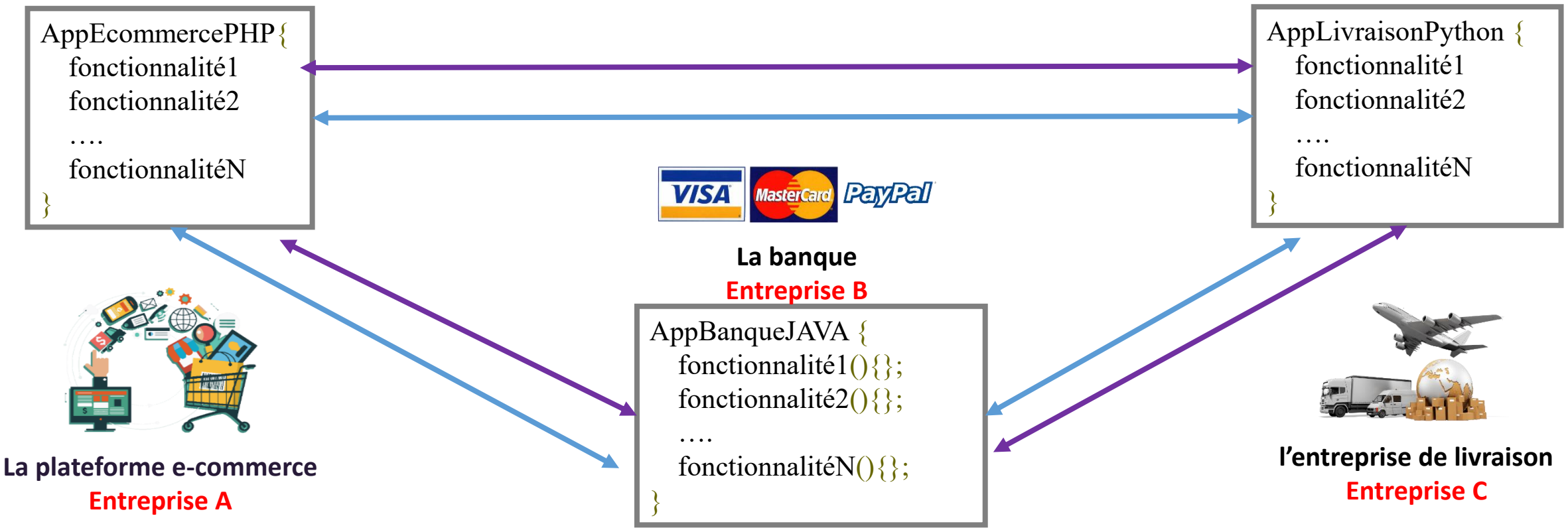
La prise en compte des commandes (La plateforme e-commerce)

La validation du paiement (La banque)

La gestion de la livraison des produits (l'entreprise de livraison)

Principe de découplage

Exemples des achats sur Internet : la collaboration de plusieurs applications appartenant à différentes organisations



Principe de découplage

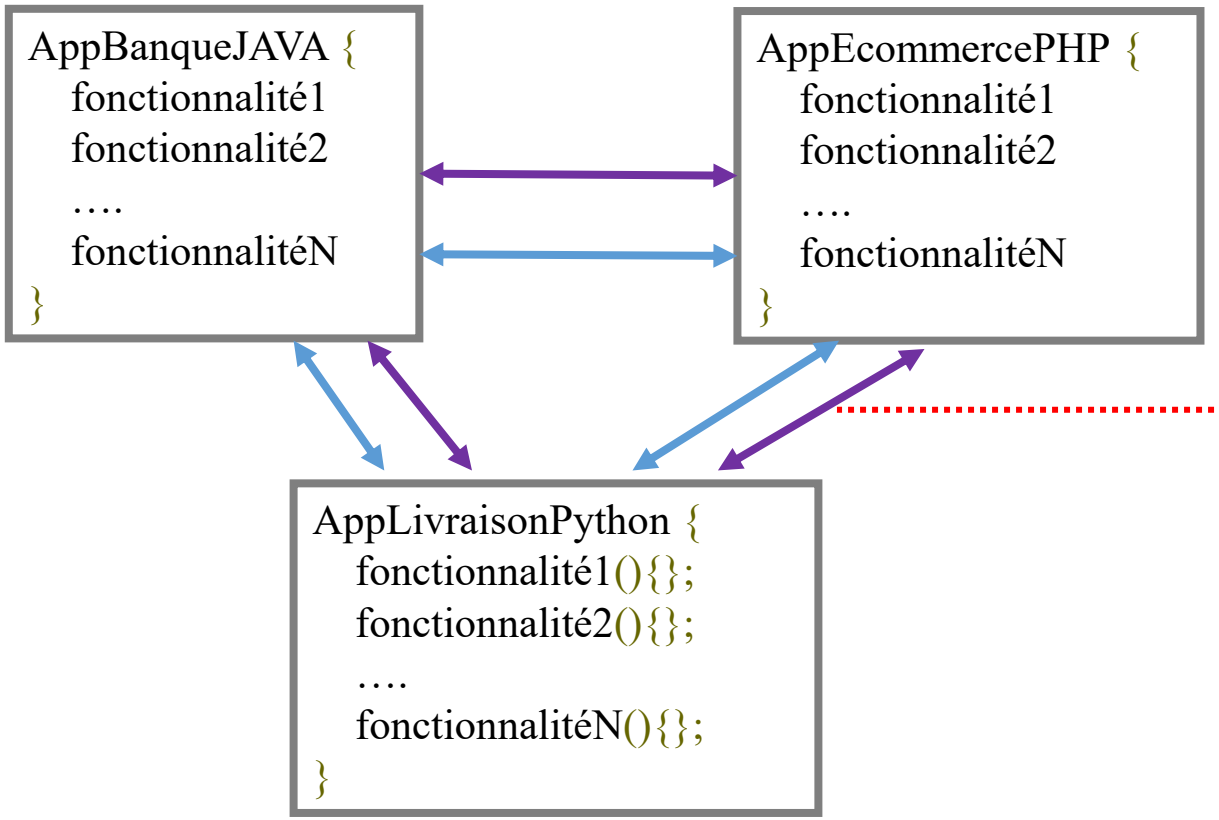
Exemples des achats sur Internet : la collaboration de plusieurs applications appartenant à différentes organisations

Chacune de ces applications est élaborée de **manière autonome** : l'application de l'entreprise A ne dépend pas directement des applications de l'entreprise B ou C, et vice versa.

Ces 3 applications sont **découplés**, ce qui signifie que les changements dans une application n'impactent pas directement les autres. Par exemple, l'entreprise A peut mettre à jour son application de gestion des commandes sans affecter l'application de gestion des paiements de l'entreprise B.

Principe de découplage

Exemples des achats sur Internet : la collaboration de plusieurs applications appartenant à différentes organisations



Attention

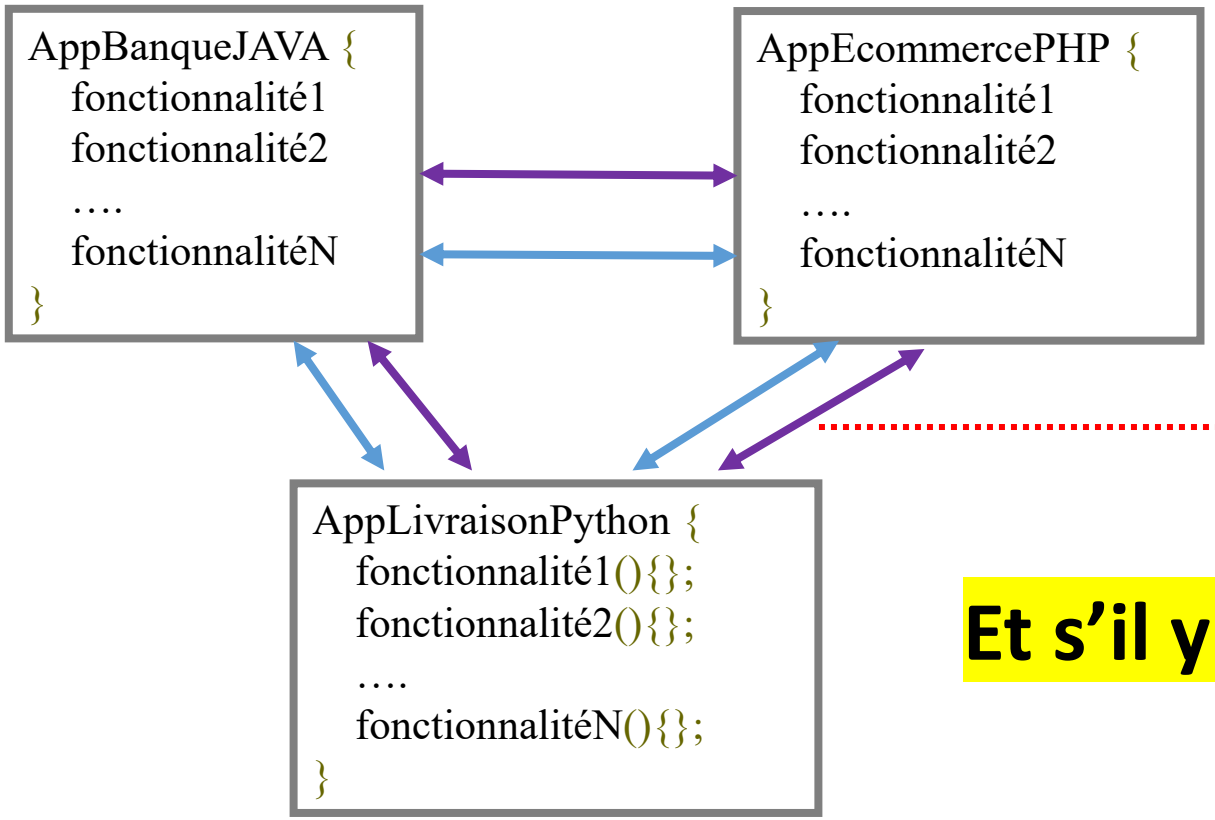
- Les développeurs doivent prévoir une **intégration point-à-point très compliquée** (connectivité, traduction des modèles de données, etc.)

Principe de découplage

Dans le contexte du web actuel, les applications sont de plus en plus complexes et reposent principalement sur l'intégration de plusieurs applications d'entreprises.

Principe de découplage

Exemples des achats sur Internet : la collaboration de plusieurs applications appartenant à différentes organisations

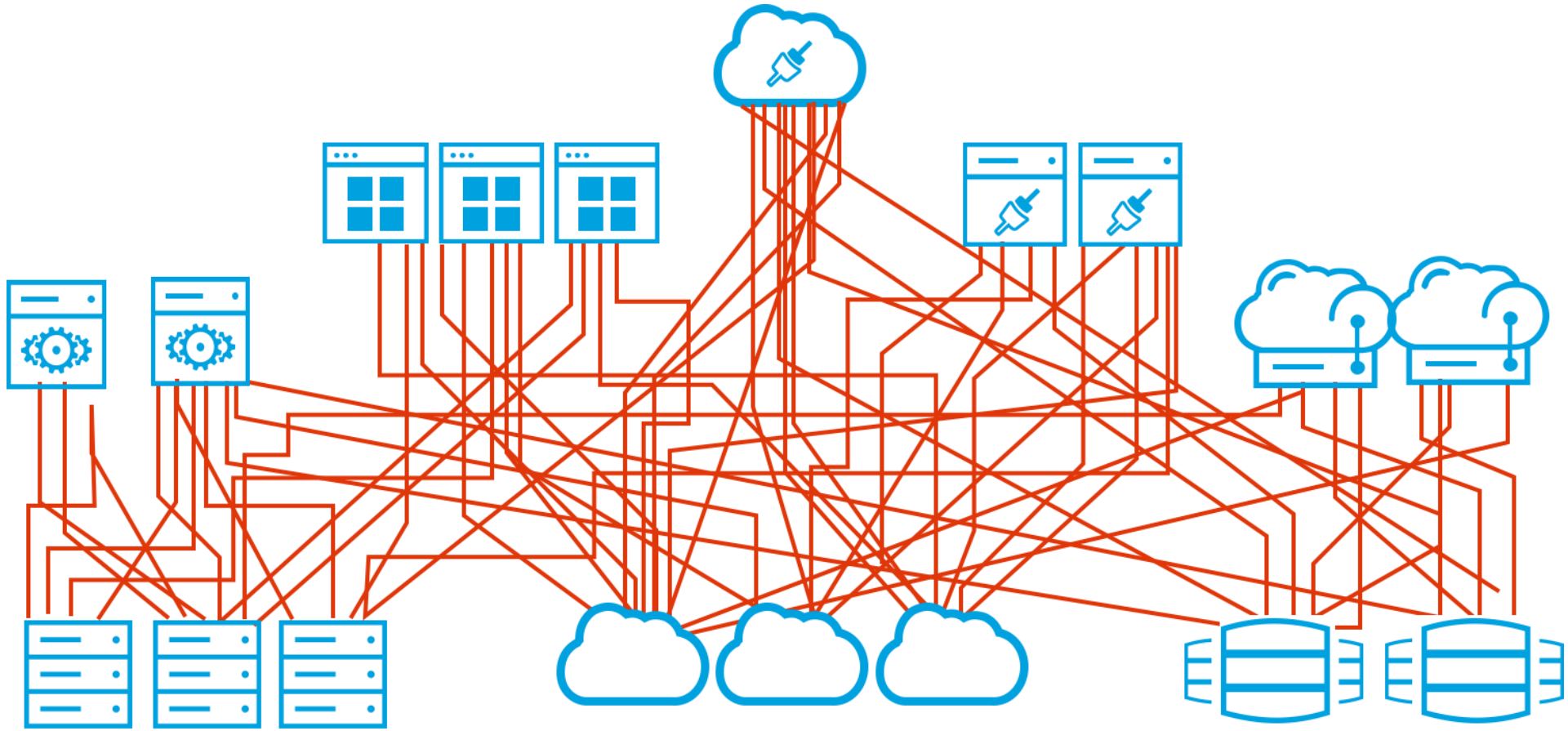


Attention

- Les développeurs doivent prévoir une **intégration point-à-point très compliquée** (connectivité, traduction des modèles de données, etc.)

Et s'il y a d'autres applications à intégrer ?

Principe de découplage



Principe de découplage

- Le système devient trop complexe (**syndrome de spaghetti**) : il y a tellement d'interdépendance, ce qui devient problématique (**chaque logiciel doit gérer des dizaines d'adaptateurs pour communiquer avec les autres applications**).

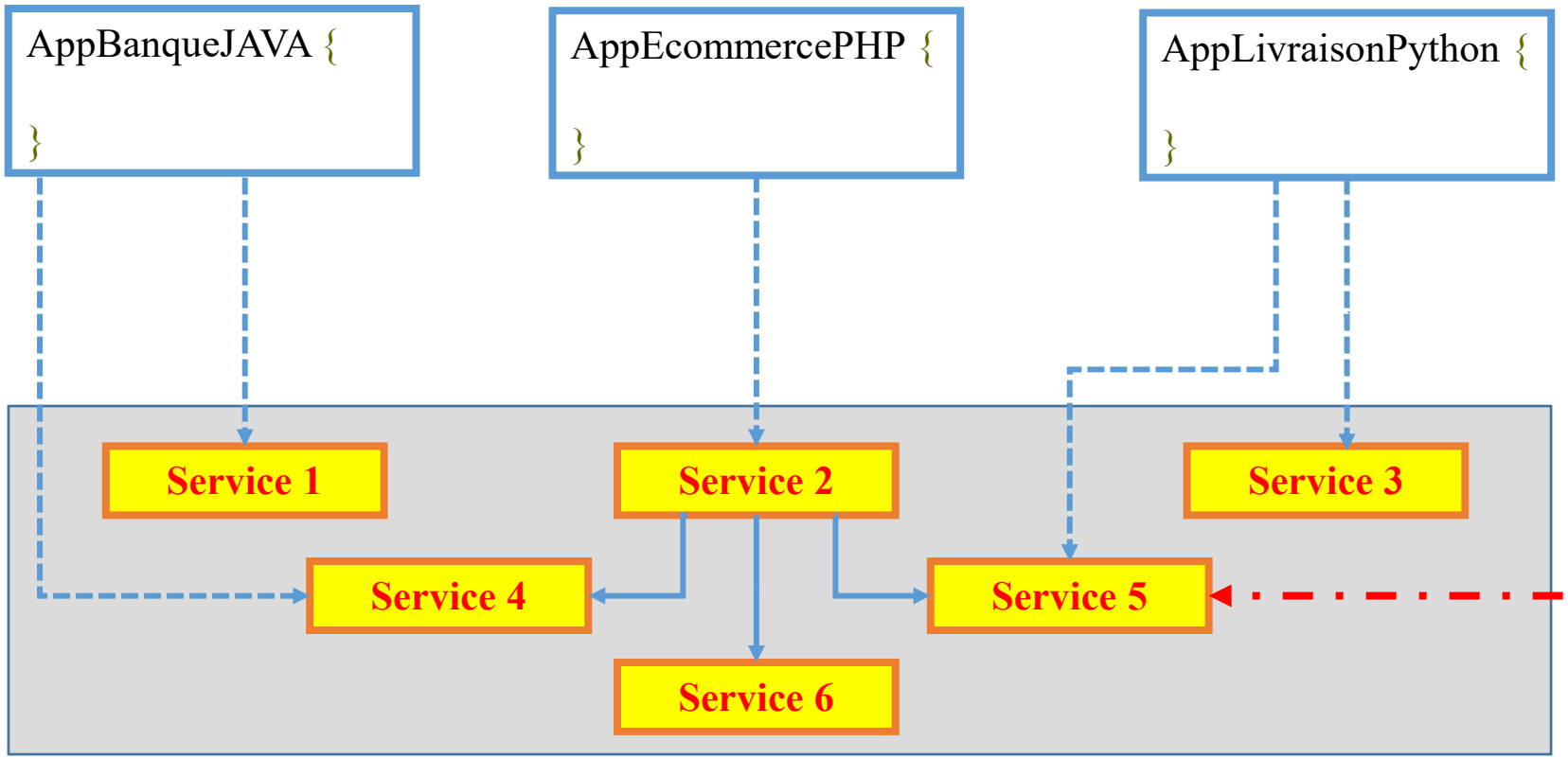
- Faire **évoluer une application devient de plus en plus difficile** : **changer la moindre fonctionnalité dans une application implique des changements dans tout les autres applications connectées avec cette dernière.**

- Le **coût de maintenance des applications devient trop élevé** à cause de l'interdépendance forte entre les composants du système.

Il faut une solution qui favorise la réutilisation, l'évolution et l'intégration des applications.

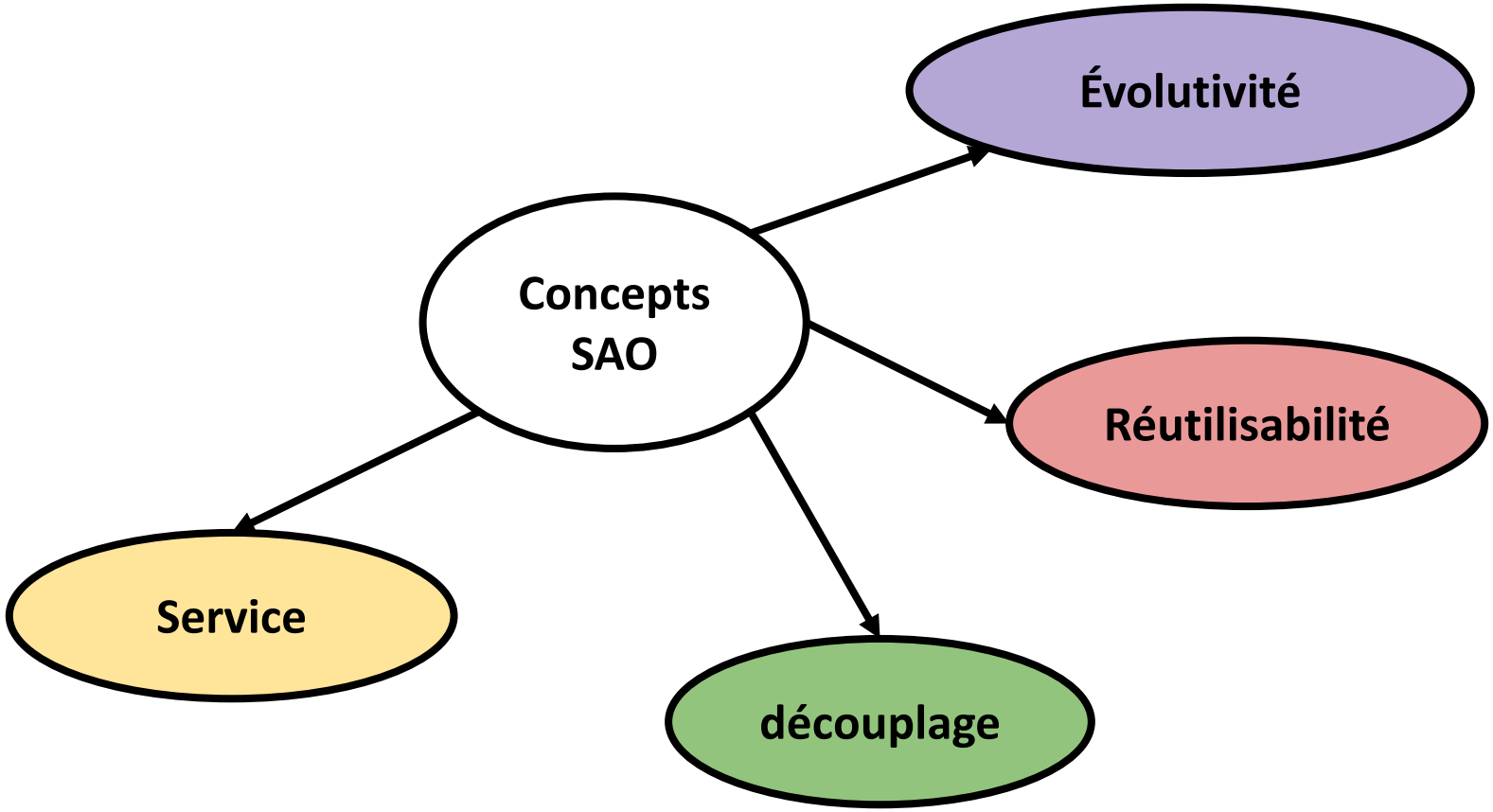
La solution est simple ! organiser le besoin métier (application) en une série de briques logicielles indépendantes appelées « services »

Quelque chose comme ça …



Chaque service a un nombre de fonctionnalités cohérentes qui sont indépendantes des autres services.

Les fonctionnalités des applications à développer sont exposées sous forme d'un ou plusieurs services.


Évolutivité
Concepts
SAO
Réutilisabilité
Service
découplage

Les services

Fondamental

- Un service est une unité autonome de fonctionnalités qui permet d'exécuter une tâche précise (une fonction ou un besoin métier).

Fondamental

- Caractéristiques d'un service :
  - Réutilisables ;
  - Forte cohérence interne ;
  - Faiblement couplés ;
  - Peut être utilisé indépendamment de la plate-forme et du langage de programmation.
  - Les services peuvent être liés entre eux pour exécuter un besoin métier.

Architecture orientée services

Imaginez que nous somme sollicité pour développer une application mobile qui permet de gérer les ventes dans une boutique en ligne.

Nous allons adopter une architecture orientée services

Les services

Fondamental

- Les services doivent assurer **une forte cohérence interne** (unité autonome de fonctionnalités) et **un couplage faible** avec les autres services (un service doit être indépendant des autres services)**.**

Service Commande

Gérer les commandes

```
ServiceCommande {
   addCustomer
   saveOrder
}
```

Les services

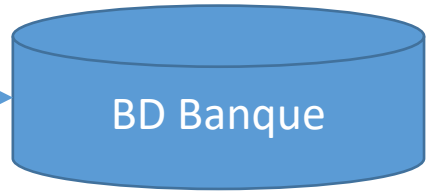
Fondamental

- Les services doivent assurer **une forte cohérence interne** (unité autonome de fonctionnalités) et **un couplage faible** avec les autres services (un service doit être indépendant des autres services)**.**

Service Paiement

Gérer les transactions bancaires

```
ServicePaiement {
   validateCreditCard
   validatePayment
}
```



Architecture orientée services

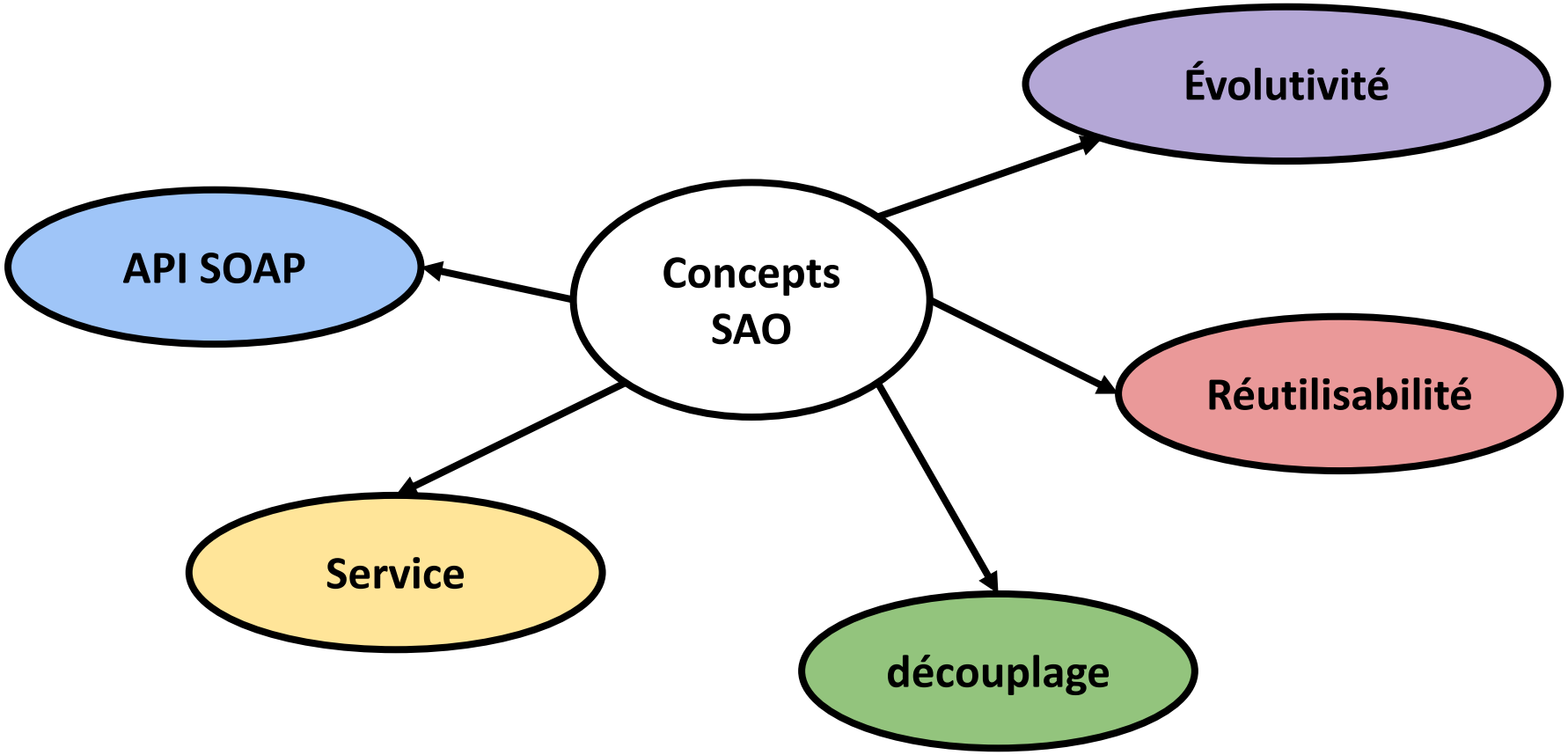
Une fois mis en production, le service « commande » va appeler le service « paiement » pour valider les commandes enregistrées (Il a besoin de confirmer la paiement de la commande pour commencer la procédure de livraison).

Les services

Fondamental

- Les services peuvent être liés entre eux pour exécuter un besoin métier.

Quelle approche adopter pour permettre la communication entre ces deux services ?


Évolutivité
API SOAP
Concepts
SAO
Réutilisabilité
Service
découplage

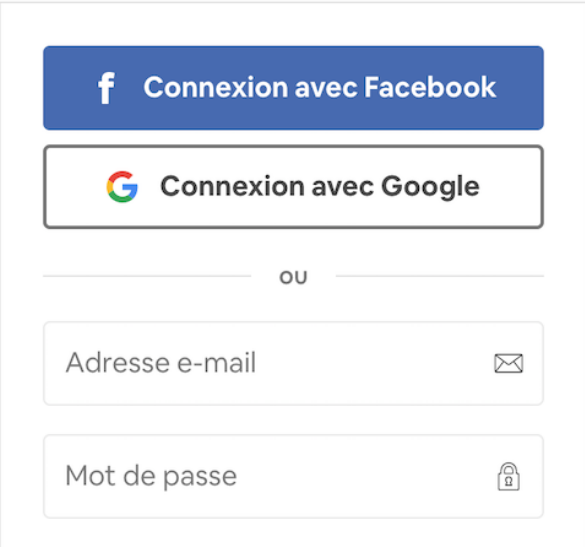
Les développeurs créent des API afin que d'autres systèmes puissent communiquer avec leurs applications

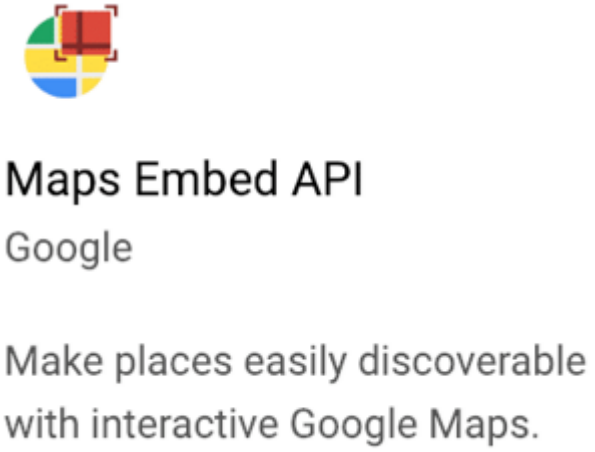
La magie des API !

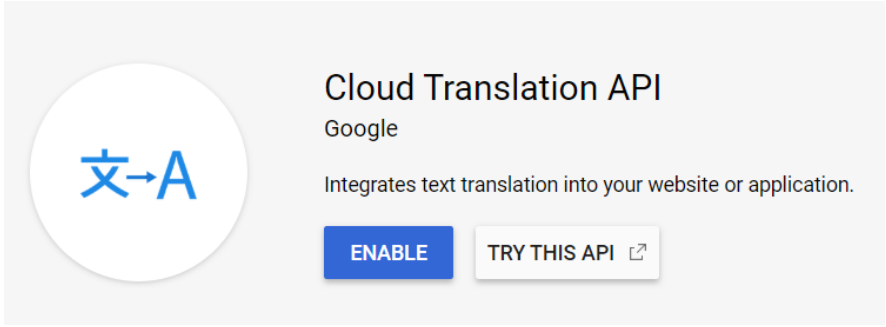
Architecture orientée services

Fondamental

- **API (Application Programming Interface/interface de programmation)** : représente un ensemble organisé de classes/fonctions/méthodes a qui servent **de façade à une application**. Elle offre aux autres applications un moyen structuré d'accéder aux services fournis par cette application (analogie encapsulation en POO).







Les API SOAP …

Problème 1 : Comment indiquer au service « commande » où trouver le service « paiement », les fonctionnalités que propose ce service et comment les appeler ?

- La réponse est simple : il faut établir un « **contrat** » qui permet d'expliquer clairement comment fonctionne chaque service (**un contrat selon une structure standard connue de tous les services**).

Les principes SOAP

Fondamental

- Chaque service est défini par **un document XML appelé WSDL** (Web Service Description Language). Dans ce contrat, seule la description des services (l'URL, les fonctions, les paramètres, type de réponse, …) est nécessaire : les détails liés à son implémentation ne sont pas exposés (**analogie : encapsulation en POO**).

```
<!-Structure d'un WSDL -->
<definitions name="MonService" targetNamespace="http://test.com/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
<!-- Définitions abstraites -->
<types> data type definitions..... </types>
<message> definition of the data being communicated.... </message>
<portType> set of operations..... </portType>
<!-- Définitions concrètes -->
<binding> protocol and data format specification.... </binding>
<service> service descripyion.... </service>
</definitions>
```

Les principes SOAP

Le service « commande » n'a plus qu'à consulter le contrat (document WSDL) du service « paiement » pour pouvoir faire appel à ce dernier !

Les principes SOAP

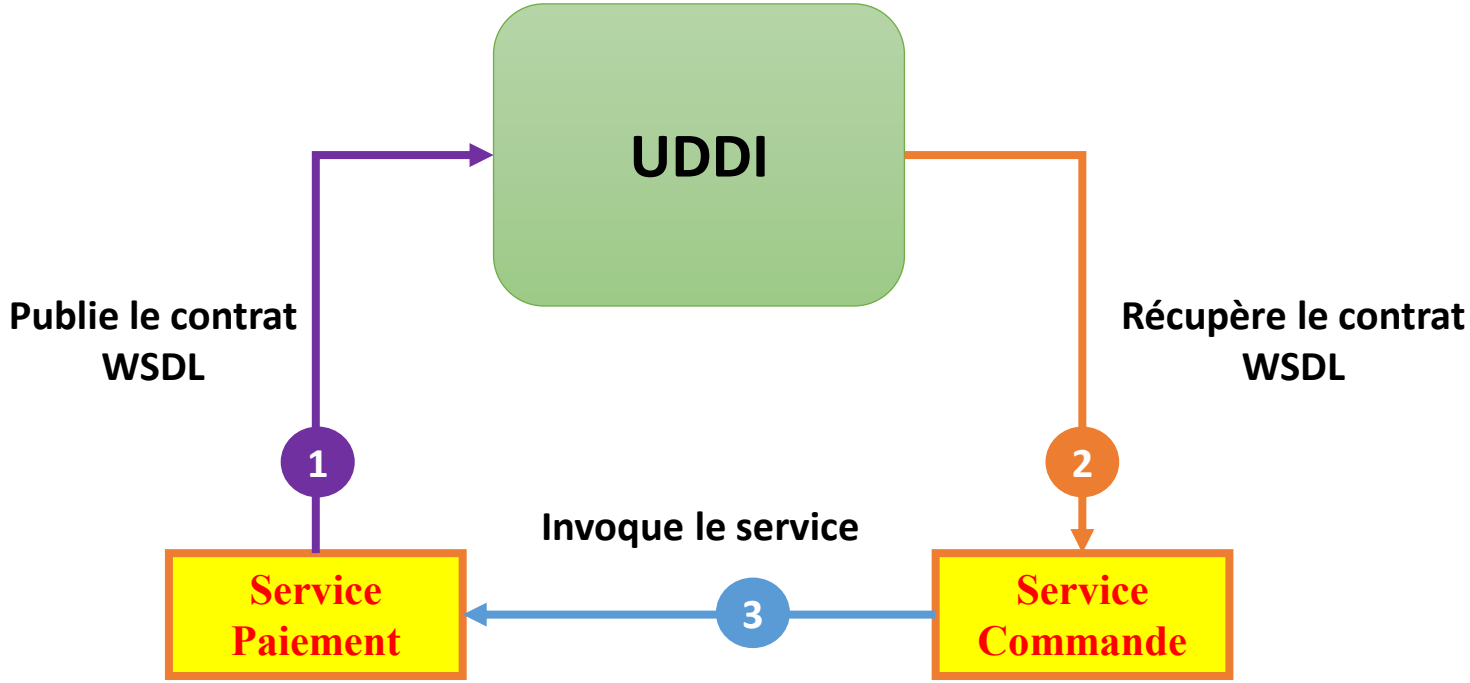
Problème 2 : Comment les services peuvent-ils trouver les contrats (document WSDL) des autres services ?

- La solution est de créer un « **annuaire de services** » qui permet de centraliser tous les contrats WSDL dans un serveur connu par tout les services.

Les principes SOAP

Fondamental

- Les contrats WSDL des services sont publiés dans **un annuaire appelé UDDI** (Universal Description Discovery and Integration). **Les différents services explorent ce registre** pour lire les contrats des autres services afin de communiquer ensemble.



Les principes SOAP

Le service « commande » est développé en JAVA alors que le service « paiement » est développé en PHP.

Les principes SOAP

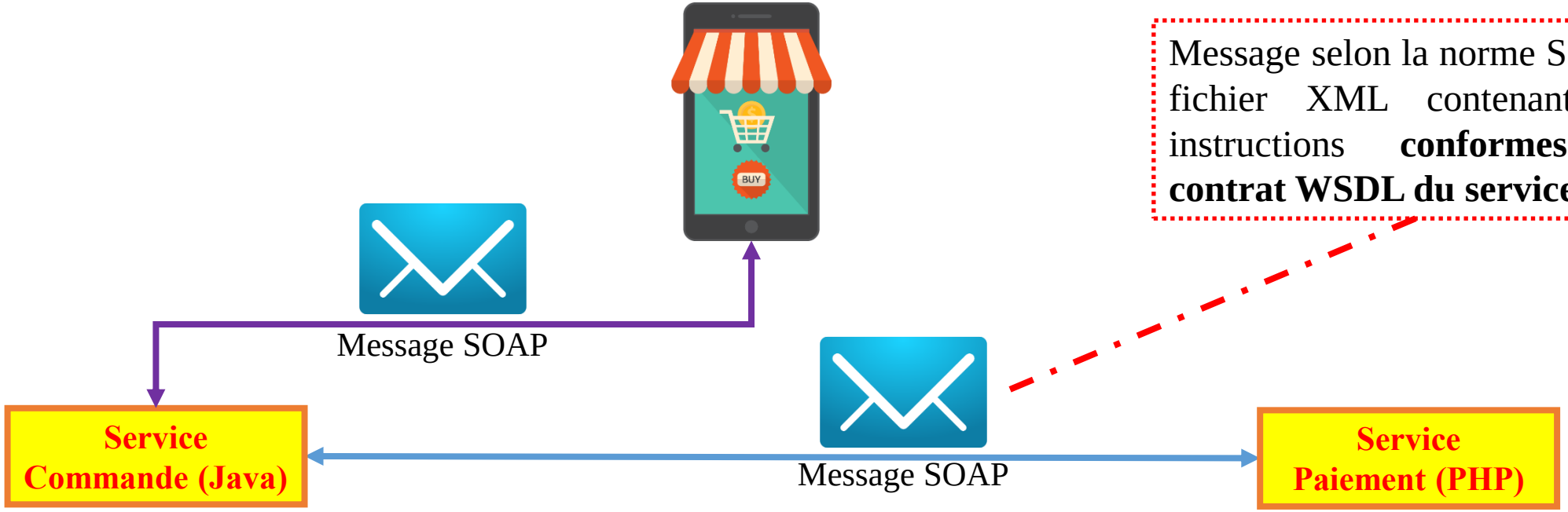
Problème 3 : Comment faire communiquer des services écrits dans des langages de programmation différents ?

- La solution est d'utiliser **un protocole de communication standard**, connu et compris de tous les services.

Les principes SOAP

Fondamental

- Les services, même développés dans différents langages, peuvent être composés pour exécuter un besoin métier. Pour ce faire, ils **communiquent entre eux grâce au protocole standard SOAP** (**Simple Object Access Protocol**) basé sur le XML.



Message selon la norme SOAP : fichier XML contenant des instructions **conformes au contrat WSDL du service**

Les principes SOAP

Fondamental

- Les services, même développés dans différents langages, peuvent être composés pour exécuter un besoin métier. Pour ce faire, ils **communiquent entre eux grâce au protocole standard SOAP** (Simple Object Access Protocol) basé sur le XML.

<!-Exemple d'un message SOAP pour appeler l'opération **additionner** -->

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:n2="http://test.com" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
    <n2:additionner>
        <n2:valeur1>13</n2:valeur1>
        <n2:valeur2>17</n2:valeur2>
    </n2:additionner>
</soapenv:Body>
</soapenv:Envelope>
```

Pour résumer …

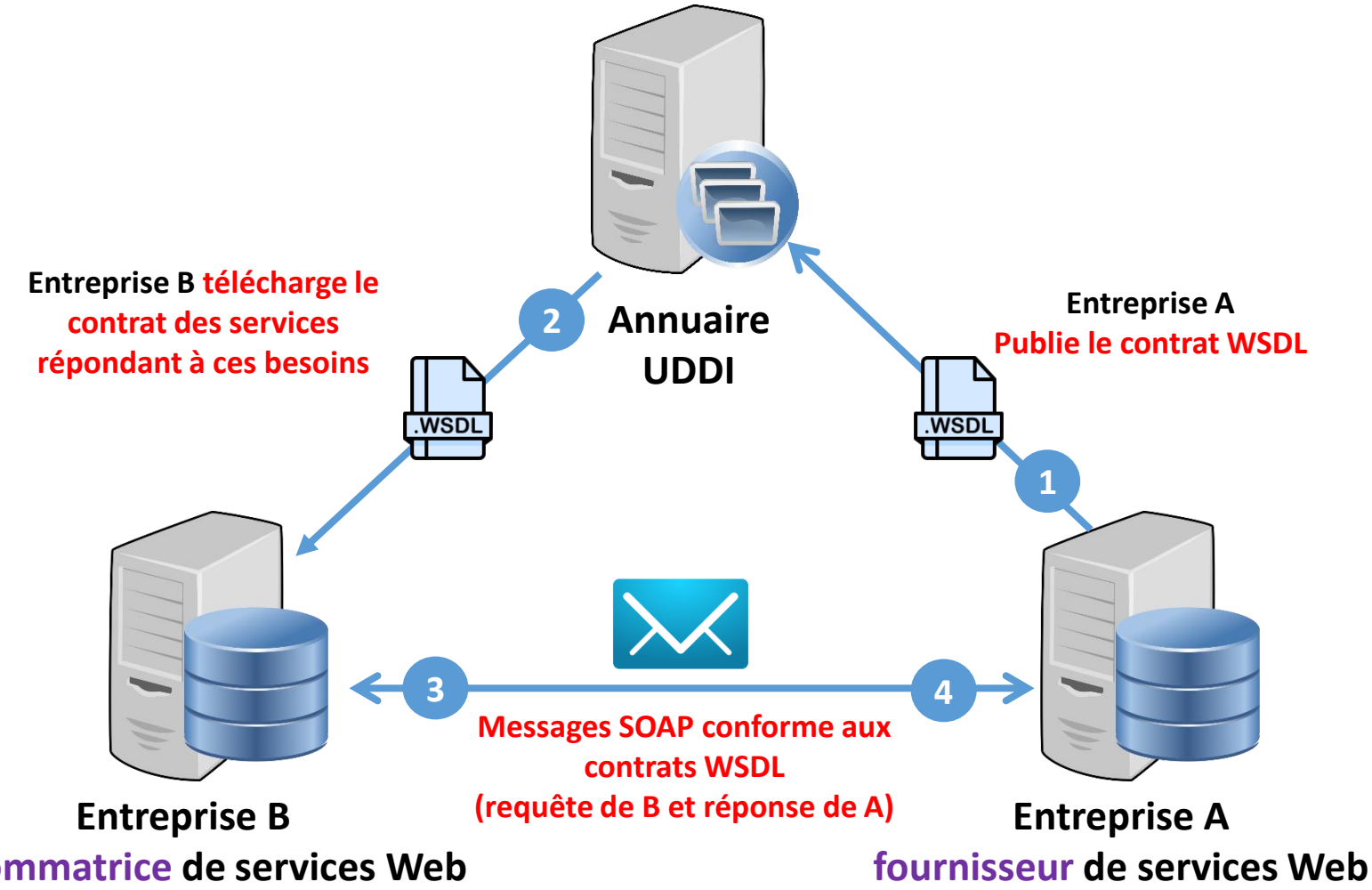
Les principes SOAP

Fondamental

- L'architecture orientée services (Service-Oriented Architecture- SOA) est une **architecture logicielle** qui rend des composants logiciels **autonomes**, **réutilisables**, **évolutifs** et **interopérables** grâce à des **interfaces de services** qui utilisent un langage commun pour communiquer via un réseau.

- **Implémentation de type SOAP** : services, annuaire UDDI, documents WSDL, messages SOAP.

Les principes SOAP



Les principes SOAP

Inconvénients de l'implémentation de type « SOAP » …

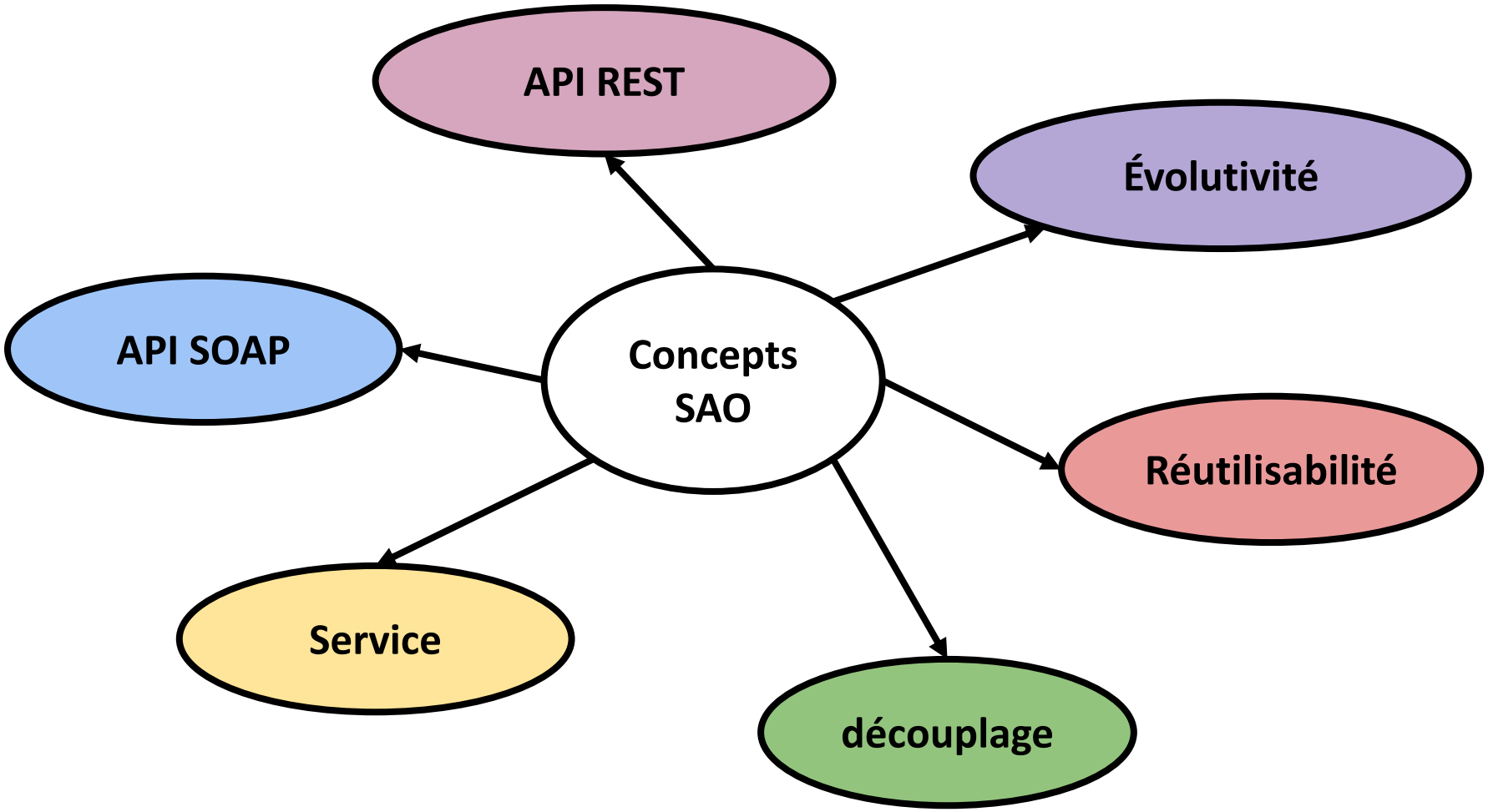
- **Complexité des concepts liés à l'implémentation de type « SOAP »** : il est difficile de configurer et de mettre en œuvre des services SOAP faciles à utiliser ;

- **Problème de performance** : le protocole SOAP alourdit considérablement les échanges à cause du nombre d'informations transmis dans les messages XML, notamment dans un contexte de Big Data. Ceci peut entraîner une surcharge de bande passante et une latence accrue.

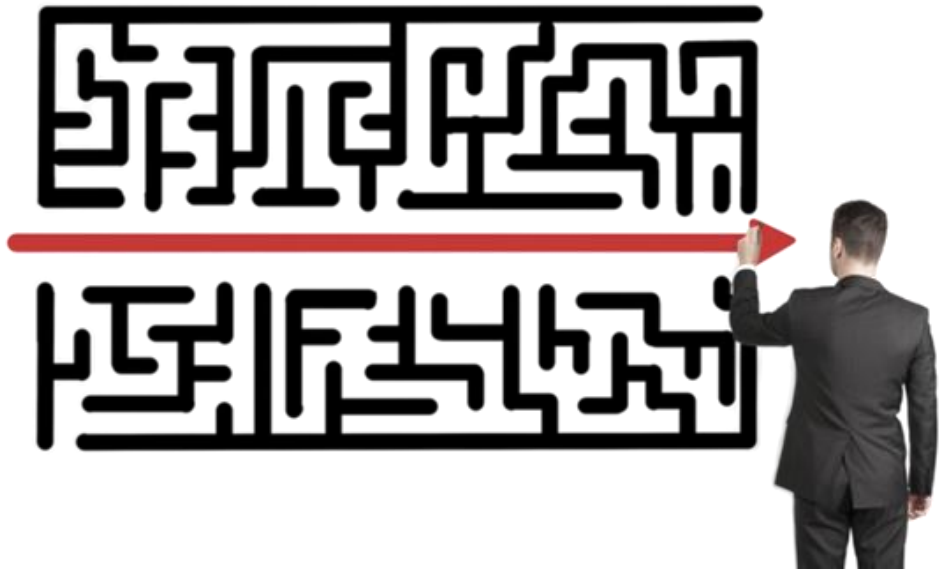
Quiz

Les affirmations suivantes sont-elles vraies ou fausses?

1. L’architecture orientée service (SOA) est un modèle de conception qui rend des composants logiciels réutilisables, grâce à des interfaces de services qui utilisent un langage commun pour communiquer via un réseau
2. Un service est une unité autonome de fonctionnalités conçue pour réaliser une tâche précise.
3. Les services sont encapsulés pour éviter les dépendances entre eux et pour faciliter leur réutilisation.
4. La réutilisation des services est un principe clé de SOA.
5. Un service est accessible via une interface standard.
6. SOAP est un langage de communication basé sur JSON
7. Les contrats WSDL sont des documents XML qui décrivent les services Web et les méthodes qu’ils exposent.
8. Aucune réponse n'est juste.


API REST
Évolutivité
API SOAP
Concepts
SAO
Réutilisabilité
Service
découplage

Pourquoi faire compliqué quand on peut faire simple ?



Architecture orientée services


```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:n2="http://test.com"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
      <n2:additionner>
            <n2:valeur1>13</n2:valeur1>
            <n2:valeur2>17</n2:valeur2>
      </n2:additionner>
</soapenv:Body>
</soapenv:Envelope>
```

Service A

Message SOAP

http://test.com/additionner?valeur1=13&valeur2=17

Client

Serveur

Les principes REST

Fondamental

- L'architecture orientée services (Service-Oriented Architecture- SOA) est une **architecture logicielle** qui rend des composants logiciels réutilisables, grâce à des interfaces de services qui utilisent un langage commun pour communiquer via un réseau.

Implémentation de type REST : la solution http pour les services Web

Simplicity wins : more than 92% of web services use REST APIs

Les principes REST

Fondamental

- **REST (REpresentational State Transfert)** : est un style d'architecture logicielle qui définit **un ensemble de contraintes** sur la façon dont une API doit fonctionner. Il tire son inspiration de la structure du Web et utilise le protocole **HTTP** comme principal moyen de communication. Les services Web qui adoptent l'architecture REST sont appelés **services Web RESTful.**

Attention

- Contrairement à SOAP et HTTP, **REST n'est pas un protocole, un standard, ou un format**, mais il s'agit d'une approche (**ensemble de bonnes pratiques à respecter**) pour construire des services web.

Les principes REST

Les principes clés pour les API REST …

Les principes REST

#1

Les principes REST

Fondamental

- **Le client et le serveur doivent être des entités distinctes** : le client interagit avec le serveur sans avoir une connaissance préalable de la structure ou du contenu des données stockées sur le serveur.

Les principes REST

#2

Les principes REST

Fondamental

- REST exige que **les ressources doivent être identifiées par des URIs (**Uniform Resource Identifiers**)**. Les ressources sont les informations que les services fournissent à leurs clients. Par exemple, les ressources peuvent être un texte, des images, des vidéos, ou tout autre type de données.

/univ-jijel → **Identifie**

L'URI est une chaîne de caractères qui identifie de manière unique une ressource particulière

Les principes REST

Les URIs sont des identificateurs uniques qui permettent d'identifier vos ressources

URI	Ressources
/orders/9876	Cette URI identifie de manière unique **la ressource commande** avec **l'identifiant** 9876.
/users/12345	Cette URI identifie **un produit spécifique, dont l'identifiant est 12345.**
/products	Cette URI identifie une **collection de produits**
/books/87/comments	Cette URI identifie les commentaires associés au livre ayant l'identifiant 87

Les principes REST

#3

Les principes REST

Fondamental

- **Les réponses des requêtes doivent représenter des ressources.** Par exemple, si une requête est envoyée pour récupérer la **ressource** « **/users/18** », la réponse doit être un utilisateur au format **JSON** (**JavaScript Object Notation**) ou **XML**.

```
{
"id": 18
"name": "Mohammed",
"age": 25,
"email": Mohammed@gmail.com
"adresse": {
    "rue": "123 Rue Principale",
    "ville": "Jijel",
    "pays": "Algérie"
  }
}
```

La ressource « /users/18 » au format JSON

Le format **JSON** (JavaScript Object Notation) est le plus utilisé pour les messages, car, il est léger et facile à comprendre

Les principes REST

#4

Les principes REST

Fondamental

- **L'architecture REST se base les spécifications originelles du protocole HTTP** au lieu de créer de nouvelles spécifications, contrairement à ce que fait SOAP.

- POST : Créer de nouvelles ressources (**Create**)
- GET : Récupérer des ressources (**Read**)
- PUT : Mettre à jour des ressources existantes (**Update**)
- DELETE : Supprimer des ressources (**Delete**)

Les principes REST

 POST :  Créer de nouvelles ressources (**Create**)

Les principes REST

Création d'une nouvelle ressource de type « user »

```
{
"name": "Mohammed",
"age": 45,
"email": "Mohammed@gmail.com"
"id": 1
 }
```

Service A


```
{
"name": "Mohammed",
"age": 45,
"email": "Mohammed@gmail.com"
 }
```

POST https://example.com/rest/users

Client

Serveur

Les principes REST

☑ GET : Récupérer des ressources (**Read**)

Les principes REST

Récupération de la représentation d'une ressource de type « user »

```
{
"name": “Mohammed”,
"age": 45,
"email": “Mohammed@gmail.com”
"id": 1
 }
```

Service A

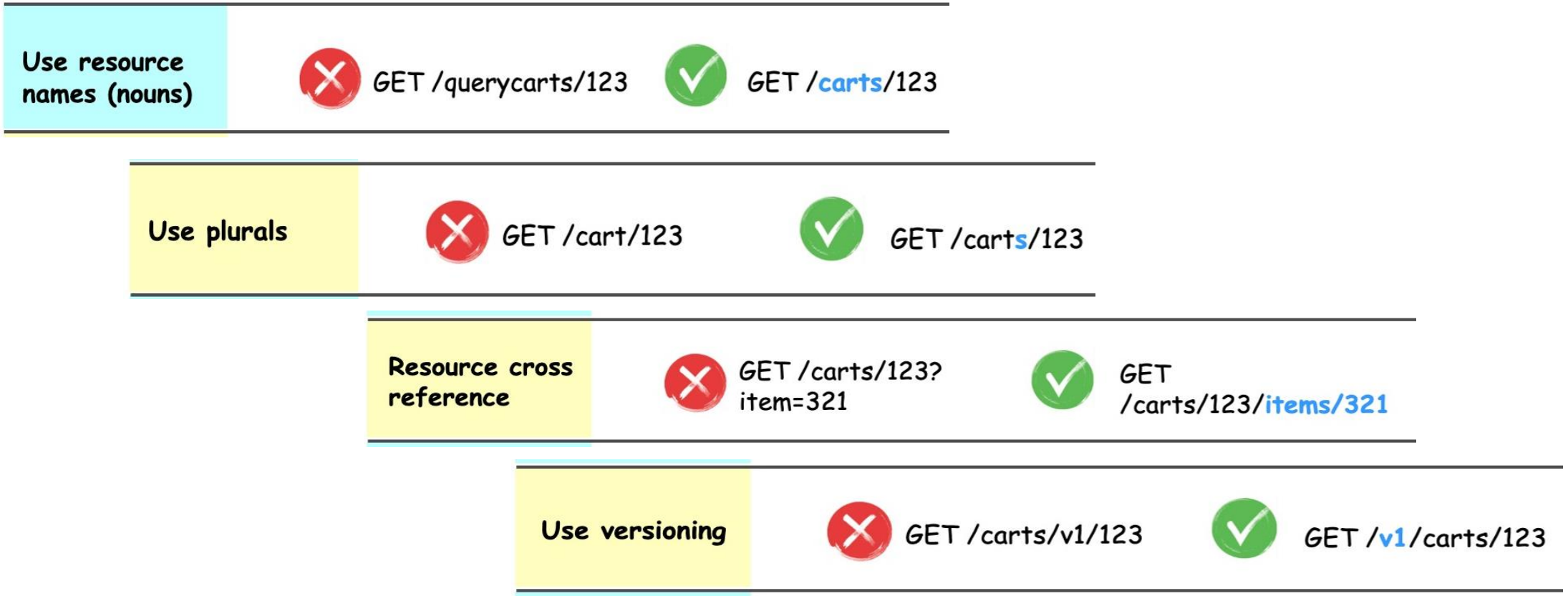
Client

GET https://example.com/rest/users/1

Serveur

Les principes REST

Conseils à suivre pour la conception de vos APIs



Les principes REST

PUT : Mettre à jour des ressources existantes (**Update**)

Les principes REST

Modification d'une ressource de type « user »

```
{
"name": "Mohammed",
"age": 49,
"email": "Mohammed2023@gmail.com"
"id": 1
 }
```

Service A


```
{
"age": 49,
"email": "Mohammed2023@gmail.com"
 }
```

PUT https://example.com/rest/users/1

Client

Serveur

Les principes REST

❌ DELETE : 🗑 Supprimer des ressources (**Delete**)

Architecture orientée services

Les principes REST

Suppression d'une ressource de type « user »

```
{
"name": "Mohammed",
"age": 49,
"email": "Mohammed2023@gmail.com"
 }
```

Service A

Client

`DELETE https://example.com/rest/users/1`

Serveur

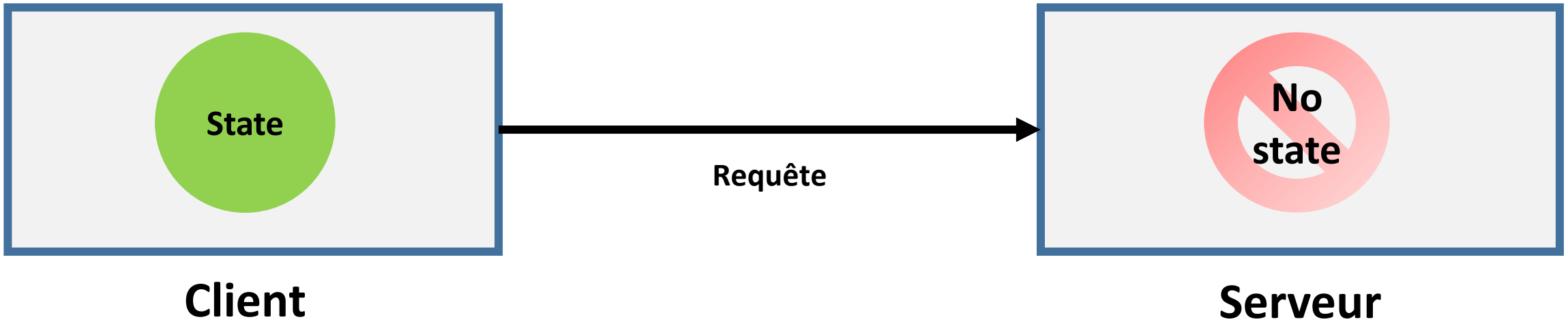
Les principes REST

#5

Les principes REST

Fondamental

- **Les API REST sont « Sans état » (Stateless)** : chaque requête du client doit contenir toutes les informations nécessaires à la compréhension et au traitement de cette requête. En d'autres termes, chaque requête est autonome, et le serveur n'a pas besoin de se souvenir des requêtes précédentes du client (**état/session**) pour traiter la requête en cours.



Les principes REST

Le fait d'être « sans état » rend vos requêtes très compréhensible …

Les principes REST

Imaginons que vous commandiez des frites à un serveur. Il se dirige vers la cuisine, récupère vos frites, et les apporte à votre table.

Cependant, au moment où il vous les sert, vous vous souvenez soudainement que vous vouliez aussi du ketchup. Vous demandez poliment au serveur : « Excusez-moi, pourrais-je avoir du ketchup, s'il vous plaît ? »

À votre surprise, le serveur répond : « **Avec quoi ?** »

Un serveur « sans état » ne serait pas au courant de votre commande précédente de frites, car il ne se souvient pas des commandes passées. Son rôle se limite à transmettre les demandes de la cuisine au client sans conserver d'information sur les commandes antérieures.

Requête correcte : « Est-ce que je pourrais avoir du ketchup sur les frites que j'ai commandées à la table 5 ? »

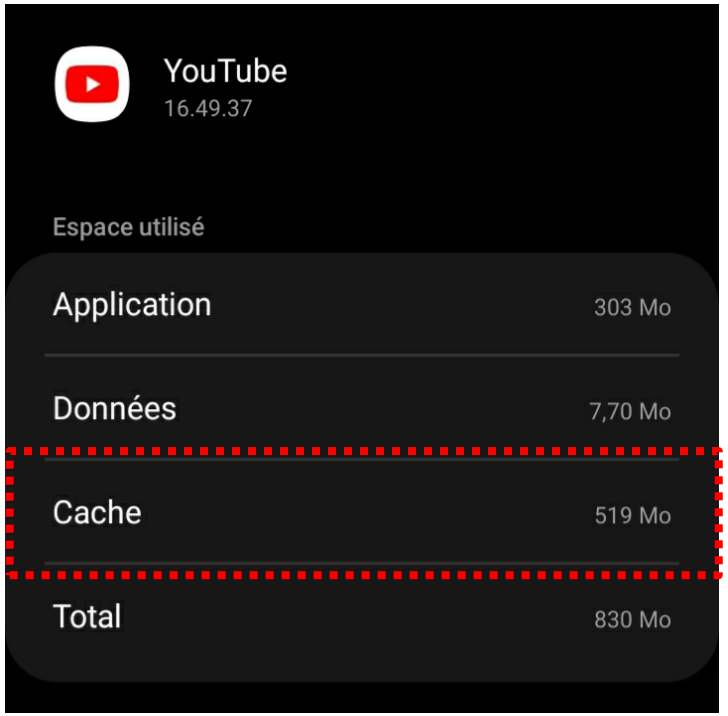
Les principes REST

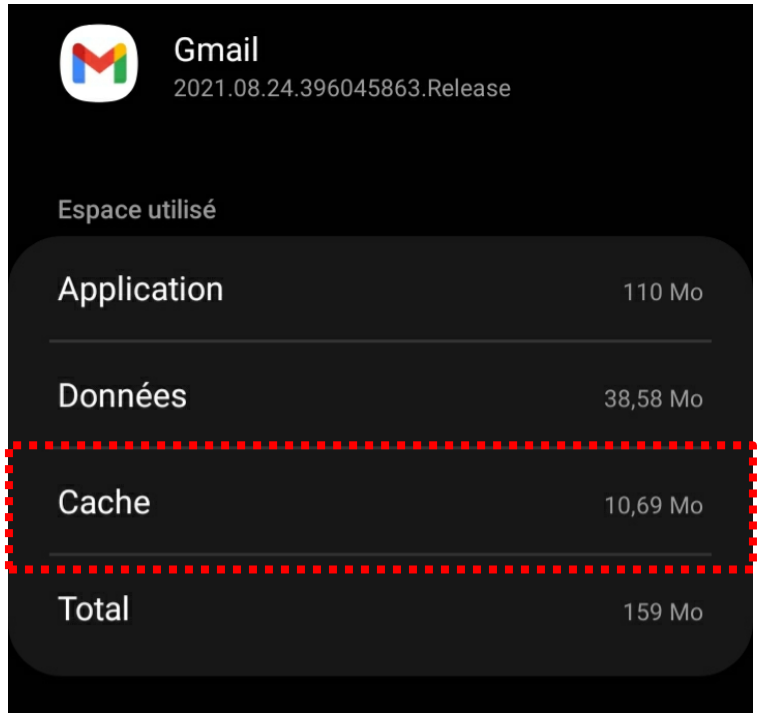
#6

Les principes REST

Fondamental

- Les services Web RESTful prennent en charge la mise en cache, qui consiste à stocker certaines réponses du serveur en mémoire côté client pour améliorer le temps de réponse du serveur.





Pour résumer …

Les principes REST

Fondamental

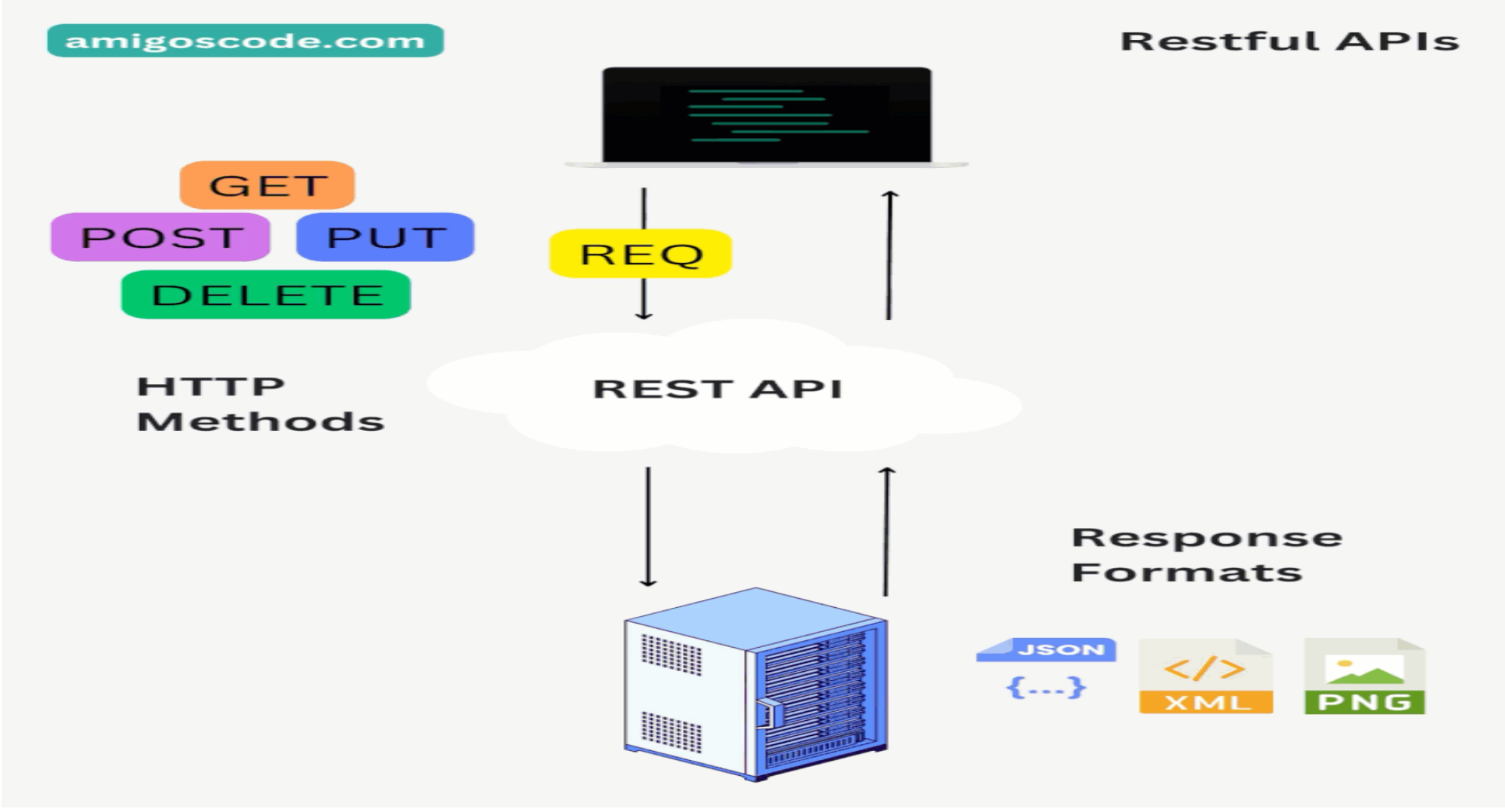
- L'architecture orientée services (Service-Oriented Architecture- SOA) est une **architecture logicielle** qui rend des composants logiciels réutilisables, grâce à des interfaces de services qui utilisent un langage commun pour communiquer via un réseau.

Implémentation de type REST : la solution http pour les services Web

Simplicity wins : more than 92% of web services use REST APIs

Architecture orientée services

Les principes REST



Les principes REST

Avantages

- Simplicité : L'architecture REST est simple et légère
- Évolutivité : les systèmes qui implémentent des API REST sont conçus pour être évolutif
- Portabilité/Interoperabilité : les API REST sont indépendantes de la technologie utilisée

Inconvénients

- Tout dépend du serveur (maillon faible)
- Cout élevé des machines serveurs

Architecture orientée services

Demo online

https://serpapi.com/

Travail demandé

- Apprendre comment développer un service Web selon les principes REST (appelée API ou services web RESTful).
- Développer une application web qui utilise deux services web REST développés dans deux langages de programmation différents (P. ex. Java et PHP) pour répondre à un besoin métier particulier (selon votre choix) ;
- Un de ces deux services fait appel à une API REEST disponible sur le Web (p. ex. API Google, API Microsoft, API Facebook, …) ;