Partie 3 : Modèles d'architecture

Partie 3 : lien vers la présentation

https://drive.google.com/file/d/11VS88SlWiXcJIMAjLclt9ILHIvfAo-5F/view?usp=sharing

Pour l'architecture et pourquoi un modèle d'architecture ?

- Les modèles d'architecture aident à définir les caractéristiques de base et le comportement d'une application.
- Par exemple, certains modèles d'architecture se prêtent naturellement à des applications hautement scalables, alors que d'autres modèles d'architecture se prêtent naturellement à des applications très agiles.
- Il est nécessaire de connaître les caractéristiques, les forces et les faiblesses de chaque modèle d'architecture afin de choisir celui qui répond aux besoins et aux objectifs spécifiques de votre entreprise.

En tant qu'architecte, vous devez toujours justifier vos décisions en matière d'architecture.

Avertissement sur l'évaluation et la comparaison des modèles

- Pour chaque modèle, une grille d'analyse est proposée comprenant 6 critères de notation.
  L'évaluation de ces critères est parfois subjective et peut dépendre des choix technologiques et du contexte du projet.

- Chaque modèle est présenté de façon indépendante mais dans la pratique, des concepts de plusieurs modèles peuvent être combinés entre eux.
  C'est ce que nous verrons dans la partie concernant les choix technologiques.

- Il n'y a pas de "bon" ou de "mauvais" modèle, chaque modèle correspond à un contexte précis et des exigences particulières.

2 catégories de modèles

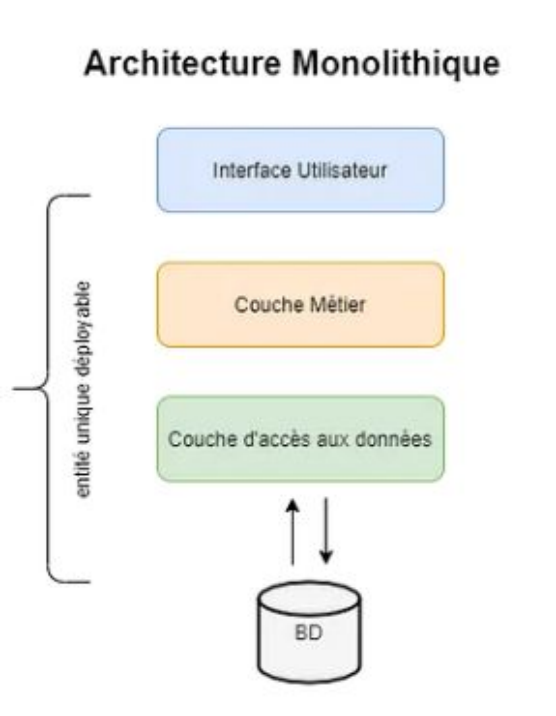
Les modèles d'architecture qui vont être décrits dans cette partie peuvent séparés en deux catégories :

- Modèles d'architecture d'application : comme son nom l'indique, décrit l'architecture d'une application
- Modèles d'architecture d'intégration : décrit l'architecture du SI ou d'une partie du SI constitué de plusieurs applications et/ou services

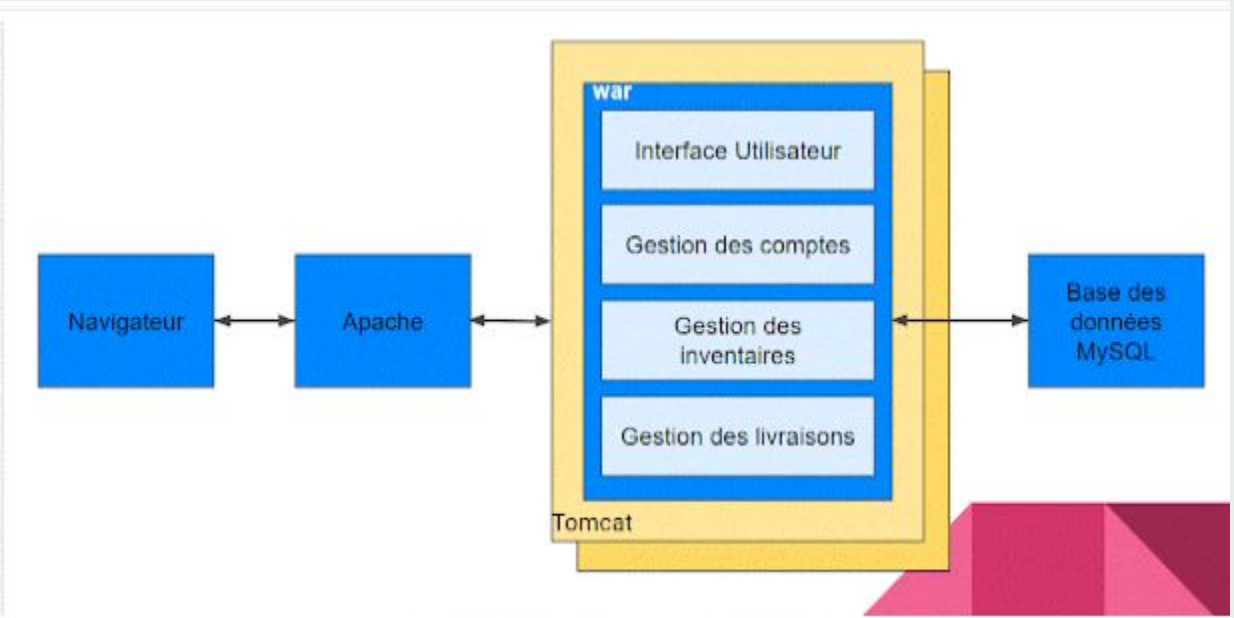
Modèle 1 : Architecture monolithique

- Dans ce contexte, « monolithique » signifie formé d'un seul bloc
- Un logiciel monolithique est conçu pour être autonome ; ses composants sont interconnectés et interdépendants plutôt qu'associés de manière flexible comme dans le cas des programmes modulaires.
- Dans ce type d'architecture étroitement intégrée, chaque composant et ceux qui lui sont associés doivent être présents pour permettre l'exécution ou la compilation du code.
- Une évolution technologique impacte toute l'application

Description du modèle (1/2)



Description du modèle (2/2)



Exemple d'une architecture monolithique

Grille d'analyse (1/3)

- **Agilité globale : faible**
  L'agilité globale est la capacité de répondre rapidement à un environnement en constante évolution. Il n'est pas possible de faire évoluer les composants de manière indépendante, mais uniquement l'application dans son ensemble.

- **Facilité de déploiement : fort**
  Les applications monolithiques sont faciles à déployées car le déploiement consiste généralement à gérer seulement un fichier ou un répertoire.

Grille d'analyse (2/3)

- **Testabilité : faible**
  Les différents composants de l'application peuvent être difficiles à isoler et à tester séparément.

- **Performance : fort**
  A priori, ce modèle offre de bonnes performances notamment car il n'y a pas de latence réseau puisque tous les services sont situés dans le même flux de travail.

Grille d'analyse (3/3)

- **Scalabilité : faible**
  La seule option de scalabilité d'une application monolithique est d'augmenter la puissance du serveur qui l'héberge (scalabilité verticale). Mais cette option est coûteuse et vite limitée.

- **Facilité de développement : fort**
  La facilité de développement obtient un score relativement élevé, principalement parce que ce modèle est bien connu et n'est pas excessivement complexe à mettre en œuvre.

Modèle 1 : Architecture monolithique : conclusion

Agilité globale : faible	Facilité de déploiement : fort
Testabilité : faible	Performance : fort
Scalabilité : faible	Facilité de développement : fort

- La grille d'analyse peut sembler finalement être assez favorable pour le choix de ce modèle d'architecture
- Néanmoins, les points faibles de ce modèle sont très limitants et bloquants

Modèle 2 : Architecture en couches

- Ce modèle peut être aussi connu sous le nom de modèle d'architecture n-tier mais il y a une différence de signification :
  - Une couche (layer) est une unité de composants logiques
  - Un niveau (tier) est l'unité physique (machine, serveur d'application) de déploiement des couches.

- Ce modèle est la norme de facto pour la plupart des applications Java EE et est donc largement connu de la plupart des architectes, concepteurs et développeurs.

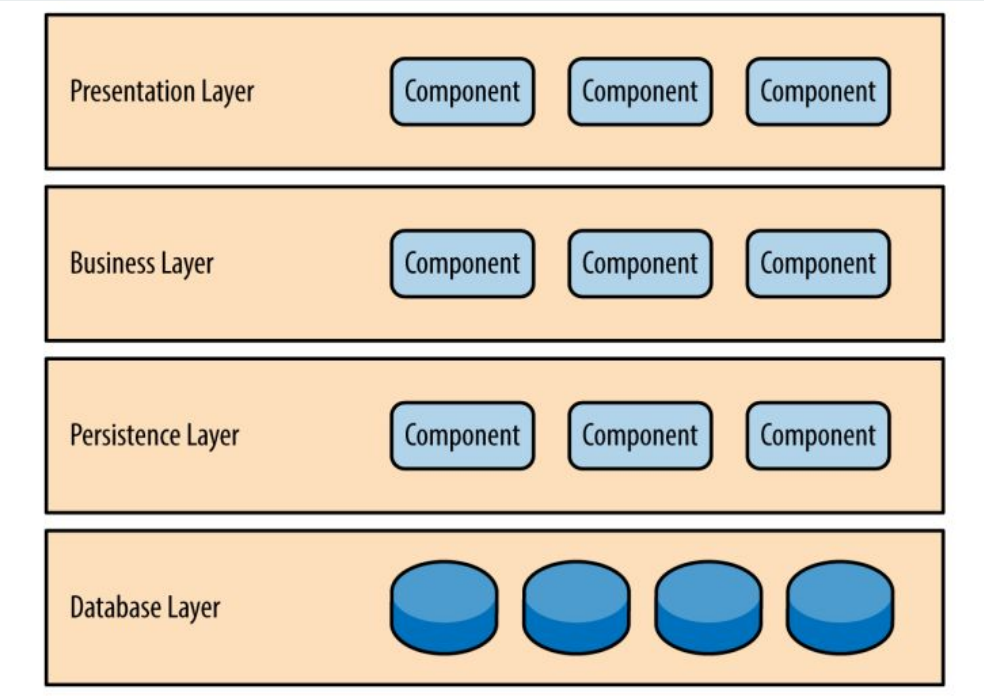
- Le modèle d'architecture en couches correspond étroitement aux structures traditionnelles de communication et d'organisation de l'informatique qu'ont la plupart des entreprises, ce qui en fait un choix naturel pour la plupart des efforts de développement d'applications d'entreprise.

- Date : fin des années 1990

Description du modèle (1/4)

- Les composants du modèle d'architecture en couches sont organisés en couches horizontales, chaque couche jouant un rôle spécifique dans l'application (par exemple, la logique de présentation ou la logique métier).
- Bien que le modèle d'architecture en couches ne spécifie pas le nombre et les types de couches qui doivent exister dans le modèle, la plupart des architectures en couches sont constituées de quatre couches standard : présentation, métier, persistance et base de données.

Description du modèle (2/4)



Description du modèle (3/4)

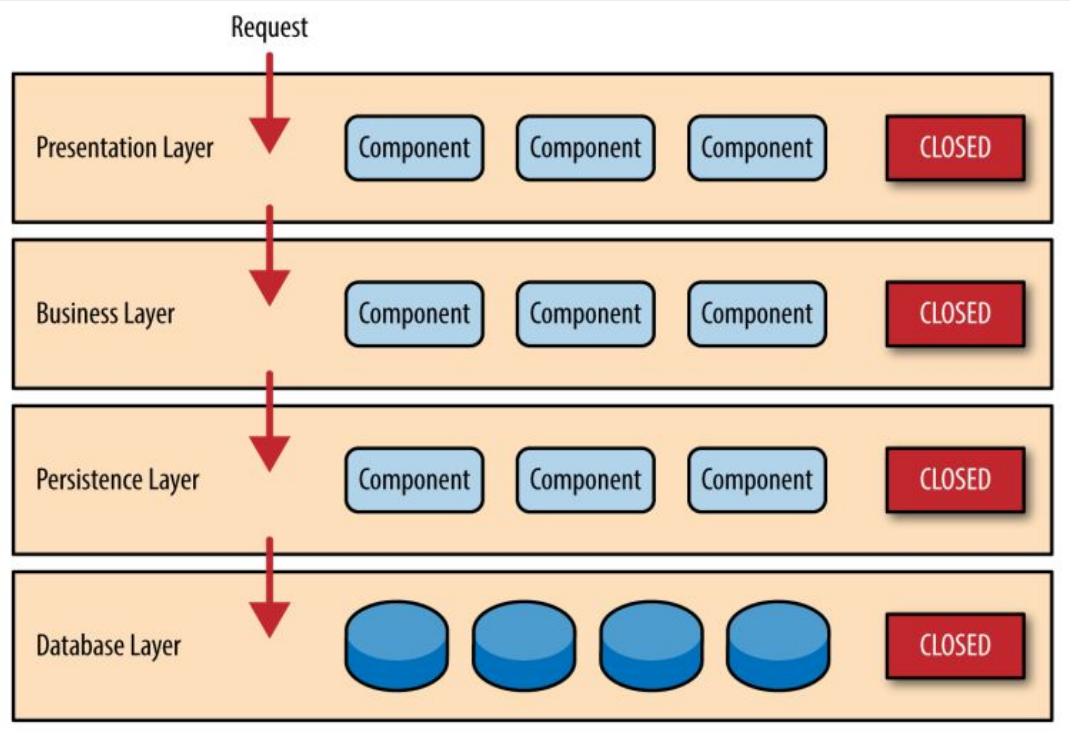
Dans certains cas, la couche métier et la couche de persistance sont combinées en une seule couche métier, particulièrement lorsque la logique de persistance est intégrée dans les composants de la couche métier.

Ainsi, les petites applications peuvent n'avoir que trois couches, alors que les applications plus grandes et plus complexes peuvent contenir cinq couches ou plus.

Description du modèle (4/4)

- Chaque couche du modèle d'architecture en couches a un rôle et une responsabilité spécifiques dans l'application.
- Par exemple, une couche de présentation de présentation est responsable de la gestion de l'interface utilisateur et de la logique de communication avec le navigateur, tandis qu'une couche métier serait l'exécution de règles métier spécifiques associées à une requête.
- Chaque couche de l'architecture forme une abstraction autour du travail à effectuer pour satisfaire une requête
- Par exemple, la couche de présentation n'a pas besoin de savoir ou de se préoccuper de la manière d'obtenir des données sur les clients.
- Par exemple, la couche de présentation n'a pas besoin de savoir ou de se préoccuper de la façon d'obtenir des données sur les clients, elle doit juste afficher ces informations sur un écran dans un format particulier.
- De même, la couche métier n'a pas besoin de se préoccuper de la façon de formater les données du client pour les afficher sur un écran, ni même de leur provenance. Il lui suffit d'obtenir les données de la couche de persistance, d'exécuter la logique métier sur les données (par exemple, calculer des valeurs ou calculer des valeurs ou agréger des données) et transmettre ces informations à la couche de présentation.

Concept clé : notion de fermeture



Une couche fermée signifie que lorsqu'une requête se déplace d'une couche à l'autre, elle doit passer par la couche juste en dessous pour atteindre la couche suivante.
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 finalement la couche de base de données.

Concept clé : notion de fermeture
Illustration d'une situation où le concept n'a pas (du tout !) été respecté

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ page import="java.io.*,java.util.*,java.sql.*"%>
<%@ page import="javax.servlet.http.*,javax.servlet.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>

<html>
<head>
<title>SELECT 操作</title>
</head>
<body>
<!--
JDBC 驱动名及数据库 URL
数据库的用户名与密码，需要根据自己的设置
useUnicode=true&characterEncoding=utf-8 防止中文乱码
 -->
<sql:setDataSource var="snapshot" driver="com.mysql.jdbc.Driver"
     url="jdbc:mysql://localhost:3306/w3big?useUnicode=true&charact
     user="root"  password="123456"/>

<sql:query dataSource="${snapshot}" var="result">
SELECT * from websites;
</sql:query>
```

```
<h1>JSP 数据库实例 - 本教程</h1>
<table border="1" width="100%">
<tr>
   <th>ID</th>
   <th>站点名</th>
   <th>站点地址</th>
</tr>
<c:forEach var="row" items="${result.rows}">
<tr>
   <td><c:out value="${row.id}"/></td>
   <td><c:out value="${row.name}"/></td>
   <td><c:out value="${row.url}"/></td>
</tr>
</c:forEach>
</table>

</body>
</html>
```

Dangereux ! A ne pas reproduire chez vous !

Concept clé : notion de fermeture

- Alors pourquoi ne pas permettre à la couche de présentation d'accéder directement à la couche de persistance ou à la couche de base de données ?
- Après tout, l'accès direct à la base de données à partir de la couche de présentation est beaucoup plus rapide que de passer par un ensemble de couches inutiles juste pour récupérer ou enregistrer les informations de la base de données.
- La réponse à cette question réside dans un concept clé connu sous le nom de couches d'isolation.

Concept clé : couches d'isolation

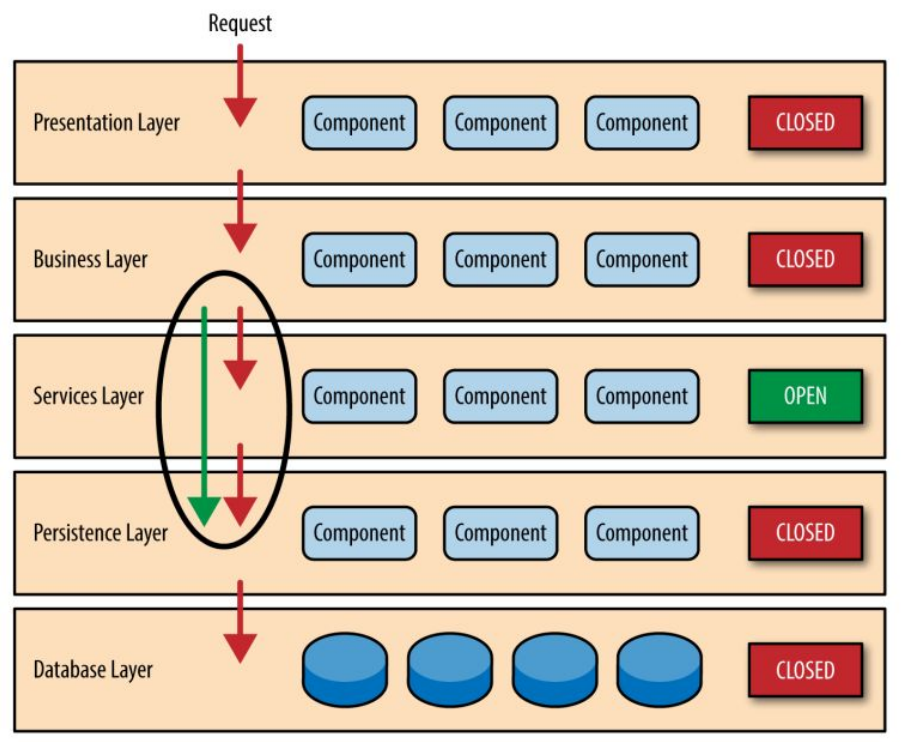
- Le concept de couches d'isolation signifie que les changements apportés dans une couche de l'architecture n'ont généralement pas d'impact ou d'incidence sur les composants sur les composants des autres couches

- La modification est isolée dans les composants de cette couche, et éventuellement à une autre couche associée (telle qu'une couche de persistance contenant le SQL).

Notion de couche ouverte (1/2)

- Alors que les couches fermées facilitent les couches d'isolation et aident donc à d'isoler les changements au sein de l'architecture, il est parfois utile que certaines couches soient ouvertes.
- Exemple : on souhaite ajouter une couche de services partagés à une architecture contenant des composants de services communs auxquels accèdent les composants de la couche métier (par exemple, des classes utilitaires de données et de chaînes ou des classes d'audit et de journalisation).
- La création d'une couche de services est généralement une bonne idée dans ce cas car, d'un point de vue architectural, elle limite l'accès aux services partagés à la couche métier (et non à la couche de présentation).
- Dans ce cas, la nouvelle couche de services sera située sous la couche métier pour indiquer que les composants de cette couche services ne sont pas accessibles par la couche présentation.

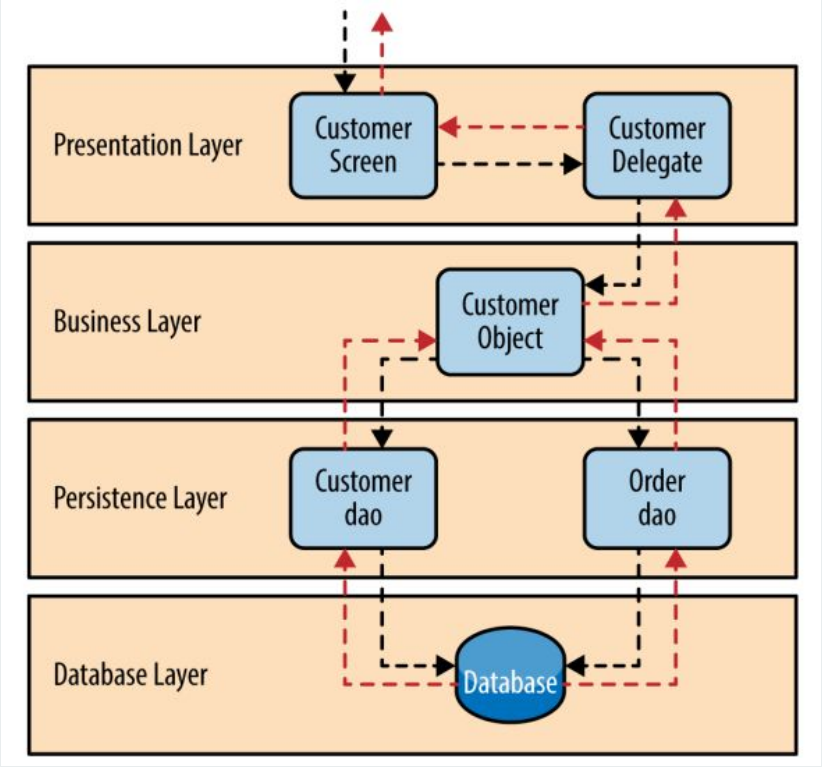
Notion de couche ouverte (2/2)



La couche services est marquée ouverte, ce qui signifie que les requêtes peuvent contourner cette couche ouverte et aller directement à la couche inférieure.

Puisque la couche services est ouverte, la couche métier est maintenant autorisée à la contourner et aller directement à la couche de persistance, ce qui est parfaitement logique.

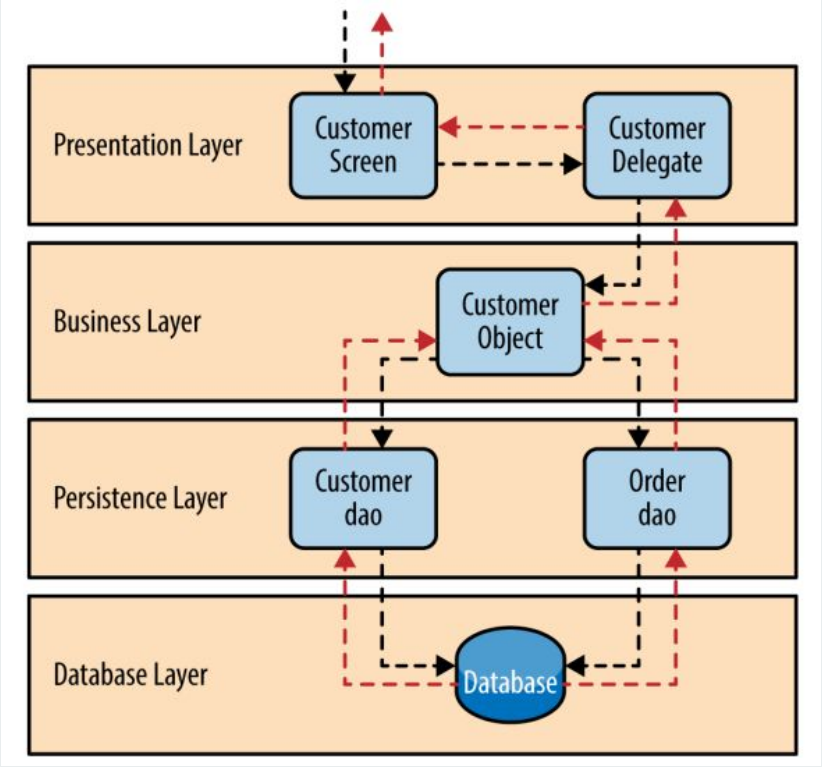
Exemple



Exemple : application CRM : fiche client avec ses commandes

Les flèches noires montrent la requête descendante vers la base de données pour récupérer les données du client et les flèches rouges montrent la réponse qui remonte vers l'écran pour afficher les données.

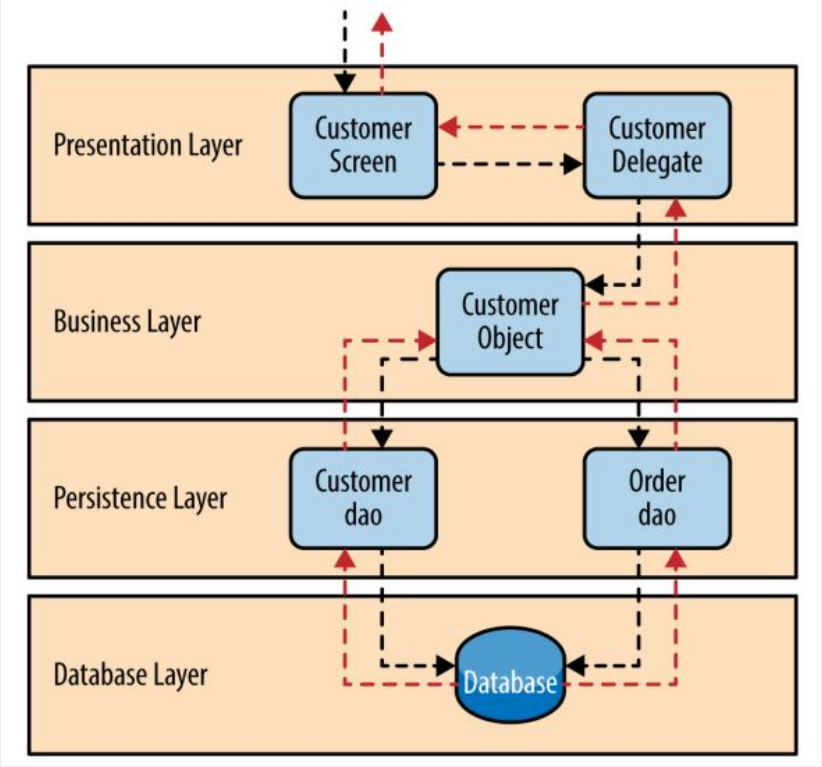
Exemple



L'écran client est responsable de la demande et de l'affichage des informations du client.
Il ne sait pas où se trouvent les données, comment elles sont récupérées ou combien de tables de la base de données doivent être interrogées pour obtenir les données.

Lorsque l'écran client reçoit une demande d'informations pour un individu particulier, il transmet cette demande au module Customer Delegate.
Ce module est responsable de la connaissance des modules de la couche métier qui peuvent traiter cette demande, mais aussi comment accéder à ce module et quelles sont les données dont il a besoin (le contrat).

Exemple



L'objet client (Customer Object) de la couche métier est responsable de l'agrégation de toutes les informations nécessaires pour la demande de l'entreprise (dans ce cas, pour obtenir des informations sur le client). Le module fait appel au module dao (objet d'accès aux données) client dans la
la couche de persistance pour obtenir des données sur les clients, ainsi qu'au module order dao pour obtenir des informations sur les commandes.
Ces modules exécutent à leur tour le SQL pour récupérer les données correspondantes et les renvoyer à l'objet client dans la couche à l'objet client dans la couche métier.
Une fois que l'objet client reçoit les données, il les agrège et transmet ces informations au module Customer Delegate qui les transmet ensuite à l'écran du client pour qu'elles soient présentées à l'utilisateur.

Points d'attention

- Le modèle d'architecture convient à de nombreux cas d'utilisation
- Il peut y avoir des situations où certaines couches semblent inutiles.
  Exemple : affichage d'informations qui ne nécessitent pas d'agrégation
- La tentation est grande de "court-circuiter" certaines couches mais cela représente un danger pour faire évoluer l'application.
- Situation connue sous le nom de "sinkhole anti-pattern" (sinkhole = glissement de terrain)

Grille d'analyse (1/3)

- **Agilité globale : faible**
  L'agilité globale est la capacité de répondre rapidement à un environnement en constante évolution. Bien que le changement puisse être isolé par le biais des couches d'isolation de ce modèle, il est toujours fastidieux et long d'effectuer des changements avec ce modèle d'architecture en raison de la nature monolithique de la plupart des implémentations, ainsi que du couplage étroit des composants.

- **Facilité de déploiement : faible**
  Selon la façon dont est implémenté ce modèle, le déploiement peut devenir un problème, en particulier pour les grosses applications. Une petite modification d'un composant peut nécessiter un redéploiement de l'application entière (ou d'une grande partie de l'application), ce qui entraîne des déploiements qui doivent être planifiés et exécutés en dehors des heures de travail ou le week-end.
  En tant que tel, ce modèle ne se prête pas facilement à un pipeline de livraison continue, ce qui pénalise encore l'évaluation du critère déploiement.

Grille d'analyse (2/3)

- **Testabilité : fort**
  Comme les composants appartiennent à des couches spécifiques de l'architecture, les autres couches peuvent être simulées (mock). Un développeur peut simuler un composant de présentation ou un écran pour isoler les tests dans un composant métier. La couche métier peut aussi être simulée pour tester certaines fonctionnalités de l'écran.

- **Performance : faible**
  S'il est vrai que certaines architectures en couches peuvent avoir de bonnes performances, ce modèle ne se prête pas aux applications à haute performance en raison de l'inefficacité de devoir passer par plusieurs couches de l'architecture pour répondre à une requête.

Grille d'analyse (3/3)

- **Scalabilité : faible**
  En raison de la tendance vers des implémentations monolithiques et étroitement couplées de ce modèle, les applications construites à l'aide de ce modèle d'architecture sont généralement difficiles à faire monter en charge.
  On peut faire évoluer une architecture en couches en répartissant les couches dans des déploiements physiques distincts ou en répliquant l'ensemble de l'application sur plusieurs nœuds, mais dans l'ensemble, la granularité est trop large, ce qui rend la mise à l'échelle coûteuse.

- **Facilité de développement : fort**
  La facilité de développement obtient un score relativement élevé, principalement parce que ce modèle est bien connu et n'est pas excessivement complexe à mettre en œuvre.
  Comme la plupart des entreprises développent des applications en séparant les compétences par couches (présentation, métier, base de données), ce modèle d'architecture devient un choix naturel pour la plupart des développements d'applications d'entreprise.

Modèle 2 : Architecture en couches : conclusion

Agilité globale : faible	Facilité de déploiement : faible
Testabilité : fort	Performance : faible
Scalabilité : faible	Facilité de développement : fort

- la grille d'analyse peut sembler finalement être peu favorable pour le choix de ce modèle d'architecture
- Néanmoins, certains points faibles de ce modèle peuvent être améliorés par des choix de technologies

Modèle 3 : Architecture orientée événements

- Le modèle d'architecture orientée événements (ou EDA : event driven architecture pattern) est un modèle d'architecture asynchrone et distribuée, utilisé pour produire des applications hautement scalables.
- Le modèle d'architecture orientée événements se présente sous deux variantes : le médiateur (mediator) et le courtier (broker).

- La variante “médiateur” est utilisée lorsque vous devez orchestrer plusieurs étapes au sein d'un événement par l'intermédiaire d'un médiateur central.

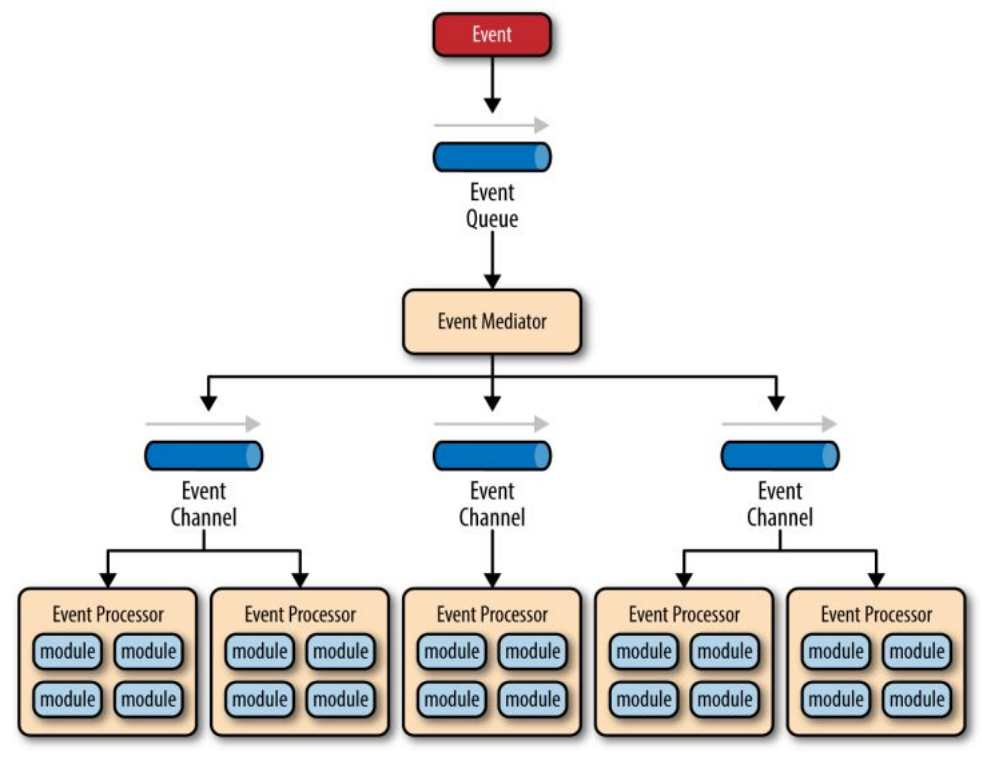
- La variante “courtier” est utilisée lorsque vous souhaitez enchaîner des événements sans avoir recours à un médiateur central.

- Date : années 2000 - 2010

Variante “Médiateur” (Mediator)

- La variante “médiateur” est utile pour les événements qui comportent plusieurs étapes et qui nécessitent un certain niveau d'orchestration pour traiter l'événement.
- Exemple : cas de la transaction boursière qui comporte plusieurs étapes :
  - valider la transaction,
  - vérifier la conformité de cette transaction boursière par rapport à diverses règles de conformité,
  - attribuer la transaction à un courtier,
  - calculer la commission
  - effectuer la transaction auprès de ce courtier.
- Toutes ces étapes nécessitent un certain niveau d'orchestration pour déterminer l'ordre des étapes et celles qui peuvent être effectuées en série et en parallèle.

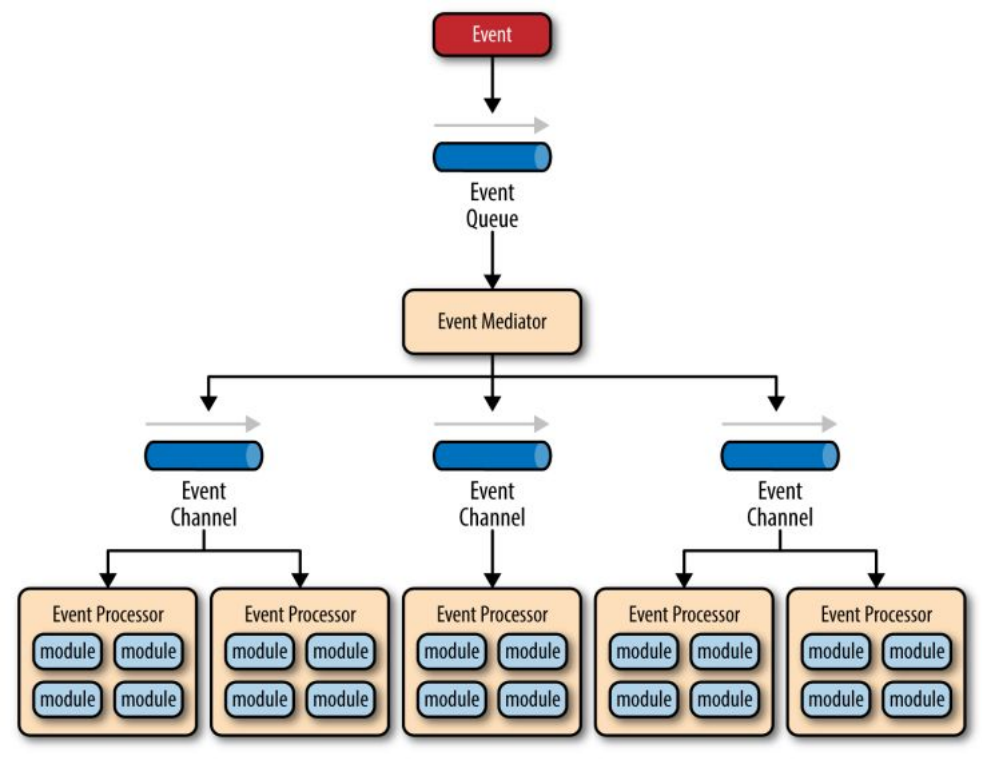
Variante “Médiateur” (Mediator)



Il existe quatre principaux types de composants d'architecture dans la variante du médiateur :

- les files d'attente d'événements,
- le médiateur d'événements,
- les canaux d'événements
- les processeurs d'événements.

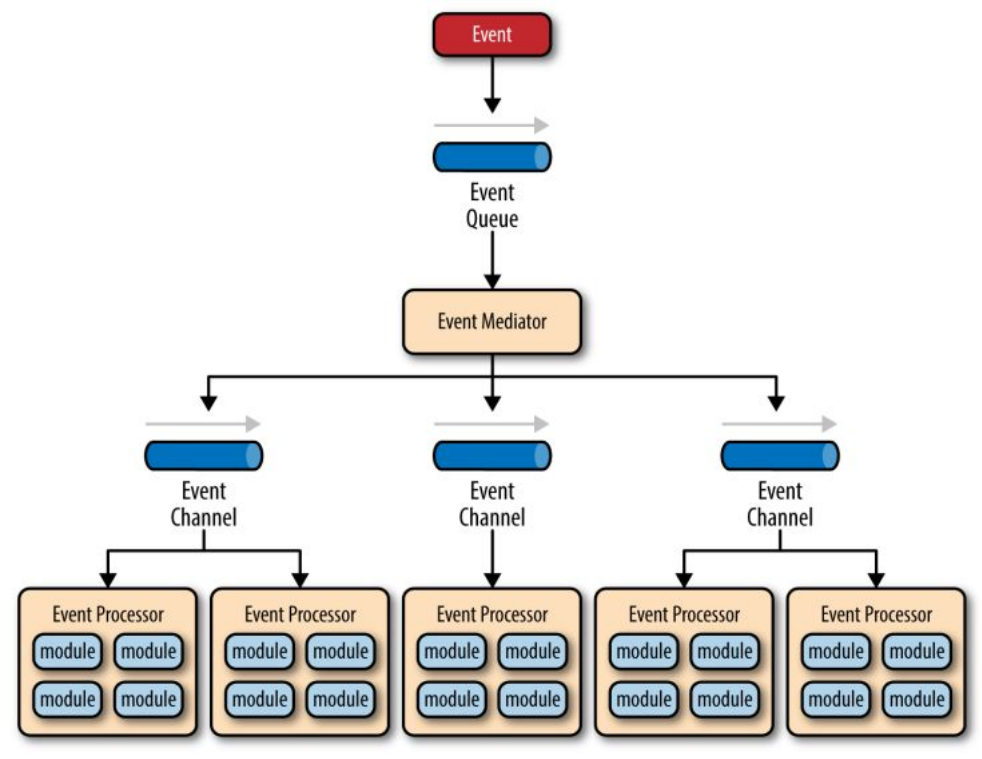
Variante “Médiateur” (Mediator)



Le flux d'événements commence avec un client qui envoie un événement à une file d'attente d'événements, qui est utilisée pour transporter l'événement vers le médiateur d'événements.

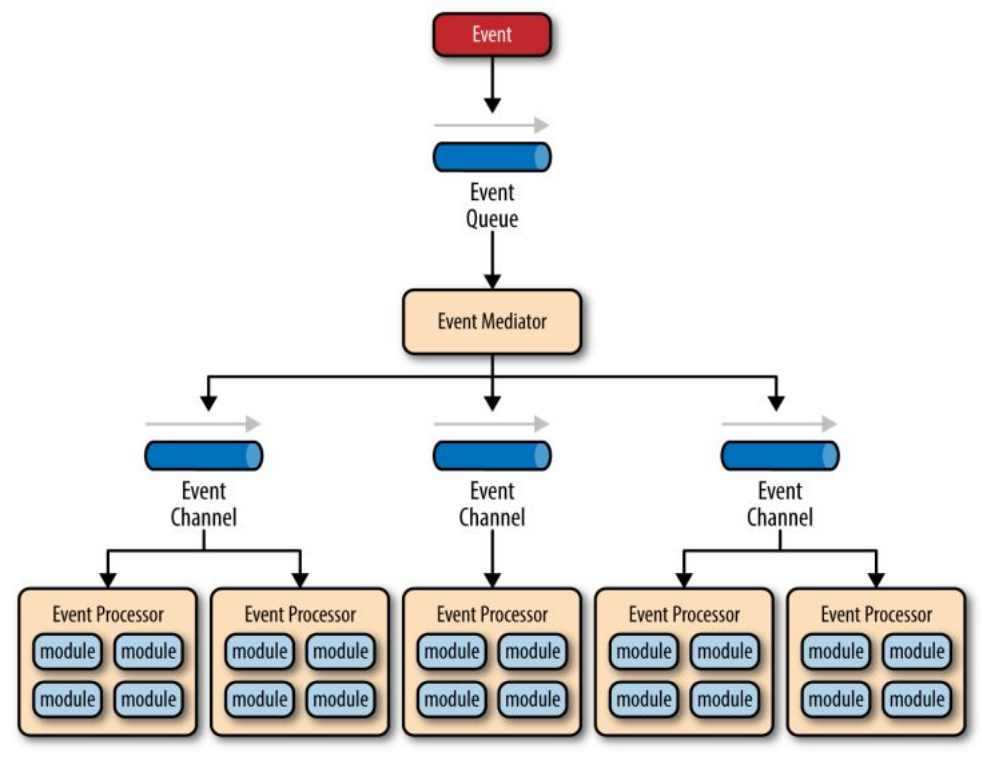
Exemple : ordre boursier

Variante “Médiateur” (Mediator)



Le médiateur d'événements reçoit l'événement initial et orchestre cet événement en envoyant des événements asynchrones supplémentaires aux canaux d'événements pour exécuter chaque étape du processus.

Variante “Médiateur” (Mediator)



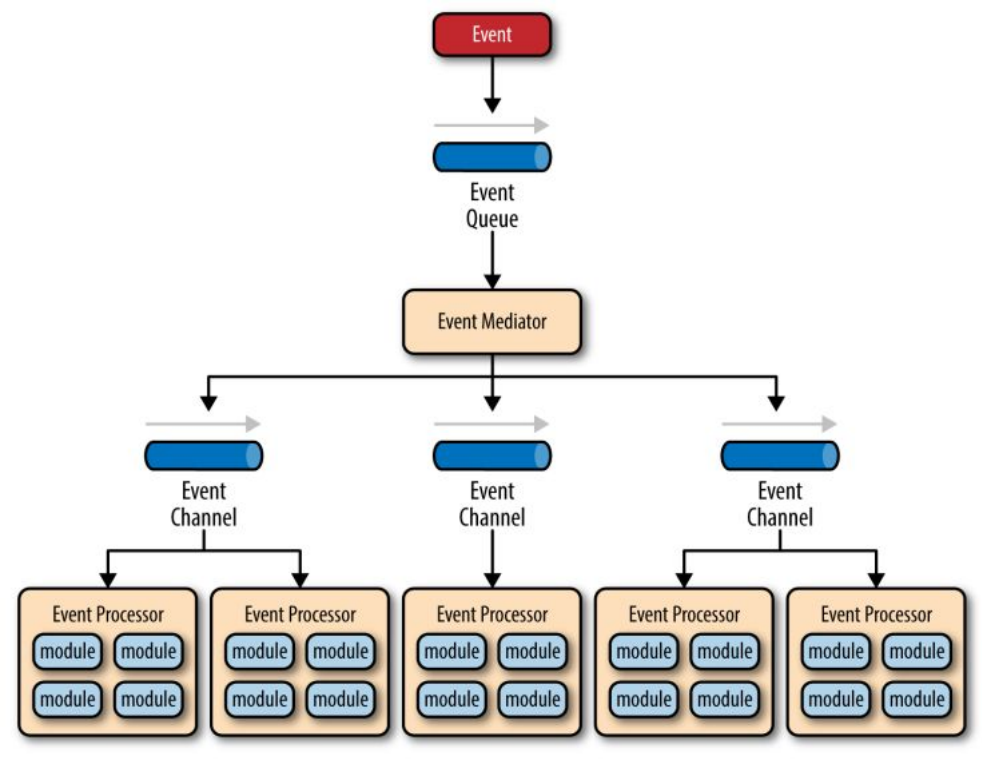
Les processeurs d'événements, qui écoutent les canaux d'événements, reçoivent l'événement du médiateur d'événements et exécutent une logique métier spécifique pour traiter l'événement.

Variante “Médiateur” (Mediator)

- Il existe deux types d'événements dans ce modèle :
  un événement initial et un événement de traitement.

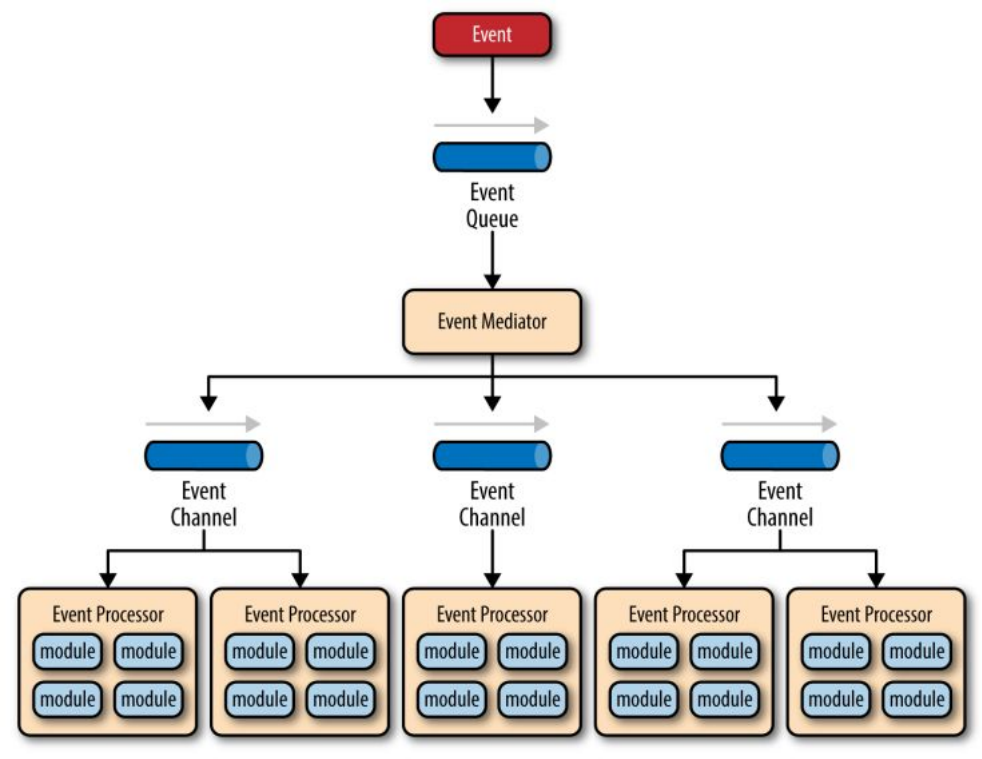
  L'événement initial est l'événement original reçu par le médiateur, tandis que les événements de traitement sont ceux qui sont générés par le médiateur et reçus par les composants de traitement des événements.

Variante “Médiateur” (Mediator)



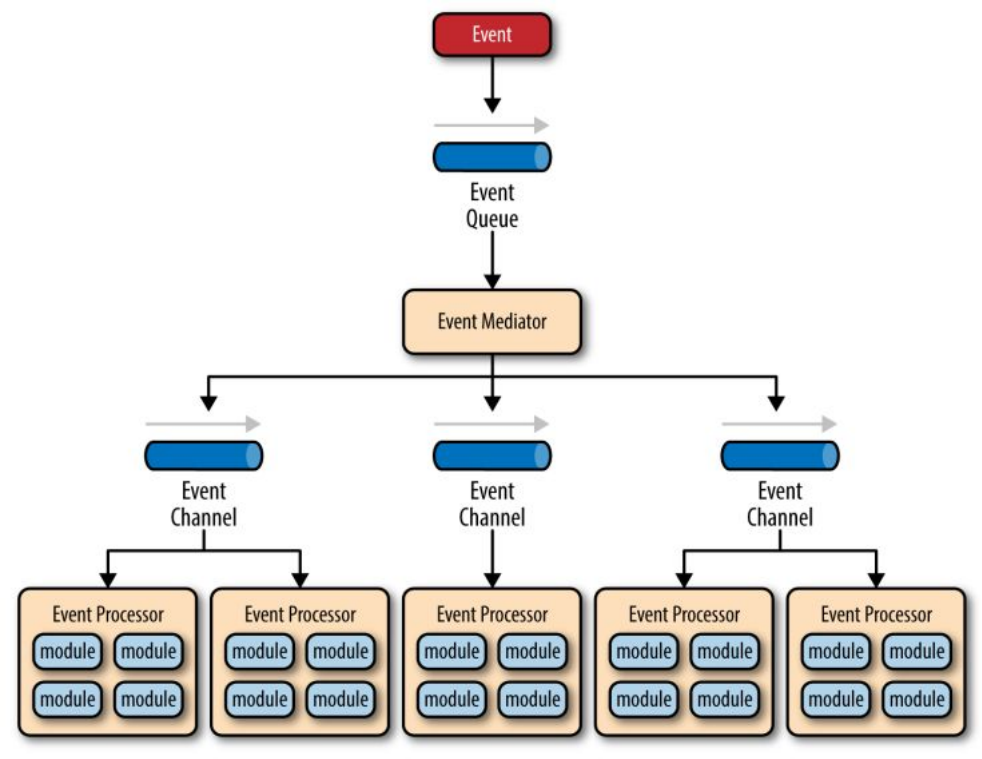
Le composant médiateur d'événements est chargé d'orchestrer les étapes contenues dans l'événement initial.
Pour chaque étape de l'événement initial, le médiateur d'événements envoie un événement de traitement spécifique vers un canal d'événements, qui est ensuite reçu et traité par le processeur d'événements.

Variante “Médiateur” (Mediator)



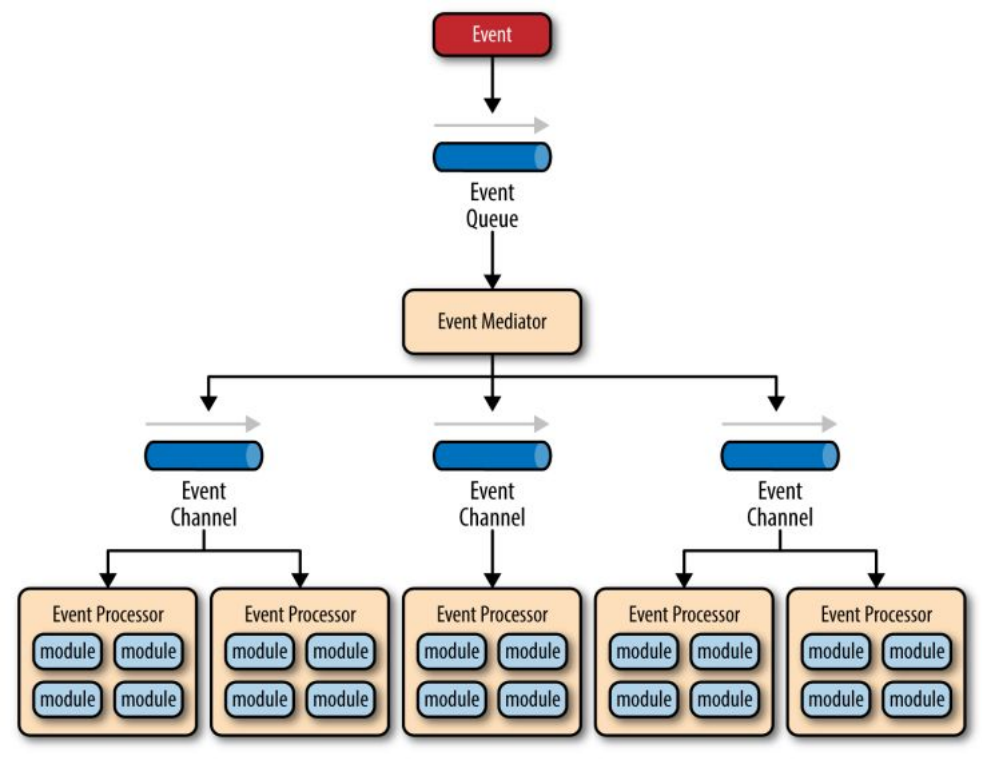
Il est important de noter que le médiateur d'événements n'exécute pas réellement la logique métier nécessaire au traitement de l'événement initial.
Il connaît plutôt les étapes nécessaires au traitement de l'événement initial.

Variante “Médiateur” (Mediator)



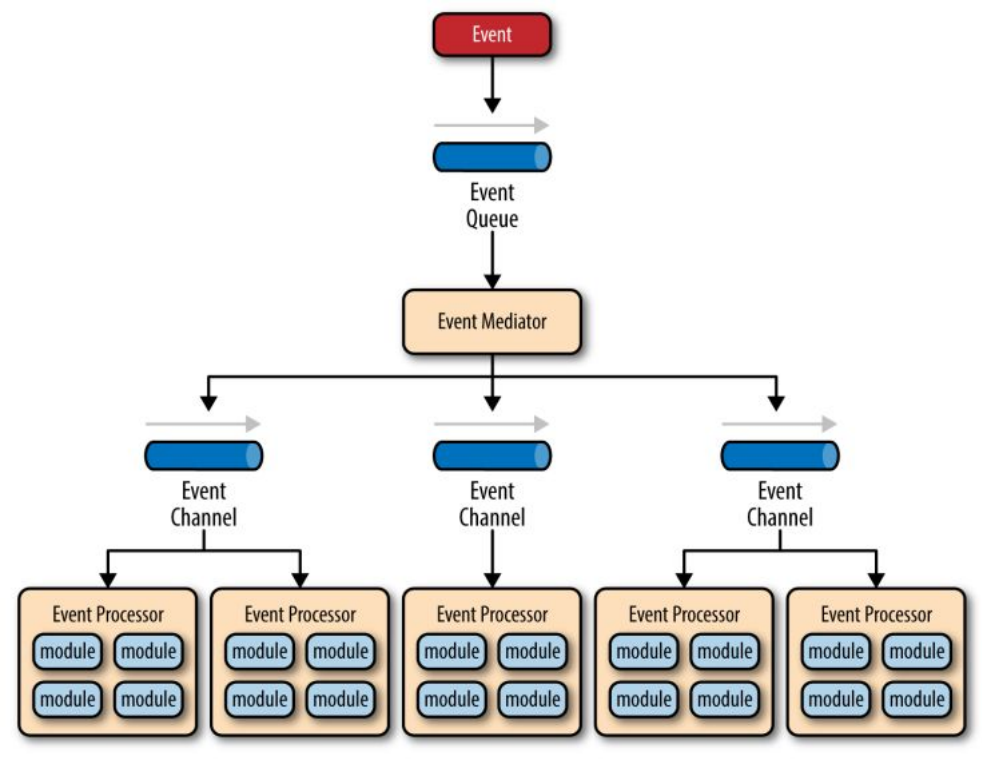
Les canaux d'événements sont utilisés par le médiateur d'événements pour transmettre de manière asynchrone des événements de traitement spécifiques liés à chaque étape de l'événement initial aux processeurs d'événements.
Les canaux d'événements peuvent être des files d'attente de messages (message queue) ou des sujets de messages (topics), bien que les sujets de messages soient le plus souvent utilisés avec la variante “médiateur” afin que les événements de traitement puissent être traités par plusieurs processeurs d'événements (chacun exécutant une tâche différente en fonction de l'événement de traitement reçu).

Variante “Médiateur” (Mediator)



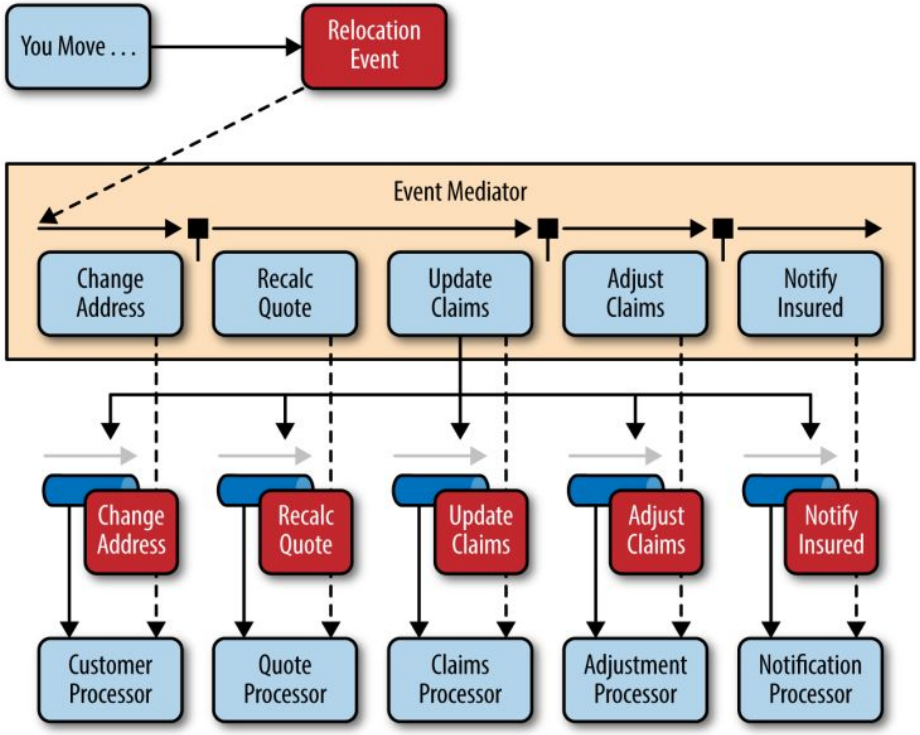
Les composants du processeur d'événements contiennent la logique métier de l'application nécessaire au traitement de l'événement. Les processeurs d'événements sont des composants d'architecture autonomes, indépendants et fortement découplés qui exécutent une tâche spécifique dans l'application ou le système.

Variante “Médiateur” (Mediator)



Bien que la granularité du composant de traitement d'événements puisse varier d'un grain fin (par exemple, calculer la taxe de vente sur une commande) à un grain grossier (par exemple, traiter une réclamation d'assurance), il est important de garder à l'esprit qu'en général, chaque composant de traitement d'événements doit effectuer une seule tâche métier et ne pas dépendre d'autres processeurs d'événements pour effectuer sa tâche spécifique.

Variante "Médiateur" (Mediator)



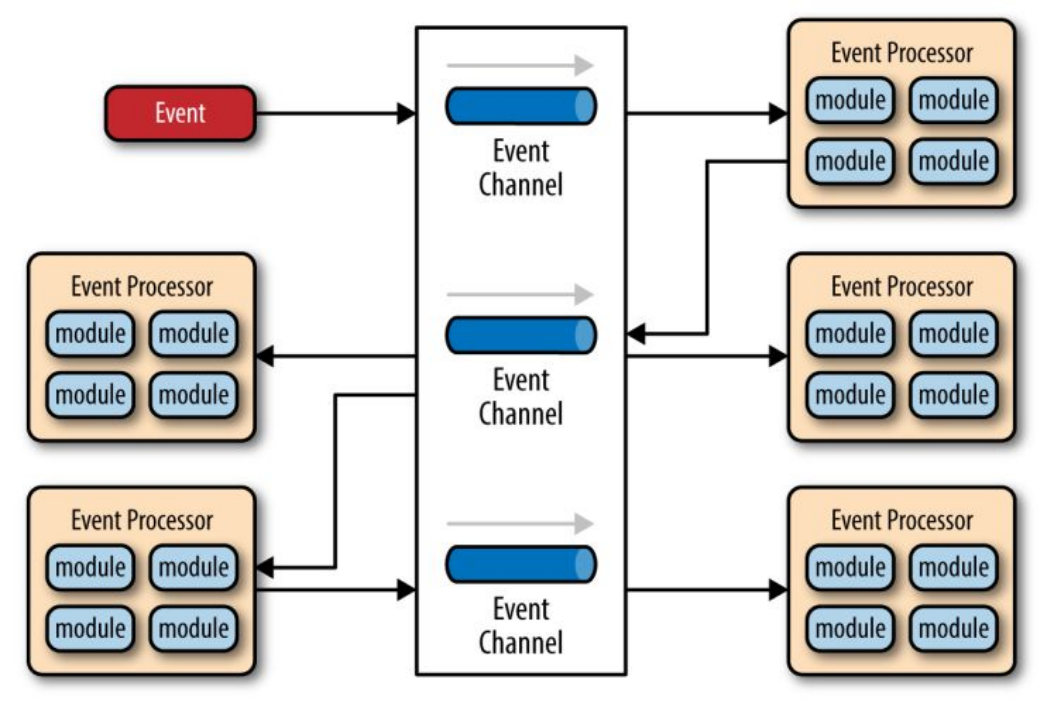
Autre exemple : cas du signalement de changement d'adresse auprès de votre assurance.

- déclaration du changement d'adresse
- mise à jour des devis et des demandes d'indemnisation en cours
- régularisation
- notification de prise en compte du changement

Variante “Courtier” (Broker)

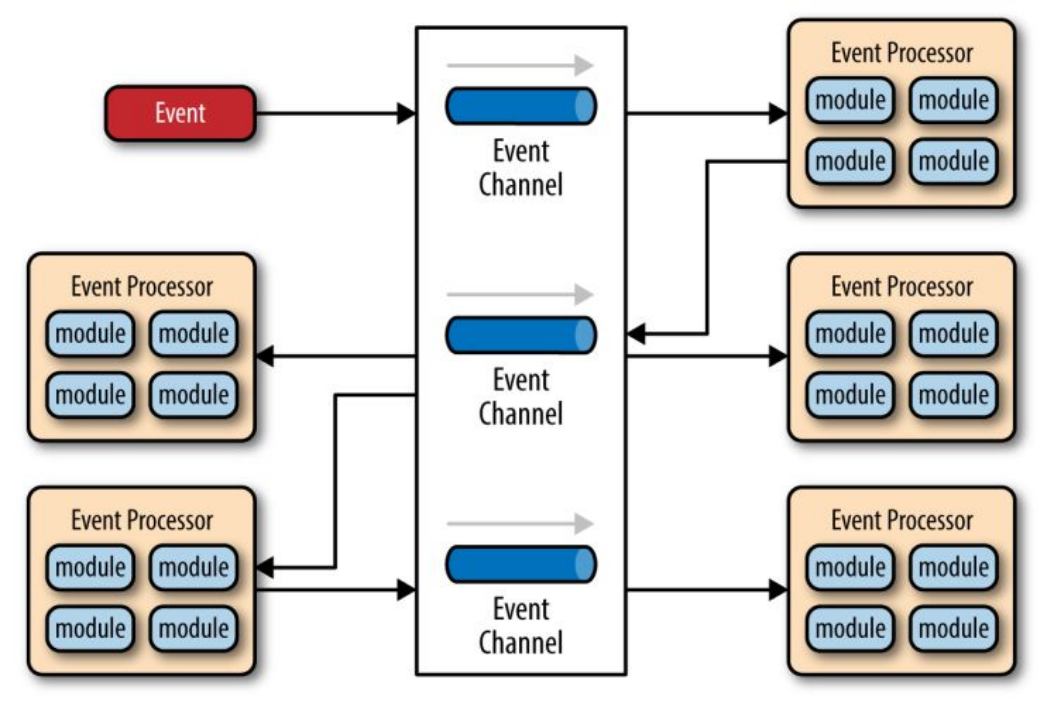
- La variante “courtier” diffère de la variante “médiateur” par le fait qu’il n'y a pas de médiateur central d'événements.
- Le flux de messages est plutôt distribué à travers les composants du processeur d'événements, à la manière d'une chaîne, par l'intermédiaire d'un courtier de messages léger.
- Cette variante est utile lorsque vous avez un flux de traitement des événements relativement simple et que vous ne voulez pas (ou n'avez pas besoin) d'une orchestration centrale des événements.
- Il existe deux principaux types de composants d'architecture dans la variante “courtier” : un composant courtier et un composant processeur d'événements. Le composant courtier peut être centralisé ou fédéré et contient tous les canaux d'événements qui sont utilisés dans le flux d'événements.

Variante “Courtier” (Broker)



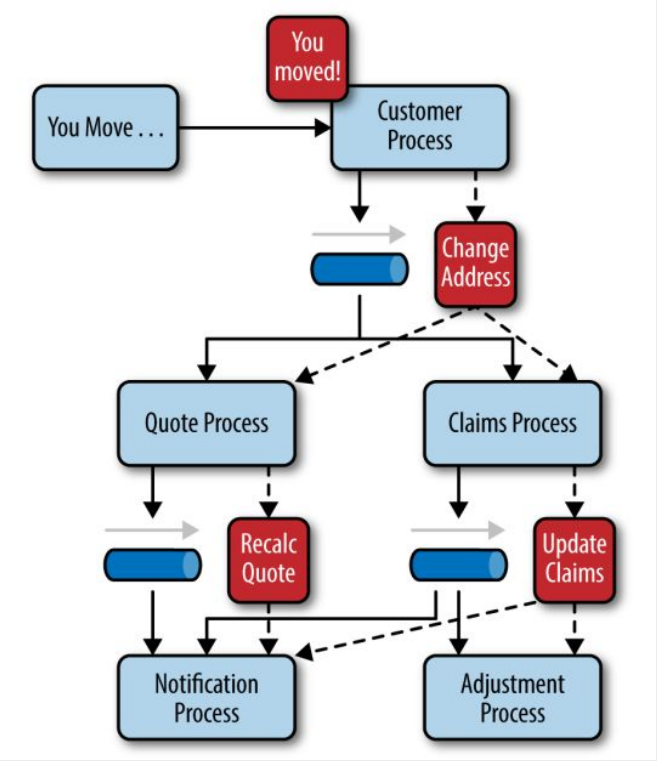
Comme il n'y a pas de médiateur central d'événements pour recevoir l'événement initial dans la variante “courtier”, le composant de traitement du client reçoit directement l'événement, modifie l'adresse du client et envoie un événement indiquant qu'il a modifié l'adresse d'un client.

Variante “Courtier” (Broker)



Dans cet exemple, deux processeurs d'événements sont intéressés par l'événement de changement d'adresse : le processus de devis et le processus d'indemnisation.
Le composant de traitement des devis recalcule les nouveaux tarifs d'assurance en fonction du changement d'adresse et publie un événement au reste du système indiquant ce qu'il a fait.

Variante “Courtier” (Broker)



Autre vue du même process :
- Evénement initial : changement d’adresse
- Le composant de traitement des devis recalcule les nouveaux tarifs d'assurance en fonction du changement d'adresse et publie un événement au reste du système indiquant ce qu'il a fait.
- Le composant de traitement des sinistres, quant à lui, reçoit le même événement de changement d'adresse, mais dans ce cas, il met à jour une demande d'assurance en cours et publie un événement dans le système.
- Ces nouveaux événements sont ensuite repris par d'autres composants de traitement d'événements, et la chaîne d'événements continue jusqu’à ce qu’il n’y ait plus d’évènement à traiter

Points d'attention

- Le modèle d'architecture orienté événements est un modèle relativement complexe à mettre en œuvre, principalement en raison de sa nature distribuée asynchrone.
- Lors de la mise en œuvre de ce modèle, vous devez résoudre divers problèmes d'architecture distribuée, tels que la disponibilité des processus distants et la logique de reconnexion des courtiers en cas de défaillance d'un courtier ou d'un médiateur.
- Une considération à prendre en compte lors du choix de ce modèle d'architecture est l'absence de transactions atomiques pour un seul processus.
- C'est pourquoi, lorsque vous concevez votre application à l'aide de ce modèle, vous devez continuellement réfléchir aux événements qui peuvent ou qui ne peuvent pas être exécutés indépendamment et prévoir la granularité de vos processeurs d'événements en conséquence.

Grille d'analyse (1/3)

- **Agilité globale : fort**
  L'agilité globale est la capacité de répondre rapidement à un environnement en constante évolution. Les composants des processeurs d'événements étant à usage unique et complètement découplés d'autres composants de processeurs d'événements, les changements sont généralement isolés à un ou quelques processeurs d'événements et peuvent être effectués rapidement sans avoir d'impact sur les autres composants.

- **Facilité de déploiement : fort**
  Dans l'ensemble, ce modèle est relativement facile à déployer en raison de la nature découplée des composants du processeur d'événements. La variante "courtier" tend à être plus facile à déployer que la variante "médiateur", principalement parce que le médiateur d'événements est assez étroitement couplé aux processeurs d'événements.

Grille d'analyse (2/3)

- **Testabilité : faible**
  Bien que les tests unitaires individuels ne soient pas trop difficiles, ils nécessitent une sorte de client de test spécialisé ou un outil de test pour générer des événements.
  Les tests sont également compliqués par la nature asynchrone et chronologique de ce modèle.

- **Performance : fort**
  Ce modèle d'architecture peut atteindre des performances élevées grâce à ses capacités asynchrones, c'est-à-dire la possibilité d'exécuter des tâches découplées et parallèles.
  La capacité d'effectuer des opérations asynchrones découplées et parallèles compense largement le coût de la mise en file d'attente des messages.

Grille d'analyse (3/3)

- **Scalabilité : fort**
  La scalabilité est naturellement obtenue dans ce modèle grâce à des processeurs d'événements hautement indépendants et découplés. Chaque processeur peut être mis à l'échelle séparément, ce qui permet une scalabilité très précise.

- **Facilité de développement : faible**
  Le développement peut être assez compliqué en raison de la nature asynchrone du modèle, la création de contrats et la nécessité d'une gestion des erreurs plus avancée dans le code pour les processeurs d'événements qui ne répondent pas et les courtiers défaillants.

Modèle 3 : Architecture orientée événements : conclusion

Agilité globale : fort	Facilité de déploiement : fort
Testabilité : faible	Performance : fort
Scalabilité : fort	Facilité de développement : faible

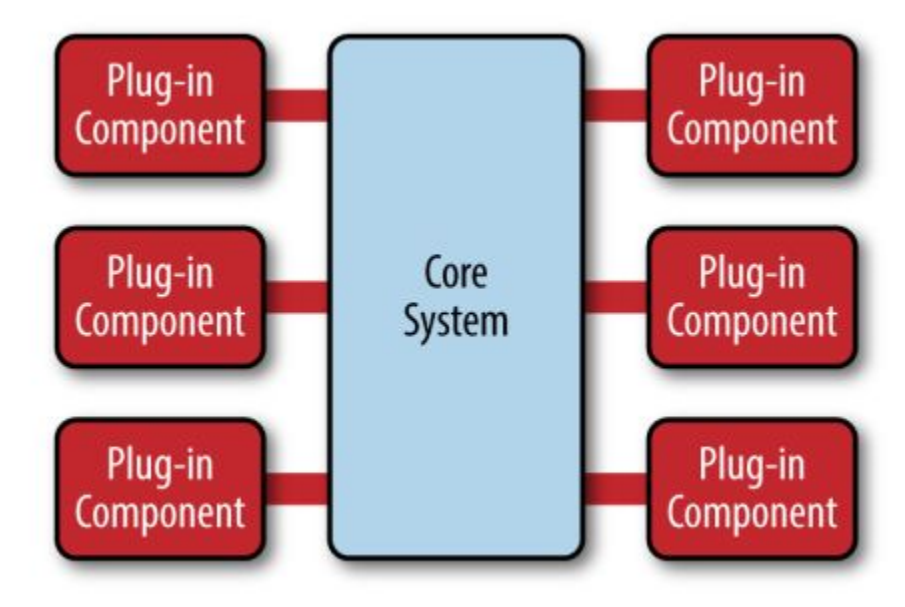
- la grille d'analyse est exactement inversée par rapport au modèle précédent
- Le point d'attention principal à retenir avant de choisir ce modèle est le besoin lié à l'aspect transactionnel des traitements.

Modèle 4 : Architecture microkernel (micro-noyau / plug-in)

- Le modèle d'architecture microkernel permet d'ajouter des fonctionnalités d'application supplémentaires sous forme de plug-ins à l'application de base.
- Ce modèle offre donc une extensibilité ainsi qu'une séparation et une isolation des fonctions.
- Ce modèle peut être adapté pour certaines applications d'entreprise ou pour des applications grand public à télécharger et à installer.

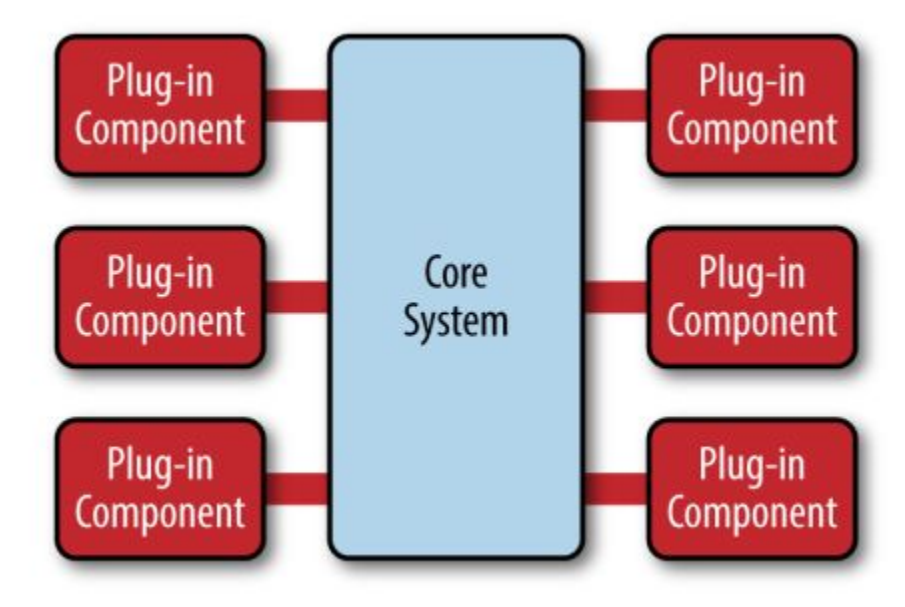
- Exemples : Wordpress, Outils de développement

Description du modèle (1/2)



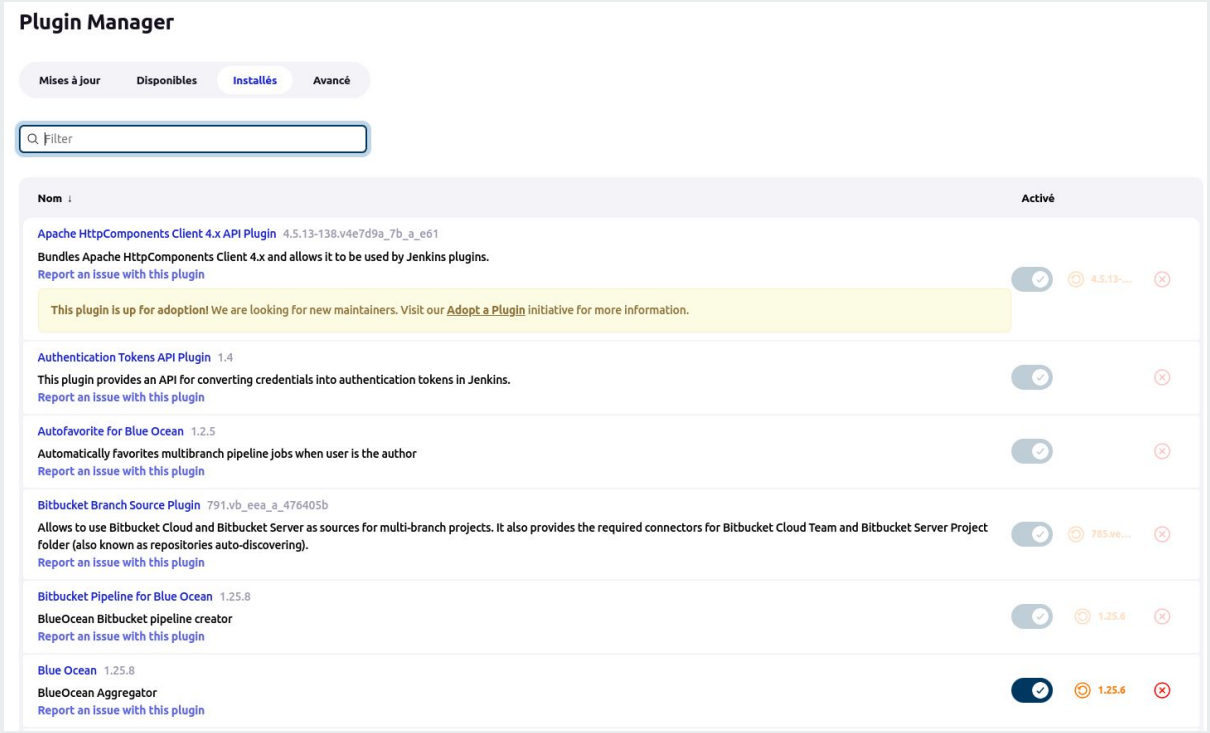
- 2 types de composants : le coeur et des plug-ins
- Le coeur du modèle d'architecture microkernel ne contient traditionnellement que les fonctionnalités minimales nécessaires à la réalisation des objectifs de l'architecture
- Les modules externes sont des composants autonomes et indépendants qui contiennent des fonctionnalités supplémentaires pour améliorer le système de base

Description du modèle (2/2)



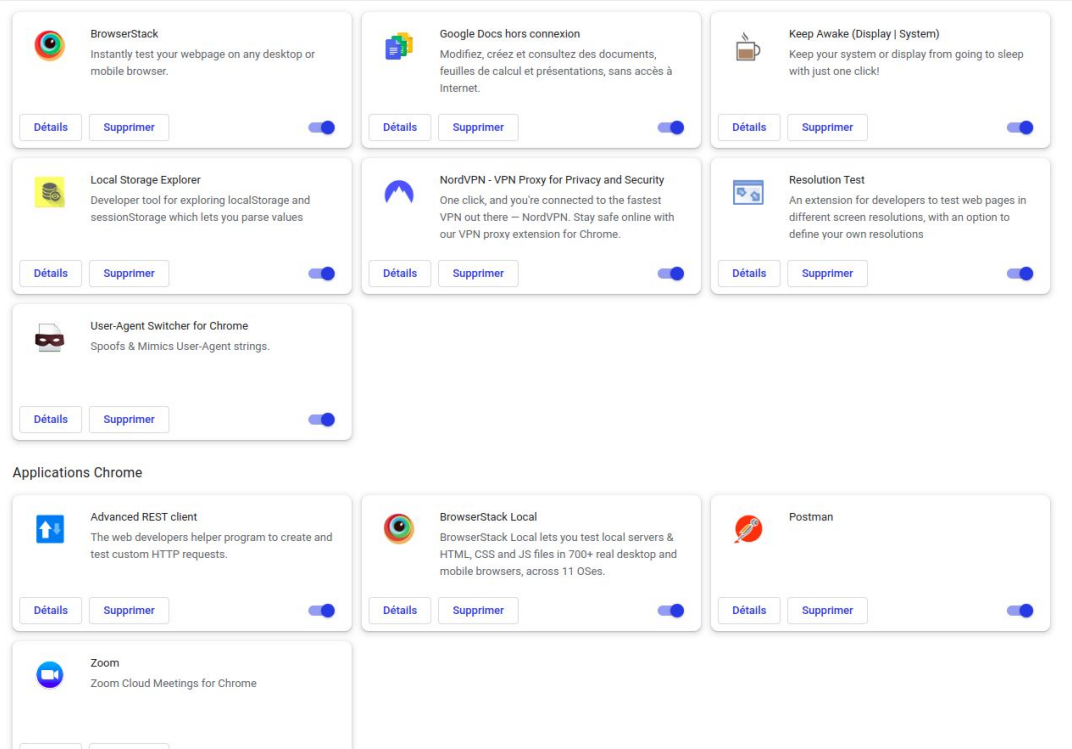
- En général, les modules externes doivent être indépendants des autres modules, mais dans la pratique, un module est souvent dépendant d'un ou plusieurs autres modules
- Le cœur a besoin de savoir quels modules externes sont disponibles et comment y accéder.
- Les relations entre le cœur et les modules externes sont définies par des contrats.

Exemple1 : Jenkins



Les plugins qui ne peuvent pas être désactivés sont ceux qui sont utilisés par d'autres plugins.

Exemple 2 : Chrome



Plus d'outils > Extensions

Points d'attention

- Le modèle d'architecture microkernel est un modèle particulièrement adapté pour une application avec des fonctionnalités relativement indépendantes les unes des autres et dont certaines peuvent être développées par des tiers.
- Une considération à prendre en compte lors du choix de ce modèle d'architecture est la contrainte de pouvoir ajouter et déployer des modules externes de façon dynamique.

Grille d'analyse (1/3)

- **Agilité globale : fort**
  L'agilité globale est la capacité de répondre rapidement à un environnement en constante évolution. Les changements peuvent être mis en œuvre rapidement grâce à des modules externes faiblement couplés. En général, le cœur de ce type d'applications nécessite peu de modifications au fil du temps.

- **Facilité de déploiement : fort**
  Selon la manière dont le modèle est mis en œuvre, les modules externes peuvent être ajoutés dynamiquement au coeur (déployés à chaud), sans interruption de service (ou très court)

Grille d'analyse (2/3)

- **Testabilité : fort**
  Les modules externes peuvent être testés de manière isolée et peuvent être facilement simulés (mock) par le cœur pour démontrer ou prototyper une fonctionnalité particulière avec peu ou pas de modifications du système central.

- **Performance : fort**
  Bien que le modèle de microkernel ne se prête pas naturellement aux applications à haute performance, en général, la plupart des applications construites à l'aide de ce modèle d'architecture ont de bonnes performances grâce à des mécanismes permettant d'activer uniquement les fonctionnalités nécessaires.

Grille d'analyse (3/3)

- **Scalabilité : faible**
  Les modules externes sont souvent conçus de telle façon que leur mise à l'échelle n'est pas prévue (pas de possibilité de scalabilité horizontale)

- **Facilité de développement : faible**
  L'architecture microkernel nécessite une conception et une gouvernance contractuelle très précises, ce qui la rend assez complexe à mettre en œuvre. Les versions des contrats, les registres de plug-ins, la granularité des plug-ins et le large choix disponible pour la connectivité des plug-ins contribuent à la complexité de la mise en œuvre de ce modèle.

Modèle 4 : Architecture microkernel : conclusion

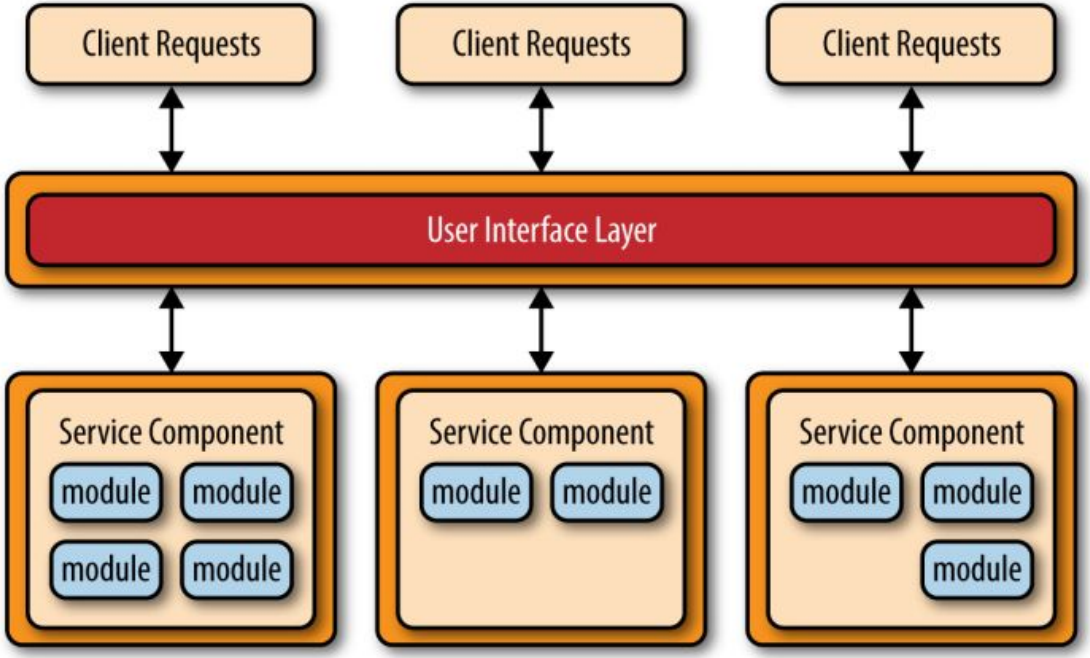
Agilité globale : fort	Facilité de déploiement : fort
Testabilité : fort	Performance : fort
Scalabilité : faible	Facilité de développement : faible

- Le point d'attention principal à retenir avant de choisir ce modèle est la complexité de mise en œuvre.
- Au-delà de la grille de critères, ce modèle sera retenu dans des contextes très particuliers.

Modèle 5 : Architecture microservices

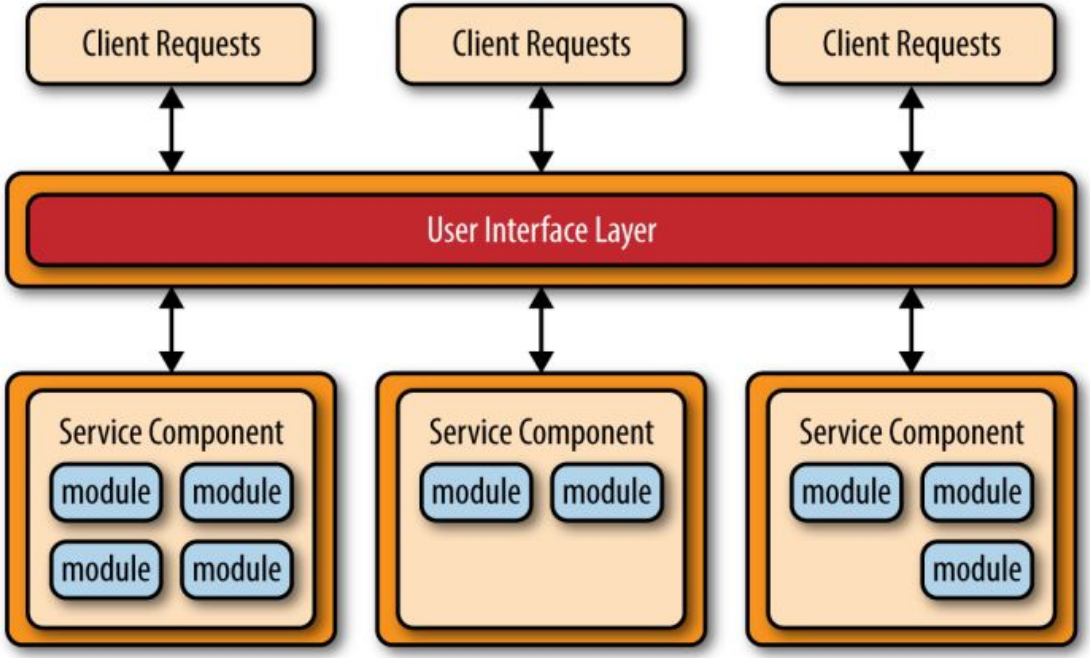
- Le modèle d'architecture microservices est reconnu comme une alternative viable aux applications monolithiques et aux architectures orientées services
- Date : début des années 2010

Description du modèle (1/2)



- Le concept le plus important est la notion d'unités déployées séparément.
- Chaque composant de l'architecture microservices est déployé en tant qu'unité distincte, ce qui permet une meilleure évolutivité et un degré élevé de découplage des composants de l'application.
- Les services (Service component) contiennent un ou plusieurs modules (par exemple, des classes Java) qui représentent aussi bien une petite fonctionnalité qu'une fonctionnalité plus importante d'une application.

Description du modèle (2/2)



- Les services (Service component) sont complètement indépendants pour leur développement, leurs tests, leur déploiement.
- Ils communiquent avec la couche UI par des protocoles d'accès distant.

Comparaison Architecture microservices et SOA

- SOA : Service Oriented Architecture
- Les deux architectures microservices et SOA s'appuient sur des services en tant que composant principal, mais leurs caractéristiques varient considérablement .

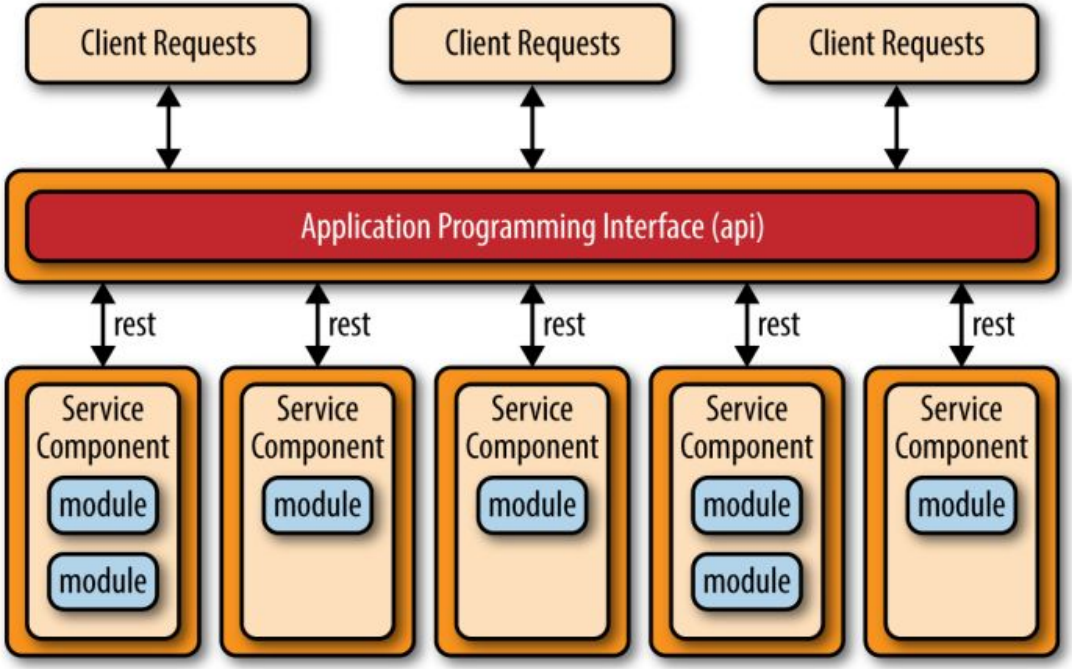
  2 différences principales :
  - La portée: l'architecture orientée services (SOA) a une portée "entreprise", tandis que l'architecture des microservices a une portée "application"

  - Évolutivité: L'architecture microservices est plus souple et donc évolutive que l'architecture orientée services (SOA) .

Architecture microservices : typologies

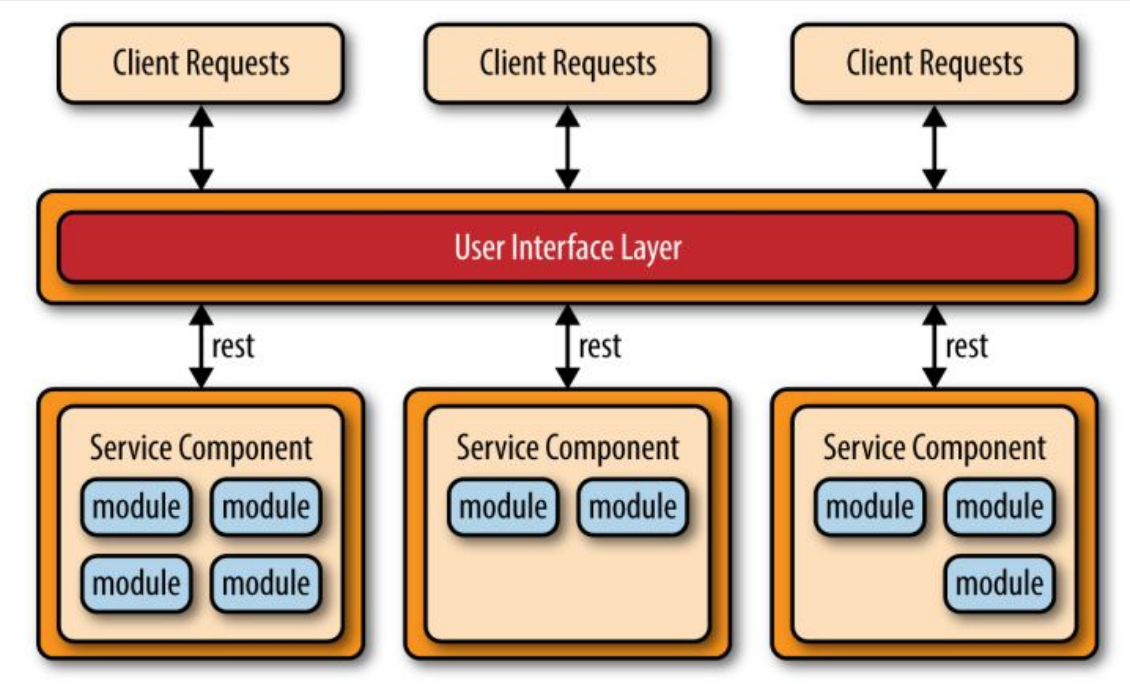
- Il y a un grand nombre d'implémentations possibles du modèle d'architecture microservices
- Néanmoins, on peut citer trois types d'implémentation :
  - API REST-based
  - Application REST-based
  - Centralized messaging

Typologie API REST-based



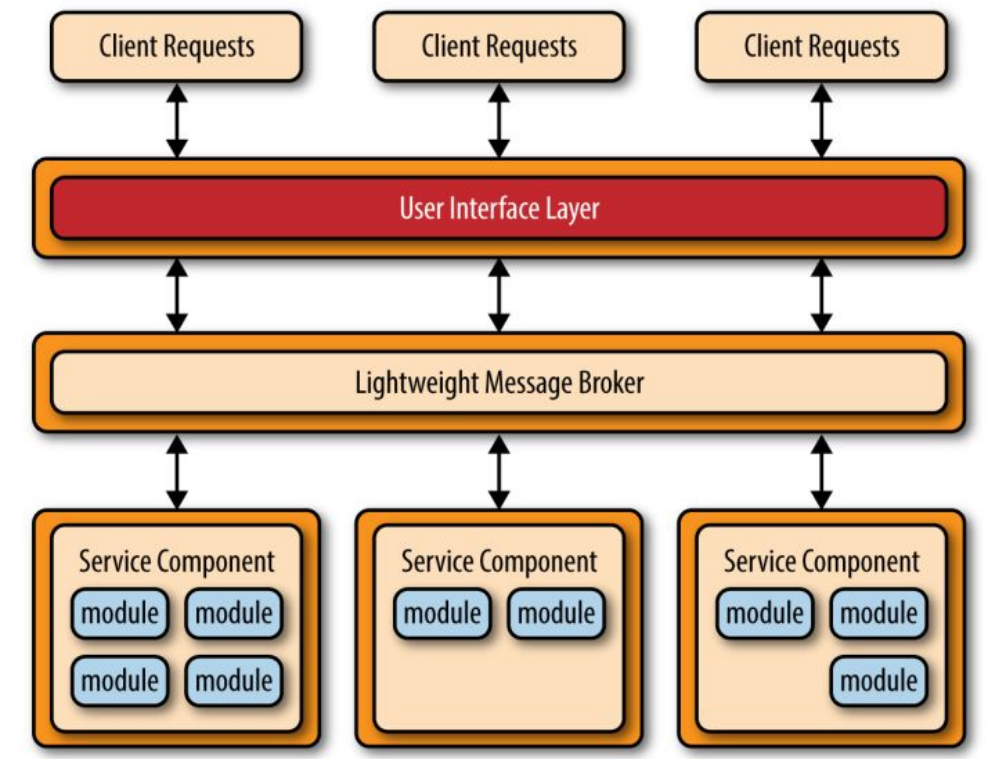
- Services à granularité très fines
- typologie adaptée pour des applications web qui exposent certains de leurs services via une couche dédiée (API)
- Exemple : service d'authentification Google

Typologie Application REST-based



- Services à granularité moyenne
- typologie adaptée pour des applications métiers de taille modeste

Typologie centralized messaging (“messagerie centralisée”)



- typologie adaptée pour des applications métiers de taille plus importante qui nécessite d’autres types de communication que ceux apportés par REST : files d’attente, asynchronisme

Points d'attention

- La principale difficulté dans l'utilisation du modèle d'Architecture microservices est de trouver la bonne granularité des services.
- Il peut y avoir des situations qui nécessitent que deux services partagent une même information. Dans ce cas, le recours à une base de données partagée est possible.
- En raison de la nature distribuée de ce modèle, il est très difficile de maintenir une seule unité transactionnelle entre les composants du service. Cela nécessite la mise en place de mécanismes de compensation de transactions.
- Un autre avantage de ce modèle est qu'il permet de d'effectuer des déploiements isolés et donc à impacts limités (pas d'effet big bang)
- Le modèle d'architecture microservices est une architecture distribuée, il partage certains des problèmes complexes que l'on retrouve dans le modèle d'architecture orienté événements, notamment la création, la maintenance et la gouvernance des contrats, la gestion de la disponibilité des systèmes, ainsi que l'authentification et l'autorisation de l'accès à distance.

Grille d'analyse (1/3)

- **Agilité globale : fort**
  L'agilité globale est la capacité de répondre rapidement à un environnement en constante évolution. Les modifications sont généralement isolées à un nombre de composants de service réduit, ce qui permet un déploiement rapide et facile. De plus, les applications construites à l'aide de ce modèle tendent à être très peu couplées, ce qui facilite également le changement.

- **Facilité de déploiement : fort**
  Les services sont généralement déployés en tant qu'unités logicielles distinctes, ce qui permet d'effectuer des "déploiements à chaud" à tout moment. Le risque global lié au déploiement est également réduit de manière significative, dans la mesure où les déploiements ratés peuvent être restaurés plus rapidement et ont un impact limité aux services concernés.

Grille d'analyse (2/3)

- **Testabilité : fort**
  Les tests peuvent être ciblés par composant ce qui rend leur développement plus facile et leur exécution plus rapide.
  Par contre, les tests globaux sont plus compliqués à réaliser.

- **Performance : faible**
  Ce modèle d'architecture ne se prête pas naturellement aux applications à hautes performances en raison de sa nature distribuée

Grille d'analyse (3/3)

- **Scalabilité : fort**
  La mise à l'échelle peut être gérée par service en fonction de leurs besoins et leurs sollicitations.

- **Facilité de développement : fort**
  Les développements sont limités à chaque composant ce qui les facilite et réduit les risques de régression.

Modèle 4 : Architecture microservices : conclusion

Agilité globale : fort	Facilité de déploiement : fort
Testabilité : fort	Performance : faible
Scalabilité : fort	Facilité de développement : fort

- Le point d'attention principal à retenir avant de choisir ce modèle est la difficulté pour définir la bonne granularité des services.
- Le critère "Performance" noté faible peut être compensé par certains choix technologiques

Architecture microservices : un peu d'histoire

SOMMAIRE

Plus qu'une nouvelle révolution dans les architectures informatiques, les microservices sont une évolution de l'architecture SOA (pour *Service Oriented Architecture*) dont on a beaucoup parlé au début des années 2000. Une évolution logique qui résulte d'un long retour d'expérience dans la mise en place d'architecture SOA, et qui vient gommer certains de ses défauts. Jérôme Mainaud, architecte logiciel chez Ippon Technologies, livre sa définition des microservices : "c'est un style d'architecture logicielle dont les différents composants sont développés sous la forme de services autonomes qui s'exécutent avec leur propre processus. Ce modèle s'oppose ainsi aux architectures monolithiques où les composants écrits sous la forme de bibliothèque, modules ou classes sont tous déployés dans le même processus."

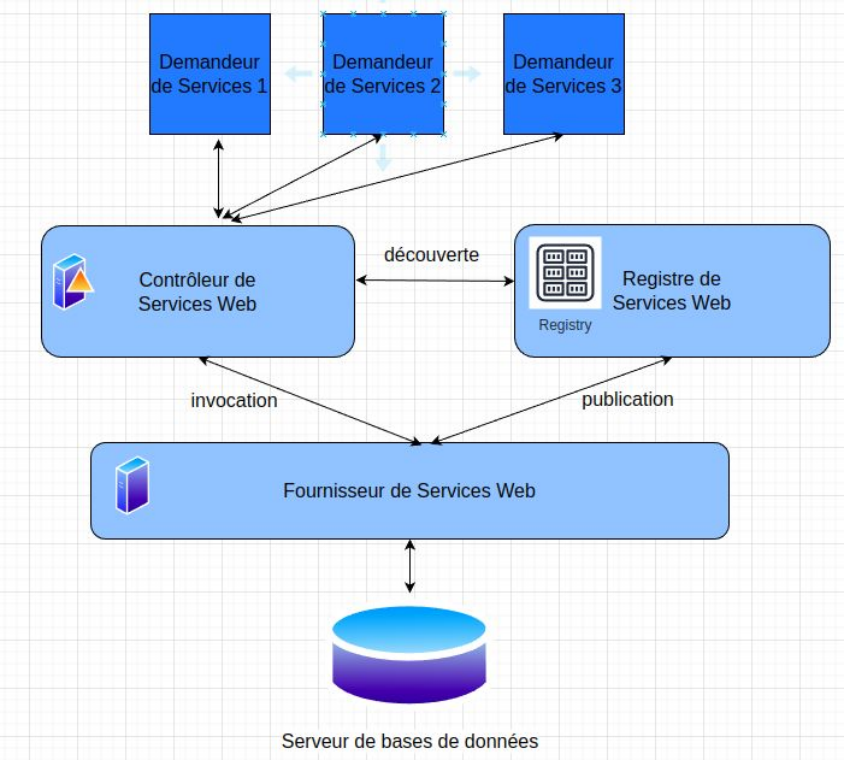
Extrait d'un article paru dans le Journal du Net, le 17 novembre 2015

Modèle 6 : Architecture Orientée Services

- L'architecture orientée services (ou SOA, Service-Oriented Architecture) 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.
- L'architecture SOA permet à des composants logiciels déployés et gérés séparément de communiquer et de fonctionner ensemble sous la forme d'applications logicielles communes à différents systèmes.

- Date : début des années 2000

Description du modèle



- 3 types de composants :
  - Le registre de services (Web Services Registry) : fournit les informations sur les services disponibles
  - Le contrôleur de services web interprète une demande de services et recherche la fonction dans le registre. Il exécute ensuite cette fonction et renvoie le résultat au demandeur.
  - Le fournisseur de services crée des services web qu'il met à disposition dans un registre de services. Il est responsable des conditions d'utilisation du service.

Points d'attention et exemples

- Le modèle Architecture Orientée Service peut parfois être confondu avec le modèle Architecture Micro Services.
  Une des grandes différences entre ces deux modèles est leur portée : l'Architecture Orientée Service a une portée "entreprise" alors que l'Architecture Micro Services a une portée "application".
  Les composants mis en œuvre ne sont pas non plus les mêmes.

- Exemple d'utilisation d'Architecture Orientée Service : cas du transporteur qui met à disposition un service de livraison à une plateforme de commerce en ligne

- Autres exemples : convertisseur de devises en ligne, informations sur les vols commerciaux en temps réel

Grille d'analyse (1/3)

- **Agilité globale : fort**
  L'agilité globale est la capacité de répondre rapidement à un environnement en constante évolution. Les modifications sont généralement isolées à un nombre de composants de service réduit, ce qui permet un déploiement rapide et facile. La présence d'un registre permet de gérer la publication de nouvelles versions de services.

- **Facilité de déploiement : fort**
  Les services sont généralement déployés en tant qu'unités logicielles distinctes, ce qui permet d'effectuer des "déploiements à chaud" à tout moment. Le risque global lié au déploiement est également réduit de manière significative, dans la mesure où les déploiements ratés peuvent être restaurés plus rapidement et ont un impact limité aux services concernés.

Grille d'analyse (2/3)

- **Testabilité : fort**
  Les tests peuvent être ciblés par composant ce qui rend leur développement plus facile et leur exécution plus rapide.
  Par contre, les tests globaux sont plus compliqués à réaliser.

- **Performance : faible**
  Ce modèle d'architecture ne se prête pas naturellement aux applications à hautes performances en raison de sa nature distribuée

Grille d'analyse (3/3)

- **Scalabilité : fort**
  La mise à l'échelle peut être gérée par service en fonction de leurs besoins et leurs sollicitations.

- **Facilité de développement : fort**
  Les développements sont limités à chaque composant ce qui les facilite et réduit les risques de régression.

Modèle 6 : Architecture orientée services : conclusion

Agilité globale : fort	Facilité de déploiement : fort
Testabilité : fort	Performance : faible
Scalabilité : fort	Facilité de développement : fort

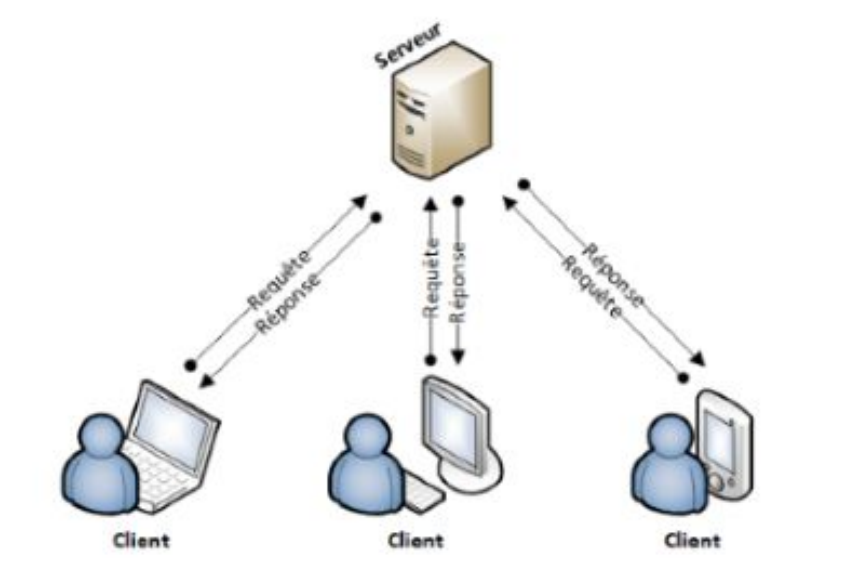
- Le point d'attention principal à retenir avant de choisir ce modèle est la difficulté pour définir la bonne granularité des services et de bien définir leur contrat.

Modèle 7 : Architecture client - serveur

- Une architecture client-serveur représente l'environnement dans lequel des applications de machines clientes communiquent avec des applications de machines de type serveurs.
- Caractéristiques d'un serveur :
  - Il est à l'écoute, prêt à répondre aux requêtes envoyées par des clients,
  - Dès qu'une requête lui parvient, il la traite et envoie une réponse.
- Caractéristiques d'un client :
  - Il envoie des requêtes au serveur,
  - Il attend et reçoit les réponses du serveur.
- Le client et le serveur doivent bien sûr utiliser le même protocole de communication. Un serveur est généralement capable de servir plusieurs clients simultanément.

- Date : fin des années 1980

Schéma


Serveur
Requête
Réponse
Requête
Réponse
Réponse
Requête
Client
Client
Client

Grille d'analyse (1/3)

- **Agilité globale : faible**
  L'agilité globale est la capacité de répondre rapidement à un environnement en constante évolution. Les modifications impactent potentiellement toute la partie serveur et dans certains cas les clients.

- **Facilité de déploiement : faible**
  Le déploiement de la partie serveur ne pose pas de problème.
  Par contre, en fonction du type de clients (lourd, léger), les modifications peuvent nécessiter une mise à jour de tous les clients.

Grille d'analyse (2/3)

- **Testabilité : faible**
  Il peut être difficile d'isoler les différentes parties à tester.

- **Performance : fort**
  L'aspect centralisé de cette architecture permet d'offrir de bonnes performances

Grille d'analyse (3/3)

- **Scalabilité : faible**
  L'aspect centralisé de cette architecture rend compliqué une mise à l'échelle du système.

- **Facilité de développement : fort**
  A l'époque à laquelle ce modèle était très utilisé, de nombreux outils permettaient de rendre les développements plus faciles et rapides .

Modèle 7 : Architecture client - serveur : conclusion

Agilité globale : faible	Facilité de déploiement : faible
Testabilité : faible	Performance : fort
Scalabilité : faible	Facilité de développement : fort

- De nombreux exemples d'utilisation de ce modèle :
  - client de messagerie : outlook
  - serveur d'impression
  - ERP

Architecture client - serveur : un peu d'histoire

Publié le 10 févr. 1993 à 1:01

Avant de devenir une réalité au sein des entreprises, l'architecture « client-serveur » a envahi les plaquettes commerciales, des éditeurs de logiciels aux constructeurs informatiques en passant par les SSII. Tous les acteurs de l'industrie informatique sont devenus subitement « compétents » en la matière, alors que le concept émergeait à peine il y a quatre ans. Pourtant, plus qu'un nouvel effet de mode, il s'agit bien d'un mouvement de fond qui répond aux aspirations technologiques et économiques des utilisateurs.

Le « client-serveur » est une architecture informatique (c'est-à-dire un ensemble de règles d'assemblage et de fonctionnement) qui repose sur deux types de matériels: les uns « clients » et les autres « serveurs ». Les systèmes clients, généralement des micro-ordinateurs (souvent sous MS-DOS ou Windows), gèrent l'application et son interface utilisatrice. Les serveurs, micros ou systèmes intermédiaires (souvent sous Unix), gèrent la base de données. Les deux types de systèmes étant reliés par un réseau de type Token Ring ou Ethernet.

extrait d'un article paru dans Les Echos le 10 février 1993

Comparaison entre différents modèles

Monolithes vs microservices - 1'33
https://www.youtube.com/watch?v=YfGBvNGSOss

Introduction to Microservices and Event-Driven Architecture - 12'50
https://www.youtube.com/watch?v=tZPFsv846kg

Modèle 8 : Architecture hexagonale

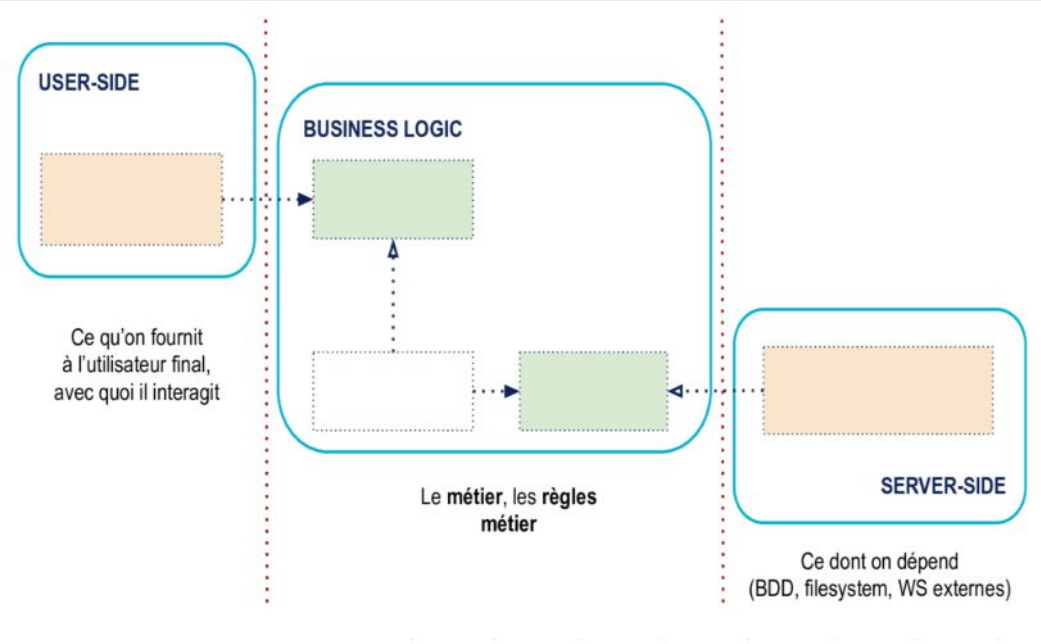
- Objectif de ce modèle : Permettre à une application d'être pilotée aussi bien par des utilisateurs que par des programmes, des tests automatisés ou des scripts batchs, et d'être développée et testée en isolation de ses éventuels systèmes d'exécution et bases de données.

- L' architecture hexagonale s'appuie sur trois principes et techniques :
  - Séparer explicitement User-Side, Business Logic et Server-Side
  - Les dépendances vont vers la Business Logic
  - On isole les frontières par des Ports et Adapters

- Date : 2005 (Alistair Cockburn)

Principe 1 : Séparer explicitement User-Side, Business Logic et Server-Side

Le premier principe est de séparer explicitement le code en trois grandes zones formalisées.



Principe 1 : Séparer explicitement User-Side, Business Logic et Server-Side

- À gauche, la zone User-Side (ou Left Side)

C'est le côté par lequel l'utilisateur ou les programmes extérieurs vont interagir avec l'application. On y trouve le code qui permet ces interactions. Typiquement, votre code d'interface utilisateur, vos routes HTTP pour une API, vos sérialisations en JSON à destination de programmes qui consomment votre application sont ici.

C'est le côté où l'on retrouve les acteurs qui pilotent la Business Logic.

- Au centre, la Business Logic (Hexagone ou Center)

C'est la partie que l'on veut isoler de ce qui est à gauche et à droite. On y trouve tout le code qui concerne et implémente la logique métier. Le vocabulaire métier et la logique purement métier, ce qui se rapporte au problème concret que résout votre application, tout ce qui en fait la richesse et la spécificité est au centre.

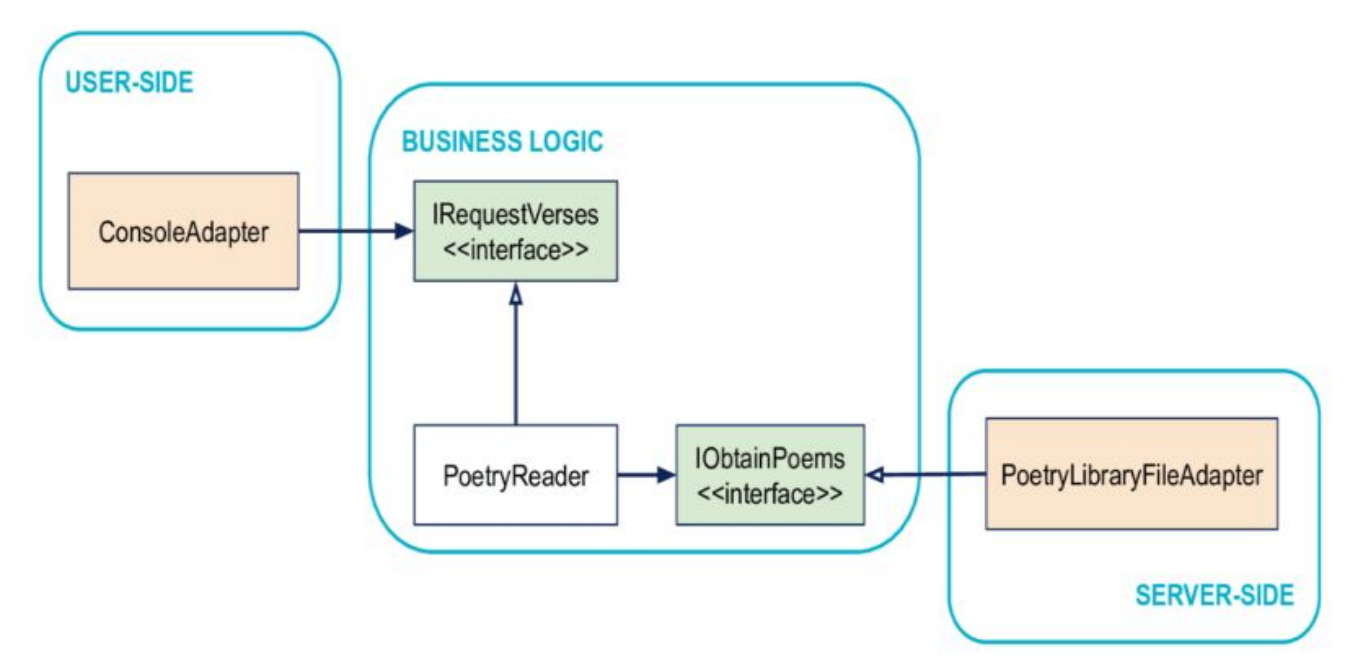
- À droite, la zone Server-Side (ou Right Side)

C'est ici qu'on va retrouver ce dont votre application a besoin, ce qu'elle pilote pour fonctionner. On y trouve les détails d'infrastructure essentiels comme le code qui interagit avec votre base de données, les appels au système de fichier, ou le code qui gère des appels HTTP à d'autres applications dont vous dépendez par exemple.

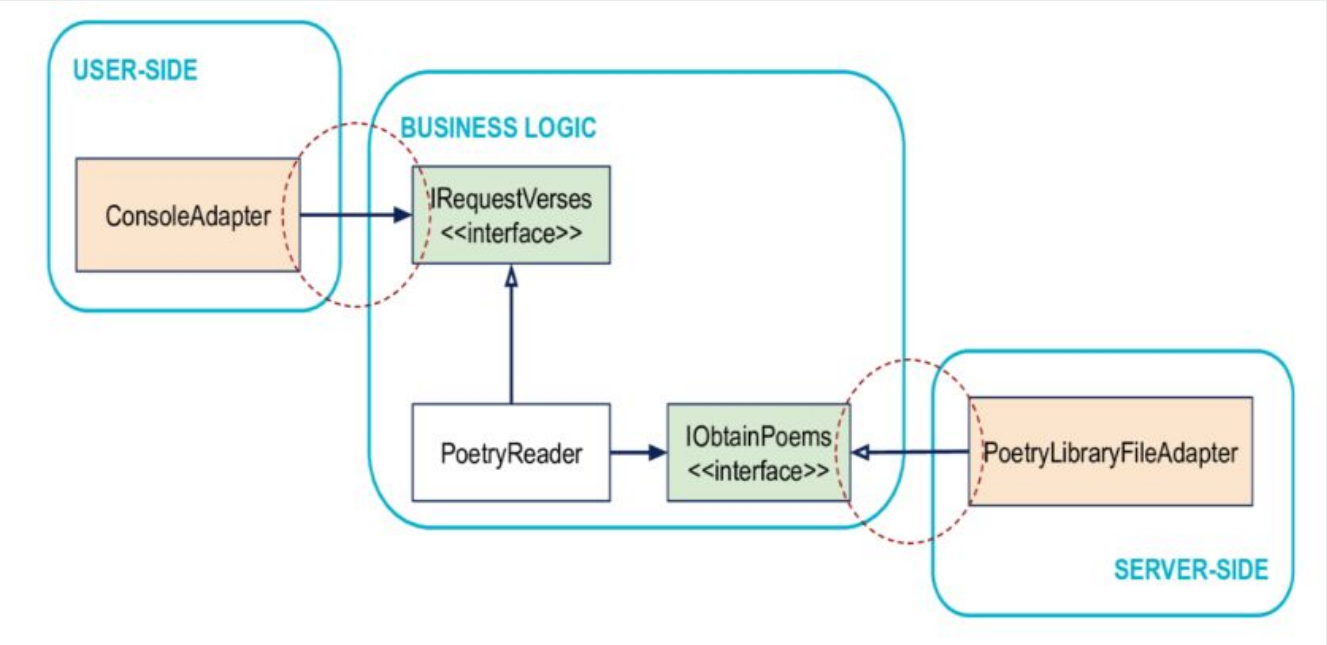
C'est le côté où l'on retrouve les acteurs qui sont pilotés par la zone Business Logic.

Principe 1 : Séparer explicitement User-Side, Business Logic et Server-Side

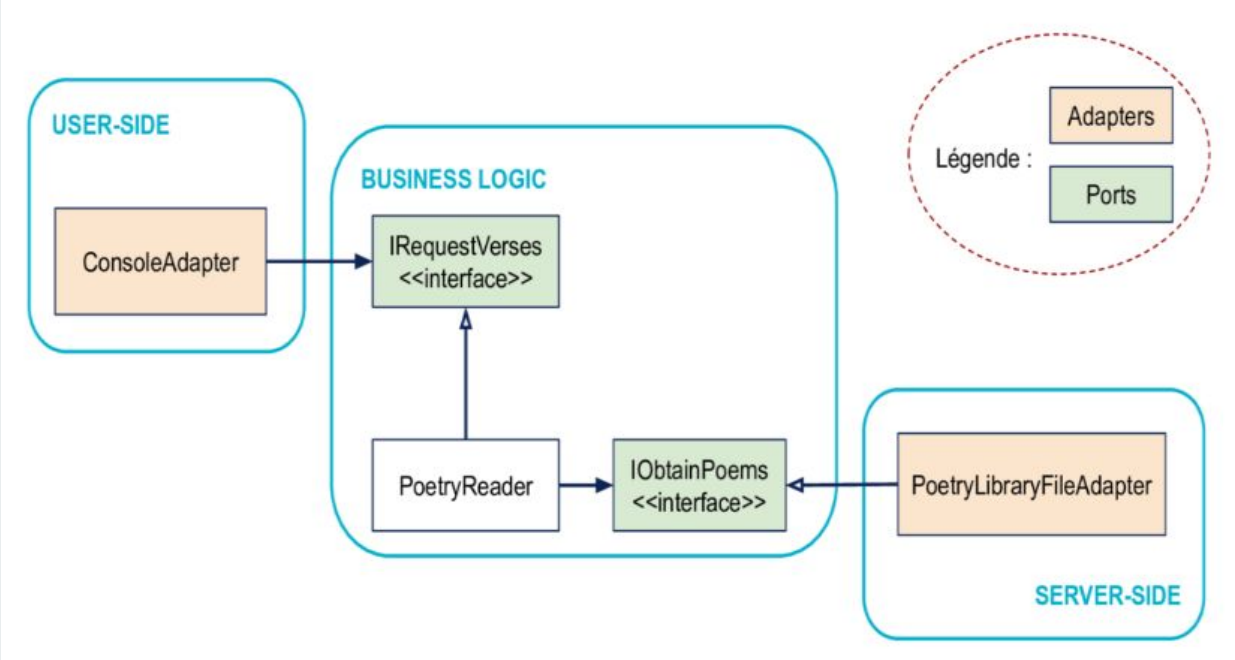
Exemple : une petite application qui va chercher des poèmes dans un système extérieur : un fichier.txt et qui les écrit dans la console



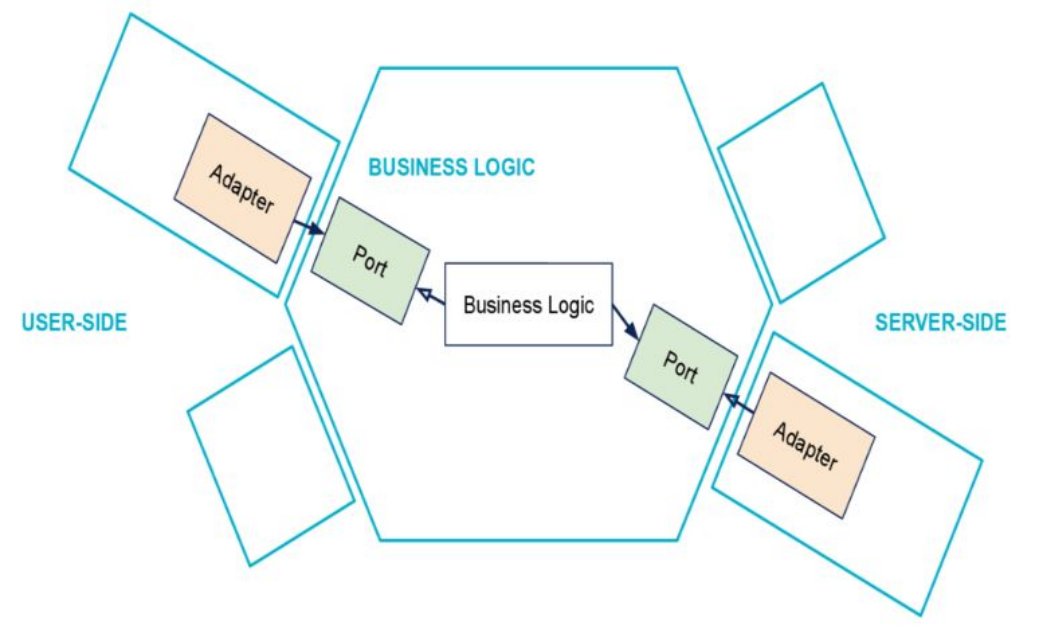
Principe 2 : Les dépendances vont vers la Business Logic



Principe 3 : on isole les frontières par des interfaces



Pourquoi “hexagonale” ?



La raison principale est que c’est une forme facile à dessiner qui laisse la place pour représenter plusieurs ports et adapters sur les schémas. Et il se trouve que même si l’hexagone est assez anecdotique au final, l’expression Hexagonal Architecture est plus populaire que Ports & Adapters Pattern.

Grille d'analyse (1/3)

- **Agilité globale : fort**
  Le métier (Hexagone) est séparé de l'infrastructure et de ses détails è donc un changement au niveau de l'infrastructure par exemple n'engendre pas une régression sur le métier de l'application.

- **Facilité de déploiement : fort**
  Les 3 parties sont déployées en tant qu'unités logicielles distinctes, ce qui permet d'effectuer des "déploiements à chaud" à tout moment. Le risque global lié au déploiement est également réduit de manière significative, dans la mesure où les déploiements ratés peuvent être restaurés plus rapidement.

Grille d'analyse (2/3)

- **Testabilité : fort**
  Il est facile d'isoler les différentes parties à tester.

- **Performance : faible**
  Ce n'est pas le critère prépondérant de ce modèle mais il peut être compensé par des choix technologiques.

Grille d'analyse (3/3)

- **Scalabilité : faible**
  Comme pour le critère Performance, ce n'est pas non plus le critère prépondérant de ce modèle mais il peut être compensé par des choix technologiques.

- **Facilité de développement : fort**
  Une fois les concepts de ce modèle acquis, la séparation des différentes parties rend les développements plus faciles à organiser.

Modèle 8 : Architecture hexagonale

Agilité globale : fort	Facilité de déploiement : fort
Testabilité : fort	Performance : faible
Scalabilité : faible	Facilité de développement : fort

- L'architecture hexagonale est un bon compromis complexité / puissance.
- Une solution adaptée dans les situations où la partie présentation et/ou la partie infrastructure peuvent évoluer souvent.

- Pour aller plus loin : https://www.youtube.com/watch?v=-dXN8wkN0yk

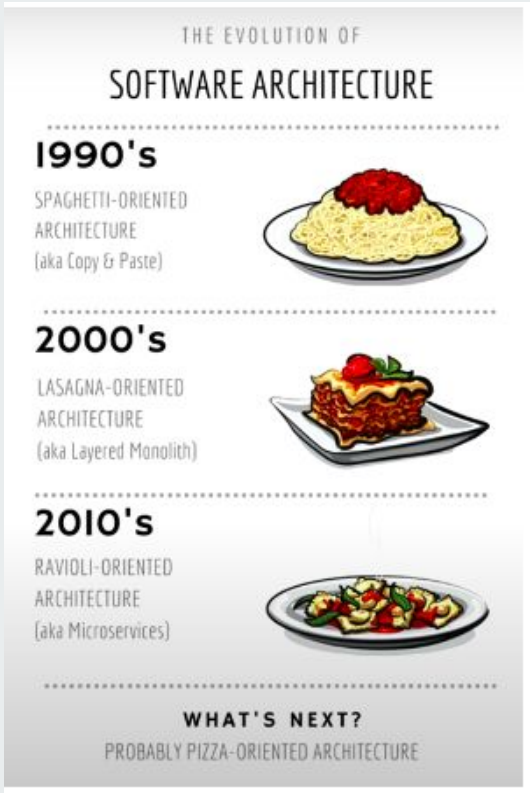
Conclusion / rappel : avertissement sur l'évaluation et la comparaison des modèles

- Pour chaque modèle, une grille d'analyse est proposée comprenant 6 critères de notation.
  L'évaluation de ces critères est parfois subjective et peut dépendre des choix technologiques et du contexte du projet.

- Chaque modèle est présenté de façon indépendante mais dans la pratique, des concepts de plusieurs modèles peuvent être combinés entre eux.

  C'est ce que nous verrons dans la partie concernant les choix technologiques.

Une autre vision de l'évolution de l'architecture logicielle …



source :
https://www.urbanisation-si.com/

Quiz 3a

https://docs.google.com/forms/d/e/1FAIpQLSfJNzWa12NKqJSUlpa_2EiADn5PjpT2QcQax5lIPbj4ZfNtUg/viewform?usp=dialog

Quiz 3b

https://docs.google.com/forms/d/e/1FAIpQLSfJNzWa12NKqJSUlpa_2EiADn5PjpT2QcQax5lIPbj4ZfNtUg/viewform?usp=dialog