



Architecture des SI : concepts, standard et nouvelles architectures

David THIBAU – 2022

david.thibau@gmail.com



Agenda

- **Introduction**

- Modèle d'un SI
- Historique et Standards

- **Support de persistance**

- Base de données relationnelles
- NoSQL
- Message Broker et Event-store

- **Client/Serveur 2-tiers**

- Caractéristiques, avantages et inconvénients

- **Application web n-tiers**

- Modèle, JavaEE
- Tiers de présentation et architecture MVC
- Interactions service métier
- Architecture de déploiements

- **Services Web**

- Webservices XML
- RestFUL JSON
- Design By Contract

- **Architecture micro-services**

- Modèle
- Services techniques
- Patterns et leurs relations

- **Clients externes**

- API Gateway et Gestionnaire d'API
- Front-end Javascript

- **Infrastructures de déploiement**

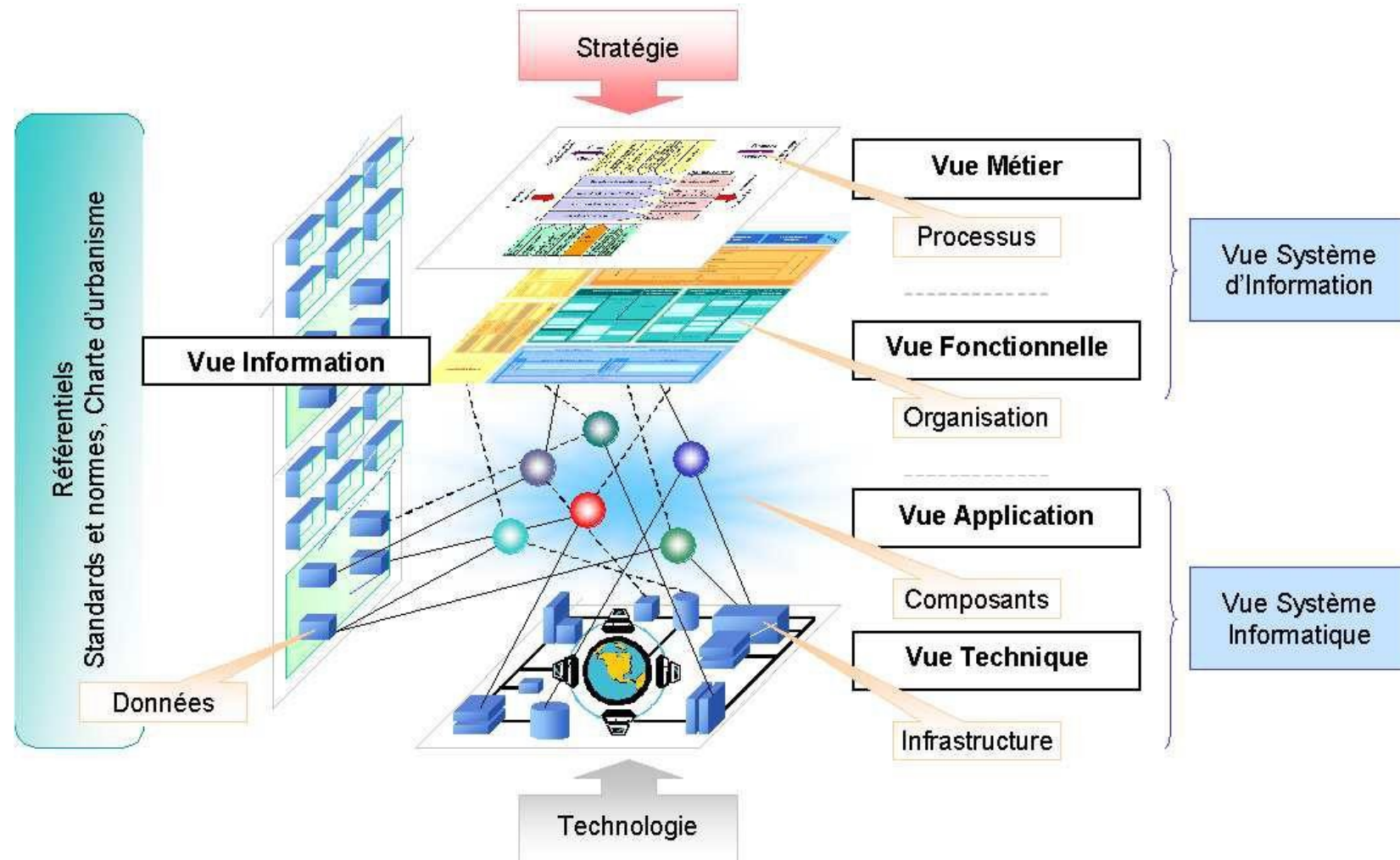
- Introduction
- Virtualisation
- Conteneurs et orchestrateur



Introduction

Architectures du SI Historique et standards

Architecture de SI





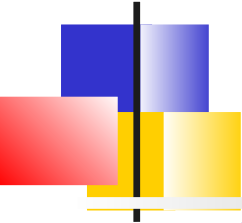
Architectures de déploiement

La distribution

Monolithe : Le logiciel est déployé sur une seule machine matérielle ou virtualisée

Distribuée : L'application est décomposée en différentes fonctions s'exécutant sur différentes machines ou nœud. Un nœud peut être utilisé dans plusieurs applications

Cluster : Un ensemble de nœuds est vu comme un seul. Les nœuds sont répliqués pour augmenter la puissance de calcul, le volume de stockage, la disponibilité



Architecture logicielle

Les responsabilités

Architecture en couche (tiers) : Permet une séparation des responsabilités, de définir des APIs permettant une meilleure évolutivité et la réutilisation (par exemple : réutiliser la couche métier)

Architecture distribuée :

- Modèle client/serveur : Un client demande un service
- Modèle pair-to-pair : Le client est également serveur et vice-versa
- Modèle consommateur/producteur : Un producteur produit un événement

Architecture orientée service : Les services définissent des contrats clairs, leur API. Ils peuvent être composés pour produire une application finale. Les services peuvent évoluer en toute indépendance. (SOA, micro-services)



Architecture logicielle

Les données

Architecture Data-centric : Centré autour du support persistant, les applications périphériques communiquent avec le composant central

Architecture en flot-de données : Plusieurs composants logiciels sont reliés entre eux par des flux de données, chaque composant effectue un traitement (transformation) sur les données. L'ensemble des traitements est souvent appelé pipeline (Ex traitement par lot et batch)

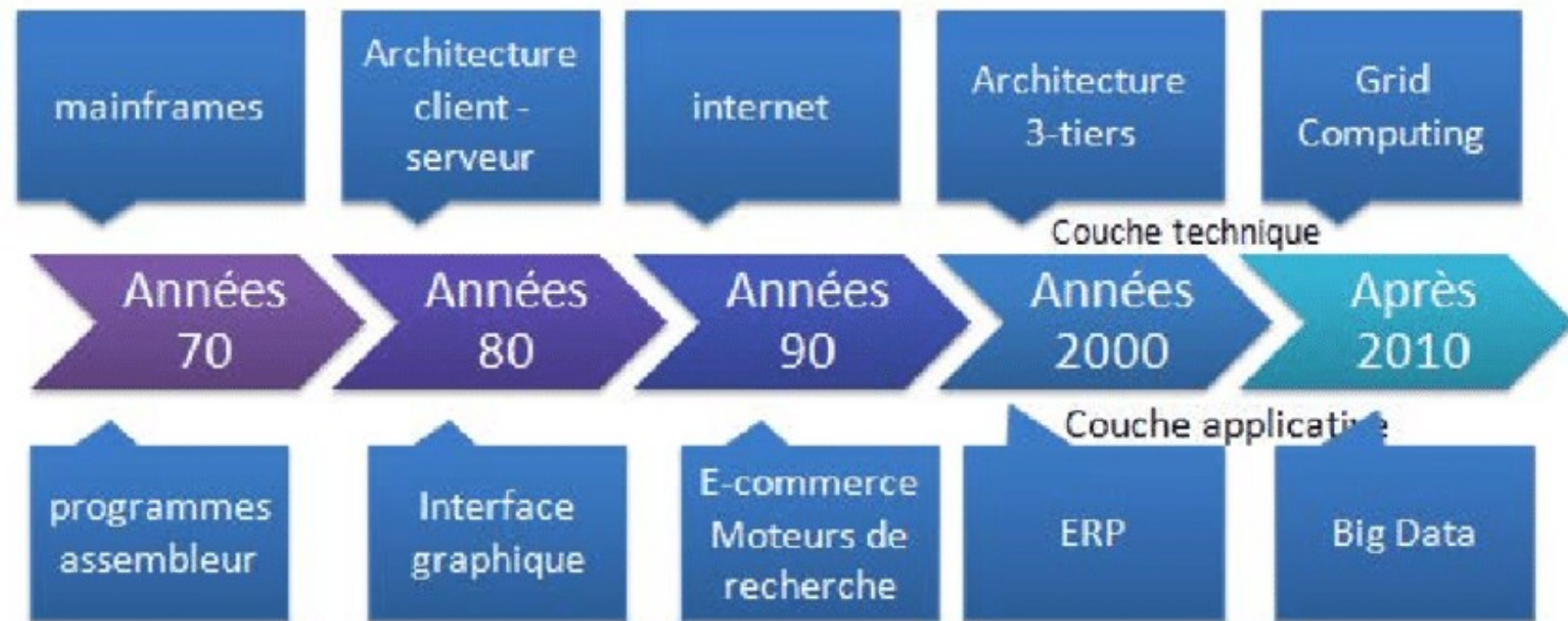
Architecture orientée événement : Sous-catégorie du flot de données. Un composant logiciel émet un événement, les intéressés réagissent et produisent également leurs événements. Centrée sur un middleware message bus



Introduction

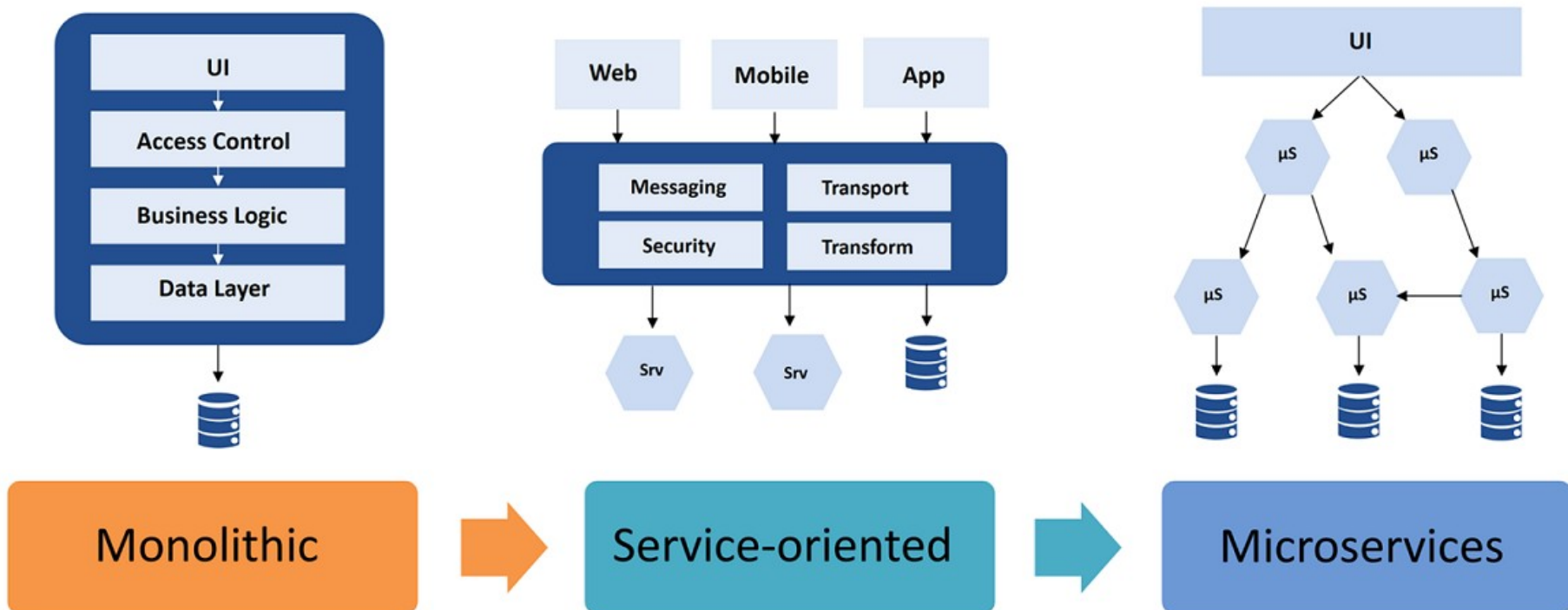
Architectures du SI
Historique et standards

Evolution → 2010



Actuellement

Evolution of Software Architectures





Standards

J2EE, JavaEE, Jakarta EE : Standard pour les applications d'entreprises n-tiers. Définit les APIs et services fournis par le serveur applicatif

WSDL, SOAP, UDDI : Standard des services web XML. Formats de description du service. Enveloppes de communication des messages (XML)

JMS, AMQP : Standards pour l'échange de messages

RestFul : Style architectural. (JSON, YAML)

OpenAPI : Format de description d'un service RestFul

Open Container Initiative : Format des containers et de leur plateforme d'exécution

https/SSL/TLS, OpenID, SAML oAuth2, JWT : Standard, protocoles relatifs à la sécurité



Supports de persistance

Introduction

Bases de données relationnelles

NoSQL

Message Broker et Event Store



Définition

La **persistance** signifie que l'état d'un système survit au processus qui l'a créé.

- Ceci est réalisé en stockant l'état sous forme de données dans un système de stockage (disque)
- Les programmes doivent pouvoir transférer des données vers et depuis les périphériques de stockage
- Ils doivent fournir des mappings entre les structures de données du langage de programmation natif et les structures de données du périphérique de stockage



Différentes approches pour la persistance

Réseau et hiérarchique : Stratégie navigationnelle (Ex : Système de fichiers/répertoires)

ISAM (Indexed Sequential Access Method) : Recherche directe en utilisant un **index** (Ex : Moteur de recherche)

Relationnel : Modèle dominant, structuration du métier, navigation via des relations

Objet : Stockage direct d'un Objet direct.
Ne se sont pas imposées

XML : Stockage de documents et fragments XML.
Souffre du manque de performance

Graph : Représentation en nœuds et relations. (Ex : Neo4J)

Document : Stockage de document JSON



Supports de persistance

Introduction

Bases de données relationnelles

NoSQL

Message Broker et Event Store



Les apports

Les bases de données relationnelles sont le support de persistance le plus répandu dans le SI.

Ses apports :

- **Performance et concurrence** : Accès rapide et simultanés pour l'écriture et la lecture pour de gros volumes de données (Par rapport à un système de fichier)
- **SQL** : Un langage standard pour les interactions
- **Modélisation des données métier**, de leurs relations, leurs contraintes d'intégrité
- **Transactions** : Propriétés ACID sur un groupe d'opérations de lecture/écriture



Acteurs du marché

Gratuit :

- MariaDB, Postgresql, ...
- HyperSonic, H2 (développement uniquement)

Hybride :

- MySQL

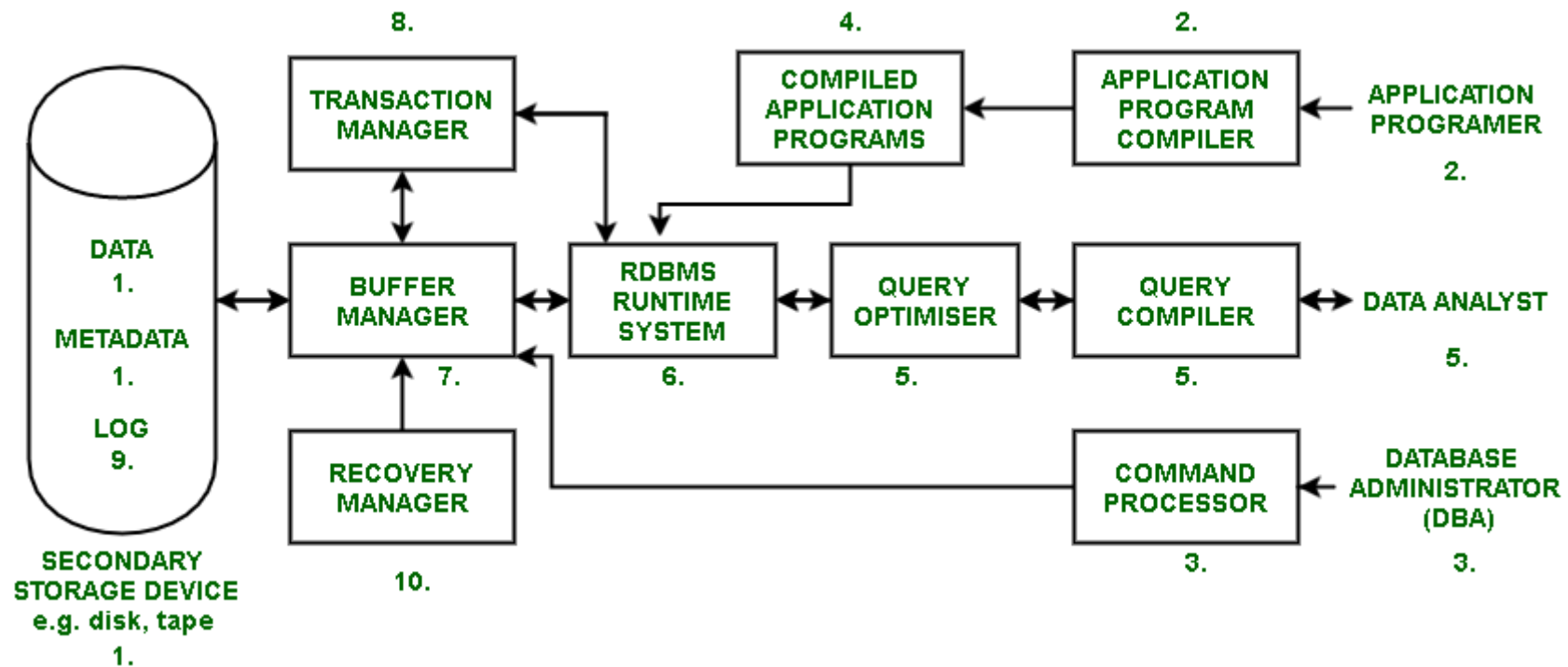
Commercial :

- Oracle, Microsoft Sql Server, DB2, ...

Cloud

- AWS, Azure, OpenStack, ...

Architecture interne

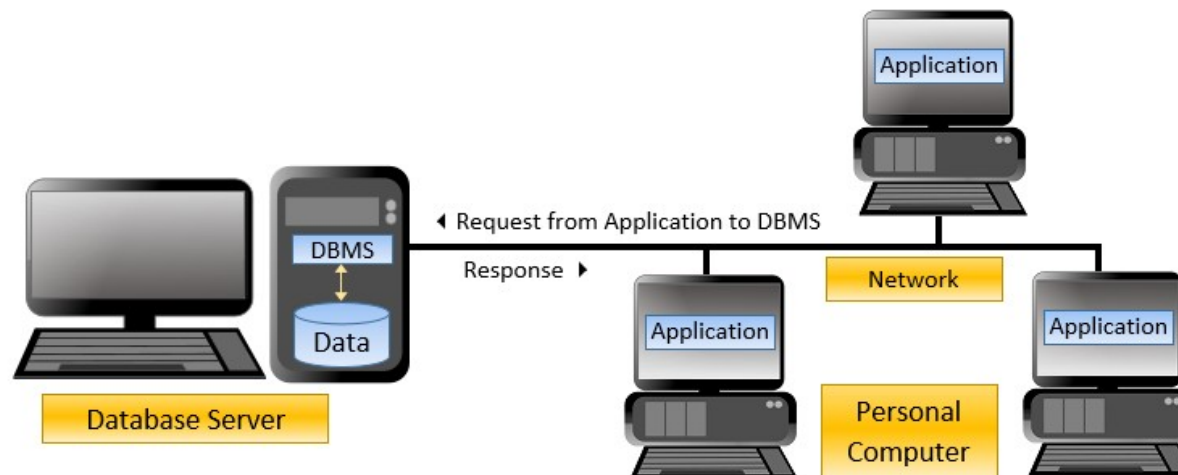


Usage client

Les SGDB définissent des utilisateurs auxquels correspondent des droits :

- Lecture, Écriture, Création de tables

Chaque client se connecte sous l'identité d'un compte utilisateur





SQL

SQL est le langage standard permettant :

- ✓ d'ajouter, modifier, supprimer, des données (**C**reate **R**ead **U**ppdate **D**eleate)
- ✓ de créer des schéma (**D**ata **D**efinition **L**anguage)

Exemples de requêtes SQL :

```
create table pilote  
  (matricule integer, nom char(20), prenom  
   char(20), salaire float)) ;
```

```
insert into pilote values  
(1012, 'Quiroule', 'Pierre', 25000) ;
```

```
insert into pilote values  
(2893, 'Moreau', 'Paul', 18000) ;
```

...

Extraire le matricule et le salaire des pilotes dont le nom commence par un « D » et le matricule est supérieur à 7000 :

```
select matricule, salaire from pilote where nom  
like 'D%' and matricule > 7000 ;
```

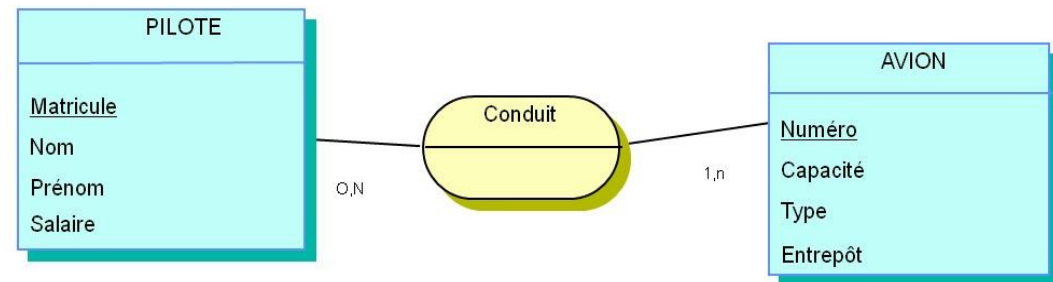
=>

Matricule	salaire
8736	20000

Modélisation des données métier

- SGBD: ensemble d'outils logiciels permettant la création et l'utilisation de bases de données.

- Fonctions:
 - Modèle conceptuel de données permettant d'échanger avec l'expert métier



- Génération modèle physique

- Génération contraintes d'intégrité

Table PILOTE			
Matricule	Nom	Prénom	Salaire
1012	Quiroule	Pierre	25000
2893	Moreau	Paul	18000
8736	Durand	Christian	20000
6392	Dupond	Philippe	23000

Table AVION			
Numéro	Capacité	Type	Entrepôt
715678	150	A320	Orly
871223	219	A310	Orly
920114	220	A310	Bourget
692003	150	A320	CDG

Table PILOTE_AVION	
Matricule	Numéro
1012	871223
1012	715678
6392	920114
6392	692003
8736	715678



Notion de Transaction

- ❖ Garantir l'intégrité des données au fil des modifications de ces dernières.
- ❖ Exemple : Transfert de fond du Compte A vers le Compte B
 - Modifications :
 - 1 : Debiter(CpteA, somme)
 - 2 : Crediter(CpteB, somme)
 - Intégrité
 - $\text{Solde}(\text{CpteA}) + \text{Solde}(\text{CpteB})$ est invariant



Propriétés ACID

Une transaction est une unité logique de travail sur les données qui respecte les propriétés suivantes :

- **Atomicité**
 - L'ensemble des modifications de la transaction aboutit ou aucune n'aboutit.
- **Cohérence**
 - La transaction amène les données d'un état cohérent à un autre état cohérent.
- **Isolation**
 - Les résultats d'une transaction ne sont pas visibles des autres transactions avant la fin de celle-ci
- **Durabilité**
 - Tous les résultats d'une transaction aboutie sont persistants (survivent à n'importe quel crash).



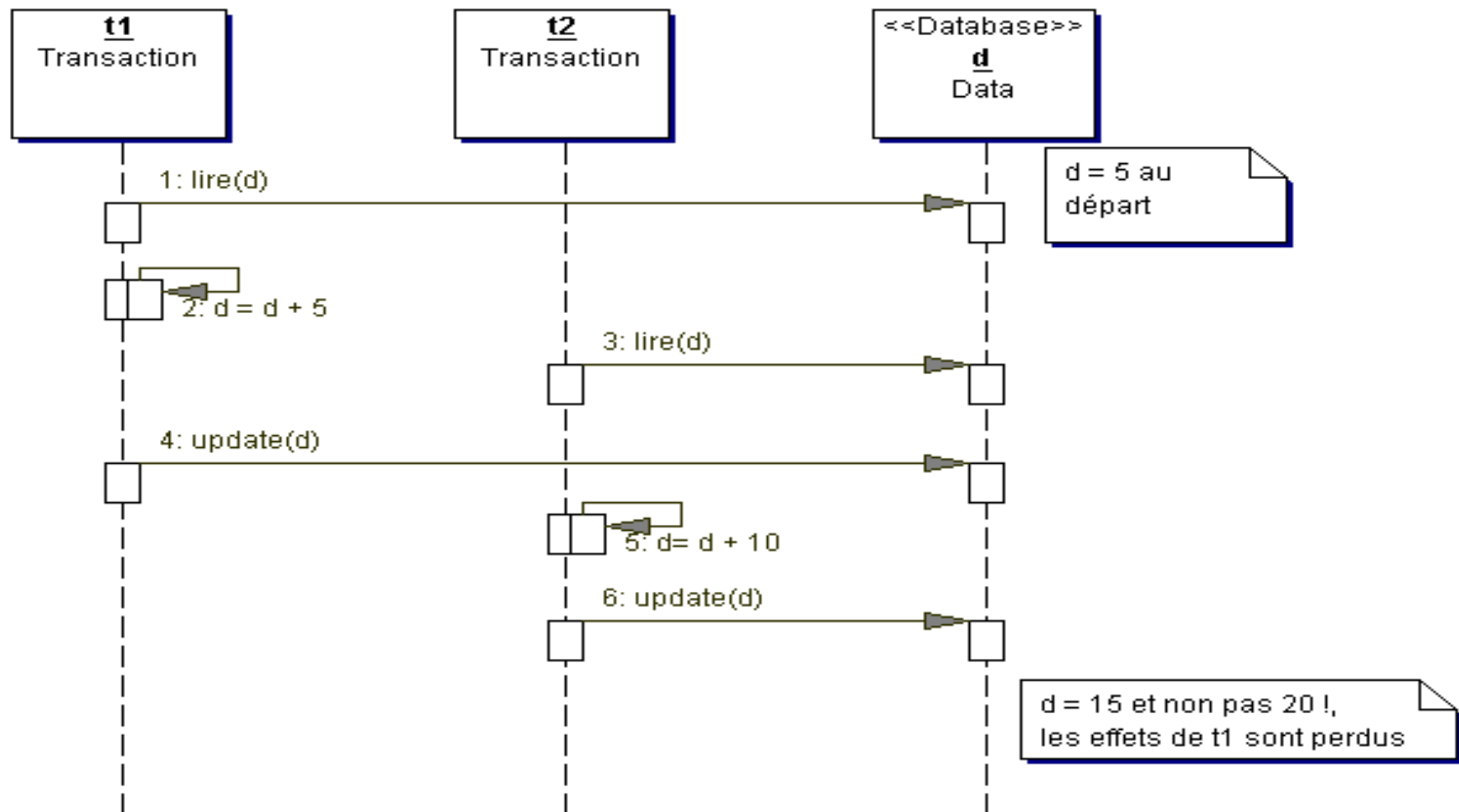
Transaction et Concurrency Problèmes

- Problèmes de concurrence
 - Surviennent quand plusieurs processus ou threads veulent accéder aux mêmes données.
 - Cas typique des applications Web
- Des problèmes peuvent survenir¹
 - Perte de mise à jour
 - Lecture de données non-validées
 - Lecture fantôme, lire une données qui sera supprimée par une autre transaction
 - ...

1. <https://www.geeksforgeeks.org/concurrency-problems-in-dbms-transactions/>

Transaction et Concurrency

Perte de mise à jour



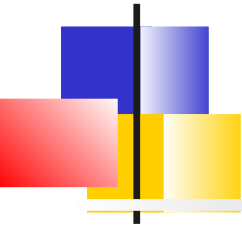
Niveaux d'isolation

Définitions

- Les BD proposent des verrous (R/W) pour contrer les problèmes liés à la concurrence
- Les verrous imposent des **niveaux d'isolation** qui permettent de faire un compromis entre la réactivité du système et les risques encourus
- 4 niveaux sont définis
 - SERIALIZABLE
 - REPEATABLE-READ
 - READ-COMMITTED
 - READ-UNCOMMITTED

Niveaux d'isolation

Risques encourus

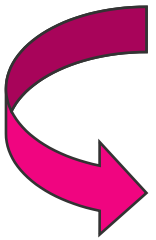


Niveau Isolation / erreurs de lecture	Dirty Read	Unrepeatable Read	Phantom
Read Uncommitted	O	O	O
Read Committed	N	O	O
Repeatable Read	N	N	O
Serializable	N	N	N



Relationnel et POO

- Les données des systèmes d'informations sont stockées majoritairement dans des SGBDR.
- Les applications utilisent de plus en plus le paradigme objet.

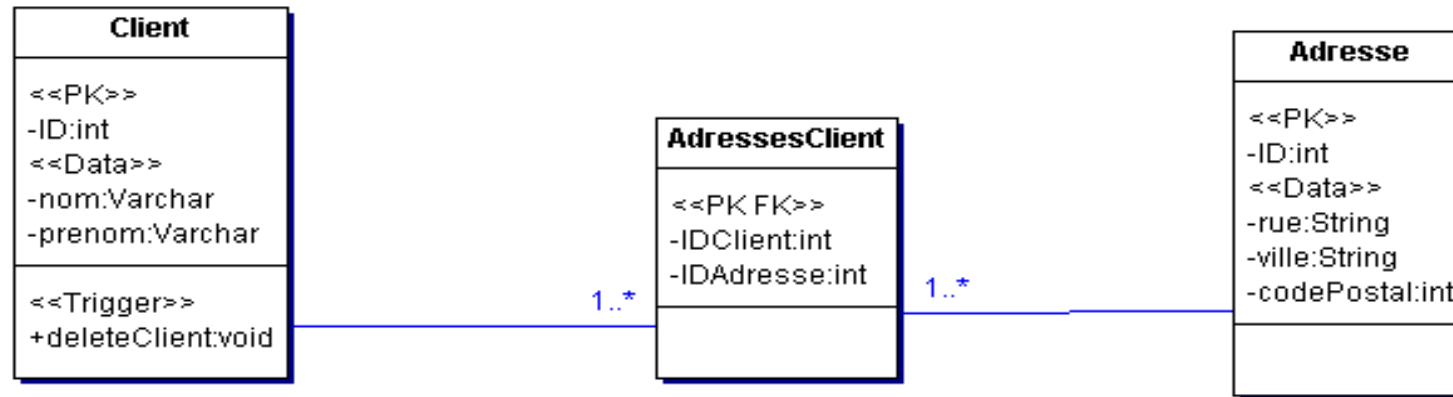


- Object / relational paradigm mismatch.
- D'ou la naissance des outils d'ORM, Hibernate, TopLink, ...

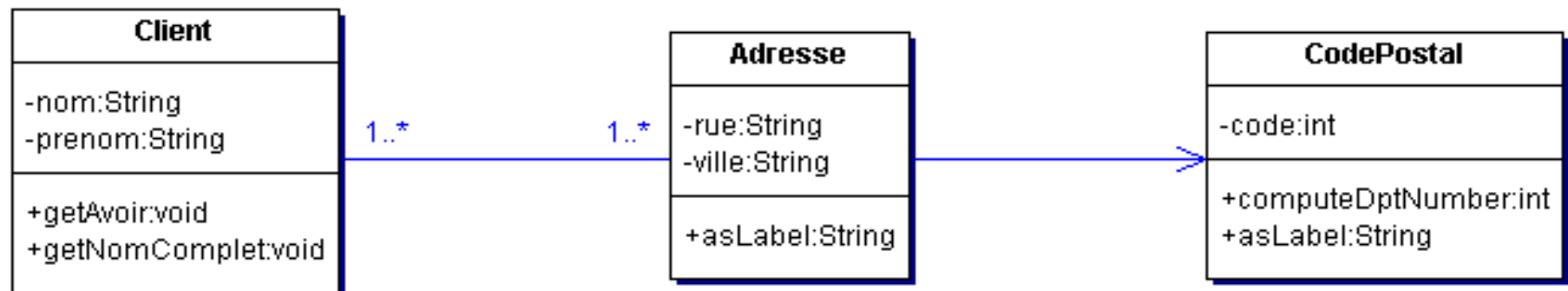
Impedence mismatch

Apparences trompeuses

Modèle de données

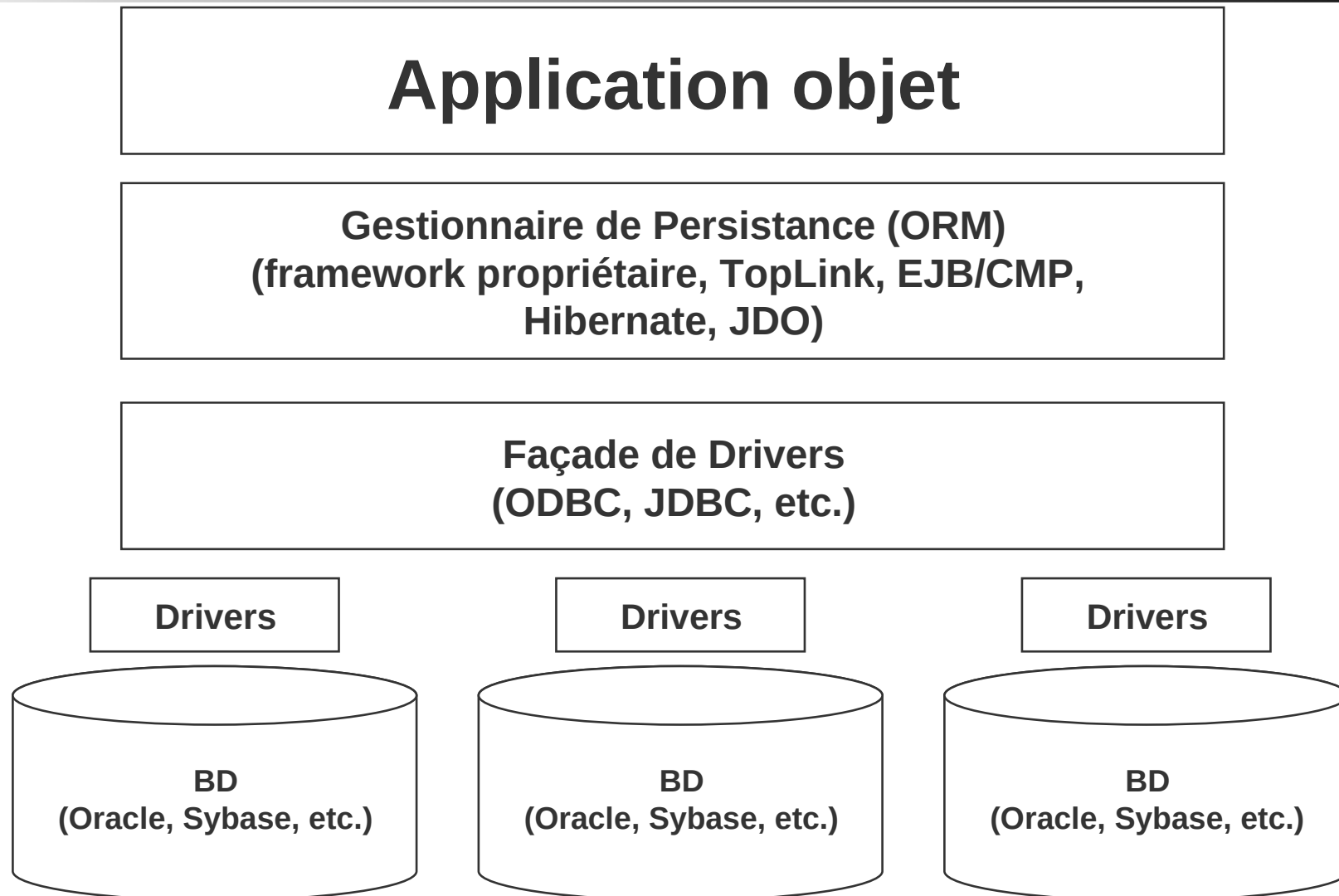


Modèle objet





Architectures de la persistance





Méta-données de la solution d'ORM

```
@Entity
@Table(name = "t_order")
public class Order {

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    private Instant date;

    private float discount;

    @Enumerated(EnumType.STRING)
    private OrderStatus status;

    @Embedded
    private PaymentInformation paymentInformation;

    @OneToMany(cascade = CascadeType.ALL, mappedBy = "order")
    List<OrderItem> orderItems = new ArrayList<>();

    @Transient
    public Float getAmount() {
        return orderItems.stream().map(i -> i.getPrice() * i.getQuantity()).reduce(0f, (a, b) -> a + b);
    }
}
```



Supports de persistance

Introduction

Bases de données relationnelles

NoSQL

Message Broker et Event Store



Introduction

NoSQL (Not Only Sql) est une alternative aux bases de données relationnelles

À l'origine, utilisé pour des bases de données géantes (Big Data, Google, Amazon.com, Facebook ou eBay¹), mais peut s'appliquer dans beaucoup de cas

Avantages :

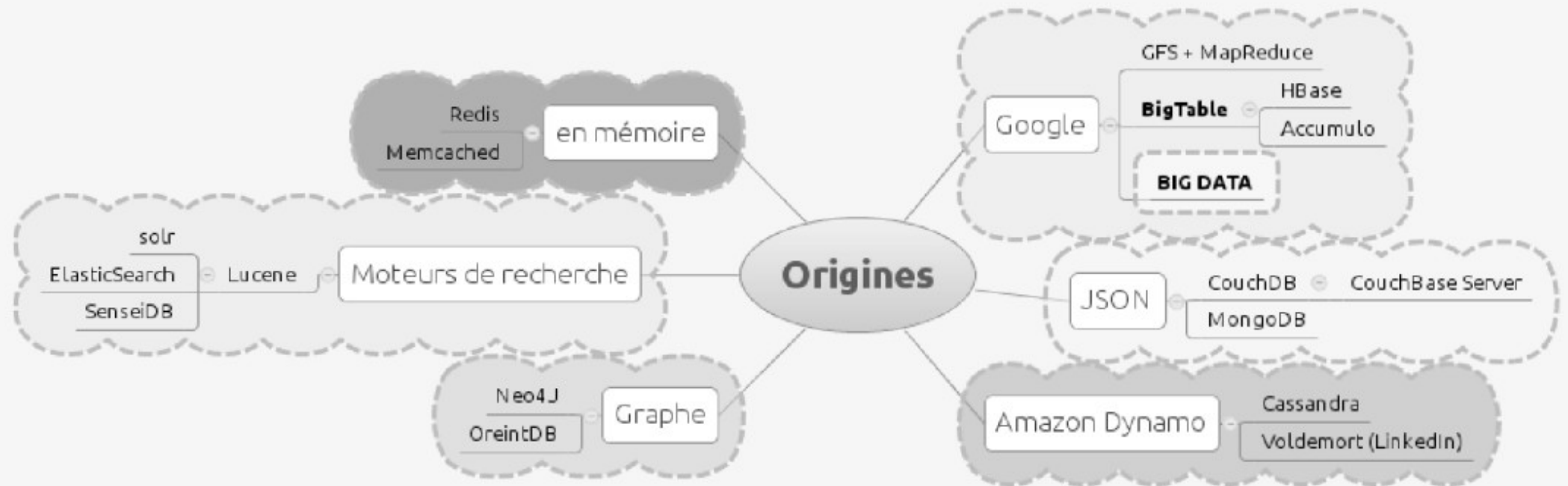
- Simplicité
- Structuration relationnelle faible

Inconvénients

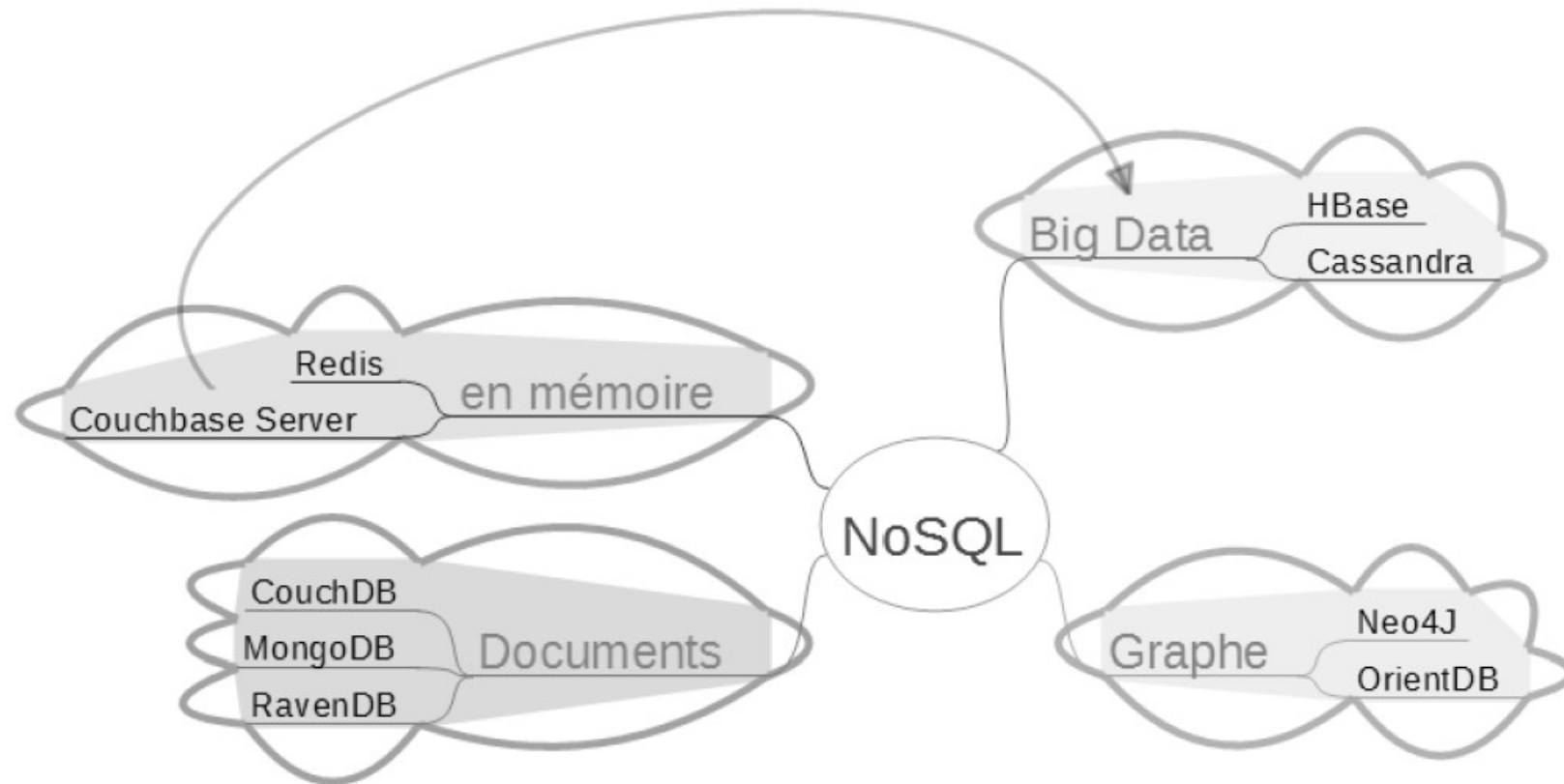
- Perte de performances compensée par la scalabilité et l'ajout de serveurs
- Pas de jointure => redondance de données

=> Applications typiques : stockage de document, analyse temps-réel, du stockage de logs (journaux), etc.

Origines du NoSQL



Les différents mondes du NoSQL





NoSQL distribué

Répondre au défi de la montée en charge

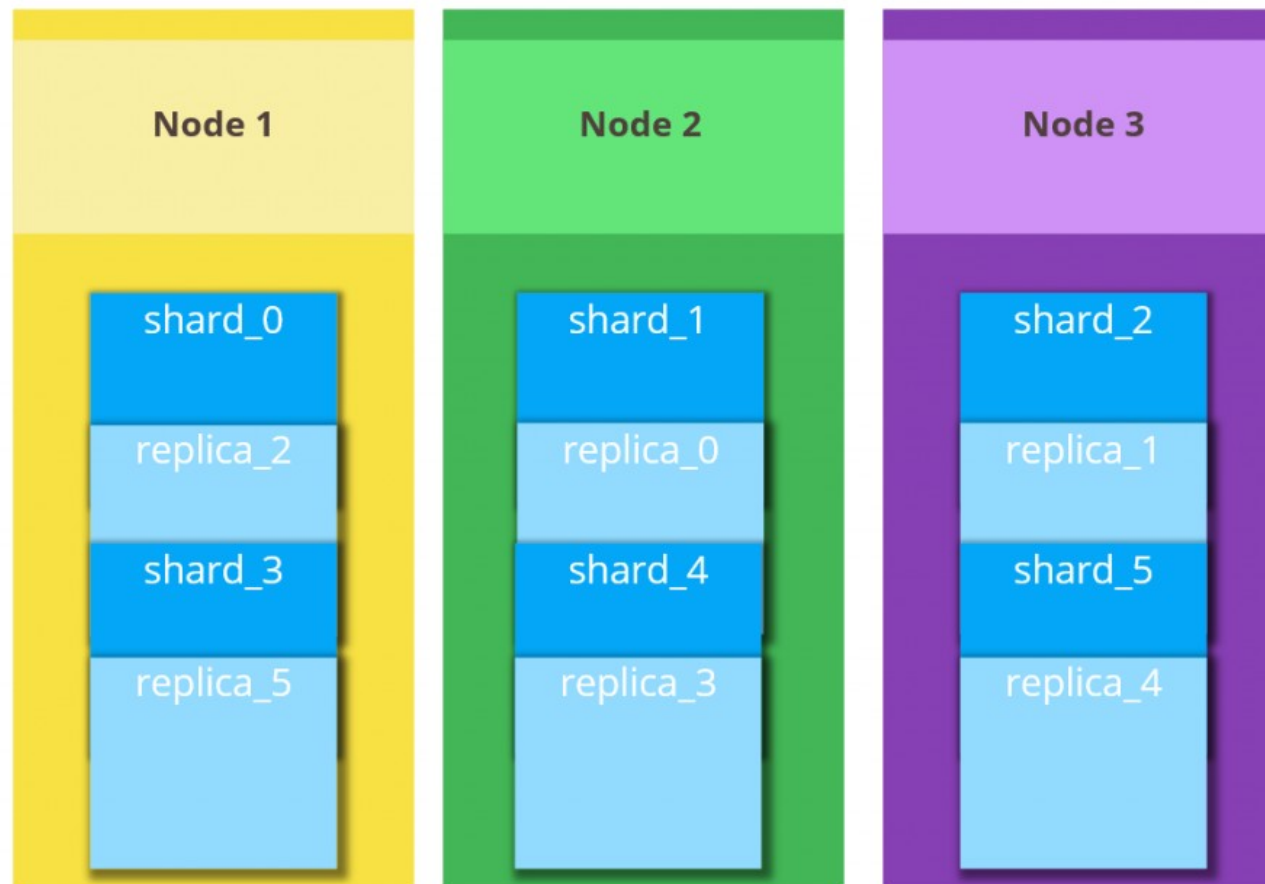
Les SGBDR ne sont pas conçus pour la scalabilité

Avec NoSQL :

- **Scale up** : Augmenter la puissance de traitement en répliquant les processus serveurs
- **Scale-out** : Augmenter la capacité de stockage en distribuant les données sur plusieurs disques

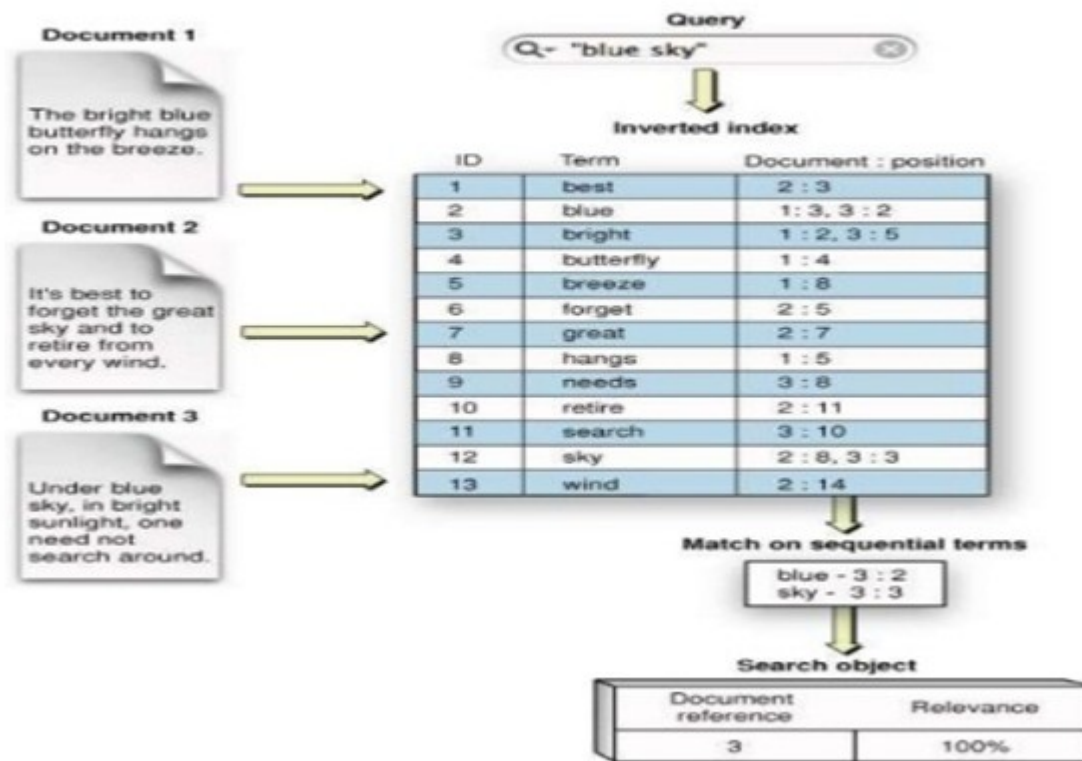
Scalabilité et tolérance aux pannes ElasticSearch

Elasticsearch Cluster



Index inversé

Lucene Internals - Inverted Index





Exemple requêtes

// MongoDB

```
db.zips.aggregate({$match:{state:"CA"}},{ $group:
  {_id:"$city",population:{ $sum:"$pop"} }},{ $sort:{population:-1}})
```

// ELS

```
GET demandes*/_search
```

```
{ "query": { "match": { "adresse": "belleville" } } }
```

// Neo4J

```
MATCH
```

```
  (neo:Database { nom:"Neo4j" } ),
```

```
  (thibaut:Personne { nom:"Thibau" } )
```

```
CREATE (thibau)-[:AMI]->(:Personne:Expert { nom:"Linda" } )-
  [:COMPETENCE]->(neo)
```



Supports de persistance

Introduction

Bases de données relationnelles

NoSQL

Message Broker et Event Store



Introduction

Messaging Pattern¹ : Un client invoque un service en utilisant une messagerie asynchrone

- Le pattern messaging fait souvent intervenir un message broker
- Un client effectue une requête en postant un message asynchrone
- Optionnellement, il s'attend à recevoir une réponse

1. <http://microservices.io/patterns/communication-style/messaging.html>



Avantages attendus

Un message broker permet de découpler le client du serveur (producteur/consommateur).

- Pas nécessaire que le récepteur soit en cours d'exécution lors de l'envoi du message
- Dépendance uniquement sur le format du message

Ce faible couplage favorise la montée en charge et l'évolutivité de l'architecture.

- Modification du traitement du message
- Scalabilité des récepteurs
- Ajout de nouveaux destinataires
- Ajout d'information dans le message



Le message broker

Un **message broker** est un agent intermédiaire qui transmet des messages entre un producteur et consommateur.

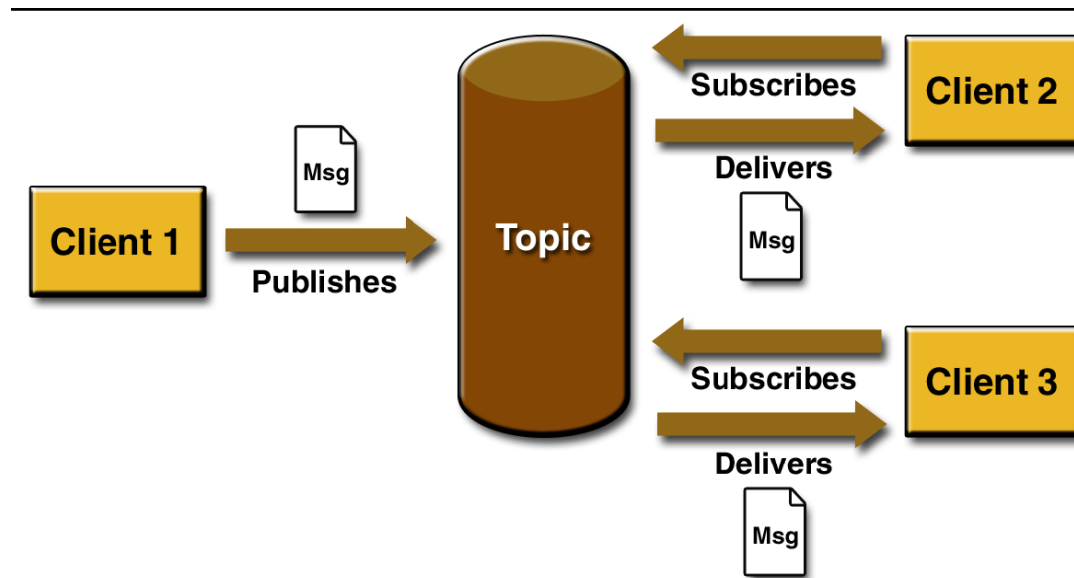
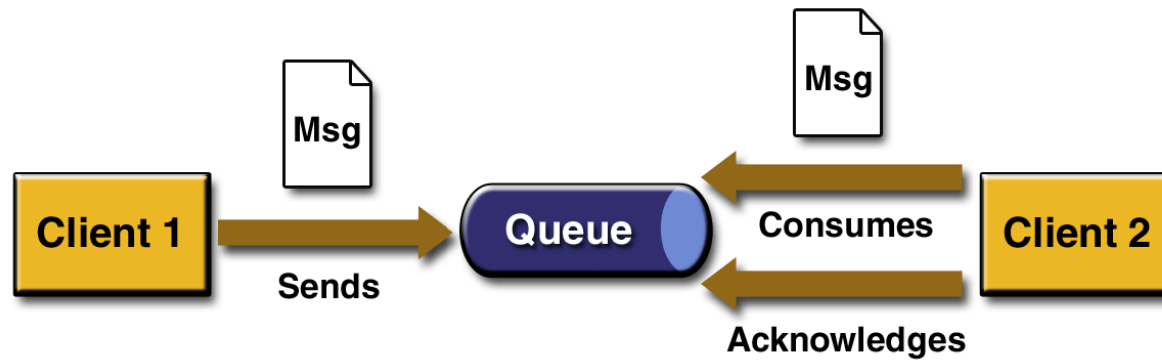
Il persiste une donnée (message)

- soit pendant le temps de sa transmission
- Soit jusqu'à une durée de péremption

Les données représentent généralement un événement

Les messages brokers sont utilisés pour l'intégration entre systèmes, les communications entre services/micro-services et récemment comme solution de stockage des événements métier

Modèles de communication





Garantie du Message broker

Les **message broker** permettent de garantir certaines propriétés :

- Garantie de livraison (At-Most-Once, At-Least-Once, Exactly-Once même en cas de failure)
- Transactions : Envois de plusieurs messages dans une seule transaction, Validation (commit) d'une réceptions
- Scalabilité : Montée en charge des consommateurs

De plus, différentes Qualité de Service peuvent être configurées :

- Dimensionnement (~capacité de bufferisation)
- Dead-letter (messages non délivrés)
- Ordre des messages, priorité
- Durée de vie



Standards

JavaEE a bien proposé le standard *JMS* mais cela ne permettait que d'intégrer du Java entre eux.

Les éditeurs se sont alors tournés vers d'autres spécifications comme **AMQP** (*Advanced Message Queuing Protocol*) qui n'ont pas d'exigence sur le langage

=> La flexibilité multilingue est devenue réelle pour les courtiers de messages open source.

D'autres protocoles existent : *STOMP*, *MQTT*



Produits

- ◆ AWS Simple Queue Service (SQS), Azure Service Bus
- ◆ Apache ActiveMQ, Fuse (Pro)
- ◆ HornetQ (RedHat), IBM MQ, Oracle Message Broker, TIBCO, WSO2
- ◆ Redis
- ◆ Apache Kafka

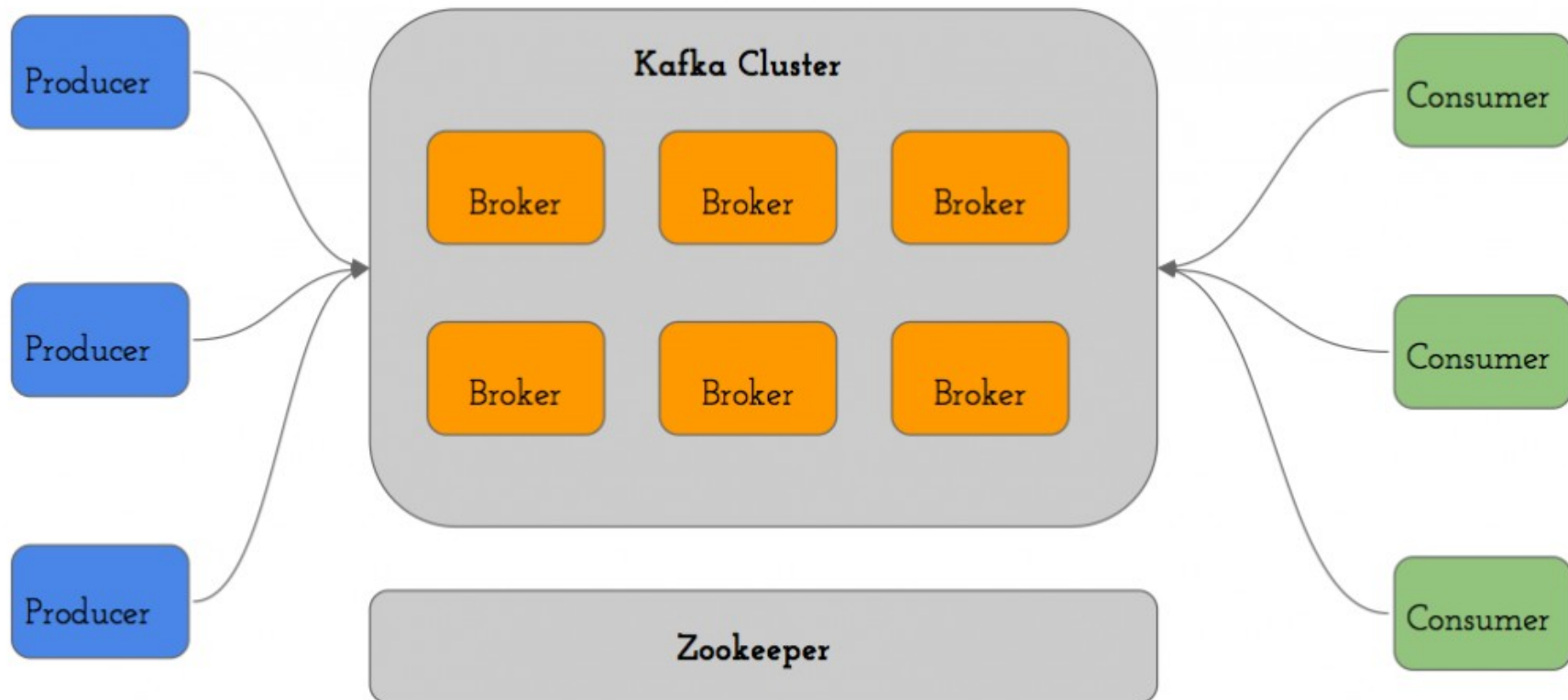


Spécificités de Kafka

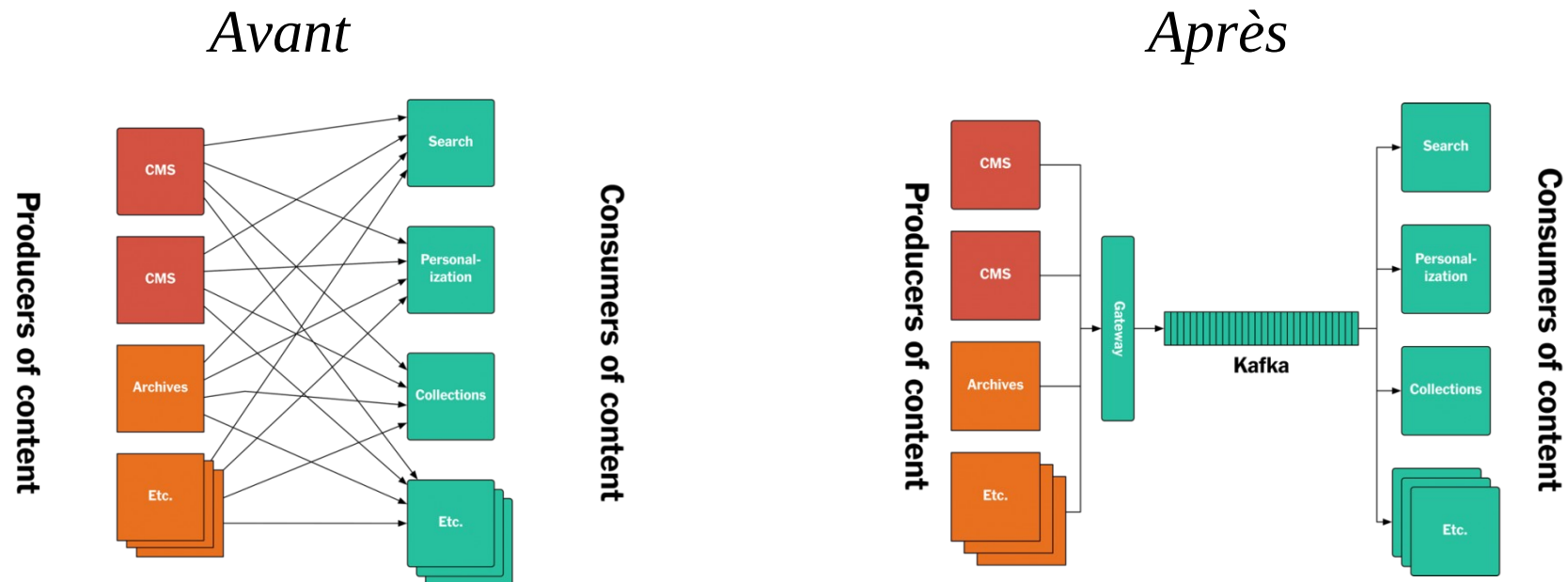
Kafka a réinventé le sujet en l'adaptant au
BigData :

- Ne supporte que le modèle Pub&Sub
- A la différence des autres brokers, les messages ne sont pas supprimés lorsqu'ils ont été reçus par leur destinataires mais après une date de péremption
=> Les messages peuvent donc être traités à posteriori et on peut donc rejouer un historique
- Taillé pour le BigData, il fonctionne en cluster et est capable de conserver et traiter un volume très important de données

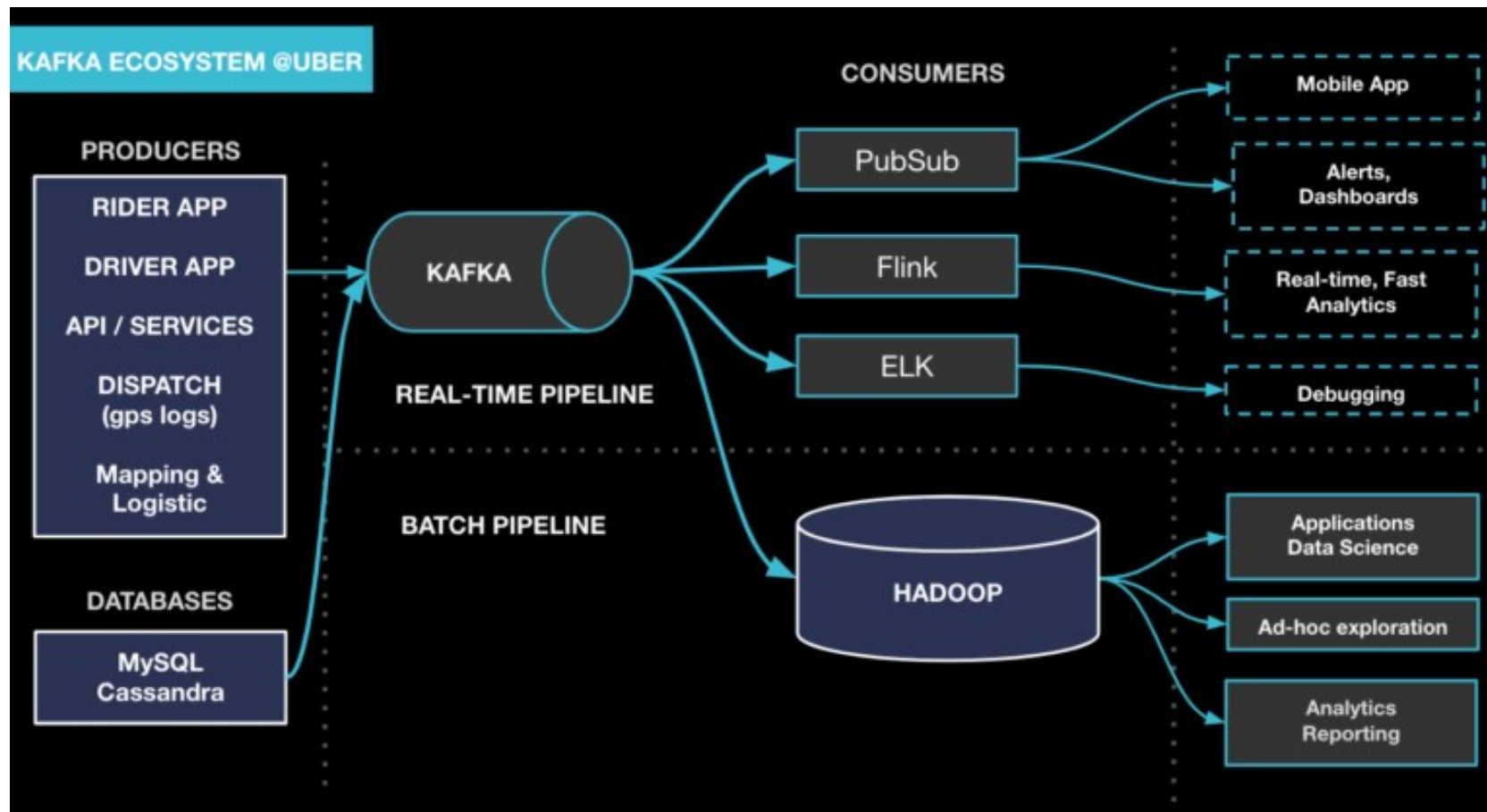
Kafka



Cas d'usage intégration : New York Times



Cas d'usage : traitement flux d'évènements





Kafka comme système de stockage

Les enregistrements sont écrits et répliqués sur le disque.

La structure de stockage est très scalable.

Kafka fonctionne de la même manière avec 50 Ko ou 50 To de données sur le serveur.

=> Kafka peut être considéré comme un système de stockage.

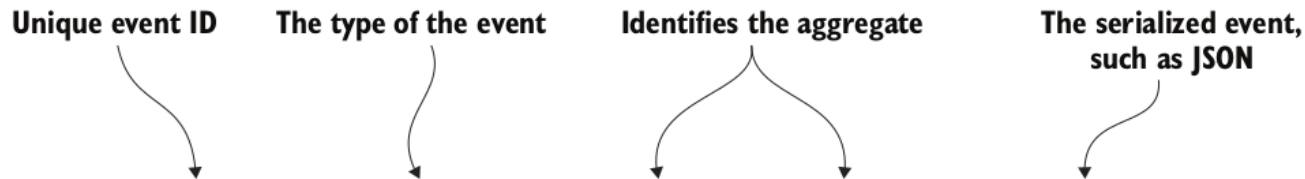
A la différence d'une BD, il stocke l'intégralité de l'historique des données plutôt qu'un simple instantané

Voir par exemple le projet *ksqlDB*
(<https://ksqldb.io/overview.html>)



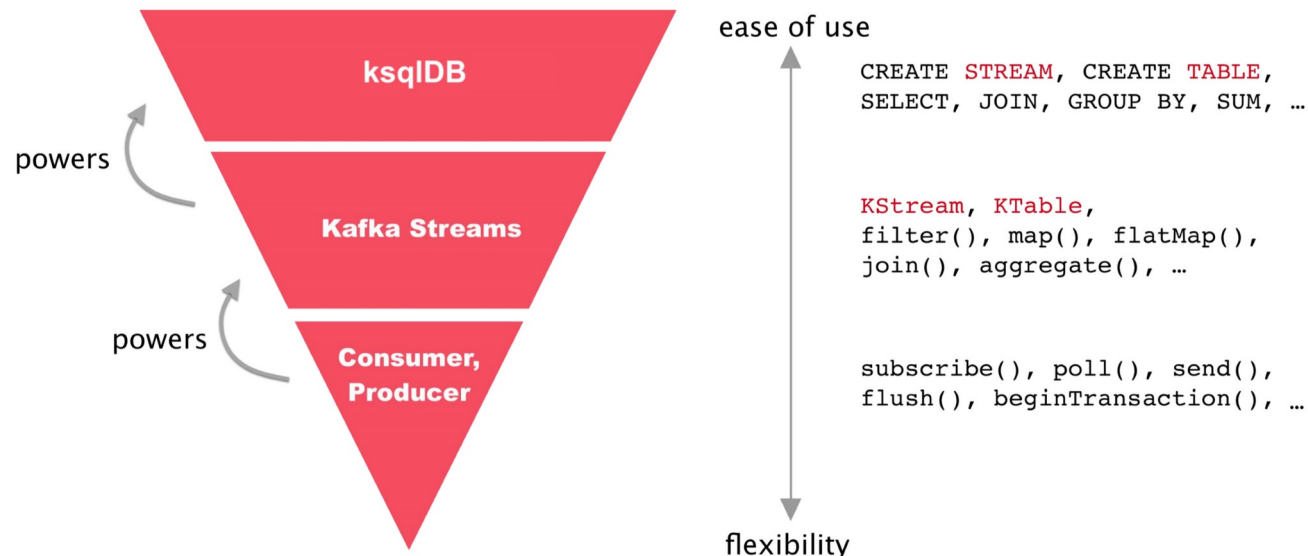
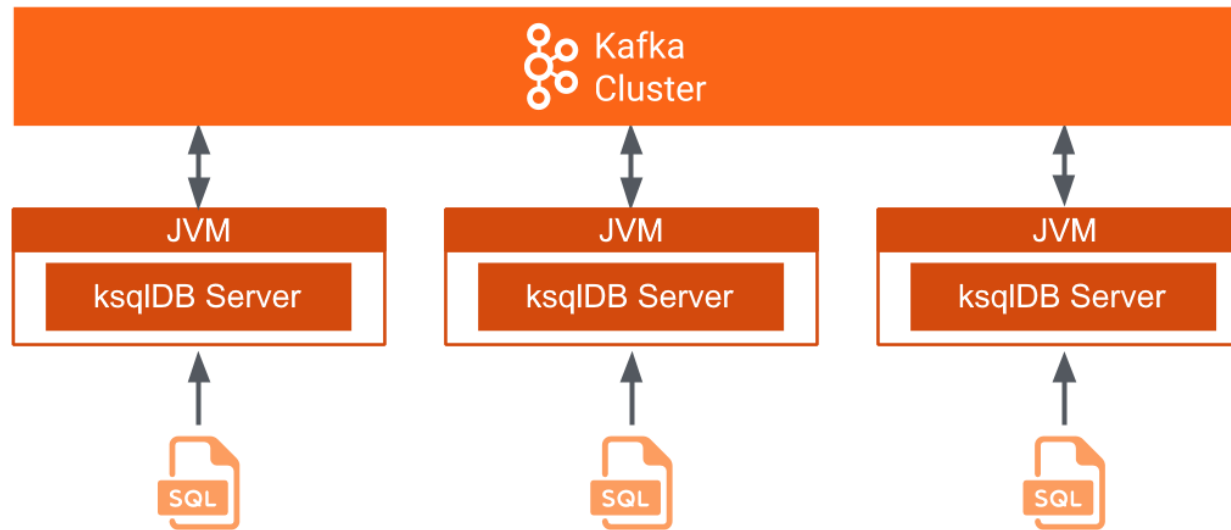
Event Store

Au lieu de stocker l'agrégat dans un schéma traditionnel classique, l'agrégat est stocké sous forme d'événements dans un ***EventStore***



event_id	event_type	entity_type	entity_id	event_data
102	Order Created	Order	101	{...}
103	Order Approved	Order	101	{...}
104	Order Shipped	Order	101	{...}
105	Order Delivered	Order	101	{...}
...

EVENTS table





Anciennes architectures

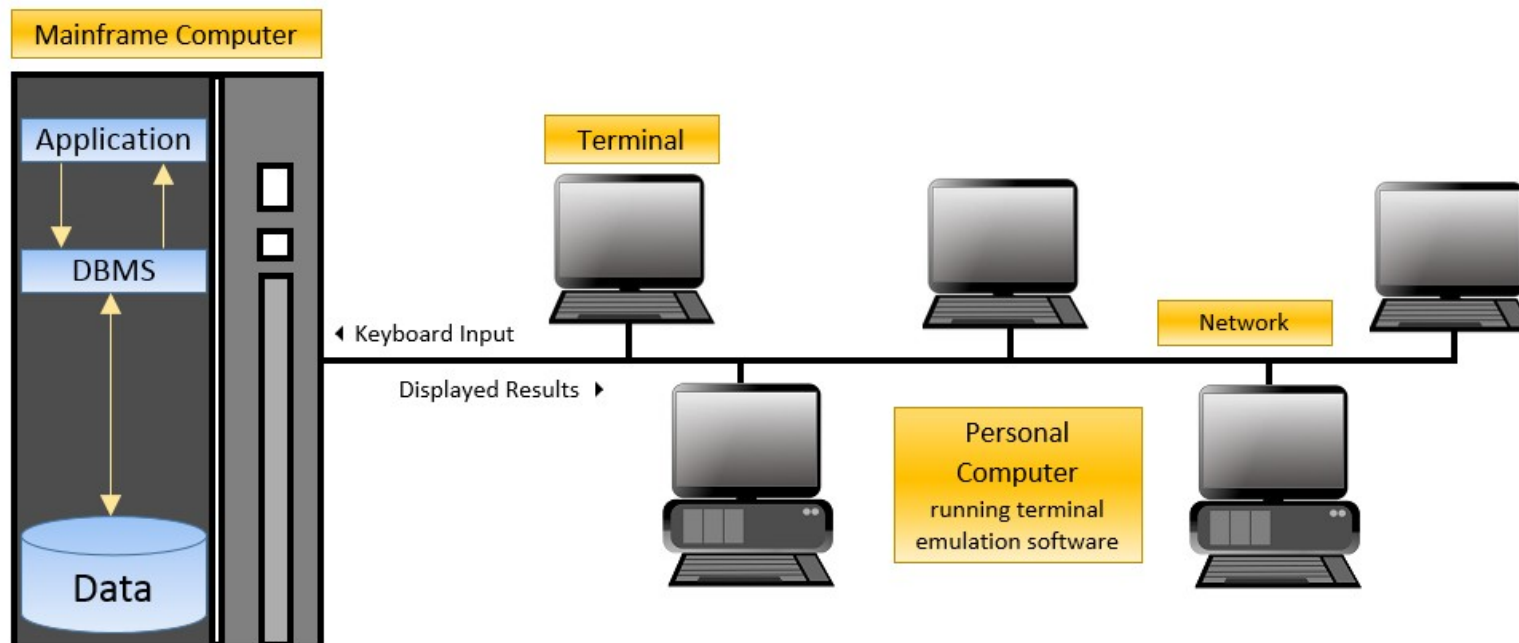
Mainframe et Client/Serveur

Caractéristiques, avantages et inconvénients

Mainframe et terminal passif

Les données et traitement sont centralisés sur la mainframe

Le clavier et l'écran sont déportés sur le terminal.





Avantages / inconvénients

Avantages :

- Traitement co-localisé au support de persistance => Rapidité
- Evolutivité simplifiée

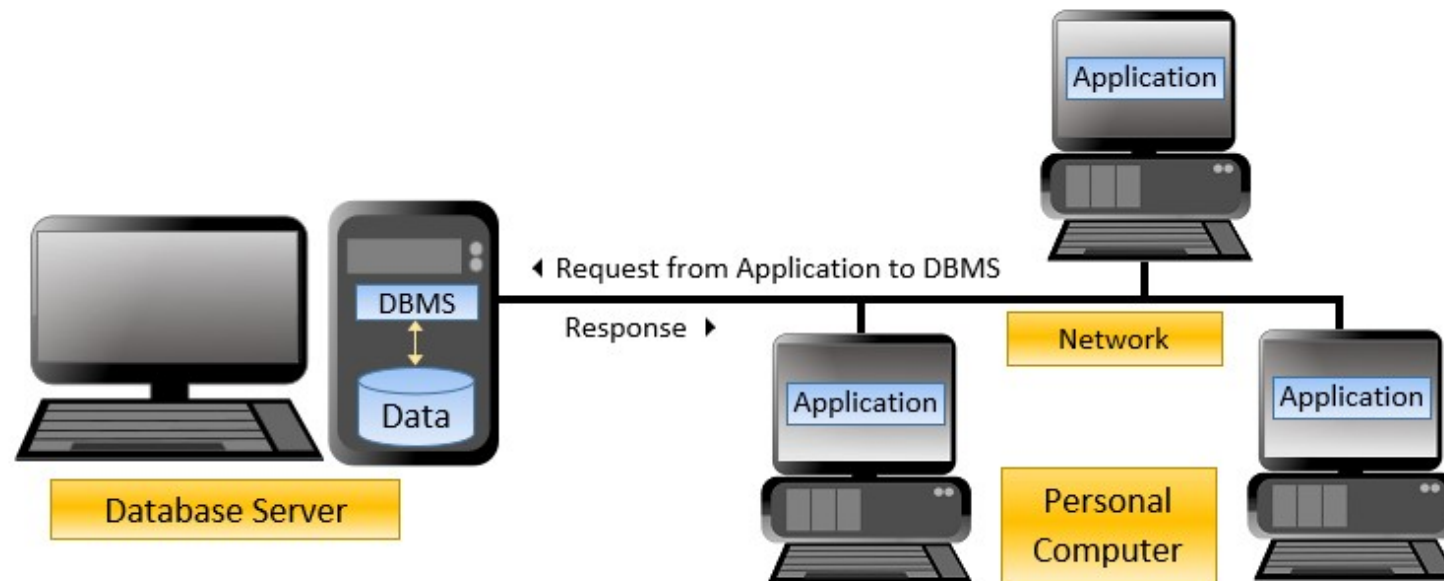
Inconvénients :

- Toute les demandes de puissance sont centralisés sur le mainframe
- Tout traitement nécessite un aller-retour réseau
- Les logiques de persistance, du métier et de la présentation sont entremêlés
- Compétence difficile à trouver

Client serveur ou 2-tiers

L'application est séparée des données, le transfert des données s'effectue par le réseau

Le serveur gère de multiples connexions simultanées





Avantages / inconvénients

Avantages :

- Fait bon usage de la puissance de traitement de plusieurs ordinateurs
- Gère de grandes quantités de données et de transactions
- Peut prendre en charge des milliers d'utilisateurs simultanés

Inconvénients :

- Le logiciel d'application est distribué, ce qui peut entraîner des problèmes de contrôle de version
- Le logiciel SGBD est relativement coûteux. Certains éditeurs facturant au nombre d'utilisateurs simultanés



Application Web n-tiers

Modèle JavaEE

Présentation et MVC
Appels services métier
Déploiements



Architecture N-tiers

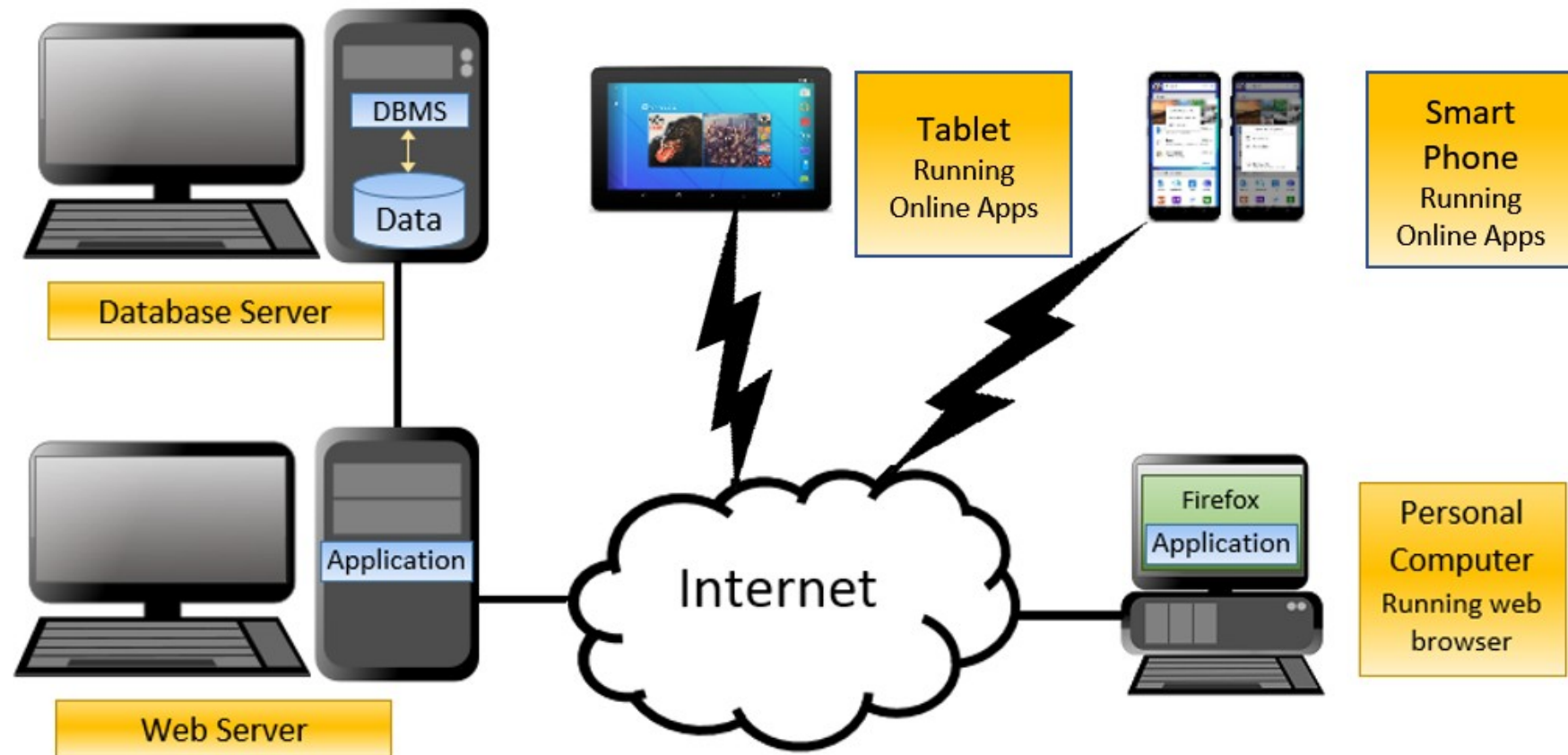
La logique applicative est divisée en composants selon leurs fonctions (les tiers)

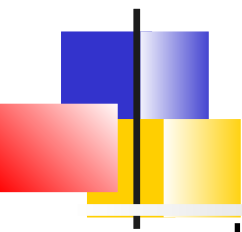
Les composants peuvent être installés sur des machines différentes

L'architecture la plus courante est une architecture 3-tiers : Les machines clientes, le serveur d'application et le support de persistance

=> Ce modèle étend les architectures précédentes client-serveur 2 tiers en plaçant un serveur d'application multi-threadé entre le client et le back-end de persistance.

Architecture 3-tiers





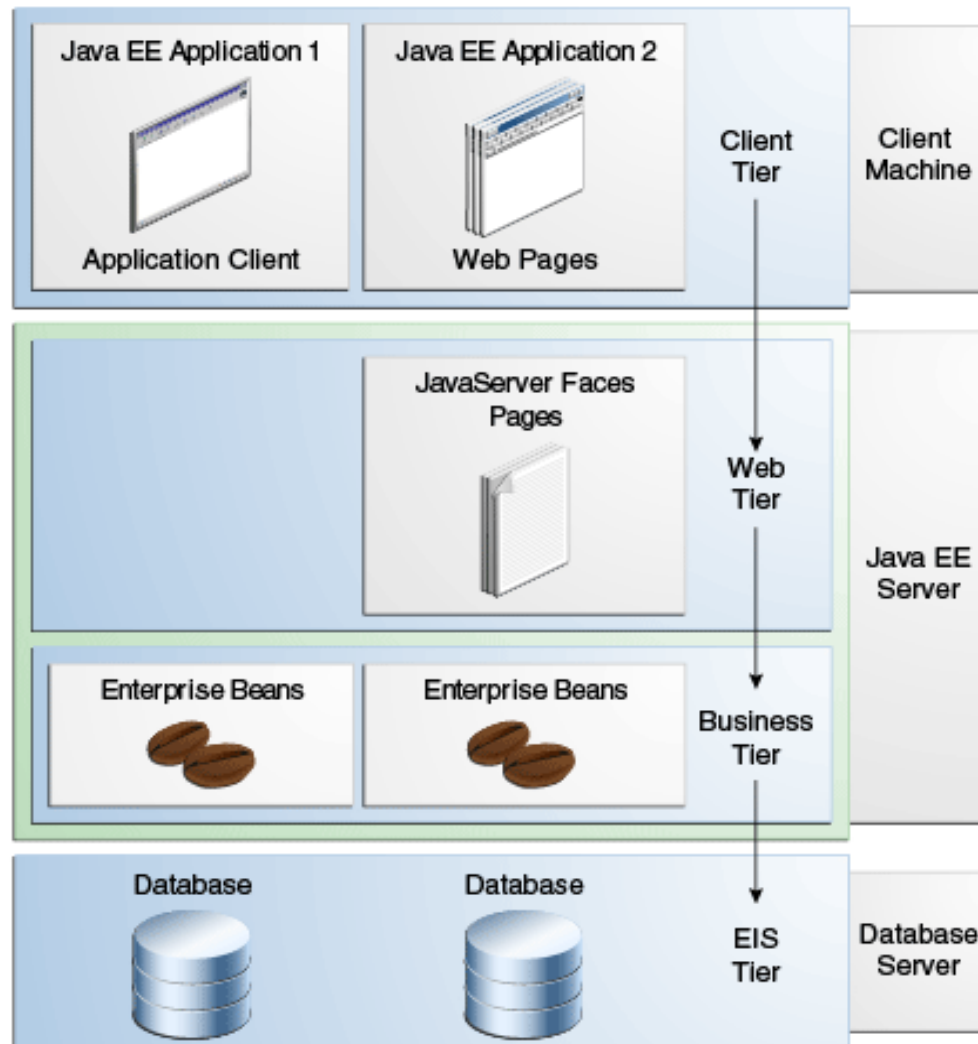
Plateforme de déploiement

La plateforme de déploiement est constitué de **serveurs applicatifs** multi-threadés qui offrent de nombreux services :

- Transactions distribuées (JTA)
- Sécurité (Authentification, Autorisation)
- Service de nommage (JNDI)
- Pools de threads, de connexions BD, d'EJBs
- Caches distribués
- MOM
- Injection de dépendance (CDI)
- Clustering

Des descripteurs de déploiement (XML, Annotation) sont packagés dans l'application pour configurer les services utilisés par les applications

Tiers Java EE





Contexte d'exécution des composants

- ❖ Avant qu'un composant puisse être exécuté, il est **assemblé** dans un module et **déployé** dans son container.
- ❖ 3 principaux formats sont fournis par la spéc :
 - **Enterprise Archive (.ear)** : archive qui contient d'autres modules qui forme une application
 - **Web Archive (.war)** : Module Web : contrôleurs et vues
 - **EJB Archive (.jar)** : Module EJ



Services configurables des containers

- ❖ L'assemblage de modules consiste à personnaliser pour chaque composant les services offerts
 - **Sécurité** : Les ressources applicatives ne sont accédées que par les utilisateurs autorisés
 - **Transaction** : Délimitation de transaction via l'annotation des méthodes métier
 - **Nommage de ressources** : Accès unifié aux ressources grâce à un système de nommage
 - **Connectivité distante** : Exécuter des composants distants de façon transparente



Modèles de programmation

Couche de présentation :

- Architecture MVC
- Langages de templating : JSF, Velocity, Freemaker, Thymeleaf

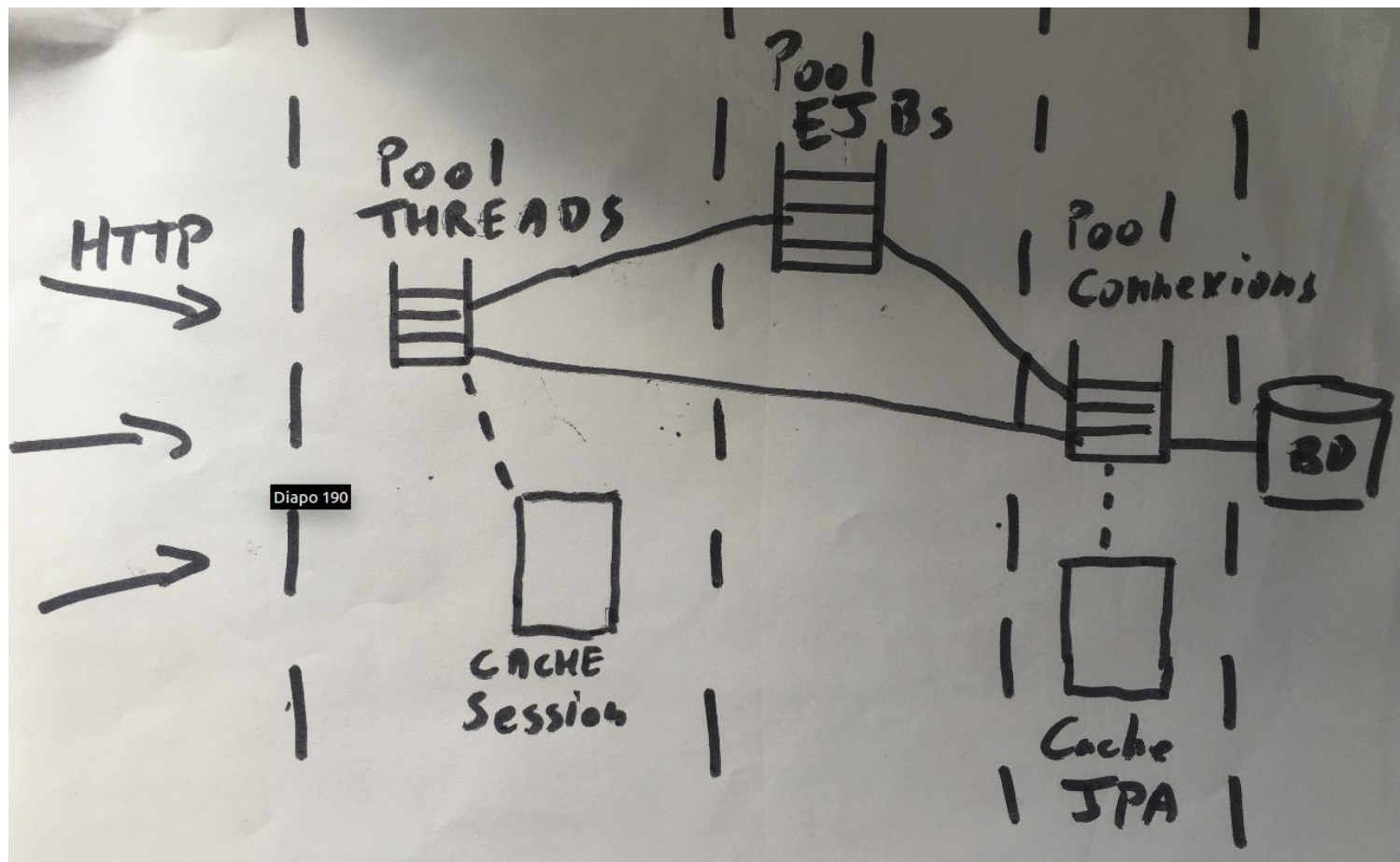
Couches métier :

- EJBs gérés via des pools avec méthodes transactionnelles
- Appel synchrone ou asynchrone via JMS

Couche de persistance :

- Outils d'ORM

Pools et Cache





Application Web n-tiers

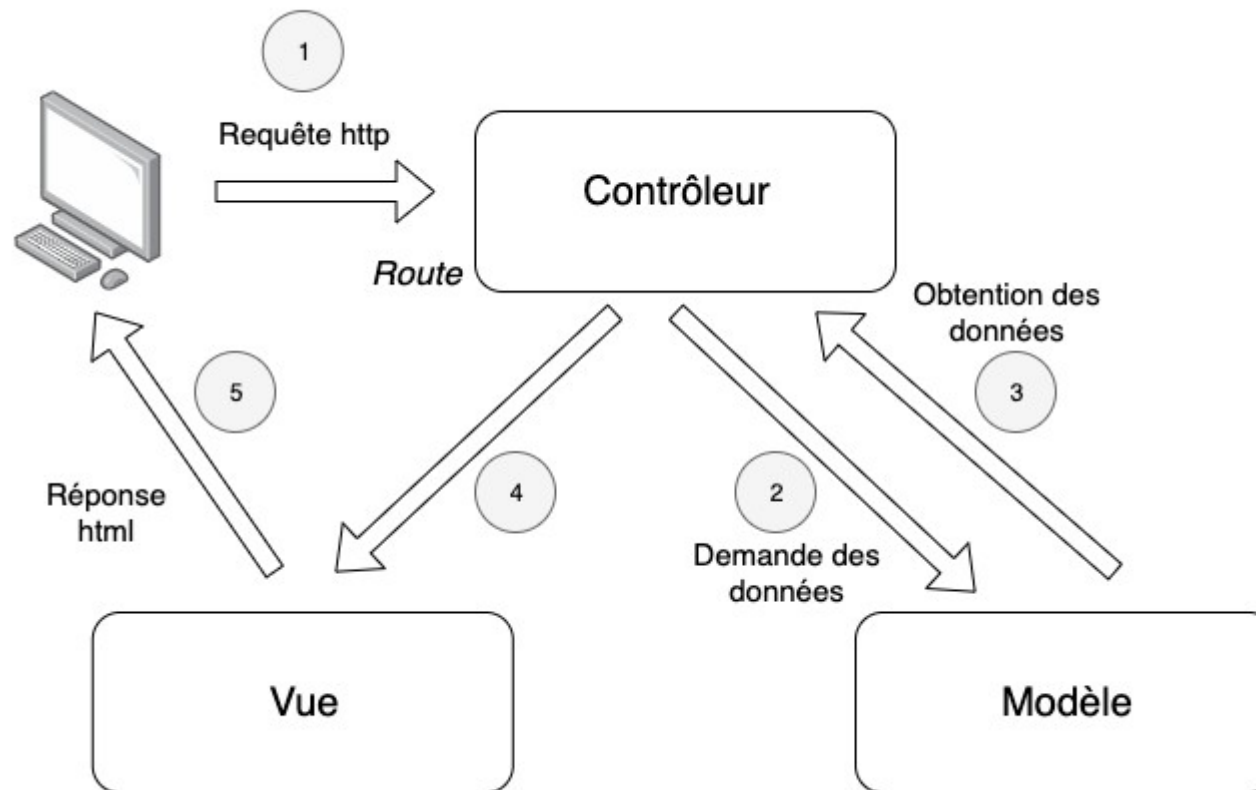
Modèle JavaEE

Présentation et MVC

Appels services métier

Déploiements

Pattern MVC





Frameworks Java

Frameworks (Main Contrôleur, conversion objet/String, validation)

- Struts
- JSF
- Spring MVC

Technologies de vue (template) :

- JSP (Struts, anciennement Spring MVC)
- Facelet (JSF)
- Velocity, Freemarker, Thymeleaf (Spring MVC)



Exemple Spring MVC

```
@Controller
@SessionAttributes("loggedUser")
public class MembersController {

    @Autowired protected MemberRepository memberRepository;

    @RequestMapping(path = "/register", method = RequestMethod.GET)
    public String showRegisterForm(Model model) {
        model.addAttribute("user", new Member());
        // Redirect to view register.jsp|html
        return "register";
    }

    @RequestMapping(path = "/register", method = RequestMethod.POST, consumes=Application.UTF-8)
    public String register(@Valid Member member, Model model) {
        member.setRegisteredDate(new Date());
        member = memberRepository.save(member);
        // Préparation du modèle utilisé par la vue
        model.addAttribute("loggedUser", member);

        // Envoi d'un ordre de redirection au navigateur
        return "redirect:documents";
    }
}
```




Template Thymeleaf

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:th="http://www.thymeleaf.org"
xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity3">
<head> <title>Liste des documents</title> </head>
<body>
  <div th:if="${session.loggedUser != null}">
    <h1>Your docs !</h1>
    <ul>
      <th:block>
        <li th:each="doc : *{documents}" th:text="${doc.name}"></li>
      </th:block>
    </ul>
  </div>
</body>
</html>
```



Application Web n-tiers

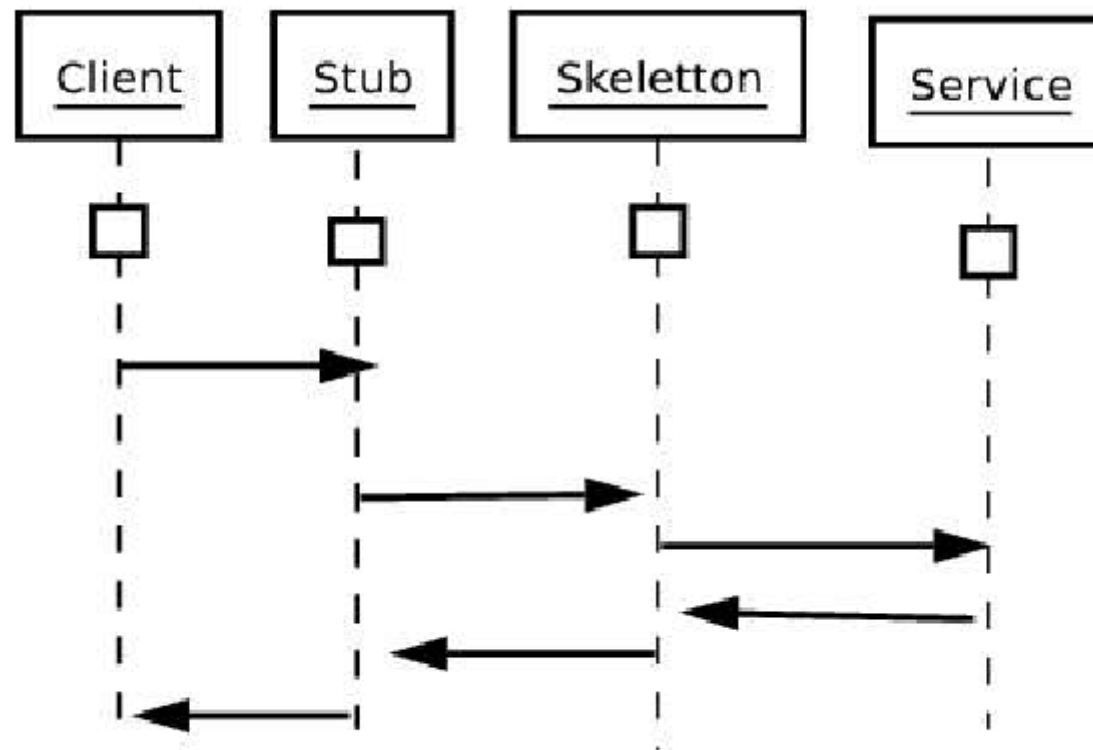
Modèle JavaEE
Présentation et MVC
Appels services métier
Déploiements



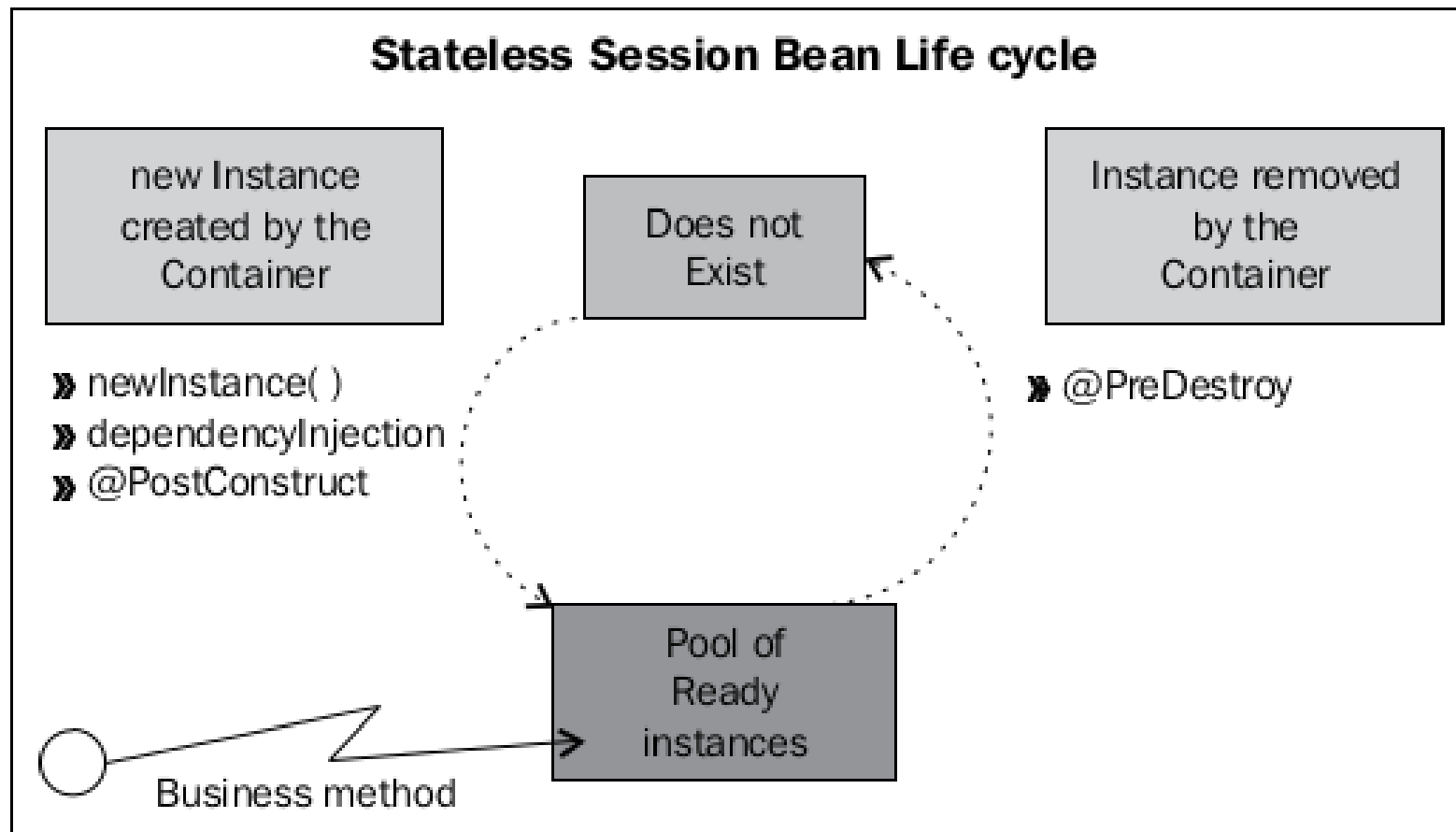
Composants métier

- ❖ **Enterprise Beans (EJB)**: Effectuent les traitements métier et interagissent avec la couche de persistance.
 - **Session beans** : Conversation avec un client.
Lorsque la session est terminée les données ne sont pas sauvegardées.
Mode singleton, stateless ou stateful
 - **Entity beans** : Représente un enregistrement dans la base de données. Le serveur Java EE peut alors s'occuper de la persistance
 - **Message-driven beans** : Réactif à des messages asynchrones JMS.

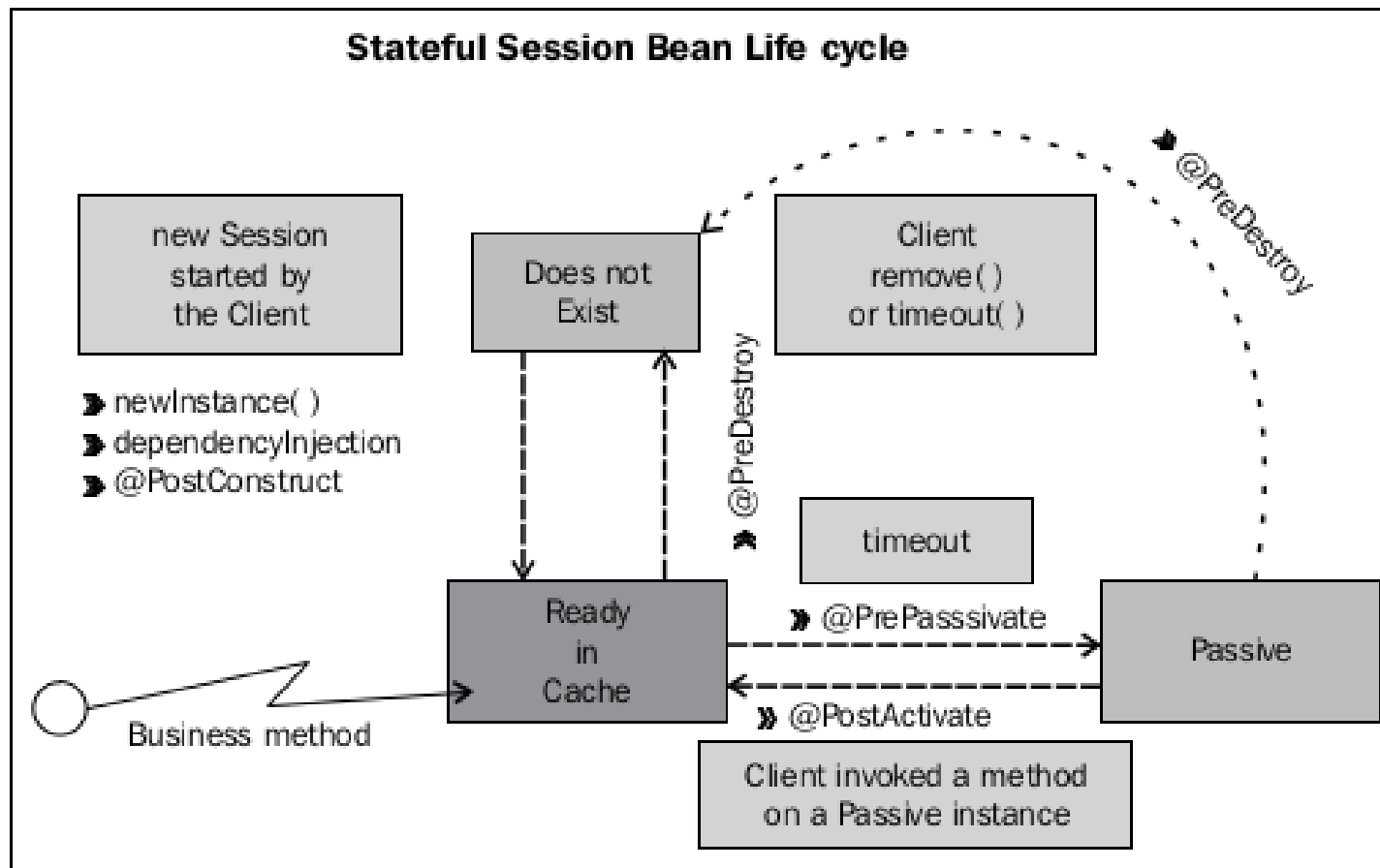
Appel EJBs distants



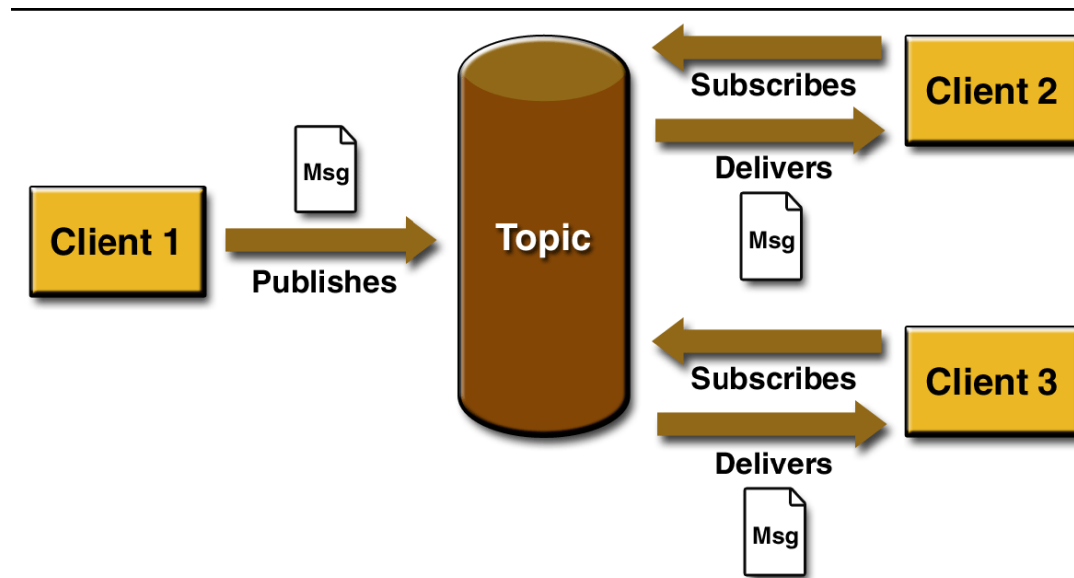
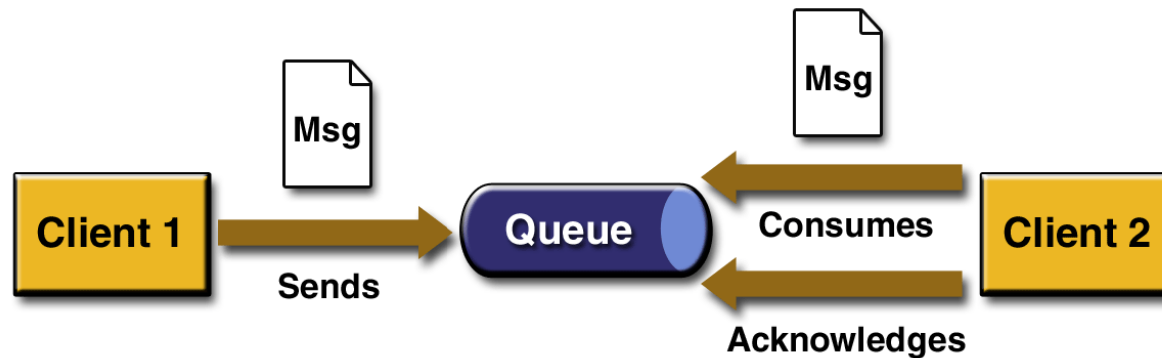
Cycle de vie d'un stateless



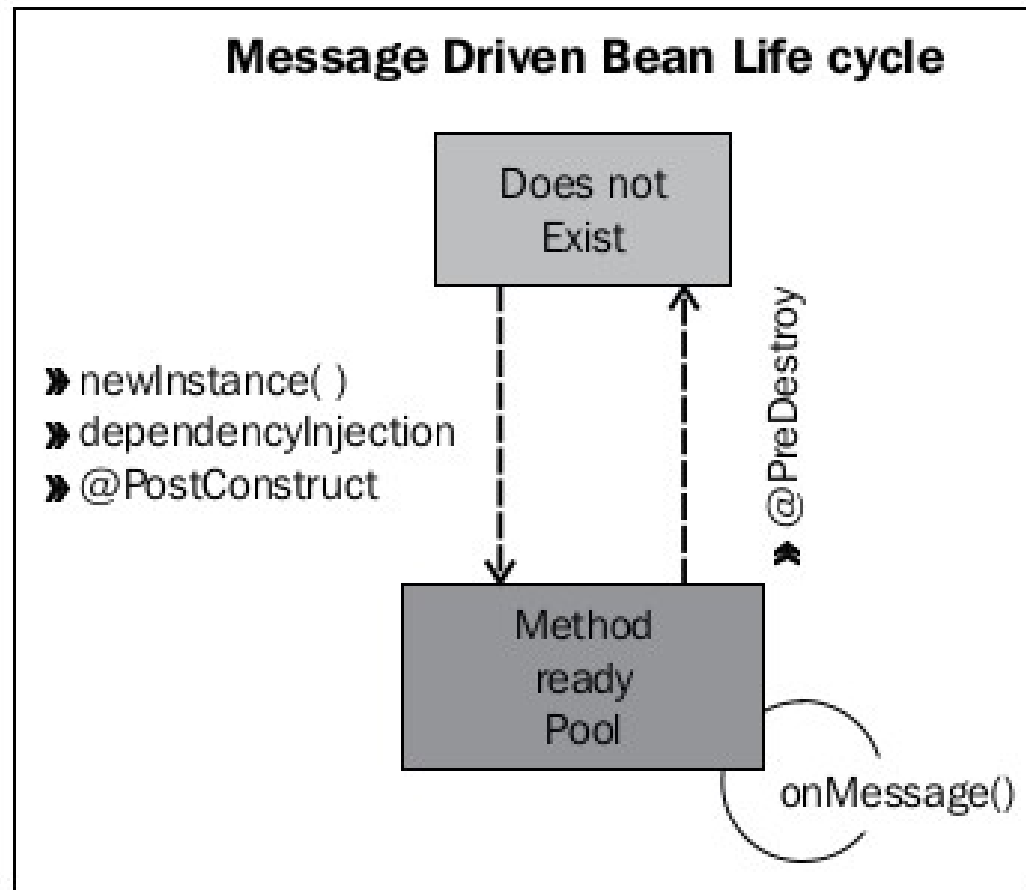
Cycle de vie d'un stateful



Asynchronisme avec JMS



Cycle de vie d'un MDB





Application sans EJB

Les EJBs n'ont jamais fait l'unanimité et de nombreux projets ont implémenté les services métier sans les EJBs

C'est le parti pris du framework Spring qui propose via des annotations les mêmes services techniques que les EJBs sans leur lourdeur, à savoir :

- La transaction
- La sécurité déclarative



Exemple : service transactionnel Spring

@Service

```
class UserManagementImpl implements UserManagement {  
  
    private final UserRepository userRepository;  
    private final RoleRepository roleRepository;  
  
    public UserManagementImpl(UserRepository userRepository,  
        RoleRepository roleRepository) {  
        this.userRepository = userRepository;  
        this.roleRepository = roleRepository;  
    }  
}
```

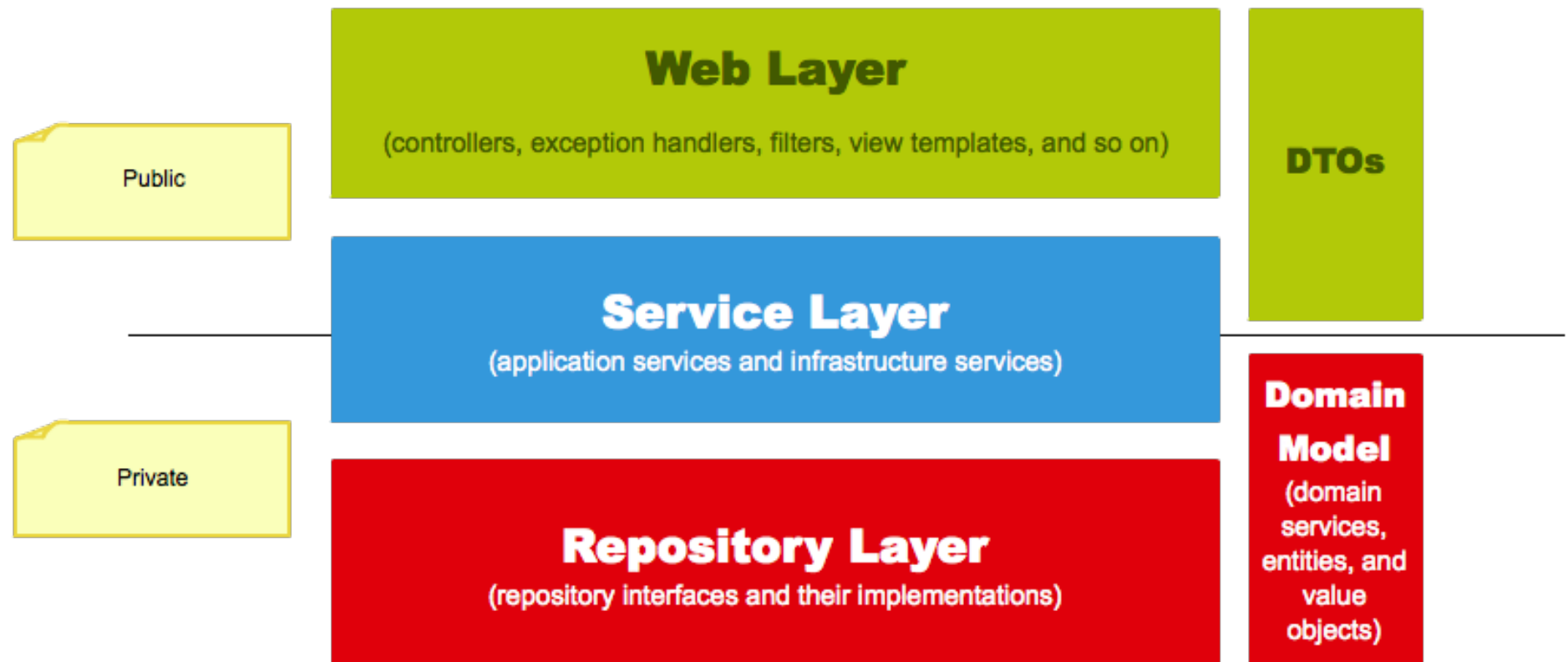
@Transactional

@RolesAllowed("ROLE_MANAGER")

```
public void addRoleToAllUsers(String roleName) {  
  
    Role role = roleRepository.findByName(roleName);  
  
    for (User user : userRepository.findAll()) {  
        user.addRole(role);  
        userRepository.save(user);  
    }  
}
```



Architecture classique Spring MVC + Data





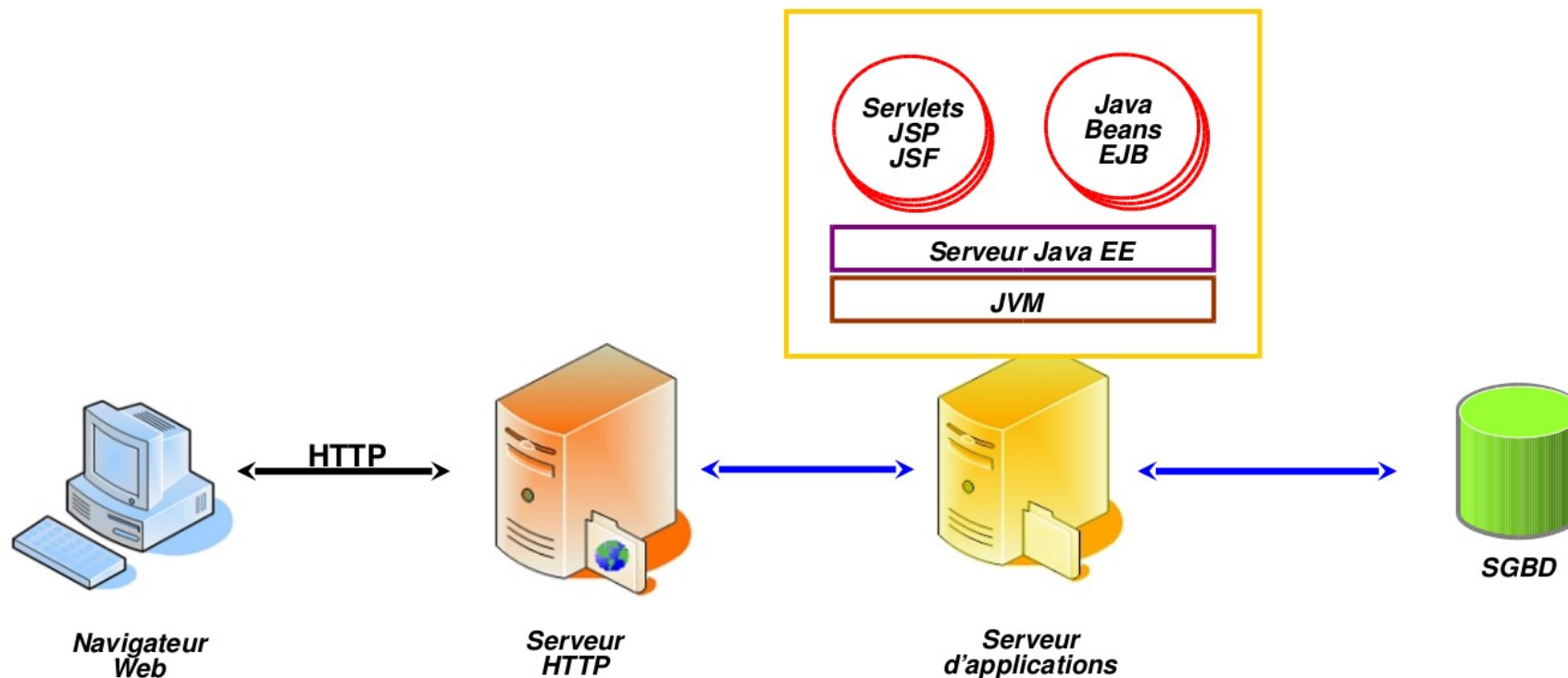
Application Web n-tiers

Modèle JavaEE
Présentation et MVC
Appels services métier
Déploiements

Monolithique non-scalable

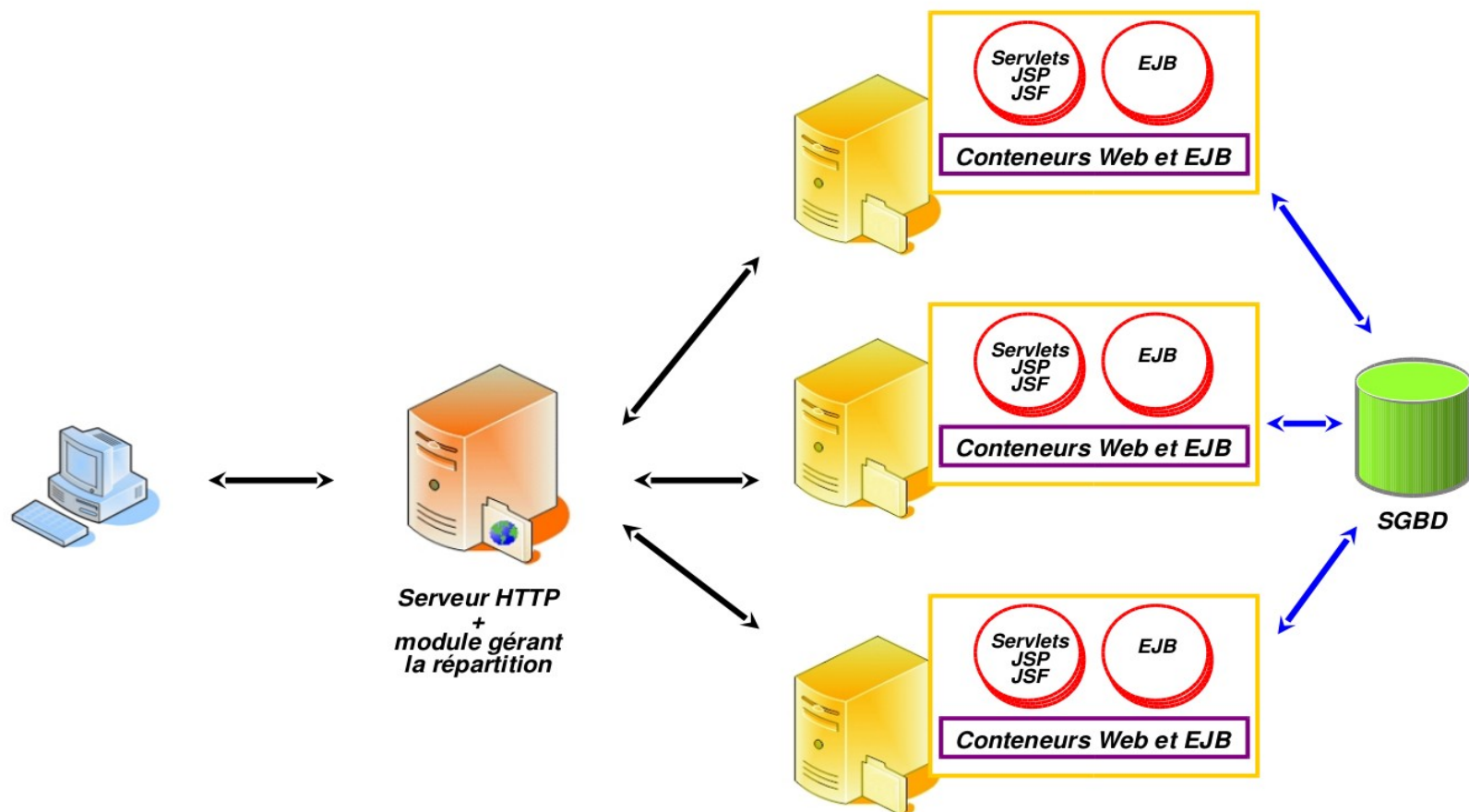
Application Web classique

- Framework MVC JSP/HTML, Services métier EJB (en option), Persistance via JPA. Packagé au format *.ear* ou *.war*
- Seveur HTTP en frontal/proxy



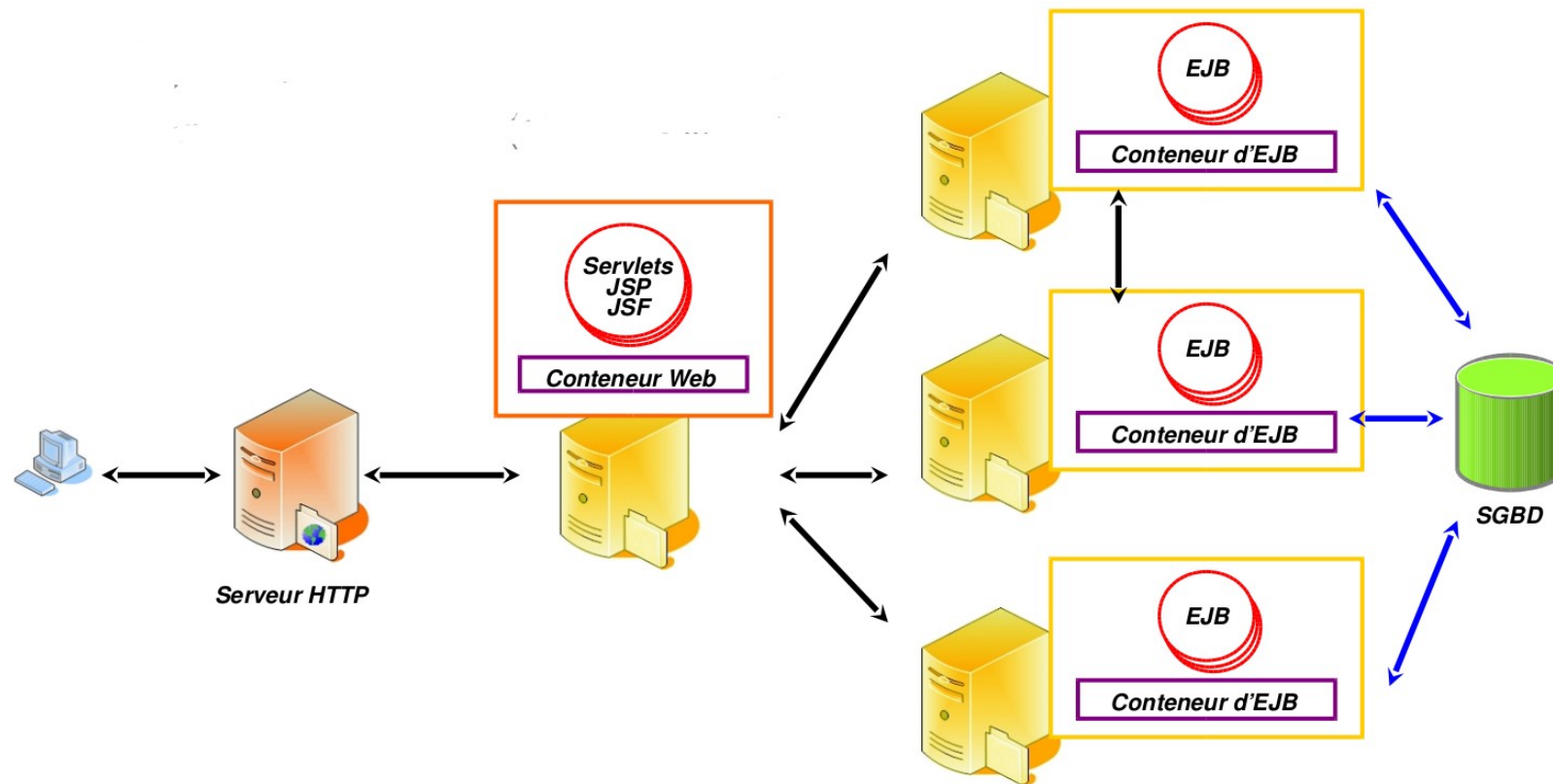
Monolithique scalé

Serveur HTTP comme répartiteur de charge



Scaling couche métier

Répartition de charge via EJB proxy





Services Web

XML Webservices

Restful JSON

Design By Contract



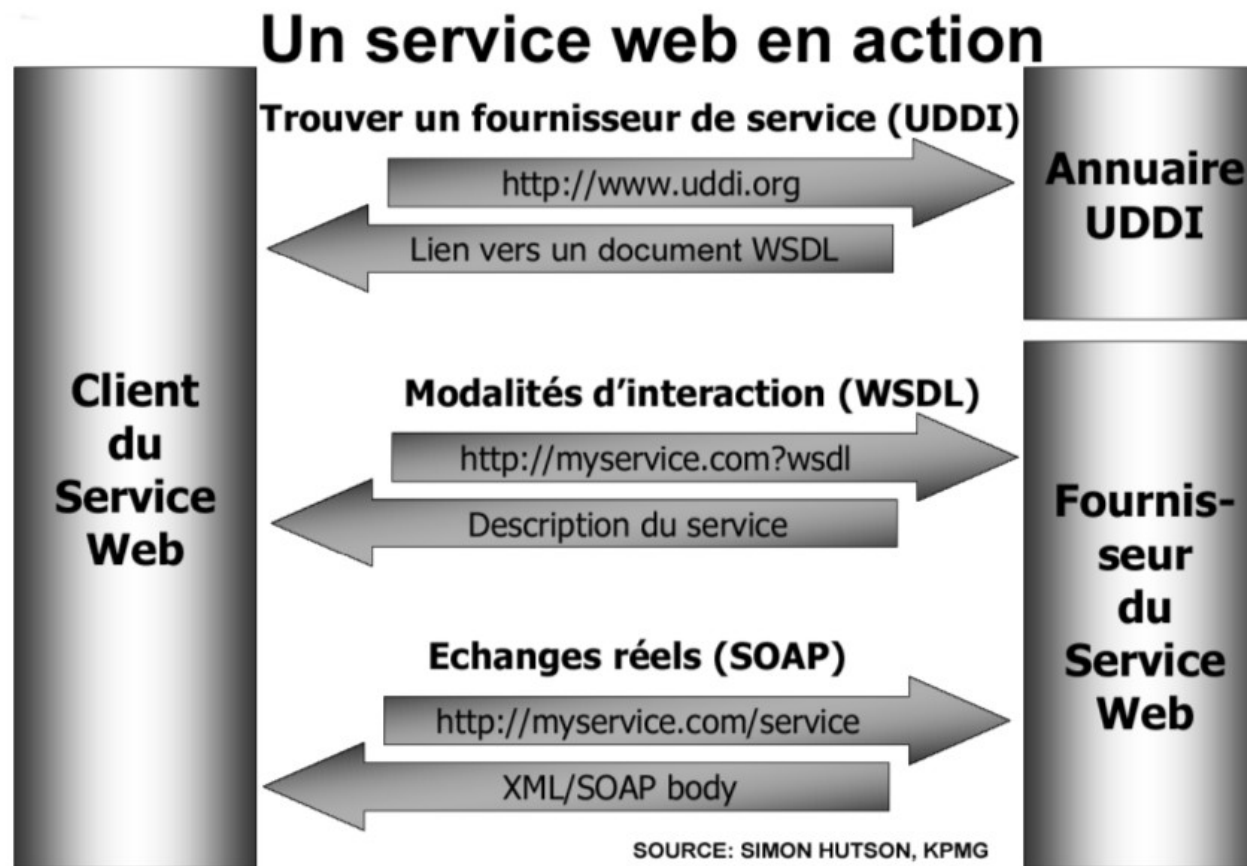
Services Web SOAP

Les API métier peuvent également être consommées par d'autres systèmes via les **services web SOAP** caractérisés par :

- leur grande interopérabilité,
- leur extensibilité
- et leurs descriptions pouvant être traitées automatiquement (WSDL).

Ces fonctionnalités sont apportées par XML

Scénario complet , service Web





SOAP

- **SOAP** est un protocole RPC. Il effectue des appels à des objets distants via un protocole de transport. (HTTP, SMTP, FTP)
- Bien que les objets soient sérialisés en XML, les clients SOAP voient des objets.
- L'enveloppe SOAP est constituée de 3 partie :
 - ◆ Une entête
 - ◆ Un corps de message
 - ◆ A l'intérieur du corps, éventuellement une faute



Exemple SOAP

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:doubleAnInteger xmlns:ns1="urn:MySoapServices">
      <param1 xsi:type="xsd:int">123</param1>
    </ns1:doubleAnInteger>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```



WSDL

- **WSDL** est une normalisation regroupant la description des éléments permettant de mettre en place l'accès à un Web Service
- C'est un document XML qui débute par la balise **<definition>** et qui contient les balises suivantes :
 - ♦ **<message>** : définition de données en entrée et sortie du service
 - ♦ **<portType>** : description des méthodes disponibles sur le WS
 - ♦ **<binding>** : protocole de communication (RPC/DOCUMENT)
 - ♦ **<service>** : URL du service



Exemple (1/2)

```
<?xml version="1.0" ?>
<definitions name="WSbibliotheque"
targetNamespace="http://corail1.utt.fr:8080/ws/WSbiblio.wsdl"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns="http://schemas.xmlsoap.org/wsdl/" >
  <!-- Données d'entrée/sortie -->
  <message name="getPrixRequest">
    <part name="livre" type="xsd:string" />
  </message>
  <message name="getPrixResponse">
    <part name="return" type="xsd:float" />
  </message>
  <!-- Méthodes disponibles du service web -->
  <portType name="WSbiblioPortType">
    <operation name="getPrix">
      <input message="getPrixRequest" name="getPrix"/>
      <output message="getPrixResponse" name="getPrixServeur"/>
    </operation>
  </portType>
```



Exemple (2/2)

<!-- Protocole de communication -->

```
<binding name="WSbiblioBinding" type="WSbiblioPortType">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="getPrix">
    <soap:operation soapAction="" />
    <input><soap:body use="encoded" namespace="urn:cite_biblio"/></input>
    <output><soap:body use="encoded" namespace="urn:cite_biblio"/></output>
  </operation>
</binding>
```

<!-- URL du service -->

```
<service name = "WSBibliotheque">
  <documentation>Prix d'un livre</documentation>
  <port name = "WSbiblioPortType" binding="WSbiblioBinding">
    <soap:address location="http://corail1.utt.fr:8080/soap"/>
  </port>
</service>
</definitions>
```



Librairies Framework

Java : JAX-WS, Apache Axis 2, Apache CXF, Spring WS

.NET : ASP.NET Web API, WCF

Php : Zend Soap

Javascript : *node-soap* (node.js), librairies clientes, package soap npm, ...



Exemple JAX-WS

```
package helloservice;
```

```
import javax.xml.ws.WebService;
```

```
import javax.xml.ws.WebMethod;
```

```
@WebService
```

```
public class Hello {
```

```
    private final String message = "Hello, ";
```

```
    public Hello() {  
    }  
}
```

```
@WebMethod
```

```
    public String sayHello(String name) {
```

```
        return message + name + ".";
```

```
    }
```

```
}
```



Consommation

```
import helloservice.endpoint.HelloService;
import javax.xml.ws.WebServiceRef;

public class HelloAppClient {
    @WebServiceRef(wsdlLocation =
        "http://localhost:8080/helloservice-war/HelloService?WSDL")
    private static HelloService service;

    private static String sayHello(java.lang.String arg0) {
        helloservice.endpoint.Hello port = service.getHelloPort();
        return port.sayHello(arg0);
    }
}
```



Inconvénients de XML

Rigueur : L'utilisation de schéma crée des exigences supplémentaires

Verbosité : Le volume des données à échanger est fortement augmenté par les balises, les références aux schémas, les espaces de noms

Exigences sur le client : Le client doit intégrer un parseur XML



Librairies Framework

Java : JAX-WS, Apache Axis 2, Apache CXF, Spring WS

.NET : ASP.NET Web API, WCF

Php : Zend Soap

Javascript : *node-soap* (node.js), librairies clientes, package soap npm, ...



Architectures SOA

Les **architectures SOA** s'appuient sur un ensemble de services web de granularité moyenne.

Chaque service expose une **interface** à **couplage faible** offrant un accès à une fonctionnalité **métier** ou **technique**.

On distingue :

- le SOA de surface : Ajout de services web sur des applications legacy à fins d'intégration
- Le SOA de profondeur : Conception d'une système via une décomposition en service



Services Web

XML Webservices
Restful JSON
Design By Contract



Service RESTFul

Les **API REST** sont également des services Web mais ont une infrastructure beaucoup plus légère :

- Ils peuvent être construits avec un outillage minimal et à faible coût
- Ils sont basés sur les standards du web : HTTP, Mime-type, URI
- Il remplace XML par JSON
- Ne sont pas auto-descriptifs mais des outils comme Swagger et OpenAPI3.0 comblent ce manque
- Consommées par :
 - d'autres systèmes : Intégration, micro-services
 - Des frameworks clients qui offrent des interfaces utilisateur beaucoup plus responsive



Conception

Très important, une API bien conçue est compréhensible par un développeur sans qu'il ait à lire la documentation

- l'API est auto-descriptive.
- Une API se matérialise directement dans l'URL des requêtes HTTP envoyées au serveur exposant la ressource.

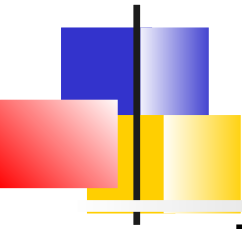
Exemple :

GET https://api.uber.com/partners/129/payments

Intuitivement on comprend que cette ressource concerne les paiement reçus par un partenaire UBER (conducteur indépendant).

Règles de nommage

Règle	Description
Noms	Utiliser les noms pour décrire la ressource Exemple : POST /payments au lieu de POST /createPayments
Pluriel	Utiliser le pluriel pour gérer les deux types de ressources Collection de ressources ou Instance d'une ressource Exemple Collection : POST /payments Exemple Instance : GET /payments/007
Casse cohérente : <i>snake_case</i> en général <i>Upper Kebab-Case</i> pour les headers	Utiliser le format snake_case car plus lisible en minuscule (fréquemment utilisée par les GAFA). Exemple : short_label et non pas shortLabel Ce format s'applique à tous les champs : L'URL (base path) & l'URI (nom de ressource), Path Param et Query Param, Body de la requête et de la réponse Seule exception, les Headers sont au format Upper Kebab-Case : Exemple : Content-Type
Format court	Utiliser des noms courts Exemple : payments_count et non pas nombre_de_paiements
Label lisible	Accompagner les codes techniques d'un label lisible .Exemple:{ "error_code": "ERR_INT_1ZE23E23", "message":"No message available« }
Pas de caractères spéciaux	Éviter les caractères spéciaux et les espaces dans le nommage des champs
Langues	Utiliser le français ou l'anglais en fonction du contexte de l'utilisateur cible



Ressource

Une ressource représente un objet « entité » décrit par un nom, sur lequel des actions / opérations (CRUD) sont effectuées.

- L'entité représentée par la ressource peut-être une abstraction de plusieurs entités persistantes.
- La conception d'une ressource revient à trouver les bonnes sources des données dans les systèmes d'informations afin de les exposer sous forme de ressources



Opération d'une ressource

- Une opération est un point de terminaison (ou « endpoint ») HTTP réel qui traite une demande HTTP/S envoyée par une application.
- Une opération est toujours exposée via une ressource. Elle définit une action qui peut être effectuée sur l'entité représentée par la ressource.
- Important : Une opération est autodescriptive grâce à la méthode HTTP qu'elle implémente (GET, POST, PUT, ...).
- Parfois, un chemin explicite est ajouté à la fin de l'URL pour affiner l'action sur la ressource.

Exemple d'une requête avec un chemin implicite :

GET

`https://api.entreprise/calendar/
meetings/{meeting_id}/attendees`

Cette requête renvoie la liste des participants à la réunion.

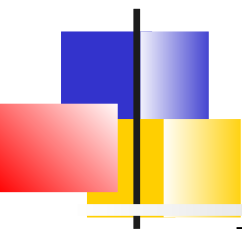
Exemple d'une requête avec un chemin explicite :

GET

`https://api.entreprise/calendar/
meetings/{meeting_id}/attendees/search`

Cette requête recherche la liste des participants à la réunion.

Le verbe « search » affine l'action si la méthode HTTP seule n'est pas suffisamment claire.



Paramètres de requête (1)

Le protocole HTTP définit 4 types de paramètres qui peuvent être transmis dans une requête :

- ***Path parameter*** : Paramètre transmis par le chemin de l'URL
- ***Query parameter*** : Paramètre de requête
- ***Header*** : Paramètre d'entêtes HTTP
- ***Body*** : Corps de la requête

Ces paramètres doivent être utilisés selon certaines bonnes pratiques.



Paramètres de requête (2)

Paramètre	Utilisation	Exemple
Path Parameter	Pour l'identification seulement : <ul style="list-style-type: none">- Uniquement un ID, toujours suivant l'entité à laquelle il se réfère- Paramètre obligatoire	http://.../accounts/ 123 /transactions
Query Parameter	Pour la gestion de résultat - filtrer, trier, ordonner, grouper les résultats (paramètres courts) : <ul style="list-style-type: none">- Paramètres techniques optionnels- Valeurs sont définies et documentées dans la spécification de l'API	http://.../transactions ? from = NOW & sort = date:desc & limit = 50
Header	Pour la gestion du contexte d'application et de la sécurité <ul style="list-style-type: none">- Utilisé par les navigateurs, les applications clientes et autres pour transmettre des informations sur le contexte de la demande- Utilisé pour transmettre les paramètres d'authentification	Authorization : Bearer XXXXXXXXXX
Body	Pour les données fonctionnelles <ul style="list-style-type: none">- Utilisé pour transmettre des informations fonctionnelles- Doit être un objet JSON	{ "name": "phone", "category": "tech", "max_price": 45 }



Codes Retours (1)

Les codes retours HTTP permettent de déterminer le résultat d'une requête ou d'indiquer une erreur au client.

Ils sont standards et doivent être utilisés d'une façon appropriée pour une gestion efficace des anomalies en production.

5 catégories :

- 100 : Informational
- 200 : Success
- 300 : Redirection
- 400 : Erreur Client
- 500 : Erreur serveur



Codes 2xx

Code	Description	Cas d'usage
200 – OK	Requête traitée avec succès	Toute requête réussie
201 – Created & Location	Requête traitée avec succès et création d'un document.	Création d'un nouvel objet. Le lien ou l'identifiant de la nouvelle ressource est envoyé dans la réponse
204 – No content	Requête traitée avec succès mais pas d'information à renvoyer.	Mettre à jour ou supprimer un objet (avec une réponse vide)
206 – Partial Content	Une partie seulement de la ressource a été transmise.	Obtenez une liste paginée d'objets



Codes 3xx

Les codes 3xx sont rarement utilisés par les ressources RestFul sauf lors de renommage de l'API

Code	Description
301 – Moved Permanently	Ressource déplacée de façon permanente
302 – Found	Ressource déplacée de façon temporaire



Codes 4xx (1)

Code	Description	Cas d'usage
400 – Bad Request	La syntaxe de la requête est erronée	<ul style="list-style-type: none">- Format des dates incorrect-Envoi de XML au lieu de JSON- Integer à la place de string- Envoi d'un objet de type non attendu- Oubli d'un paramètre obligatoire
401 - Unauthorized	Une authentification est nécessaire pour accéder à la ressource	Jeton d'authentification absent de la Requête ou invalide
403 – Forbidden	Le serveur a compris la requête, mais refuse de l'exécuter : contrairement à l'erreur 401, s'authentifier ne fera aucune différence	Le rôle de l'utilisateur n'est pas suffisant



Codes 4xx (2)

Code	Description	Cas d'usage
404 – Not Found	Ressource non trouvée	L'objet demandé n'existe pas Exemple : GET/compte/{id} et ce compte n'existe pas.
405 - Method Not Allowed	Méthode connue du serveur mais pas prise en charge pour la ressource	La méthode utilisée n'est pas supportée pour cette URL
406 – Not Acceptable	La ressource demandée n'est pas disponible dans un format qui respecterait les en-têtes "Accept" de la requête	Cela indique qu'il est impossible de servir une réponse qui satisfait aux critères définis dans les en-têtes Accept-Charset et Accept-Language.
409 – Conflict	La demande n'a pas pu être traitée en raison d'un conflit avec l'état actuel de la ressource	Tentative de création d'un nouveau profil utilisateur avec une adresse e-mail déjà existante
429 - Too Many Requests	Le client a émis trop de requêtes dans un délai donné	Throttling



Codes 5xx

Code	Description	Cas d'usage
501 - Not Implemented	Fonctionnalité réclamée non supportée par le serveur	La méthode (GET, PUT, ...) n'est connue du serveur pour aucune ressource
502 - Bad Gateway ou Proxy Error	Mauvaise réponse envoyée à un serveur intermédiaire par un autre serveur	La réponse du backend n'est pas comprise par l'API Gateway
503 – Service Unavailable	Service temporairement indisponible ou en maintenance	API hors service, en maintenance, ...
504 - Gateway Time-out	Temps d'attente d'une réponse d'un serveur à un serveur intermédiaire écoulé	Timeout dépassé



RestController

Côté Java, les requêtes de l'API sont traitées par des RestController

Les RestController s'appuient sur les couches basses (Service ou Repository)

Leur responsabilité consiste à :

- Définir le bon mapping (association avec l'URL du point d'accès Rest)
- Récupérer les paramètres de l'appel REST
- Appeler les bonnes méthodes des couches basses (service ou repository)
- Générer la bonne réponse : Code retour, entêtes et corps JSON

Dans un contexte de framework, la plupart de ce travail est effectué via des annotations



Exemple Spring MVC

```
@RestController
@RequestMapping(value="/users")
public class MyRestController {

    @RequestMapping(value="/{user}", method=RequestMethod.GET)
    public User getUser(@PathVariable Long user) {
        // ...
    }

    @RequestMapping(value="/{user}/customers", method=RequestMethod.GET)
    List<Customer> getUserCustomers(@PathVariable Long user) {
        // ...
    }

    @RequestMapping(value="/{user}", method=RequestMethod.DELETE)
    public User deleteUser(@PathVariable Long user) {
        // ...
    }

    @RequestMapping(path = "/Members", method = RequestMethod.POST)
    public ResponseEntity<Member> register(@Valid @RequestBody Member member) {

        member = memberRepository.save(member);

        return new ResponseEntity<>(member,HttpStatus.CREATED);
    }
}
```



Exemple JAX-RS

@Path("/student/data")

```
public class RestServer {
```

```
    @GET
```

```
    @Produces(MediaType.APPLICATION_JSON)
```

```
    public Student getStudentRecord(){
```

```
        Student student = new Student();
```

```
        student.setLastName("Hayden");
```

```
        student.setSchool("Little Flower");
```

```
        return student;
```

```
    }
```

```
    @POST
```

```
    @Consumes(MediaType.APPLICATION_JSON)
```

```
    public Response postStudentRecord(Student student){
```

```
        return Response.status(201).entity("Record entered: "+ student).build();
```

```
    }
```

```
}
```



Sérialisation JSON

Un des principales problématiques des interfaces REST et la conversion des objets du domaine au format JSON.

Des frameworks spécialisés sont utilisés (Jackson, Gson) mais en général le développeur doit régler certaines problématiques :

- Boucle infinie pour les relations bi-directionnelles entre entités
- Adaptation aux besoins de l'interface de front-end
- Optimisation du volume de données échangées

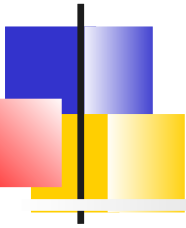


Alternatives Jackson à la sérialisation

Jackson propose une sérialisation par défaut pour les classes modèles basée sur les getter/setter.

Pour adapter la sérialisation par défaut à ses besoins, 3 alternatives :

- Créer des classes DTO spécifiques
- Utiliser les annotations de champs proposées par Jackson
- Utiliser les JsonView sur son modèle
- Implémenter ses propres *ObjectMapper*



Annotations de base Jackson

@JsonProperty, @JsonGetter, @JsonSetter, @JsonAnyGetter, @JsonAnySetter, @JsonIgnore, @JsonIgnoreProperty, @JsonIgnoreType :
Permettant de définir les propriétés JSON

@JsonRootName : Arbre JSON

@JsonSerialize, @JsonDeserialize : Indique des dé/sérialiseurs spécialisés

@JsonManagedReference, @JsonBackReference, @JsonIdentityInfo : Gestion des relations bidirectionnelles

....



SpringDoc

SpringDoc est un outil qui simplifie la génération et la maintenance de la documentation des API REST

Il est basé sur la spécification OpenAPI 3 et s'intègre avec Swagger-UI

Il suffit de placer la dépendance dans le fichier de build :

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-ui</artifactId>
  <!-- OU : springdoc-openapi-webflux-ui -->
  <version>1.5.2</version>
</dependency>
```



Fonctionnalités

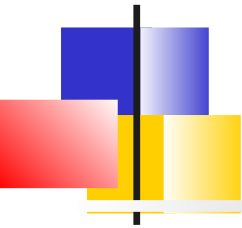
Par défaut,

- La description OpenAPI est disponible à :
<http://localhost:8080/v3/api-docs/>
- L'interface Swagger à :
<http://localhost:8080/swagger-ui.html>

SpringDoc prend en compte

- les annotations javax.validation positionnées sur les DTOs
- Les Exceptions gérées par les @ControllerAdvice
- Les annotations de OpenAPI
<https://javadoc.io/doc/io.swagger.core.v3/swagger-annotations/latest/index.html>

SpringDoc peut être désactivé via la propriété :
`springdoc.api-docs.enabled=false`



Services Web

XML Webservices
Restful JSON

Design By Contract



Design By Contract

il est important de définir précisément l'API d'un service en utilisant une sorte de langage de définition d'interface (IDL).

La définition de l'API dépend du style d'interaction :

- OpenAPI pour Rest
- Canaux de messages, format et type du message pour l'asynchrone



Consumer Driven Contract Test

Consumer-driven contract test Pattern¹ : Vérifier que la « forme » de l'API d'un fournisseur répond aux attentes du consommateur.

Dans le cas REST, le test de contrat vérifie que le fournisseur implémente un point de terminaison qui :

- A la méthode et le chemin HTTP attendus
- Accepte les entêtes attendus
- Accepte un corps de requête, le cas échéant
- Renvoie une réponse avec le code d'état, les entêtes et le corps attendus

1. <http://microservices.io/patterns/testing/service-integration-contract-test.html>



OpenAPI

Swagger est un projet open-source proposant une suite d'outils devenus la référence dans le monde de la conception et la documentation d'API.

- Site officiel : <https://swagger.io/>
- Les outils Swagger : <https://swagger.io/tools/>

L'OpenAPI Specification (OAS) était à l'origine connu sous le nom de Swagger Specification et faisait partie du framework Swagger



Example

```
openapi: 3.0.0
info:
  version: 1.0.0
  title: Sample API
  description: A sample API to illustrate OpenAPI concepts
paths:
  /list:
    get:
      description: Returns a list of stuff
      responses:
        '200':
          description: Successful response
```




Apports du contrat

A partir du contrat, on peut :

- Générer du code (Contrôleurs et mapping), classe de service client
- Générer des tests d'acceptation qui permettront de valider une implémentation
- Générer des Mock API : Permettant aux consommateurs de travailler en toute indépendance



Architecture micro-services

Modèle

Services techniques
Patterns et leurs relations



Introduction

Le terme « ***micro-services*** » décrit un nouveau pattern architectural visant à améliorer la rapidité et l'efficacité du développement et de la gestion de logiciel

C'est le même objectif que les méthodes agiles ou les approches *DevOps* :
« *Déployer plus souvent* »



Architecture

Une architecture micro-services implique la décomposition des applications en très petits services

- faiblement couplés
- ayant une seule responsabilité
- développés par des équipes full-stack indépendantes.



Bénéfices attendus

Scaling indépendant : Seuls les services les plus sollicités sont scalés
=> Économie de ressources

Mise à jour indépendantes : Les changements locaux à un service peuvent se faire sans coordination avec les autres équipes
=> Agilité de déploiement

Maintenance facilitée : Les services sont plus petits
=> Corrections, évolutions plus rapide

Hétérogénéité des langages : Utilisation des langages les plus appropriés pour une fonctionnalité donnée

Isolation des fautes : Un dysfonctionnement peut être plus facilement localiser et isoler.

Equipe DevOps autonome : Full-stack team, Intra-Communication renforcée

=> Favorise le partage et les montées en compétences



Caractéristiques

Design piloté par le métier : La décomposition fonctionnelle est pilotée par le métier (voir *Evans's DDD approach*)

Principe de la responsabilité unique : Chaque service est responsable d'une seule fonctionnalité et la fait bien !

Une interface explicitement publiée : Un producteur de service publie une interface qui peut être consommée

DURS (Deploy, Update, Replace, Scale) indépendants : Chaque service métier peut être indépendamment déployé, mis à jour, remplacé, scalé

Communication légère : REST sur HTTP, STOMP sur WebSocket, Server-Sent Events,



Contraintes

Réplication : Un micro-service doit être scalable facilement, cela a des impacts sur le design (stateless, etc...)

Découverte : La scalabilité (automatisé selon certains métriques) nécessite que la localisation des services soit dynamique => Service de discovery

Monitoring : Les services sont surveillés en permanence. Des traces sont générées puis agrégées

Résilience : Les services peuvent être en erreur. L'application doit pouvoir résister aux erreurs.

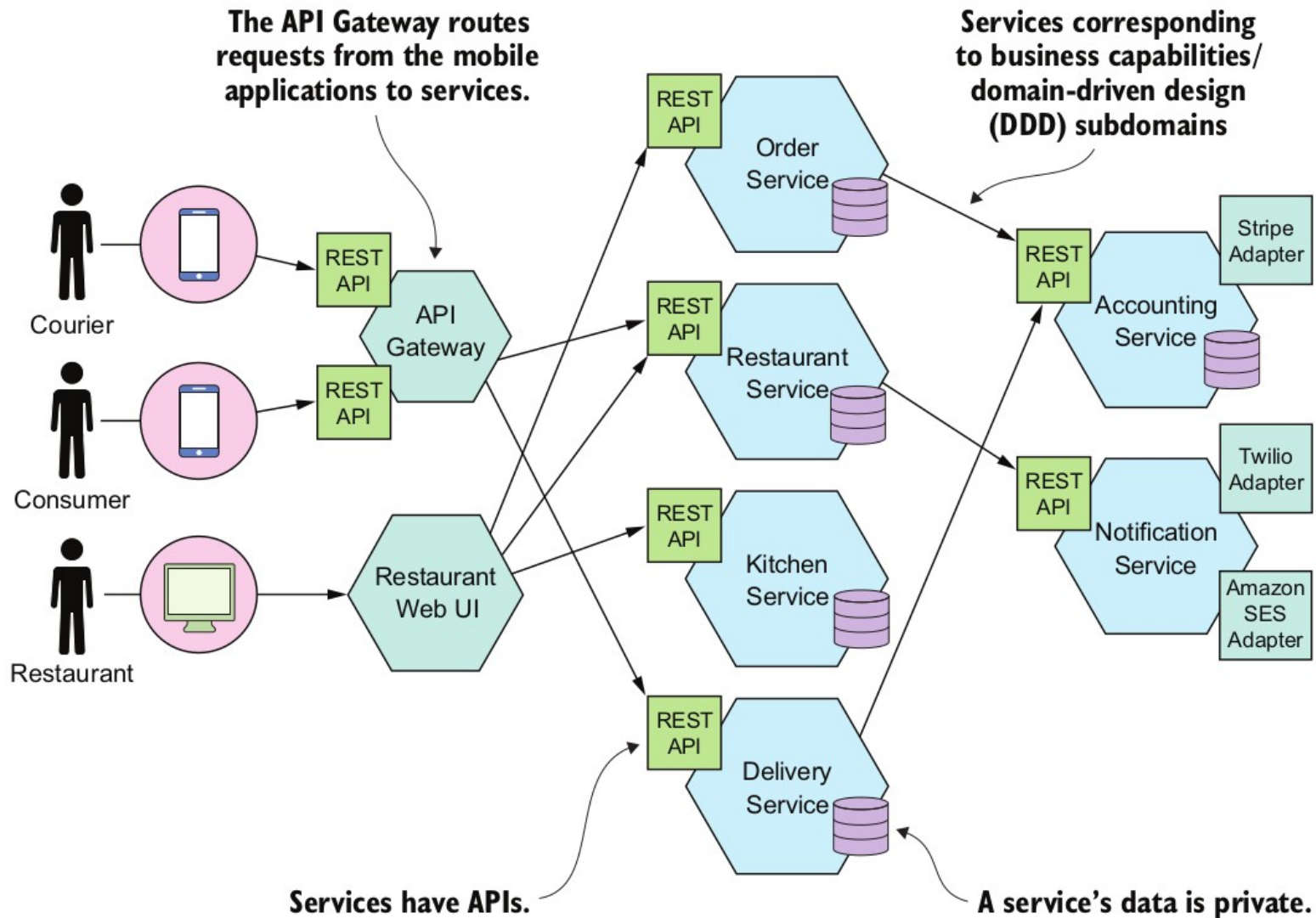
DevOps : L'intégration et le déploiement continu sont indispensables pour le succès.



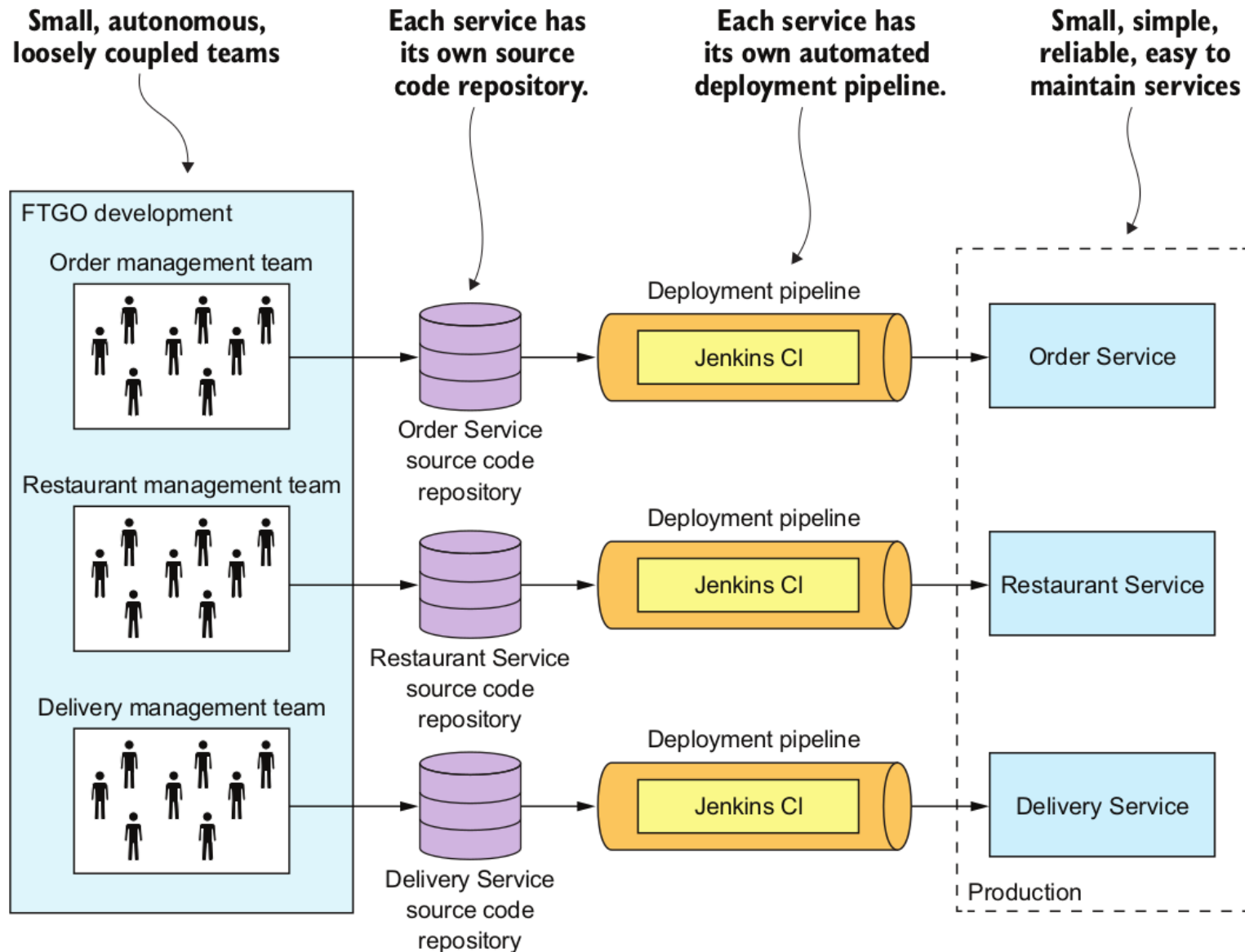
Inconvénients et difficultés

- Trouver la bonne décomposition est difficile.
Une mauvaise décomposition peut entraîner des couplages entre les micro-services
- Le côté distribué fait que le système complet est plus difficile à tester, déployer
- Le déploiement de fonctionnalités qui touche plusieurs services est plus délicat
- La migration d'une application monolithique existante vers les micro-services n'est pas simple

Une architecture micro-service



Organisation DevOps





Architecture micro-services

Modèle

Services techniques

Patterns et leurs relations



Services Transverses

De nombreux services transverses peuvent être fournis par un framework ou une infrastructure

- Service **de discovery** permettant à un micro-service de s'enregistrer et de localiser ses micro-services dépendants
- Service de centralisation de **configuration** facilitant la configuration et l'administration des micro-services
- Services **d'authentification** offrant une fonctionnalité de SSO parmi l'ensemble des micro-services, de génération et de relais de jeton
- Service de **monitoring** agrégeant les métriques de surveillance en un point central
- Support pour la répartition de charge, le fail-over, la résilience aux fautes

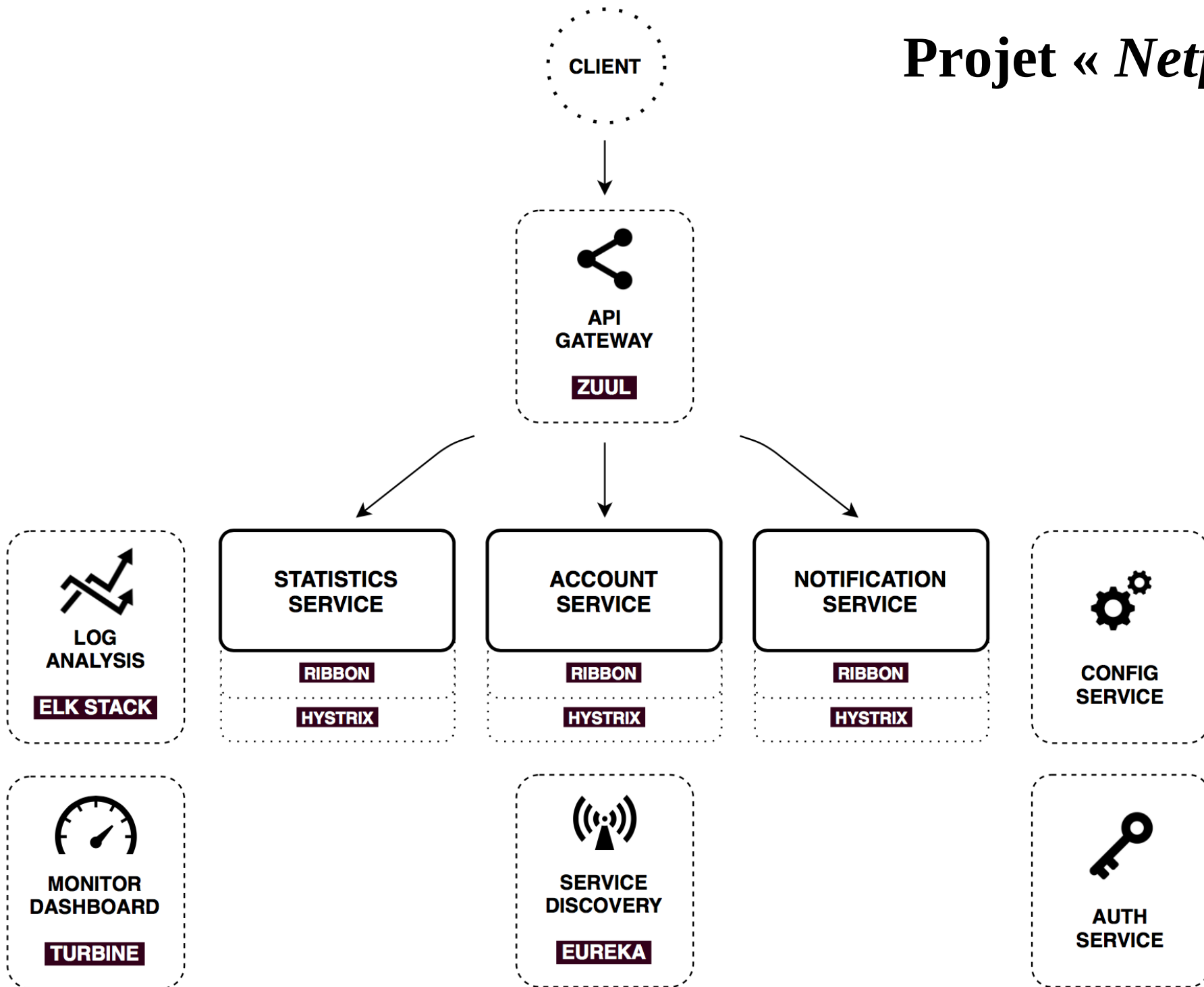


Services techniques vs Infra

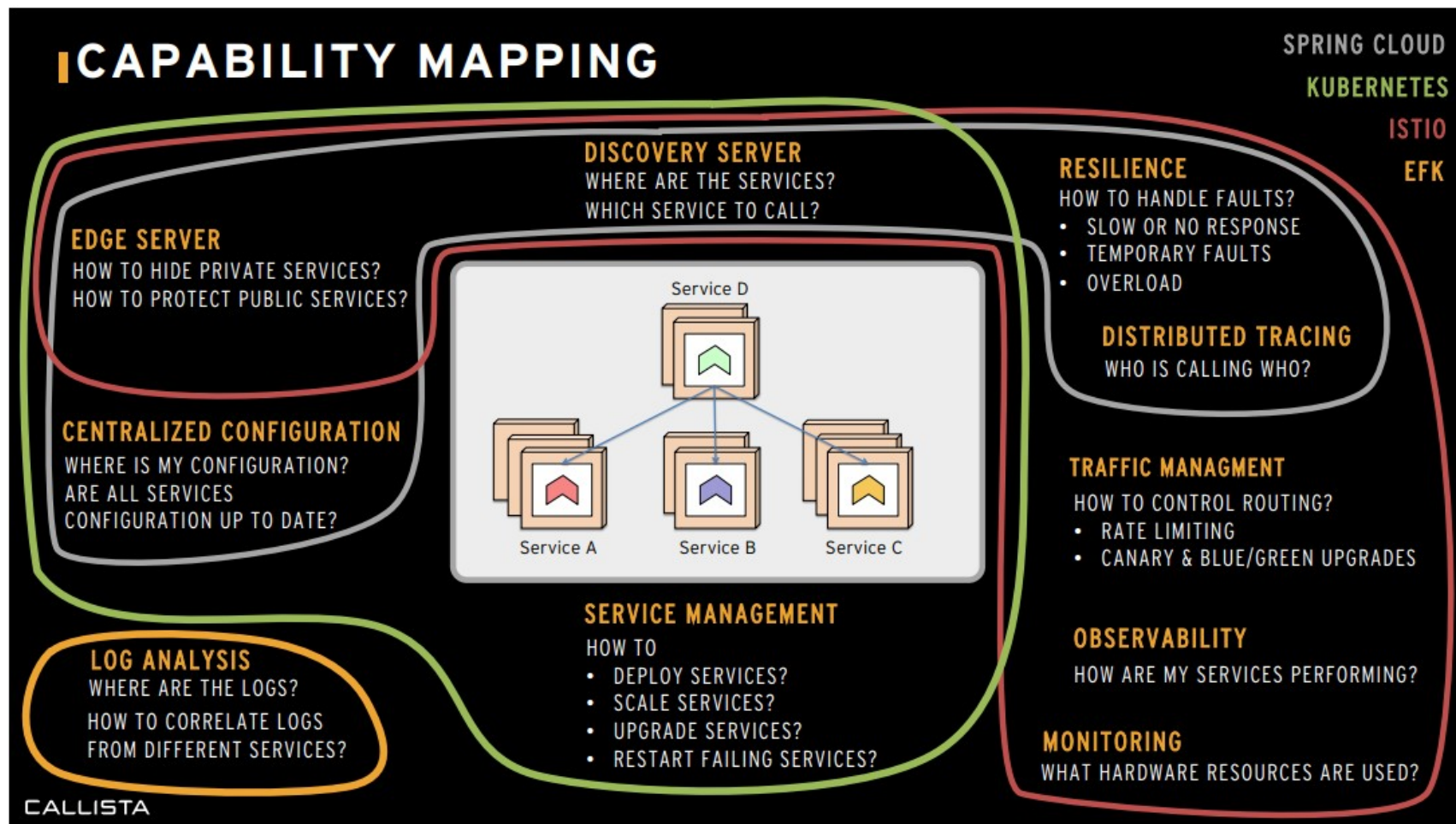
Qui fournit les services techniques ?

- Dans les premières architectures, c'est le software => Voir framework Netflix intégré dans SpringCloud
- Actuellement, de nombreux services techniques migrent vers l'infrastructure :
 - Discovery, Config, Répartition de charge offert nativement par Kubernetes
 - Résilience, Sécurité, Monitoring : Service mesh de type Istio

Projet « Netflix »



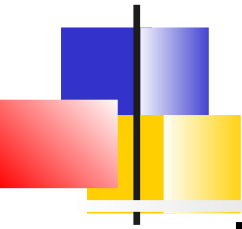
Capability Mapping





Architecture micro-services

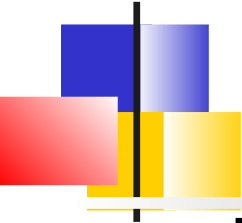
Introduction
Services techniques
Patterns et leurs relations



Patterns micro-service

Les patterns concernant les architectures micro-services peuvent être découpés en 3 domaines :

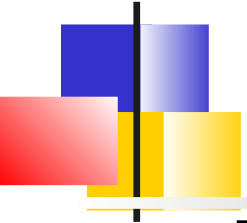
- ***Pattern d'infrastructure*** : Problématique en dehors du développement concernant l'infrastructure d'exécution des systèmes distribués
- ***Pattern applicatif d'infrastructure*** : Problématique d'infrastructure qui impacte le développement.
(Par exemple, quel mode de communication offre un message broker)
- ***Pattern applicatif*** : Problématique purement de développement.



Problèmes à résoudre et design patterns

Patterns applicatifs

- Quelle Décomposition pour mes services ?
Patterns : DDD/sous-domaines, Business Capability,
Comment définir mon API
- Comment maintenir la cohérence de mes données distribuées ?
Saga Pattern
- Comment requêter sur des données distribuées ?
CQRS Pattern
- Comment tester mes micro-services en isolation ?
Design By Contract
- Comment architecturer ma/mes bases de données ?



Problèmes à résoudre et design patterns

Patterns infrastructure applicative

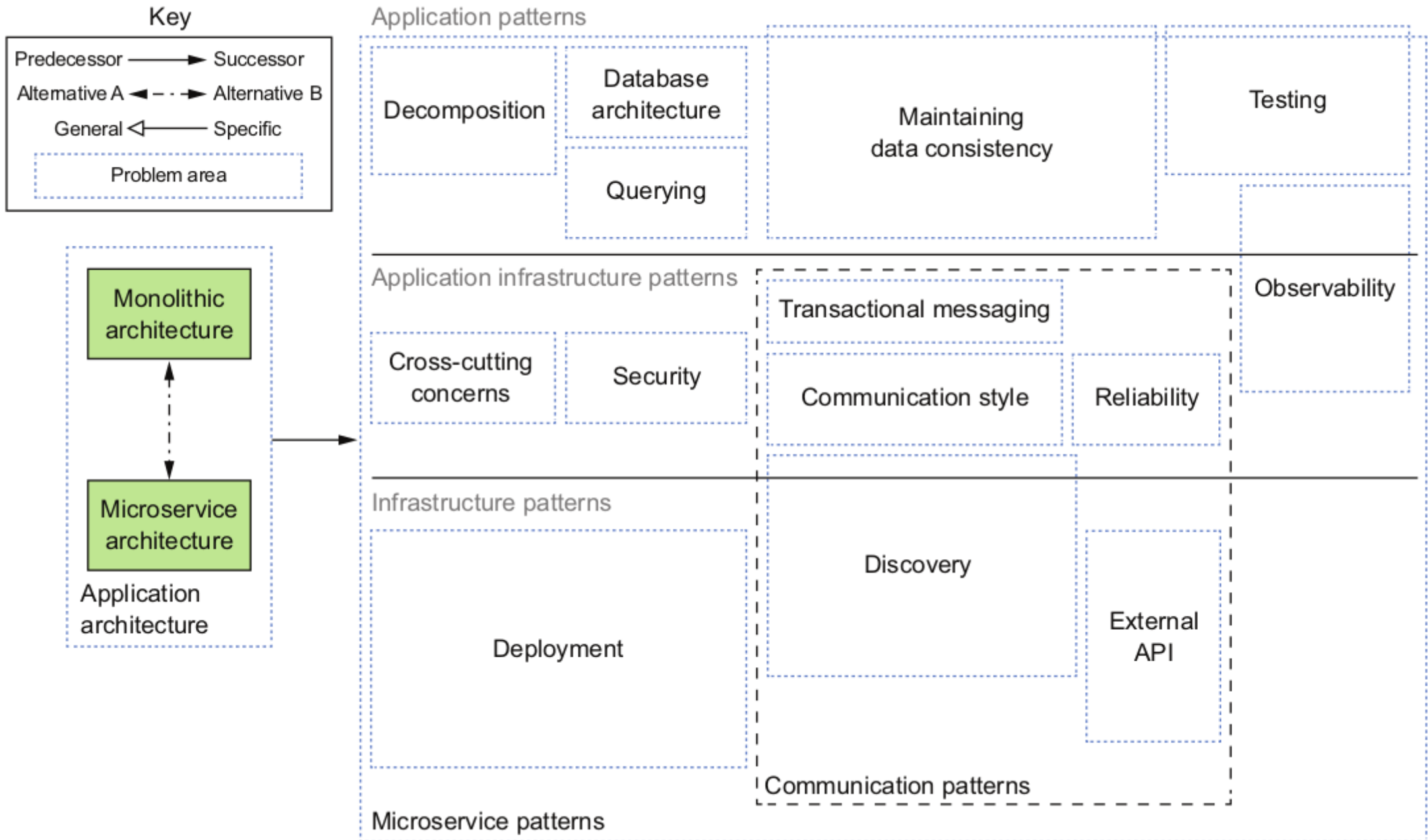
- Quelle communication entre services ?
RPC, Asynchrone, Reactive, Messagerie transactionnelle
- Comment apporter de la résilience ?
Circuit-breaker pattern, Sondes,
- Quels sont les moyens de l'observabilité ?
- Service de discovery, infrastructure ou application ?



Patterns et problèmes à résoudre

Pattern d'infrastructure

- Quelle infrastructure de déploiement est la plus adaptée ?
Hôtes uniques avec différents processus, Orchestration de Containers, Serverless
- Quels moyens pour exposer les services ?
Ips publics, Ingress Gateway





Clients REST

API manager et API gateway
Clients Javascript



API Gateway Pattern

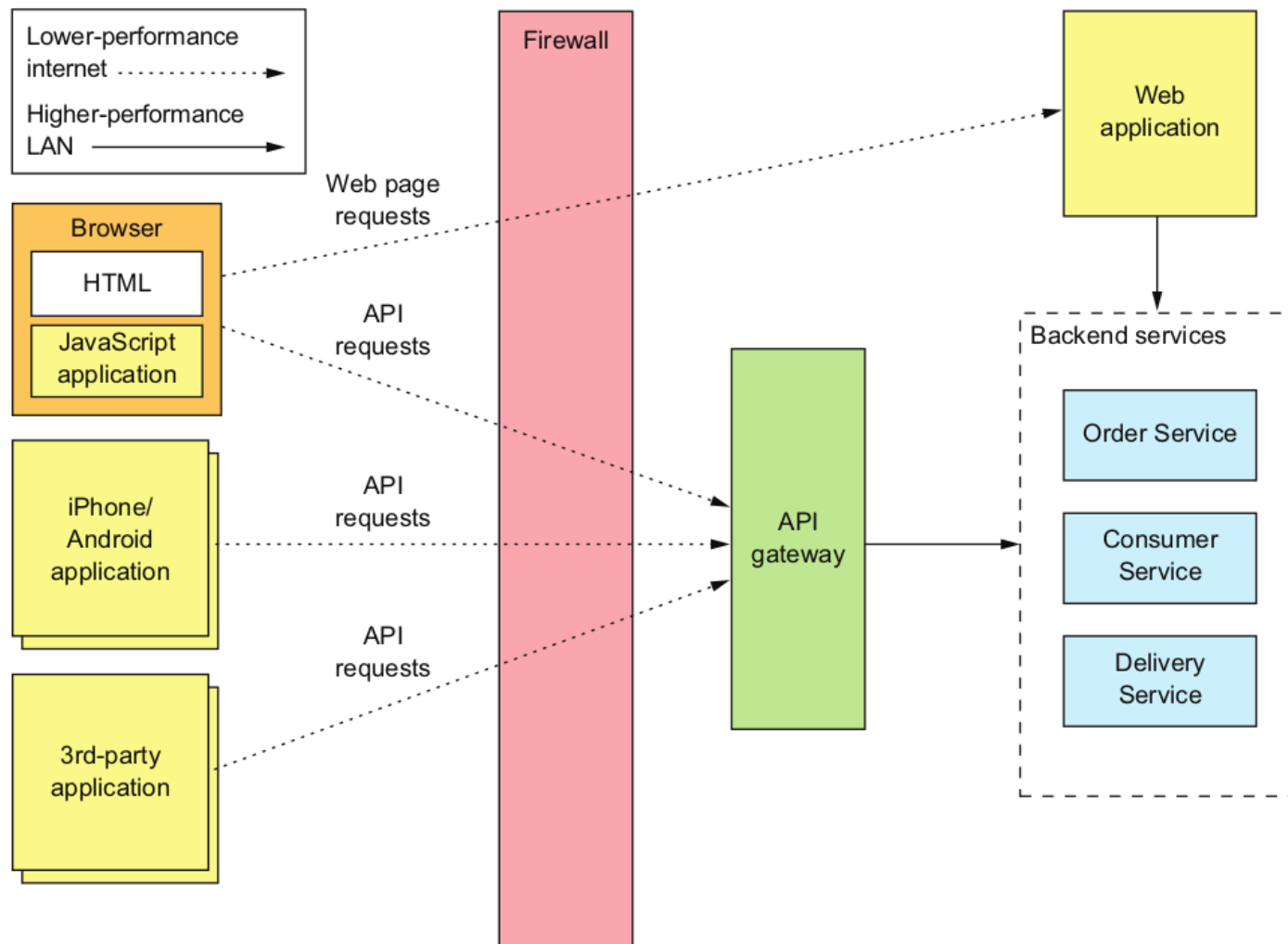
API gateway Pattern¹ : Implémente un service qui est le point d'entrée de l'application micro-service pour les clients externes

Le service API Gateway est alors responsable du routage des requêtes, de la composition d'API, de la traduction de protocole, de l'authentification et d'autres fonctions

Similaire au pattern *Facade* en Objet

1. <http://microservices.io/patterns/apigateway.html>

API Gateway





Fonctions de la Gateway

Routage : En fonction d'une table de routage, la gateway transfère les requêtes aux services backend. La table de routage peut s'appuyer sur tous les composants HTTP (URL, entêtes, paramètres de requêtes)

Composition d'API : Plusieurs appels vers les services backend sont alors agrégés

Traduction de protocoles : Traduction de requêtes GraphQL en requêtes REST par exemple

API spécifique par clients : La gateway n'offre pas la même API pour un mobile que pour les partenaires externes

Fonctions transverses (edge functions)¹ : Implémentation de l'authentification, de l'autorisation, du cache, du log de requêtes,

1. On peut également implémenter ces fonctions dans un service dédié en amont de la gateway



Solutions de gestion d'API

Les solutions de gestion d'API sont très répandues en entreprise. Elles apportent des fonctionnalités :

- Gateway : routing, sécurité, limitation de débit
- Store d'API : Documentation, mocking, souscription, monétisation
- Publication : Cycle de vie d'une API

Fonctionnalités

API Gateway	Authen/author des accès	Vérification des tokens de sécurité et de l'autorisation d'accès à la maille API	Équilibrage de charge	Répartition de la charge entre les nœuds d'une même API
	Mise en cache	Mise en cache d'une partie des résultats des appels aux API (standards HTTP)	Routage	Découverte des services et routage des échanges
	Médiation et interopérabilité	Médiation protocolaire et forçage de certains standards d'interopérabilité	Metering	Mesure de la consommation de chaque application consommatrice (par API, par jour, ...)
	Alerting	Sur atteinte des quotas ou évènement de sécurité	Throttling	Limitation de la consommation pour une application (ex: 10k appels /jour pour API A)
API Store	Catalogue et documentation	Catalogue d'API éligibles et documentation (compatible avec le format Swagger/OpenAPI)	Souscription au service	Gestion de son compte développeur et souscription aux API sous réserve d'éligibilité
	Sandbox	Outils de test des API (bouchons) par les développeurs d'applications	Suivi de la consommation	Portail de suivi des souscription, de la consommation et de la facturation
API Publisher	Gestion de quotas	Attribution de quotas sur la consommation des API (globaux et spécifiques aux comptes)	Gestion de la visibilité	Quel développeur d'application peut accéder à quelles API ?
	Cycle de vie	Gestion du cycle de vie des API (définition, test, publication, mise à jour, décommissionnement)	Gestion des droits	Gestion de droits d'accès des utilisateurs / systèmes : contrôle sur les API store et Gateway
	Niveaux de service	Différents niveaux de service pour les différentes API	Facturation	Mécanisme de facturation de l'utilisation des API (nombre d'appels, SLA souscrit ...)



Solutions



Axway est une solution d'API management déployée chez de nombreux clients en France (Total, iCDC, Fortis, BPCE). Elle a prouvé sa robustesse et sa performance dans des cas d'usages réels. Elle fournit des fonctionnalités sécurité avancées (couplage à un anti virus, lutte contre les injections de code, etc.).



La solution de CA est largement reconnue. Elle a été retenue notamment par la BNPP (ITG). C'est une solution complète, qui propose de nombreuses fonctionnalités de médiation et qui assure également une protection contre les failles du TOP 10 OWASP. Néanmoins son administration (via un client lourd) est datée et complexe.



WSO2 est la solution opensource d'APIM leader. Elle est mise en prod et/ou reconnue comme solution cible chez plusieurs de nos clients (Crédit Agricole, Société Générale IBFS, ...). Elle bénéficie également d'une communauté active. A noter qu'une nouvelle version entièrement révisée est prévue pour début 2018.



IBM était en clairement en retard sur le marché, plombé par la gamme Datapower qui a toujours rencontré un succès commercial mitigé. Cette année, sa solution API connect a entièrement été refondue et semble prometteuse. Elle a notamment été adoptées par PSA. La solution est cloud hybride, et supporte les websocket.

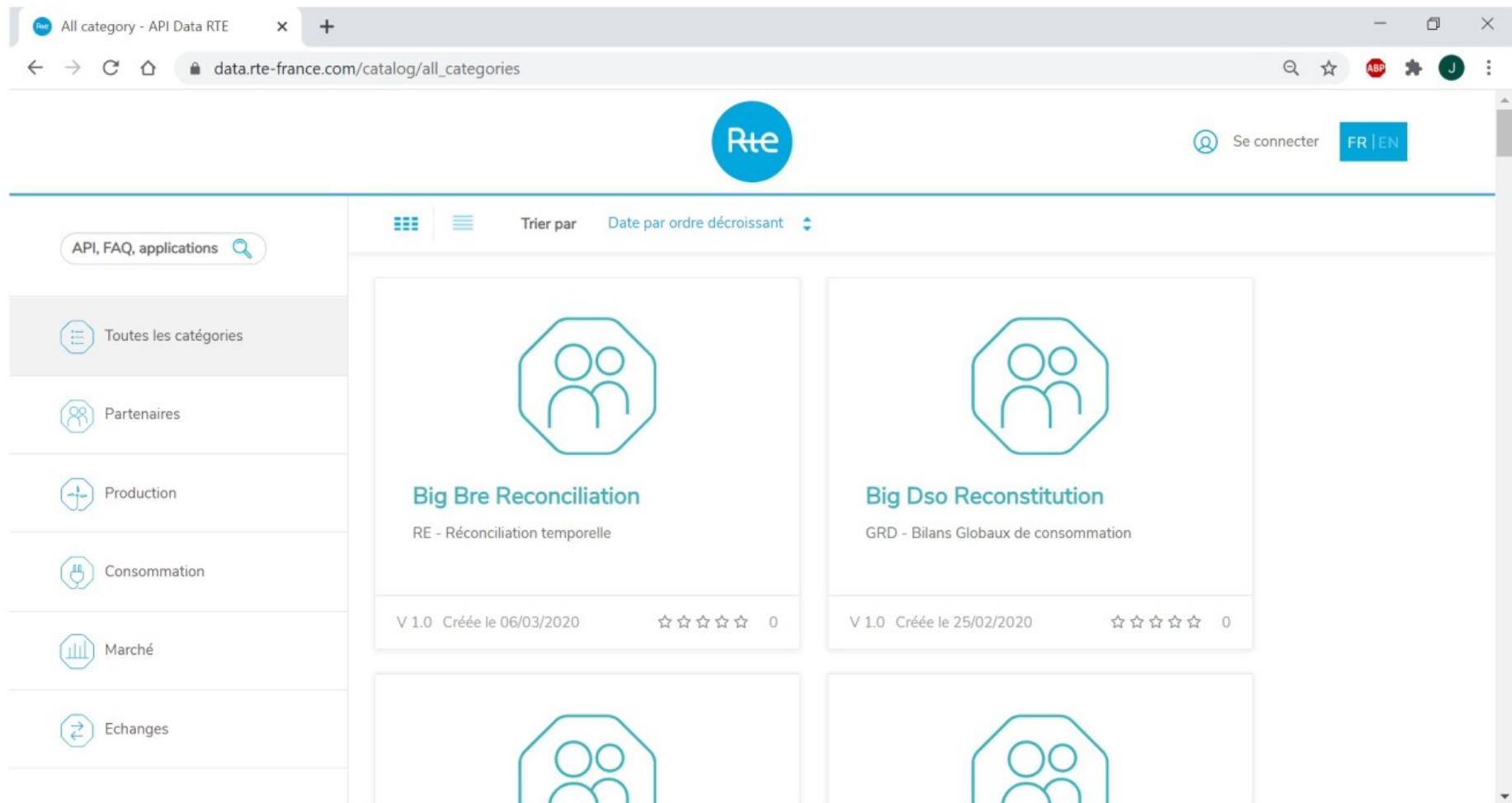


Software AG est un leader historique des middlewares SOA avec son produit phare : WebMethods (ESB). Leur solution d'API Management actuelle est clairement en retrait et peu mature par rapport aux autres éditeurs. Cependant, leur produit est en cours de refonte complète.



TIBCO est un leader sur le marché des produits middleware. Ils ont racheté et intégré à leur offre le produit Mashery, leader et pure player du marché, initialement détenu par INTEL. Nous n'avons pas de REX client sur ce produit mais beaucoup de REX positifs sur l'EAI/ESB de TIBCO et les bus de messages(PMU, AG2RLM, etc.).

Exemple API Rte



The screenshot displays the 'All category - API Data RTE' page on the website `data.rte-france.com/catalog/all_categories`. The interface features a top navigation bar with the Rte logo, a 'Se connecter' button, and language toggles for 'FR' and 'EN'. A left sidebar provides a search bar and a list of categories: 'Toutes les catégories', 'Partenaires', 'Production', 'Consommation', 'Marché', and 'Echanges'. The main content area is titled 'Trier par Date par ordre décroissant' and displays a grid of API cards. The first two cards are 'Big Bre Reconciliation' (RE - Réconciliation temporelle) and 'Big Dso Reconstitution' (GRD - Bilans Globaux de consommation), both version 1.0 and created in early 2020. The bottom row shows two more cards with a house icon, which are partially cut off. The page is viewed in a browser window with standard navigation controls and a taskbar at the bottom.

API, FAQ, applications

Toutes les catégories

Partenaires

Production

Consommation

Marché

Echanges

Trier par Date par ordre décroissant

Big Bre Reconciliation
RE - Réconciliation temporelle
V 1.0 Créée le 06/03/2020

Big Dso Reconstitution
GRD - Bilans Globaux de consommation
V 1.0 Créée le 25/02/2020

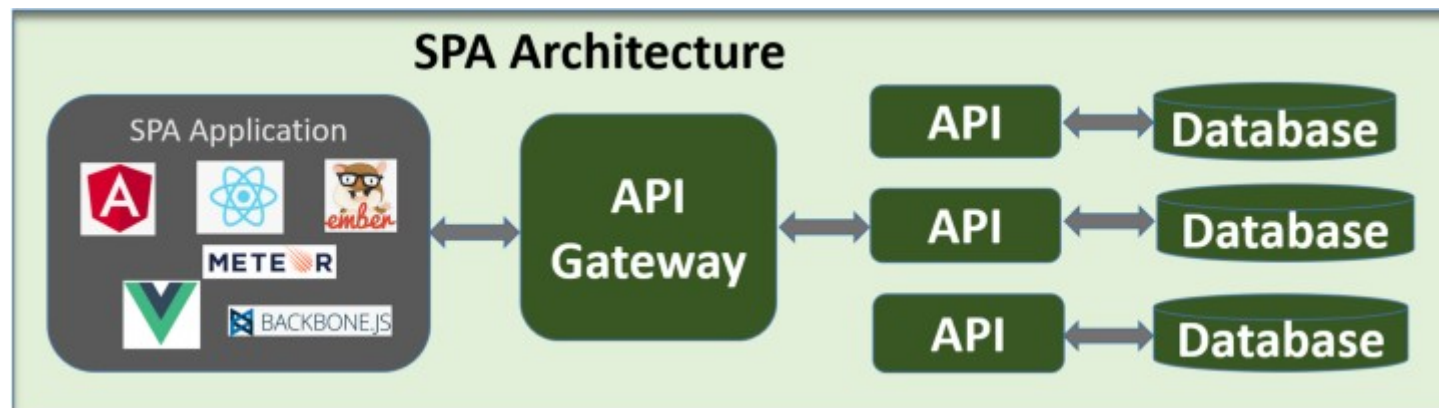


Clients REST

API manager et API gateway
Clients Javascript

Architecture SPA

Les consommateurs finaux d'API sont souvent des application SinglePage développée avec des frameworks Javascript





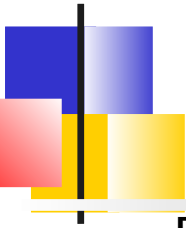
Introduction

De nombreux frameworks sont apparus ces derniers temps sur la couche cliente.

Ils sont pour la plupart basés sur Javascript.

Citons :

- *React JS (Facebook)*
- *Angular (Google)*
- *Vue JS*



Critères de choix frameworks

ReactJS :

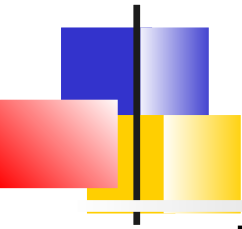
- FaceBook et Communauté importante
- Génère du HTML à partir de JSX, le navigateur ne voit que le HTML généré, idéal pour le SEO
- Mobile

Angular :

- Google
- TypeScript
- Injection de dépendances, gabarits, forms et outils de build et test

VueJS

- Taille de fichier extrêmement petite
- Courbe d'apprentissage rapide et très bonne documentation
- Facilité d'intégration à d'autres applications et langages
- Facilité de créer des composants



Angular

Développement principalement en
TypeScript,

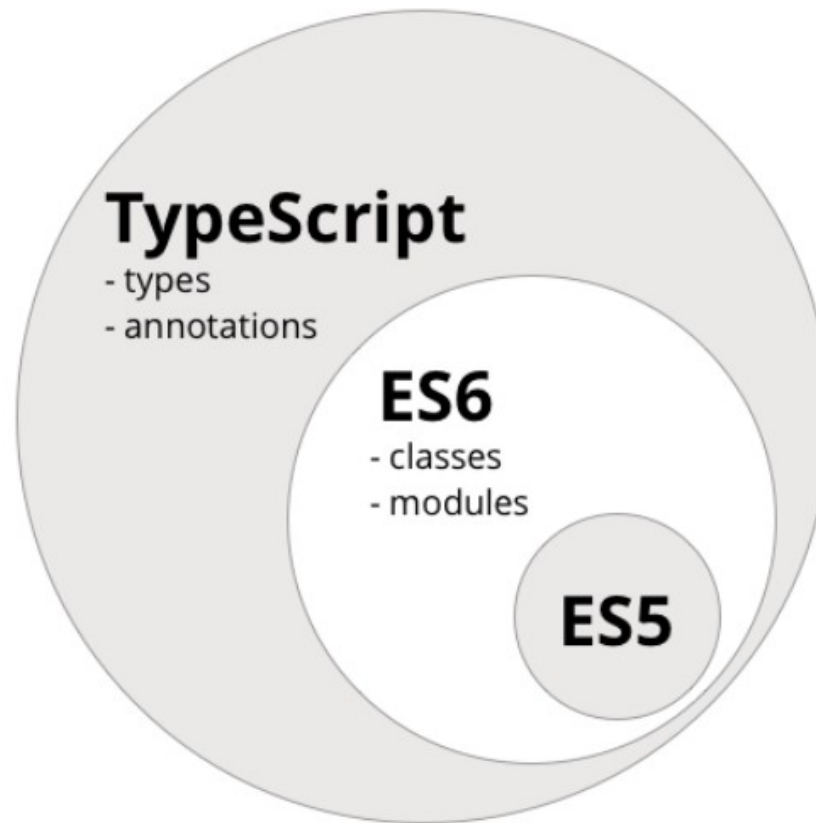
Cible Web et Mobile

Concepts cœur :

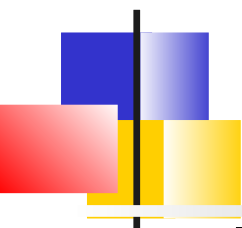
- Approche composant (réutilisable, encapsulation)
- Binding, Dependency Injection, appels asynchrones



TypeScript



ES5, ES6, and TypeScript



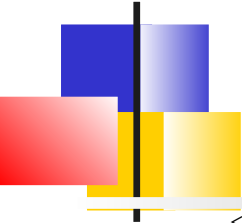
Modèle de programmation

L'interface est constitué de **composants**.

Les composants sont packagés en **modules** qui déclare les composants, les services

Chaque composant est une classe TypeScript qui

- Est associé à un sélecteur HTML (~balise), un gabarit HTML, un style
- Contient le modèle de données associé au composant
- Contient les méthodes gérant les interactions qui peuvent s'appuyer sur des services back-end injectés



Exemple Component *<my-app>*

```
<body>
  <my-app>Loading...</my-app>
</body>
```

```
-----

import { Component }           from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    <h1>{{title}}</h1>
    <nav>
      <a routerLink="/dashboard" routerLinkActive="active">Dashboard</a>
      <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
    </nav>
    <router-outlet></router-outlet>
  `,
  styleUrls: ['app/app.component.css']
})

export class AppComponent {
  title = 'Tour of Heroes';
}
```



Exemple composant

```
import { Component, OnInit } from '@angular/core';
import { Subscription } from 'rxjs/Subscription';

import { Critere } from './critere.model';
import { CritereService } from './critere.service';

@Component({
  selector: 'my-critere',
  templateUrl: './critere.component.html'
})
export class CritereComponent implements OnInit {
  criteres: Critere[];
  currentAccount: any;

  constructor(private critereService: CritereService) { }

  loadAll() {
    this.critereService.query().subscribe(
      (res: ResponseWrapper) => {
        this.criteres = res.json;
      },
      (res: ResponseWrapper) => this.onError(res.json)
    );
  }

  ngOnInit() {
    this.loadAll();
    this.principal.identity().then((account) => {
      this.currentAccount = account;
    });
    this.registerChangeInCriteres();
  }
}
```



Gabarit

```
<div class="table-responsive" *ngIf="criteres">
  <table class="table table-striped">
    <tbody>
      <tr *ngFor="let critere of criteres ;">
        <td>
          <a [routerLink]="['../critere',
critere.id ]">{{critere.id}}</a></td>
        <td>{{critere.nom}}</td>
      </tr>
    </tbody>
  </table>
</div>
```



Classe Service

@Injectable()

```
export class CritereService {

    private resourceUrl = SERVER_API_URL + 'api/criteres';

    constructor(private http: Http) { }
    create(critere: Critere): Observable<Critere> {
        const copy = this.convert(critere);
        return this.http.post(this.resourceUrl, copy).map((res: Response) => {
            const jsonResponse = res.json();
            return this.convertItemFromServer(jsonResponse);
        });
    }
    update(critere: Critere): Observable<Critere> {
        const copy = this.convert(critere);
        return this.http.put(this.resourceUrl, copy).map((res: Response) => {
            const jsonResponse = res.json();
            return this.convertItemFromServer(jsonResponse);
        });
    }
    find(id: number): Observable<Critere> {
        return this.http.get(`${this.resourceUrl}/${id}`).map((res: Response) => {
            const jsonResponse = res.json();
            return this.convertItemFromServer(jsonResponse);
        });
    }
    delete(id: number): Observable<Response> {
        return this.http.delete(`${this.resourceUrl}/${id}`);
    }
}
```




Service (Sérialisation)

```
private convertResponse(res: Response): ResponseWrapper {
    const jsonResponse = res.json();
    const result = [];
    for (let i = 0; i < jsonResponse.length; i++) {
        result.push(this.convertItemFromServer(jsonResponse[i]));
    }
    return new ResponseWrapper(res.headers, result, res.status);
}

/**
 * Convert a returned JSON object to Critere.
 */
private convertItemFromServer(json: any): Critere {
    const entity: Critere = Object.assign(new Critere(), json);
    return entity;
}

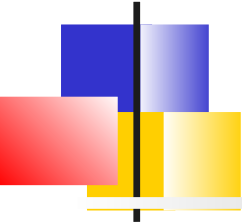
/**
 * Convert a Critere to a JSON which can be sent to the server.
 */
private convert(critere: Critere): Critere {
    const copy: Critere = Object.assign({}, critere);
    return copy;
}
```



Angular cli

Le développement est facilité par l'outil *Angular CLI* qui permet

- de générer la structure de l'application, d'un composant
- De gérer les dépendances et le build du projet
- De gérer un profil de développement et un profil de production
- De déployer

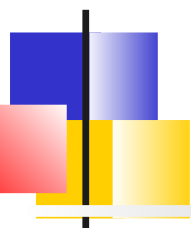


Infrastructures de déploiement

Introduction

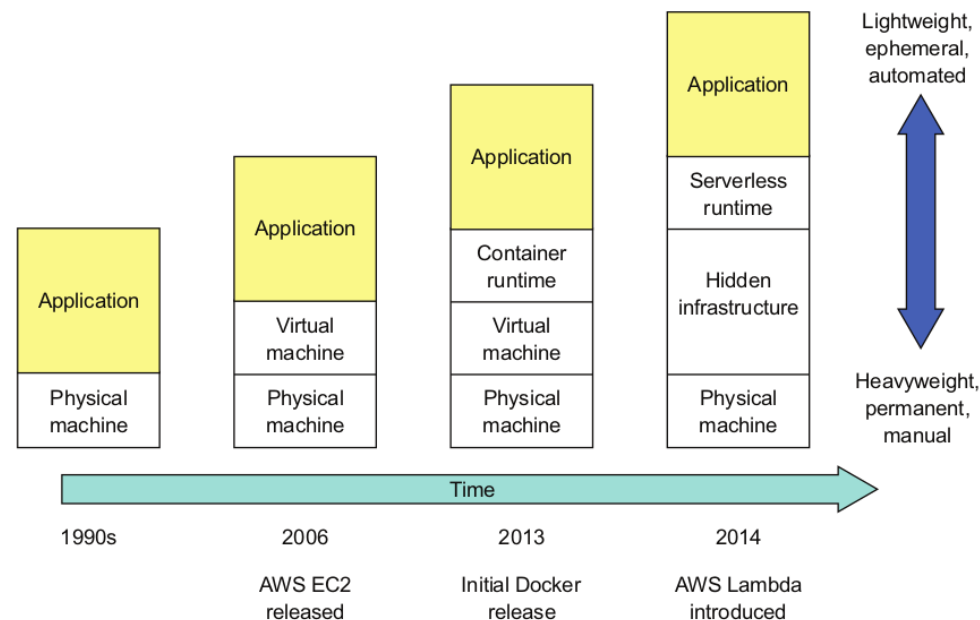
Virtualisation

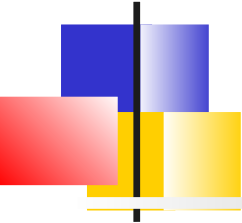
Containers et orchestrateurs



Infrastructure de déploiement

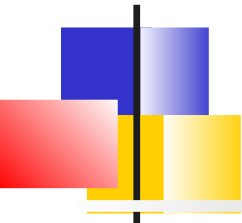
Même si plusieurs alternatives peuvent être envisagées, l'utilisation des technologies de container et d'orchestrateur de container sont à privilégier.





Infrastructures de déploiement

Introduction
Virtualisation
Containers et orchestrateurs

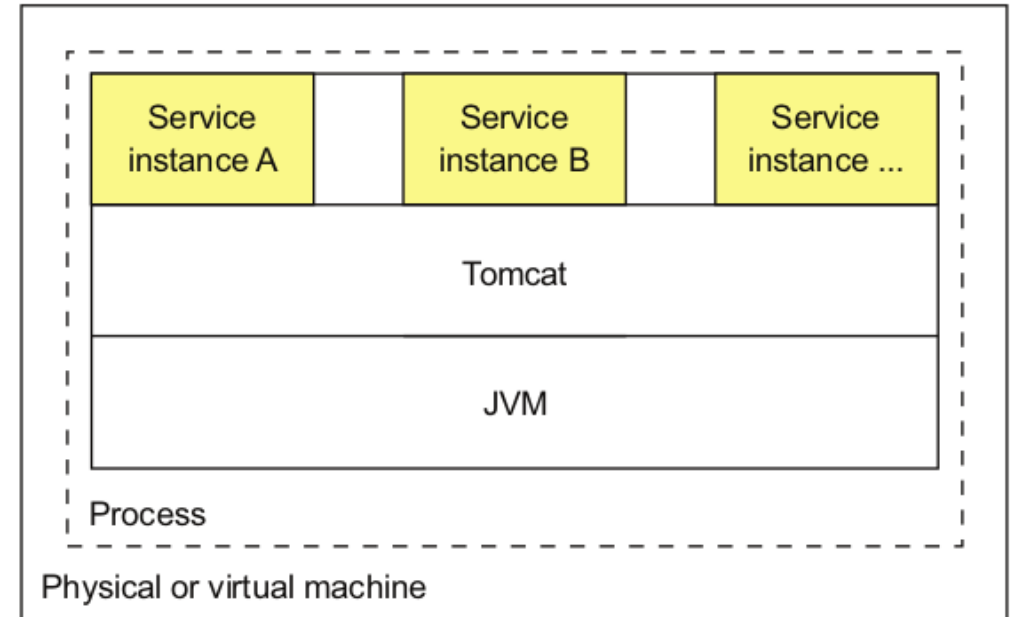
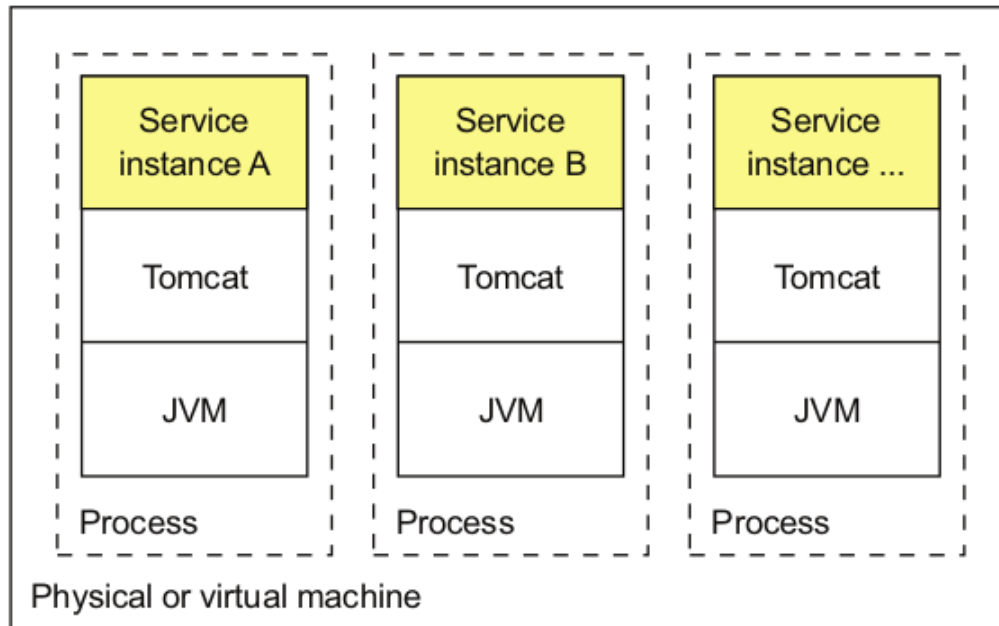


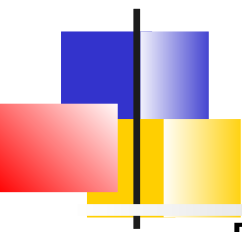
Format de packaging spécifique au langage

Une alternative est de packager les services dans un format spécifique au langage (ex : .war) et de le déployer sur une machine provisionnée (JDK + Tomcat)

- Soit chaque service est dans un processus distinct (a son propre Tomcat)
- Soit plusieurs services sont dans le même processus (plusieurs services déployés sur le même Tomcat)

Examples





Bénéfices / Inconvénients

Bénéfices

Déploiement rapide

Utilisation efficace des ressources surtout lorsque l'on exécute plusieurs instances sur la même machine ou le même processus

Inconvénients

Pas d'encapsulation de la pile technologique. Déploiements mutables.

Pas de possibilité pour limiter les ressources consommées par une instance de service.

Manque d'isolement lors de l'exécution de plusieurs instances de service sur la même machine.

Il est difficile de déterminer automatiquement où placer les instances de service.



Images VM

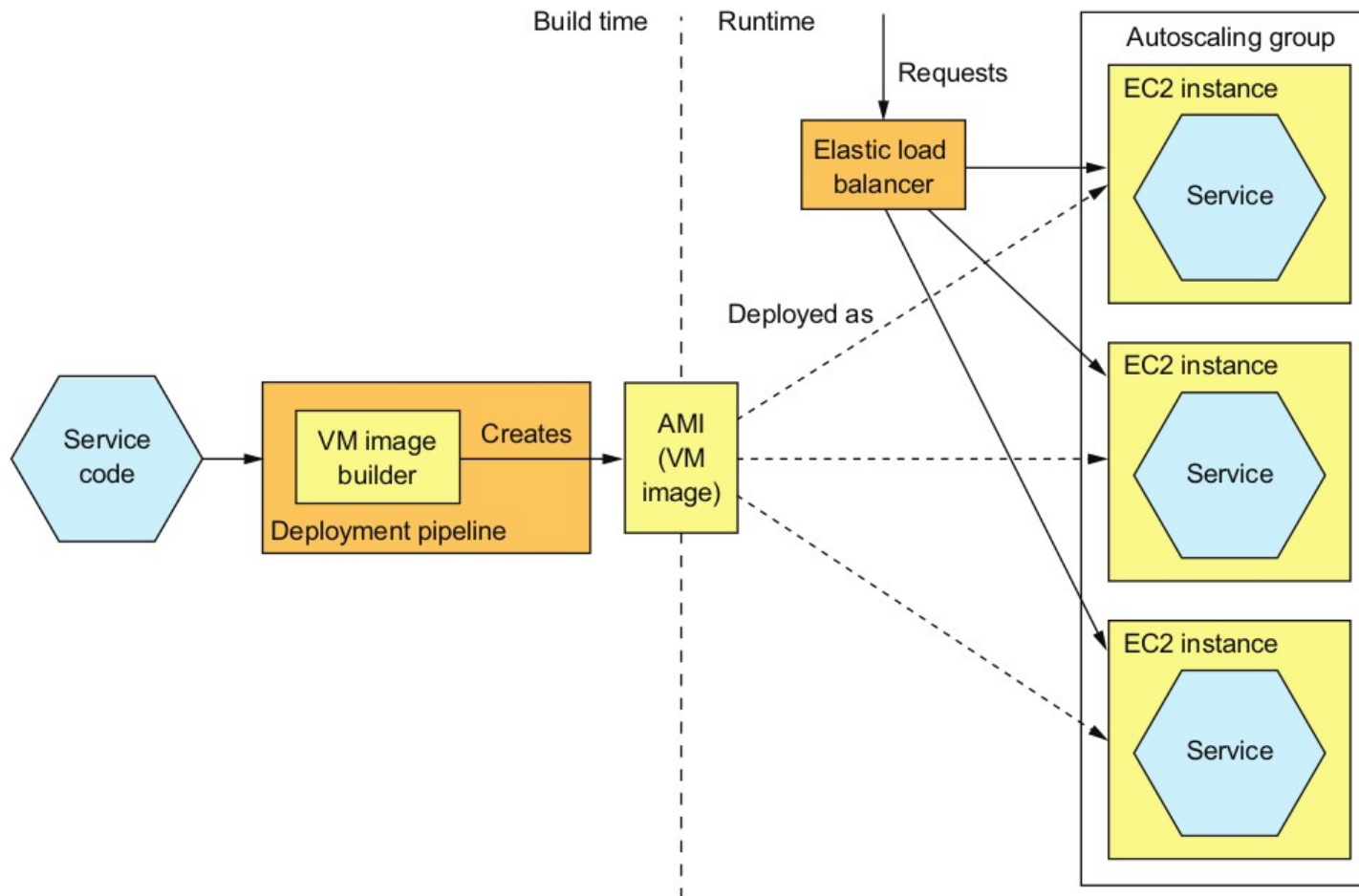
Deploy a service as a VM Pattern¹ :

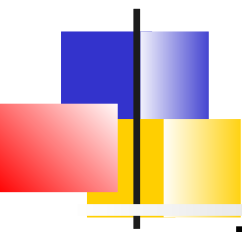
Déployer les services packagés comme des images VM. Chaque service est une VM

Le packaging peut s'automatiser dans les pipelines de build.

1. <http://microservices.io/patterns/deployment/service-per-vm.html>

Example





Bénéfices / Inconvénients

Bénéfices :

L'image VM encapsule la pile technologique =>
Déploiement immuable

Instances de service isolées.

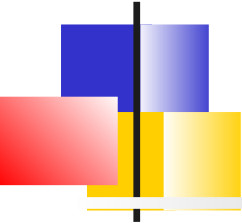
Utilise une infrastructure cloud mature.

Inconvénients :

Utilisation peu efficace des ressources

Déploiements relativement lents

Surcharge d'administration du système



Infrastructures de déploiement

Introduction
Virtualisation
Containers et orchestrateurs



Service comme Container

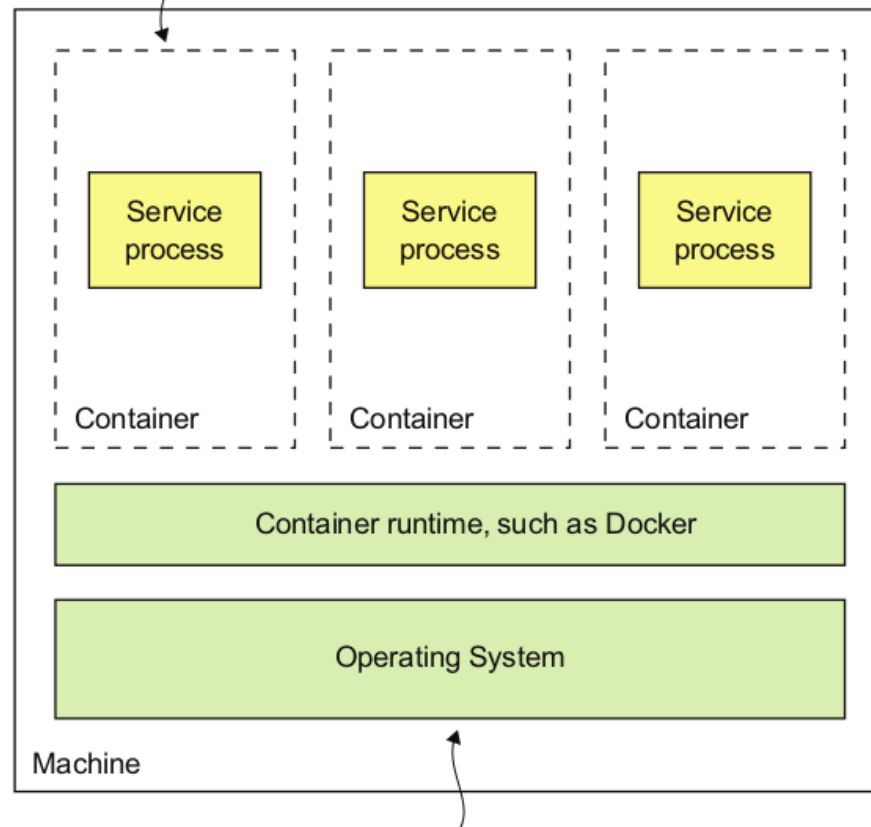
Deploy a service as a container

Pattern¹ : Déployé les services packagés comme des images de conteneur. Chaque service est un conteneur

Le packaging en image fait partie de la pipeline de déploiement

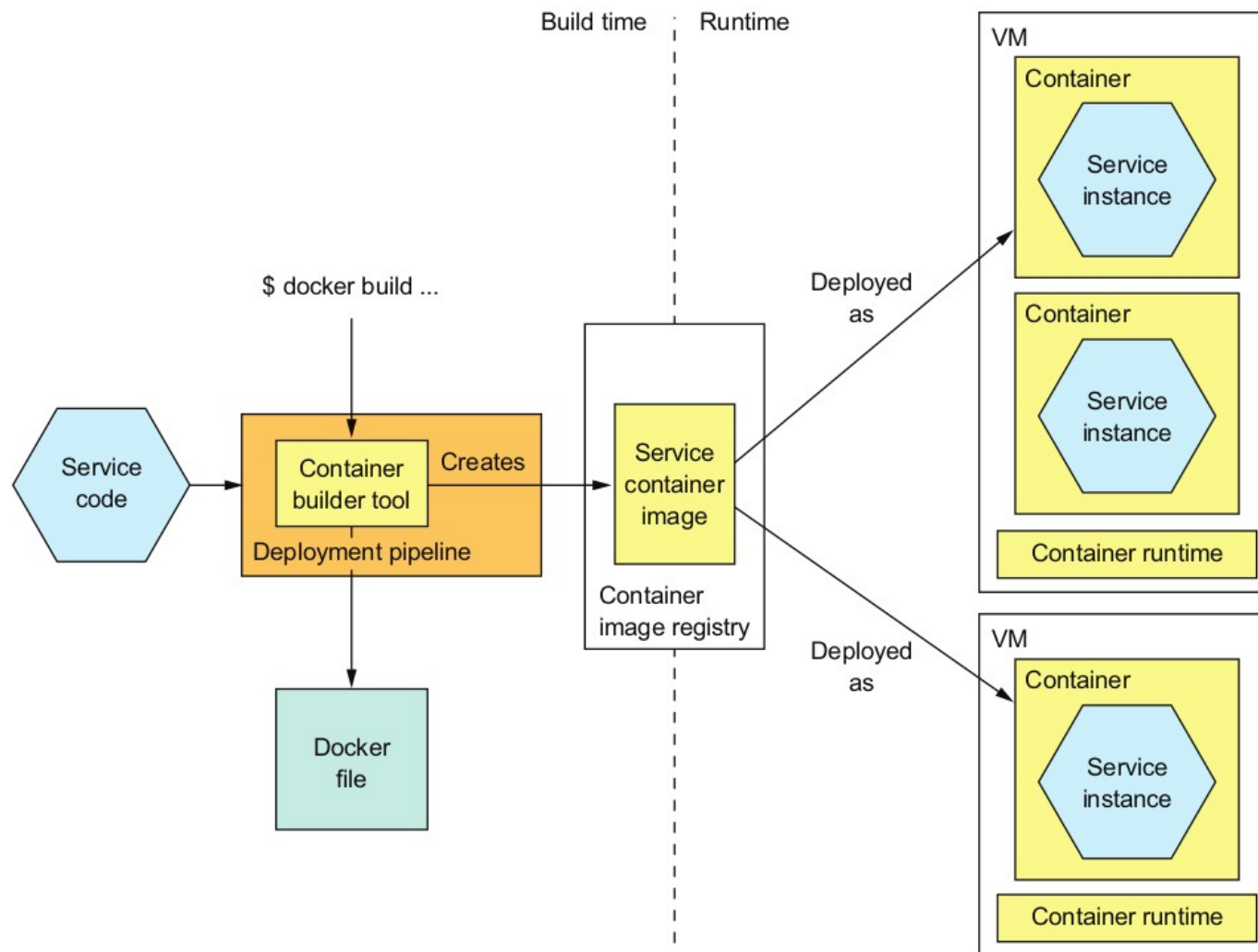
Exécution

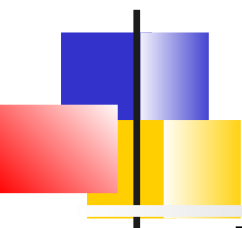
Each container is a sandbox that isolates the processes.



Shared by all of the containers

Déploiement





Bénéfices / Inconvénients

Bénéfices

Encapsulation de la pile technologique. Déploiements immuables

Les instances de service sont isolées.

Les ressources des instances de service sont limitées.

Inconvénients

Équipe DevOps responsable de l'administration des images du conteneur. (Patches de l'OS par exemple)

Administrer l'infrastructure du conteneur-runtime et éventuellement des VMs associés