

Architectures

JEE

Table des matières

I - Monde Java/JEE.....	9
1. JVM et JDK.....	9
1.1. Machine virtuelle JAVA.....	9
1.2. Historique et évolution.....	12
1.3. Principales particularités du langage JAVA.....	13
2. De Java SE à Java EE.....	16
2.1. JEE en tant qu'ensemble d'API & conteneur JEE.....	17
2.2. Principales API de JAVA (SE et EE).....	17
3. Formats de déploiement.....	18
4. Outils de développement (IDE et utilitaires).....	20
5. Avenir de l'écosystème Java.....	21
5.1. Tendances/directions.....	21
5.2. GraalVM.....	21
5.3. GraalVM Native Image.....	21
5.4. Spring Native.....	22

5.5. Quarkus (de RedHat).....	22
II - Architectures techniques.....	24
1. Revue des architectures courantes.....	24
1.1. N-tiers : Rôles/responsabilités de chaque partie.....	24
1.2. Couches logicielles classiques.....	27
1.3. Architectures logiques et physiques.....	27
1.3.a. Architectures logiques.....	27
1.3.b. Architectures physiques.....	27
1.4. évolution des architectures (début 21eme siècle).....	30
1.4.a. N-tiers JEE.....	30
1.4.b. SOA (architecture orientée services) : vers 2010.....	31
1.4.c. cloud-computing (essor rapide à partir de 2015).....	32
1.4.d. Architecture micro-services (vers 2020).....	33
2. Critères de qualité.....	34
2.1. Les principaux attributs de qualités.....	34
2.2. Quelques autres attributs de qualité.....	35
2.3. Infrastructure transactionnelle de JEE.....	37
2.4. Gestion déclarative des transactions.....	38
2.5. Sécurité J2EE/JEE5.....	39
3. Principaux choix techniques (vers 2020).....	41
3.1. Bases de données (relationnelles ou pas).....	41
3.2. Serveur d'application vs SpringBoot.....	41
3.3. synchone vs asynchrone.....	41
4. Comparaison avec .NET , NodeJs et Python.....	42
5. Tests unitaires et Test Driven Development.....	43
5.1. Présentation de JUnit 3,4,5.....	43
5.2. Présentation des anciennes versions 3 et 4.....	43
5.3. Lancement des tests unitaires.....	44
5.4. Présentation de JUnit 5 ("jupiter").....	44
5.5. Configuration maven pour JUnit 5 ("jupiter").....	44
5.6. Test Unitaire avec JUnit 5 (@Test, @BeforeEach, @AfterEach).....	45
5.7. Test Unitaire JUnit5 avec static (@BeforeAll , @BeforeAll).....	46
5.8. Spécificités de JUnit5.....	47
5.9. Suppositions (pour exécuter ou pas un test selon le contexte)	48
5.10. Tests imbriqués de JUnit5.....	49
5.11. Tests paramétrés via source/série de valeurs.....	49
5.12. Tests dynamiques / @TestFactory.....	50
5.13. Suite de Tests et Tags.....	51
5.14. Bonnes pratiques "TDD" (Test Driven Developement).....	52
6. Bénéfices de l'intégration continue.....	54
6.1. Objectifs de l'intégration continue.....	54
6.2. Présentation et installation de Jenkins.....	57
6.3. Configuration nécessaire lors du premier démarrage.....	58

6.4. Installation ou mise à jour de plugins pour Jenkins.....	58
6.5. Configuration élémentaire d'une tâche "jenkins".....	58
6.6. Différents types de "builds" (avec Jenkins).....	60
6.7. Exemple de pipeline jenkins.....	63
III - IHM Web.....	64
1. Les moteurs de Servlets.....	64
2. Java Server Pages.....	65
2.1. Pages JSP.....	65
2.2. TagLib (JSP) et JSTL.....	66
3. Modèles MVC.....	67
4. Session HTTP.....	67
5. De Struts à Java Server Faces.....	68
5.1. Struts 1.....	68
5.2. Framework JSF.....	69
5.2.a. Logique événementielle et navigation.....	70
5.2.b. Quelques traits de JSF.....	70
5.2.c. Arbre de composants JSF (racine=view).....	71
5.2.d. Implémentations de JSF.....	72
6. Wicket, Play,Struts2 ,Spring-MVC,GWT et les autres.....	73
7. Java web côté serveurs.....	75
8. Intégration Ajax et Single Page Applications.....	76
8.1. Appels de WS REST (HTTP) depuis js/ajax.....	78
8.2. Limitations Ajax sans CORS.....	82
8.3. CORS (Cross Origin Resource Sharing).....	83
IV - Clients Java.....	85
1. AWT, Swing et SWT.....	85
1.1. Exemple AWT.....	85
1.2. Exemple SWING.....	86
1.3. Exemple SWT.....	86
1.4. Eléments de base sur AWT/SWING et événements.....	87
1.4.a. Gestion des fenêtres et des contrôles.....	87
1.4.b. Composant de l'interface graphique (Component).....	87
1.4.c. Fenêtres "Container" et sous fenêtres.....	88
1.4.d. Les Contrôles (composants graphiques élémentaires).....	88
Libellé (étiquette).....	88
Champ de saisie simple (une seule ligne).....	88
Bouton poussoir.....	88
"Case à cocher" ou "bouton radio" accompagné de son libellé.....	89
Liste déroulante pour choisir un élément parmi n.....	89
Liste d'éléments (avec sélection multiple possible).....	89
Zone de saisie multi-lignes.....	89
Boite de dialogue.....	90
Menu.....	90

1.4.e. Fonctionnalités générales des composants SWING élémentaires:.....	90
1.4.f. Gestion simple du scrolling (JScrollPane).....	90
1.4.g. Boîte de message et Prompt.....	90
1.4.h. Onglets (JTabbedPane).....	91
1.4.i. Gestionnaire de répartition (LayoutManager).....	91
1.4.j. Positionnement absolu (sans LayoutManager).....	92
1.4.k. Classe Graphics – pour Dessiner (lignes, rectangles, ...).....	93
1.4.l. Gestion des événements.....	93
1.4.m. Notion d'abonnement.....	94
1.4.n. Adaptateurs.....	97
2. JavaFX.....	98
2.1. Structures fondamentales de <i>javaFx</i>	98
2.2. <i>Exemple élémentaire javaFx</i>	99
2.3. définitions <i>.fxml</i> et contrôleurs associés :.....	101
2.4. Autres fonctionnalités/spécificités de <i>javaFx</i> :	107
3. Android.....	108
3.1. Univers "Android".....	108
3.2. Développement d' <i>application android</i> (présentation).....	109
3.2.a. Android SDK.....	109
3.2.b. Android Studio (basé sur IntelliJ et gradle).....	109
3.3. Activité "android".....	109
3.4. Les ressources (<i>xml</i> , <i>png</i> , ...).....	115
3.5. Layouts.....	117
3.5.a. Principe et syntaxe XML (layout android).....	117
3.5.b. Principaux types de "Layout".....	119
3.6. Widgets simples.....	120
3.6.a. Label ("TextView").....	121
3.6.b. Zone de saisie ("EditText").....	121
3.6.c. Bouton poussoir (Button).....	121
3.6.d. Cases à cocher (non exclusives) / CheckBox.....	121
3.6.e. Boutons "radio" / RadioButton , RadioGroup.....	121
3.6.f. DatePicker (choix de date).....	122
3.7. Gestion des événements "android".....	123
3.7.a. Gestion élémentaire d'événement.....	123
3.7.b. Principaux événements.....	123
4. Déploiement Java Web Start (de java 5 à java 8).....	125
4.1. Mise en œuvre de <i>java web start</i>	125
4.2. Acquisition des permissions selon l'accord de l'utilisateur.....	126
4.2.a. Demander l'autorisation de lire un fichier.....	127
4.2.b. Demander l'autorisation d'écrire dans un fichier.....	127
4.2.c. Présentation des principaux services de JNLP.....	128
5. Quelques alternatives "non-java".....	128

V - Persistance.....129

1. JDBC.....	129
1.1. Présentation de <i>JDBC</i>	129
1.2. Paramétrage et établissement d'une connexion.....	130

1.3. Driver/Pilote JDBC et URL de connexion.....	131
1.4. Pool de connexions et DataSource JDBC.....	132
1.5. Utilisation de JDBC pour lancer des requêtes SQL.....	133
1.6. Gestion des transactions (tout ou rien).....	134
1.7. Préparer et lancer n fois un ordre Sql paramétrable.....	134
1.8. Autres fonctionnalités importantes de JDBC.....	135
2. Problématique "O.R.M."	136
2.1. Objectif & contraintes:.....	136
2.2. Eléments techniques devant être bien gérés.....	137
3. JPA et JPA 2 (Java Persistance Api).....	137
3.1. Plusieurs implémentations disponibles pour JPA.....	138
3.2. Exemple d'entité persitante (@Entity).....	139
3.3. Différents états - objet potentiellement persistant.....	140
3.4. Cycle de vie d'un objet JPA/Hibernate.....	141
3.5. Synchronisation automatique dans l'état persistant.....	142
3.6. objet persistant et architecture n-tiers.....	142
3.7. Principales méthodes JPA / EntityManager et Query.....	143
3.8. Contexte de persistance et proxy-ing.....	144
3.9. Relations (1-1, n-1, 1-n et n-n).....	145
3.9.a. Relations 1-n et n-1 (entité-entités).....	145
3.9.b. Relation n-n ordinaire.....	146
3.10. JPA et Héritage.....	148
3.11. Api "Criteria" de JPA 2.....	149
3.11.a. Apports et inconvénients de l'API "Criteria"	149
3.11.b. Exemple basique (pour la syntaxe et le déclenchement).....	149
3.12. Approches top-down , down-top ,	149
3.12.a. Configuration JPA 2.2 pour approche top-down.....	150
3.12.b. Outils pour approche down-top.....	150
4. Hibernate et les ORM.....	151
4.1. Hibernate.....	151
5. Détails d'une couche de persistance.....	152
5.1. D.A.O. (Data Access Object).....	152
5.1.a. Principe.....	152
5.1.b. DAO génériques ou DAO Spécifiques.....	153
6. Bases SQL et NoSQL.....	153
6.1. Rappel (ACID).....	153
6.2. Bases de données SQL (relationnelles) / SGBDR.....	154
6.3. Bases NoSQL (Not only SQL).....	154
NB : chaque technologie NoSQL gère à sa façon les aspects précédents (atomicité , consistance , durabilité) . On ne se passe évidemment pas de tout ça	156
C'est essentiellement l'aspect "synchronisation non immédiate" qui est mis en avant	156
6.4. Types de bases de données NoSQL :.....	157
6.5. SQL vs NoSQL.....	158
6.6. Basic NoSQL (clé,valeurs).....	159

6.7. Orienté document.....	160
6.8. Orienté colonnes.....	161
VI - Communication.....	162
1. SOAP Web services avec JAX-WS.....	162
1.1. Protocole SOAP.....	163
1.2. WSDL (Web Service Description Langage).....	166
1.3. Api java JAX-WS.....	168
1.3.a. Mise en oeuvre de JAX-WS (coté serveur).....	169
1.3.b. Implémentation sous forme d'EJB3 (pour serveur JEE5+).....	170
1.4. Tests via SOAPUI.....	171
1.5. JAX-Ws coté client.....	172
2. REST Services avec JAX-RS.....	173
2.1. Caractéristiques clefs des web-services "REST" / "HTTP".....	173
2.2. Fondamentaux sur Web Services "R.E.S.T."	174
2.3. Statuts HTTP (code d'erreur ou ...).....	176
2.4. Safe and idempotent REST API.....	177
2.5. API java pour REST (JAX-RS).....	178
2.6. Code typique d'une classe java (avec annotations de JAX-RS).....	178
2.7. Configuration de JAX-RS intégrée à un projet JEE6/CDI.....	180
3. JNDI.....	181
4. L'api javax.mail.....	183
5. Messaging asynchrone avec JMS.....	184
5.1. Présentation de JMS.....	184
5.2. JMS 1.1.....	185
5.3. Exemples (fragments) de code JMS 1.1.....	186
5.3.a. Obtention de l'objet ConnectionFactory via JNDI:.....	186
5.3.b. Obtention d'une file de message via JNDI:.....	186
5.3.c. Création d'un objet Connection via l'usine:.....	187
5.3.d. Création d'une Session à partir de l'objet Connexion:.....	187
5.3.e. Obtention de l'objet "MessageProducer" pour envois.....	187
5.3.f. Obtention de l'objet "MessageConsumer" pour réceptions.....	187
5.3.g. Déclencher le début possible des réceptions de messages:.....	188
5.3.h. Création de messages.....	188
5.3.i. Envoi et réception.....	188
5.3.j. Extraction des valeurs d'un message.....	189
5.4. Champs des entêtes d'un message JMS.....	189
5.5. Liste des principaux "Provider JMS".....	190
5.6. EJB3 de type MDB.....	190
5.7. JMS2 (version simplifiée de JMS 1.1).....	191
6. JNI et JNA.....	193
6.1. Aperçu sur JNI.....	193
6.1.a. Déclaration et appel d'une méthode native.....	193
6.1.b. Implémentation d'une méthode native.....	193
6.1.c. Etablir le lien entre JAVA et le code C:.....	194
6.2. Exemple JNA.....	195

VII - EJB et Serveurs d'applications.....198

1.	Versions des principales API de JEE / Jakarta EE.....	198
1.1.	Variantes "Full-Profile" et "Web-Profile" :.....	198
1.2.	Tableau des principales versions.....	199
1.3.	Namespaces XML pour les fichiers de configuration.....	200
2.	Serveurs d'applications Java EE.....	201
3.	Evolution de JEE.....	204
4.	Spécificités selon le serveur d'application JEE.....	205
4.1.	Principaux serveurs d'applications (JEE).....	206
4.2.	Serveur particulier "TomEE" (Tomcat EE).....	206
5.	De EJB2 à EJB 3 à EJB 4.....	207
6.	EJB (Enterprise JavaBean).....	208
6.1.	Entity et Sessions Beans.....	209
6.2.	Qualités (A.C.I.D.) d'une transaction distribuée basique.....	211
6.3.	Protocole XA pour le commit à 2 phases.....	212
6.4.	Effets du contexte transactionnel sur les EJB.....	213
6.5.	Attributs transactionnels sur EJB (approche déclarative).....	213
6.6.	Annotations sur EJB3 concernant les transactions.....	213
6.7.	Comparaison entre transactions Spring et EJB/JEE.....	214
7.	Spring vs CDI.....	214

VIII - Ecosystème Spring.....215

1.	Présentation du framework Spring.....	215
1.1.	Historique et évolution de Spring.....	215
1.2.	Architecture / Ecosystème Spring.....	216
1.3.	Configuration Spring-boot simple.....	218
1.4.	Test unitaire Spring simple.....	219
1.5.	DAO automatiques via Spring-Data.....	220
1.5.a.	Spring-data-commons.....	220
1.6.	Quelques différences/correspondances entre Spring et JEE officiel.....	224

IX - Cloud java et microservices.....225

1.	Architecture micro-services.....	225
1.1.	Grands traits de l'architecture micro-services.....	226
1.2.	Docker , kubernetes ,	233
1.3.	Docker et notion de conteneur.....	234
1.4.	"Container" vs "VM".....	236
1.5.	Isolation des conteneurs.....	237
1.6.	Référentiel d'images "docker" prêtes à l'emploi.....	238
1.7.	"Docker workflow" (utilisation classique).....	238
1.8.	Spécificités de "docker".....	239
1.9.	Kubernetes.....	240

1.9.a. Accès externe à un cluster kubernetes via LB7 / Ingress.....	244
1.10. <i>Api-gateway</i>	245
1.11. <i>Token JWT et OAuth2/OIDC</i>	251
1.12. <i>Resynchronisations asynchrones (event sourcing)</i>	259
2. Java et le cloud.....	260
2.1. <i>Spring-cloud</i>	260
2.2. <i>JEE micro-profile</i>	261
2.3. <i>Cas particulier: serveur "openLiberty"</i>	263

I - Monde Java/JEE

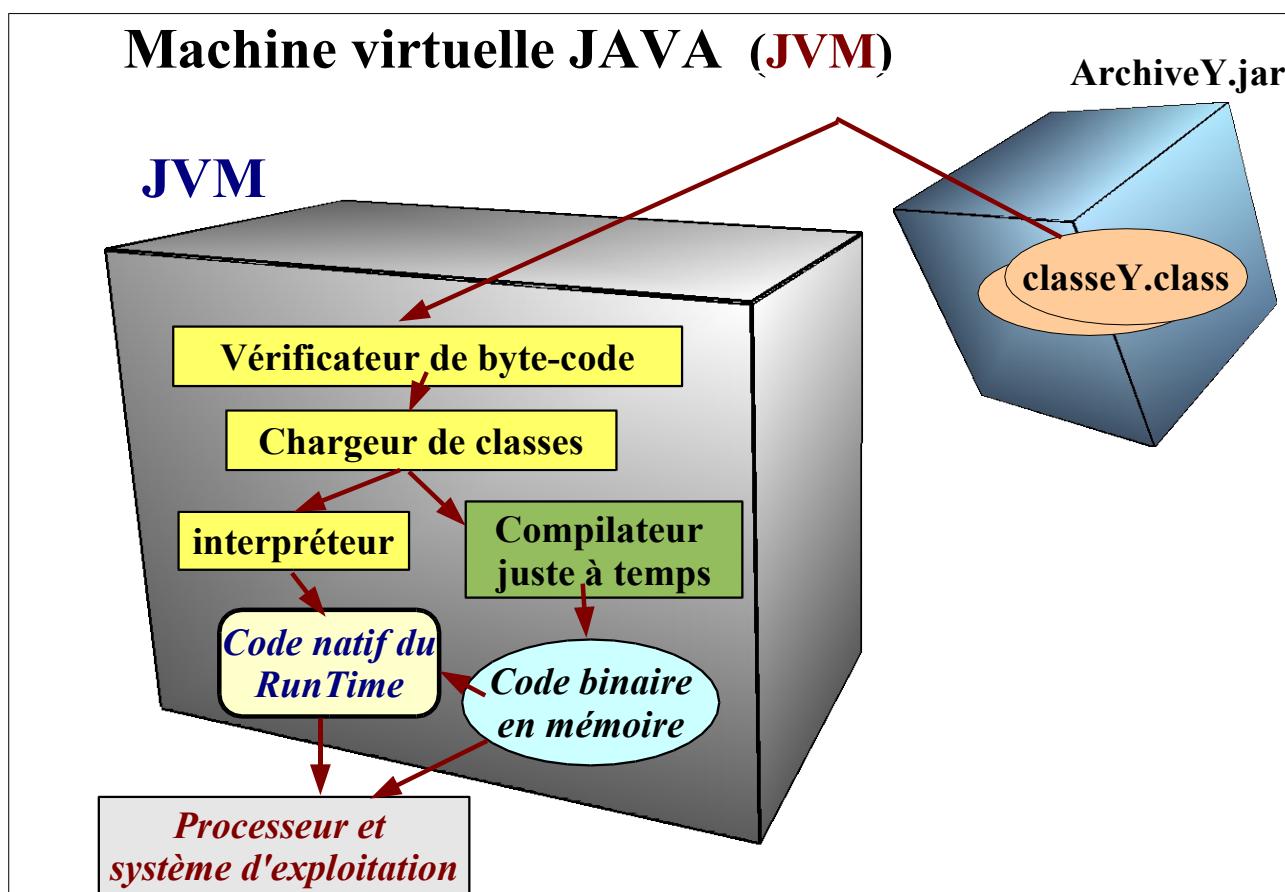
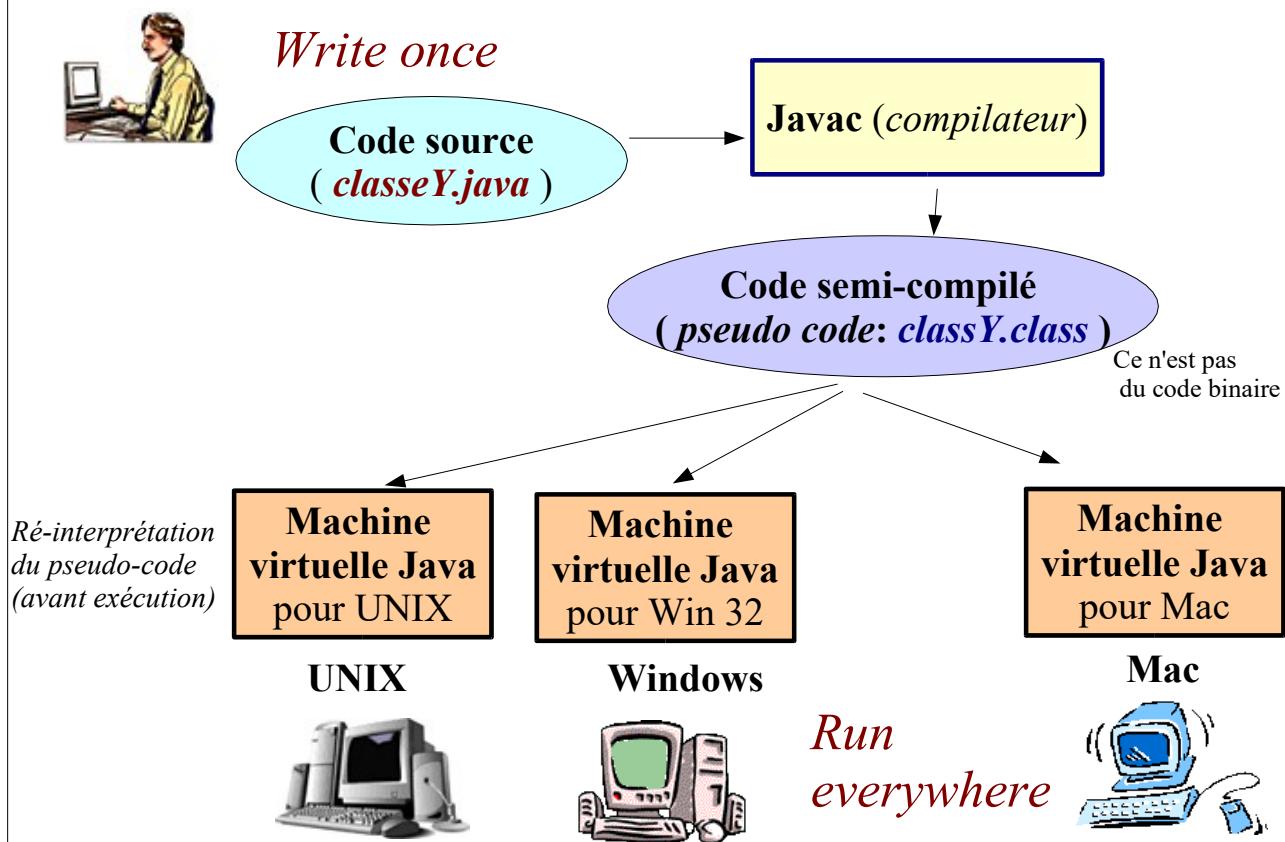
1. JVM et JDK

1.1. Machine virtuelle JAVA

La principale particularité du langage **JAVA** est d'être **multi-plateforme**:

- 1) Le développeur écrit du code source (*classeY.java*) puis le compile sur n'importe quelle sorte de machine (Windows , Unix , Linux ,).
- 2) Cette compilation partielle ne génère pas un code binaire compréhensible que par un certain type de machine mais génère un pseudo-code portable appelé "byte-code" : *classeY.class* .
- 3) Ce pseudo-code portable est ensuite packagé dans des archives (.jar) puis distribué à travers le réseau vers différentes sortes de machines.
- 4) N'étant pas dédié à un type précis de machine, ce pseudo code portable a besoin d'être ré-interprété par une application spécifique appelée "**machine virtuelle java**" .
A chaque type de plate-forme, correspond une version spécifique de la machine virtuelle java (généralement packagée dans le **JRE = Java Runtime Environnement**).

Java: langage *partiellement compilé et interprété* (via JVM)



NB:

- Il n'y a pas d'édition de liens à effectuer préalablement (les classes sont chargées au fur et à mesure des besoins du programme lancé au sein de la machine virtuelle) .
- Le **compilateur juste à temps** permet d'**améliorer sensiblement la vitesse d'exécution** d'une application java et est systématiquement activé (sauf option contraire).

Principales contraintes liées aux mécanismes de JAVA:

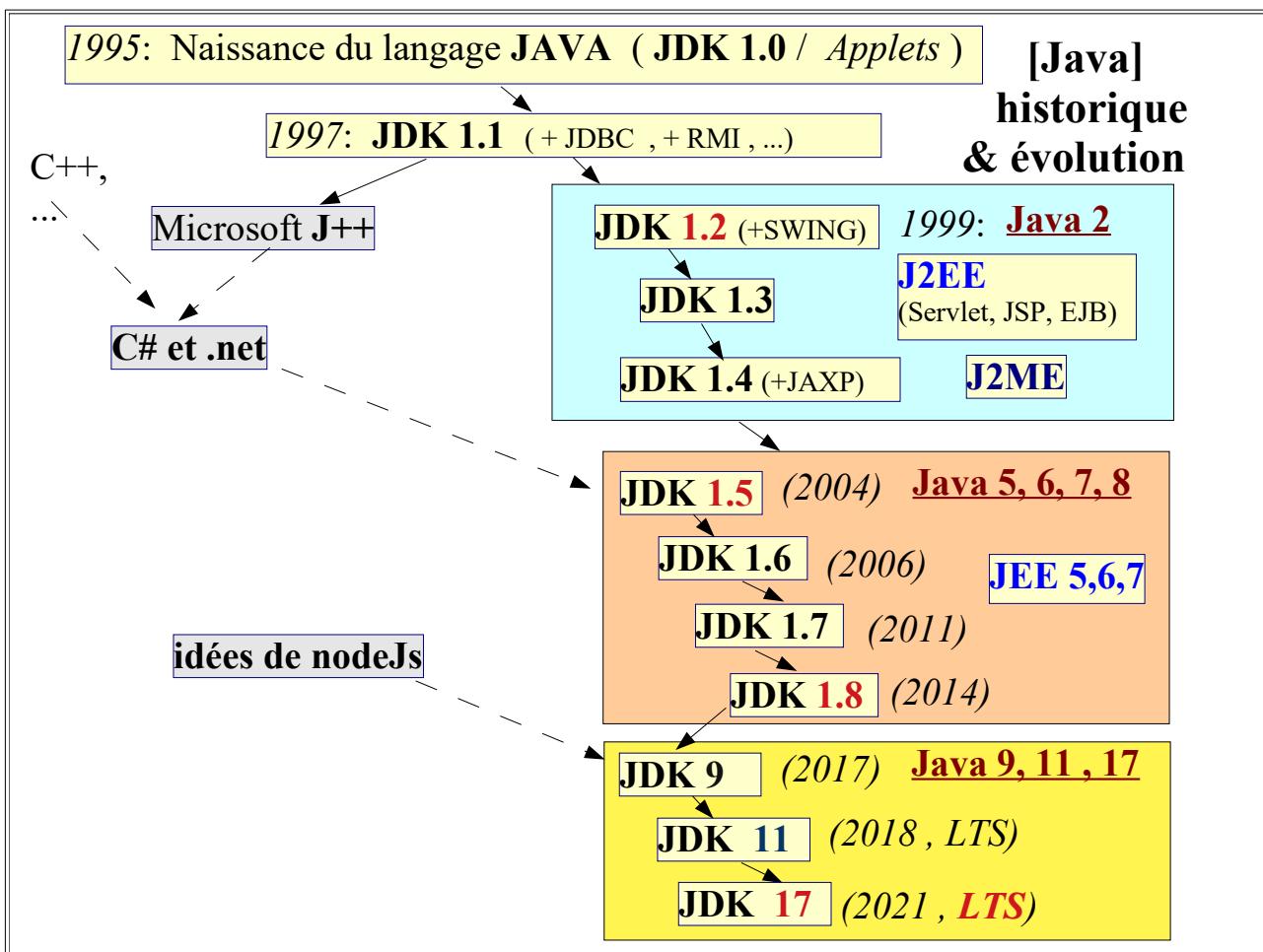
- L'initialisation de la machine virtuelle, le chargement des classes en mémoire ainsi que la compilation juste à temps sont des opérations assez lourdes qui occasionnent des **temps de démarrage relativement longs**.
- L'ensemble des constituants d'une application Java chargée au sein d'une machine virtuelle **occupe une assez grande place en mémoire vive**.

==>

Ceci explique que **Java est beaucoup utilisé coté serveur** (là où une grande consommation mémoire et un temps élevé de démarrage sont moins gênants) .

L'utilisation de java coté client (*ex*: applet , interfaces graphiques Swing, java web start, ...) nécessite des ordinateurs relativement puissants (rapides et bien dotés en mémoire vive).

1.2. Historique et évolution



JDK signifie *Java Development Kit*.

Les JDK 1.2 et 1.5 ont apportées de grandes nouveautés qui ont modifié le langage en profondeur.

Le terme plate-forme **Java 2** désigne toutes les versions de *Java à partir du JDK 1.2* et englobe également une extension pour les serveurs d'applications : *J2EE (Java 2 Enterprise Edition)*.

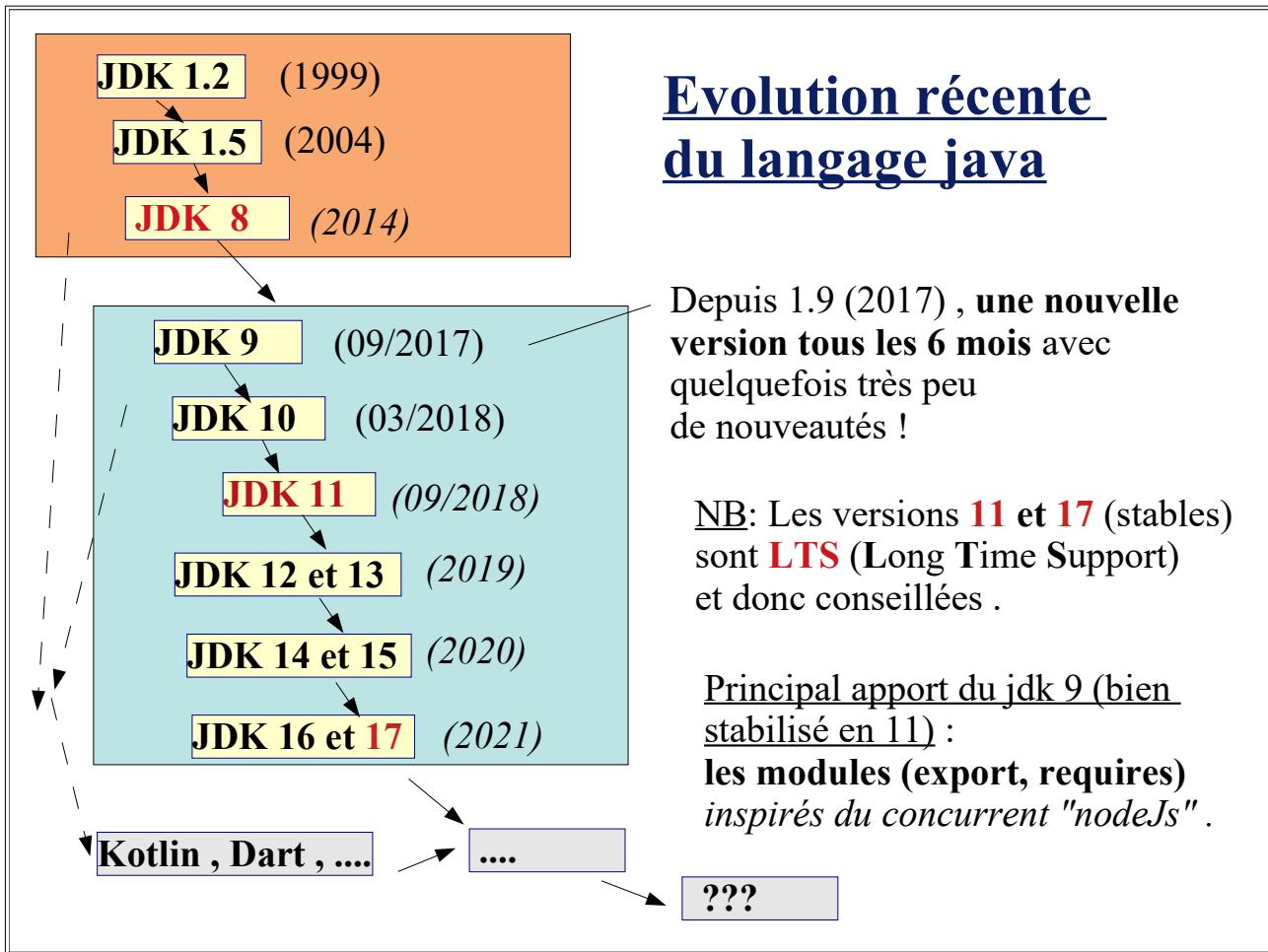
Bien que différents et incompatibles les langages Java et C# comportent beaucoup de points communs (syntaxe et architecture assez proches).

La société **SUN MicroSystem** qui a inventé le langage **JAVA** est le propriétaire officiel du langage et décide de son évolution (en tenant compte des avis de ses partenaires).

L'entreprise "SUN" a été rachetée par "Oracle".

Oracle est maintenant le **nouveau propriétaire** de "Java".

La version **1.8 du jdk** a apporté quelques grandes nouveautés syntaxiques (**lambda expressions**, streams, ...) et propose java-fx à la place de swing .



NB: à partir du jdk 1.11 , java-fx est devenu une extension facultative (désormais plus inclue dans le jdk) .

NB : a coté du jdk officiel (de l'entreprise Oracle) , il existe également le "openJdk" entièrement "open source" (un peu moins complet que le jdk d'Oracle mais sans partie potentiellement payante en production) .

1.3. Principales particularités du langage JAVA

Langage générique <i>[large palette d'applications]</i>	<u>Le langage Java peut servir à créer différentes sortes d'entités:</u> <ul style="list-style-type: none"> ◆ Applications autonomes (ne nécessitant qu'une JVM) ◆ JavaBean = composants quelconques (pour composer une interface graphique ou pour effectuer des traitements "métier"). ◆ Servlet = composant permettant de générer des pages WEB (<i>Un servlet s'exécute côté serveur</i>)
---	---

	<ul style="list-style-type: none"> ◆ Applet (mini application s'exécutant côté client , l'affichage s'effectue dans une sous fenêtre du navigateur internet) ◆ Application native android , ...
Complètement orienté objet	<p>Java est un langage résolument orienté objet .</p> <p>Les fonctions globales n'existent pas en java , <i>toutes les fonctions sont obligatoirement intégrées dans des objets</i> .</p> <p>Offrant un support à tous les principaux concepts objets (classe, instance, encapsulation, polymorphisme , héritage , ...) , Java permet de très bien structurer les programmes (==> bonne modularité).</p>
Relativement simple	<p>La gestion des références (qui sont plus simples à manipuler que les pointeurs) est en grande partie automatisée: <i>un ramasse miette libère automatiquement les blocs mémoires devenus inutiles</i>.</p> <p>Bien que yntaxiquement proche du C++ , java ne s'est inspiré que des éléments fondamentaux et simples de ce langage .</p>
Souple, expressif et bien adapté au couches hautes	<p>Souple , modulaire et étant doté de beaucoup d' API prédéfinies , Java est un langage de préférence pour les développements "3-tiers" liés à l'informatique de gestion et à internet.</p> <p>Par contre, JAVA n'est (pour l'instant) pas du tout approprié pour coder des couches de bas niveau:</p>
Vitesse d'exécution correcte et interfaçage possible avec le langage C	<p>Les langages C et C++ restent incontournables pour coder des traitements pointus devant s'exécuter très rapidement sur une plate-forme spécifique (ex: informatique industrielle, calcul scientifique, ..)</p> <p>Java peut s'interfacer avec du code C ou C++ via JNI (Java Native Interface) et des DLL locales ou via le protocole SOAP des services WEB.</p>
Robuste et fiable	<p>Langage fortement typé ==> <u>beaucoup d'erreurs détectées dès la compilation</u>.</p>

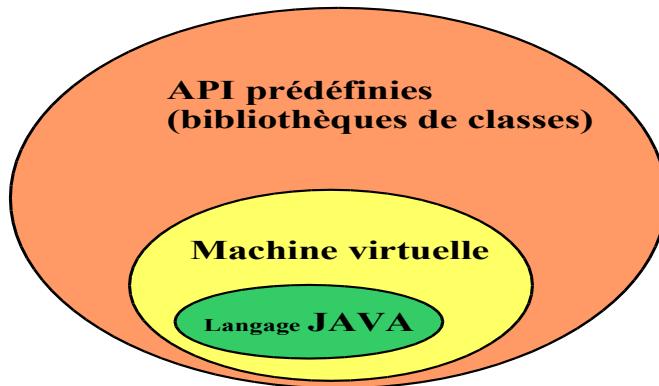
	Gestion des exceptions bien structurée (try/catch + pile d'appels ==> "Debug" simple et rapide).
Introspection	<p>Certains <u>mécanismes prédéfinis du langage JAVA permettent de récupérer automatiquement une description d'une classe Java</u> (<i>liste des attributs et des fonctions internes , types des paramètres , ...)</i> même si celle-ci n'est disponible que sous la forme compilée (<i>sans code source</i>) .</p> <p>Ceci constitue un gros point fort (vis à vis du C++) et permet d'automatiser certains traitements (persistance des données , proxy dynamique , ...)</p>
Prise en charge (en standard) du multi-threading	<p>==> Java permet d'écrire du code ré-entrant que l'on peut écrire de façon portable (sans être dépendant d'un système d'exploitation).</p>

2. De Java SE à Java EE

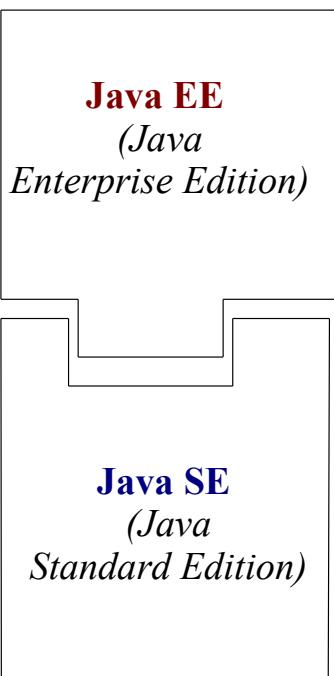
Le langage **JAVA** est associé à une machine virtuelle qui le rend portable sur une large palette de plate-formes (UNIX, Windows ,) et est accompagné d'un immense ensemble d'API (bibliothèques de classes) qui sont:

- **standards** (officialisées par SUN/Oracle , JEE , ...)
- **portables** (utilisables sur une multitude de systèmes [Linux , Windows, ...])
- **très souvent gratuites** (Open source ou ...)

Java est donc bien plus qu'un langage informatique , c'est une véritable plate-forme virtuelle:



(plate forme Java) JavaSE / JavaEE et JavaME



API supplémentaires (.jar) :
pour serveurs d'application

- Servlet / JSP (aspects web),
- JNDI (accès aux serveurs de noms) ,
- EJB (objets métiers en java)
- ...

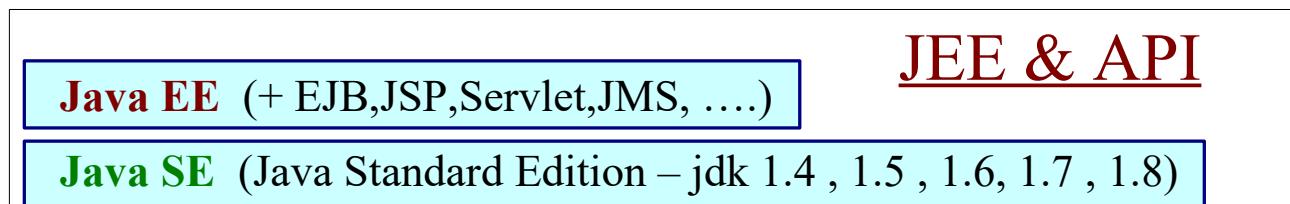
(Java Development Kit)
→ Compilateur java
+
JRE (Java Runtime Environment)
--> **machine virtuelle Java**
--> (API standards + code natif)

NB: il existe aussi **JavaME (Micro Edition)** = **JavaSE simplifié/allégé**
pour les **cartes à puce** (ex : carte bancaire) .

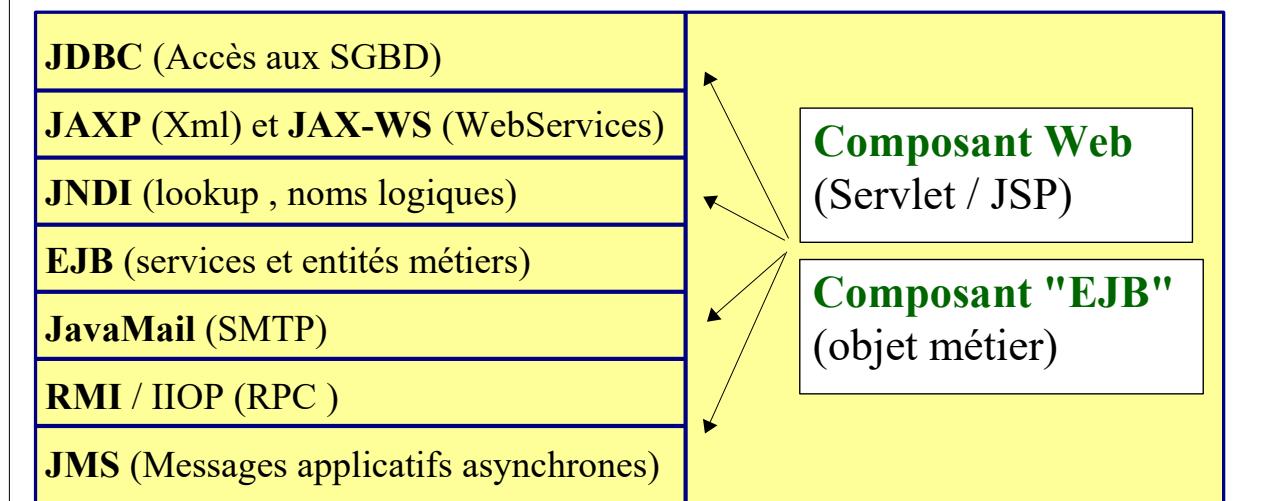
2.1. JEE en tant qu'ensemble d'API & conteneur JEE

JEE (*signifiant Java Enterprise Edition*) peut être vu comme un ensemble d'API permettant de développer des applications évoluées à déployer sur un serveur d'entreprise.

Les API de JEE se rajoutent à celles du JDK (base JSE) . Elles concernent essentiellement les aspects "présentation WEB" , "EJB" , ... et "Services Web" .



(JSE + JEE) peut être vu comme un modèle d'architecture basé sur des "container" qui offrent des services techniques orthogonaux aux composants métiers:



2.2. Principales API de JAVA (SE et EE)

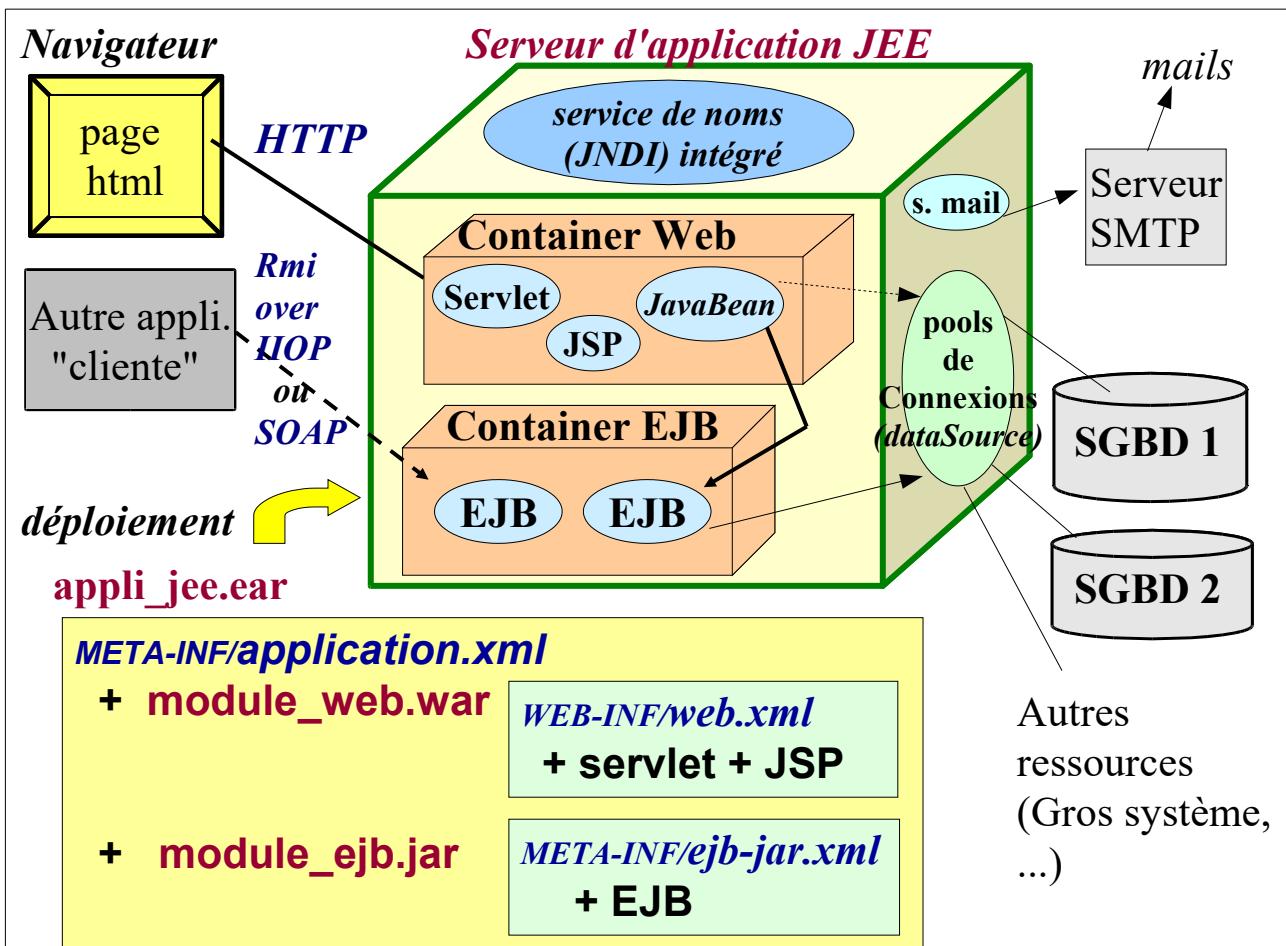
Api	intégration	fonctionnalités
JDBC	API intégré dans SE mais <i>Driver JDBC à récupérer</i>	Accès générique aux bases de données relationnelles (==> requêtes SQL)
AWT	SE (<i>depuis JDK 1.0</i>)	Bases du graphisme et du multi-fenêtrage
SWING	SE (<i>depuis JDK 1.2</i>)	Graphisme 2D et contrôles graphiques 100% java
javaFx	Hors SE (<i>depuis JDK 11</i>)	Api graphique plus moderne
SERVLET / JSP	JEE	génération de pages HTML (nécessite un conteneur Web du type Tomcat)
RMI (Remote Method Invocation)	SE (<i>depuis JDK 1.1</i>)	Appels de fonctions à travers le réseau

EJB (Enterprise Java Bean)	JEE	Objets "métier" en java (nécessite serveur JEE ex: WebSphere_AS, JBoss_AS, ...)
JPA 1 et 2	JEE	Java Persistance Api (ORM / hibernate)
DI , CDI	JEE	Injection de dépendances
JNDI	J2SE (<i>depuis JDK</i>)	Accès à des serveurs de noms (LDAP, ...)
JMS	JEE	Interface java pour MiddleWare orienté message asynchrone.
JAXP (Java Api for Xml Processing)	SE (<i>depuis JDK 1.4</i>)	Parsing XML (SAX & DOM) + transformations XSLT
JAX-WS, JAX-RS		api officielles pour Web-services (SOAP et REST)
...		

NB : à coté des spécifications officielles "JavaEE" , il existe un très bon framework "**Spring**" (standard pas officiel mais standard de fait --> écosystème java très complet) .

3. Formats de déploiement

JEE peut également être vu comme un **modèle d'architecture** pour les **serveurs d'applications**. Les **spécifications JEE** indiquent clairement le rôle des "**container**" : Ceux-ci doivent offrir aux composants applicatifs qu'ils hébergent un accès normalisé aux API standards de JEE . Autrement dit , un **composant JEE** (*ex*: servlet , EJB, ...) fonctionne exactement de la même manière au sein des serveurs WebLogic , JBoss ou WebSphere car il peut appeler les mêmes fonctionnalités (mêmes API) et qu'il expose lui même les mêmes points d'entrées pour la gestion de son cycle de vie.



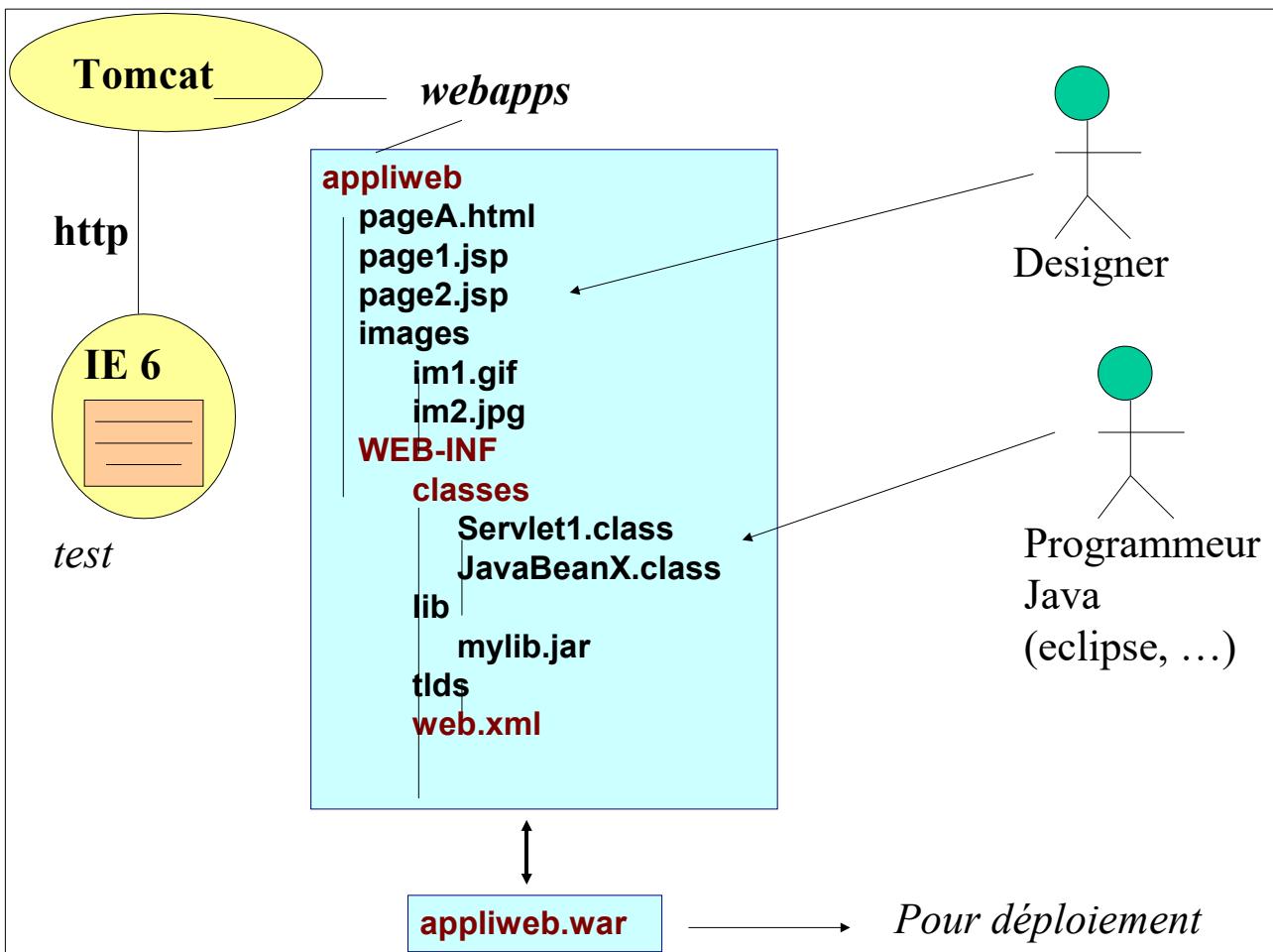
Depuis J2EE 1.2 , le **déploiement d'une application JEE est standardisé**:

- Un fichier ".war" (pour **Web ARchive**) contient tous les composants "web" et les fichiers de configurations associés (**WEB-INF/web.xml** , ...).
- Un fichier ".jar" (pour **Java ARchive**) contient tous les composants "EJB" et les fichiers de configurations associés (**META-INF/ejb-jar.xml** , ...).
- Un fichier ".ear" (pour **Enterprise ARchive**) regroupe différentes sous archives ("war", ".jar" , ...) et un fichier de configuration globale : **META-INF/application.xml** dont la balise **context-root** de l'**application WEB** indique l'**URL** relative de celle-ci.

Au lieu de parler de J2EE 1.5 , Sun/JavaSoft a préféré baptiser **JEE5,6,7** les nouvelles versions des spécifications de sa plate-forme Java de niveau entreprise .

Les principaux apports de ces nouvelles versions sont les suivants:

- **EJB3 , 3.1 , 3.2** (avec api **JPA 1 , 2.0 , 2.1** pour la persistance des données).
- Nouveau support des **services WEB "Soap"** via l'api **JAX-WS** (mieux que JAX-RPC)
Support des **services WEB "REST"** avec l'api **JAX-RS** (v1.0 pour JEE6 v2.x pour JEE7)
- Intégration du framework **JSF** (1 puis 2) dans la partie WEB
- **Partie Web supportant l'injection de dépendances (@Ejb , @Resource , @Inject CDI)**
- ...



4. Outils de développement (IDE et utilitaires)

Le **JDK (JRE + compilateur en mode texte)** ne suffit pas pour programmer de façon confortable.

Un *environnement de développement graphique* intégrant au minimum un *éditeur* et une *gestion automatisée des compilations* permet d'être efficace durant les phases de programmation .

Eclipse (*Open Source*) , **NetBeans** (*de Sun/Oracle*) et **IntelliJ** (*de JetBrains*) sont actuellement les trois **IDE Java** qui sont les plus utilisés .

Dans le cadre d'un projet d'entreprise sérieux, **un outil de gestion de version (CVS,SVN ou GIT)** permet de stocker/centraliser le code de tous les développeurs dans un **référentiel partagé en commun** .

En outre le produit "**Maven**" est très souvent utilisé pour **gérer les librairies java ("jar") et leurs inter-dépendances** .

5. Avenir de l'écosystème Java

5.1. Tendances/directions

- De plus en plus d'asynchronismes (ex : RxJava ou Reactor)
- Optimisation de la VM (ex : GraalVM , NativeImage) pour démarrer plus rapidement et optimiser la consommation mémoire
- prise en compte des besoins du cloud (java pour les microservices)

5.2. GraalVM

GraalVM est une Machine Virtuelle (VM), Open Source, issue d'un projet de recherche commencé il y a plus de 10 ans chez Oracle Labs (anciennement Sun Labs) .

Cette nouvelle VM est maintenue par une communauté d'acteurs majeurs du net (Oracle, Amazon, Twitter, RedHat notamment avec Quarkus, VMWare pour l'intégration de son framework Spring, ...).

C'est une **nouvelle génération** de VM, **Polyglotte**, c'est à dire qu'elle supporte de nombreux langages, même ceux qui ne génèrent pas de bytecode. A terme, elle pourrait remplacer l'actuelle VM HotSpot.

Techniquement :

- La VM **GraalVM** est couplée à un nouveau compilateur, **Graal**, écrit entièrement en Java (ce qui permet une compilation cyclique) :
- Il vise à remplacer le compilateur C2 utilisé pour le **JIT** de la VM **HotSpot** et qui est arrivé en fin de vie car trop complexe à faire évoluer (mélange d'assembleur, C, Java)
- Le compilateur Graal peut aussi faire de la compilation **AOT** (Ahead-Of-Time, à l'avance) aussi appelée compilation anticipée.

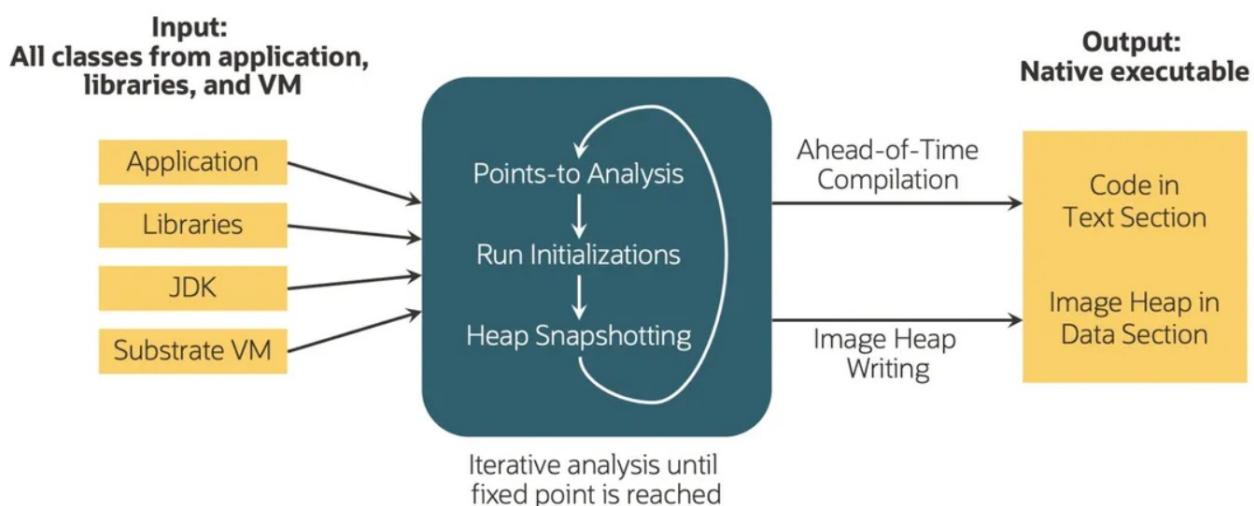
5.3. GraalVM Native Image

- **GraalVM Native Image** est une technologie de compilation ahead-of-time qui **génère des exécutables natifs**.
- **Les exécutables natifs sont idéaux pour les conteneurs et les déploiements cloud car ils sont petits, démarrent très rapidement et nécessitent beaucoup moins de CPU et de mémoire.**
- GraalVM Native Image bénéficie d'une adoption importante avec le support des principaux frameworks Java tels que Spring Boot, Micronaut, Quarkus, Gluon Substrate, etc.

Dans les grandes lignes , avec une JVM classique , la compilation et l'exécution se produisent en même temps (lors du démarrage) avec une longue période de chauffe .

Avec GraalVM Native Image, en mode AOT, le compilateur effectue toutes les compilations (jusqu'au stade binaire) pendant la construction et avant l'execution .

L'utilitaire GraalVM 'native-image' prend le bytecode Java en entrée et produit un exécutable natif. Pour ce faire, l'utilitaire effectue une analyse statique du bytecode sous une hypothèse de monde fermé. Lors de l'analyse, l'utilitaire recherche tout le code que votre application utilise réellement et élimine tout ce qui est inutile(un peu comme un bundle javascript généré par webpack et angular).



Pour approfondir le sujet :

<https://scalastic.io/graalvm-microservices-java/>

<https://www.infoq.com/fr/articles/native-java-graalvm/> (source de l'image ci-dessus)

5.4. Spring Native

En 2022, le framework spring est en train d'intégrer "GraalVM Native Image" au sein des dernières versions Spring5/SpringBoot2 et des nouvelles versions Spring6/SpringBoot3 .

Article :

<https://www.infoq.com/fr/articles/native-java-spring-boot/>

<https://docs.spring.io/spring-native/docs/current/reference/htmlsingle/>

5.5. Quarkus (de RedHat)

<https://quarkus.io/>

<https://www.redhat.com/fr/topics/cloud-native-apps/what-is-quarkus>

<https://www.ionos.fr/digitalguide/serveur/configuration/quest-ce-que-quarkus/>

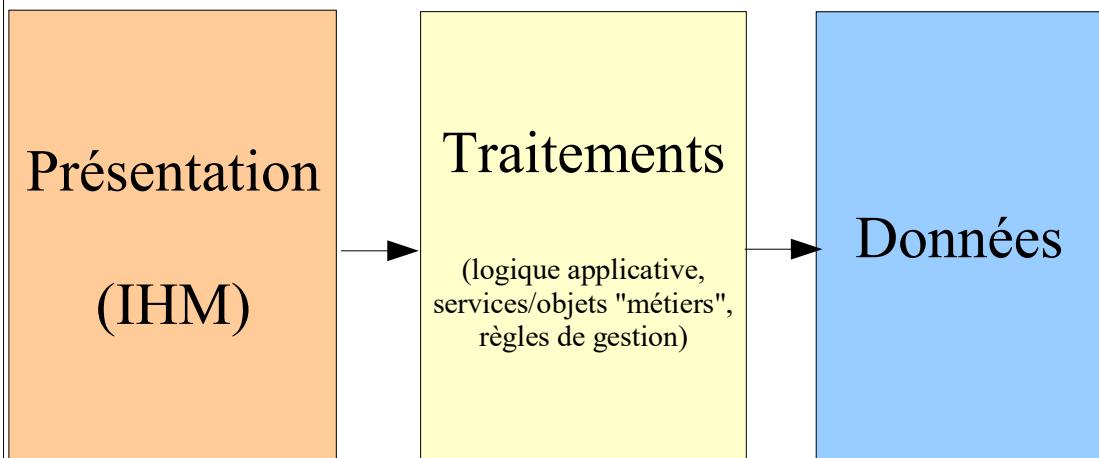
- La JVM et le JDK
- De Java SE à Java EE
- Avenir de l'écosystème Java
- Formats de déploiement
- Outils de développement

II - Architectures techniques

1. Revue des architectures courantes

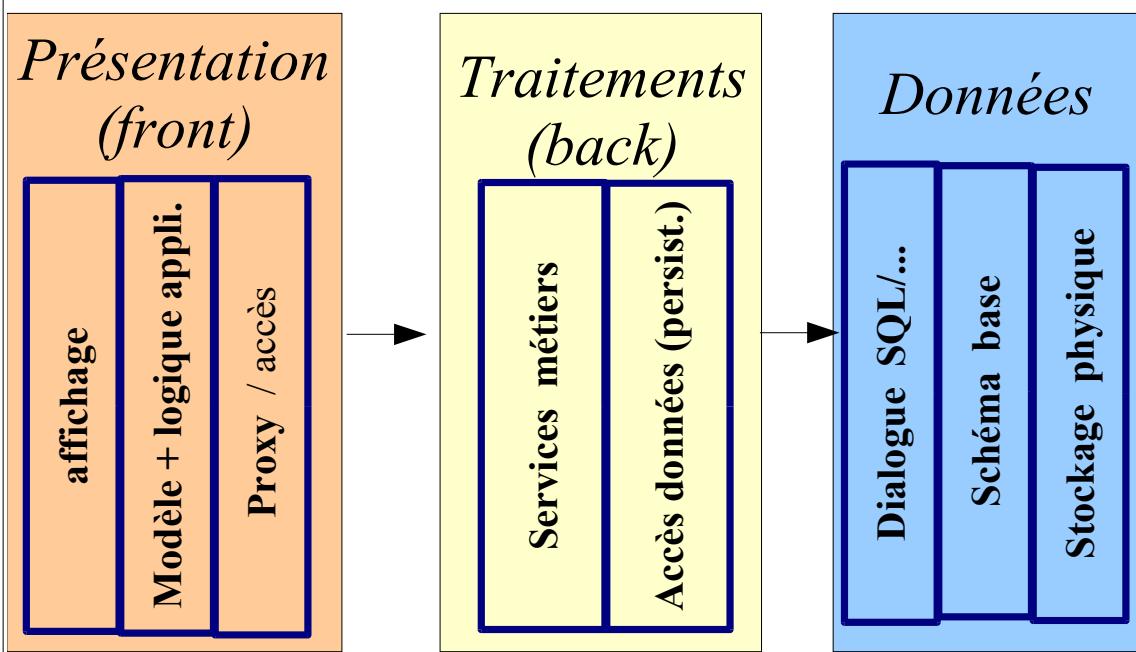
1.1. N-tiers : Rôles/responsabilités de chaque partie

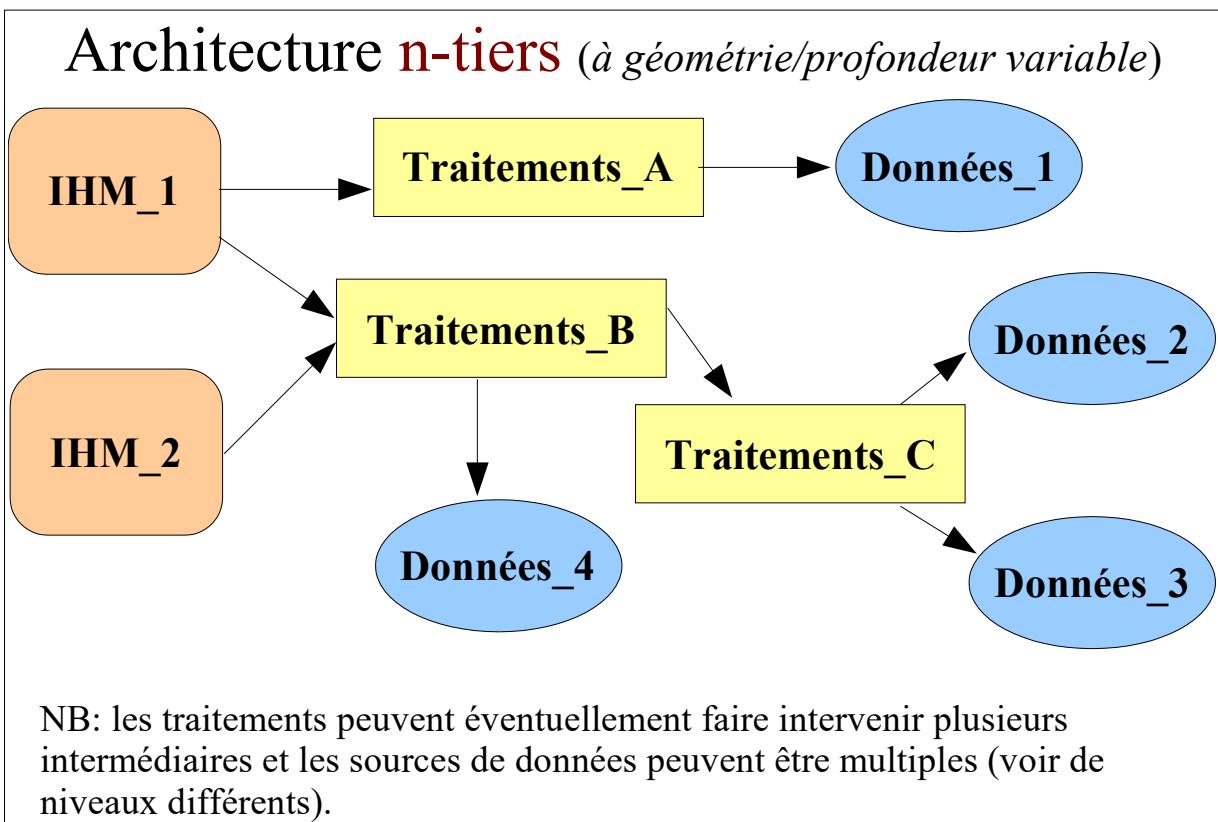
Architecture 3-tiers (modèle logique générique)



=> peut prendre pleins de formes différentes:
(écran , cpu , disque), (client léger, serv app, sgbd), ...

Architecture **n-tiers** (*modèle logique décomposé avec subdivision/répartition fine des responsabilités / ...*)





Le tiers présentation

- Affichages + Saisies/Sélections + logique événementielle
- Contrôler l'enchaînement des écrans (*selon op. valides*)
- Modèle de données locales (*modèle applicatif*)
- Communications avec les autres parties (*proxy, business delegate,*)
- NB:
il est préférable de dissocier les différents aspects précédents
[ex: framework **MVC** (Modèle / Vue / Contrôleur)]

Services "métier"

- Point de départ des transactions
- Travail de collaboration (*superviser plusieurs traitements élémentaires*)
- Fonctionnalités métiers + CRUD.
- Règles de gestion

(ex: EJB3/Jpa
ou

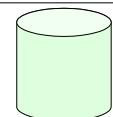
Spring/Hibernate
ou ...)

Le tiers "traitements (back)"

Entités "métier"

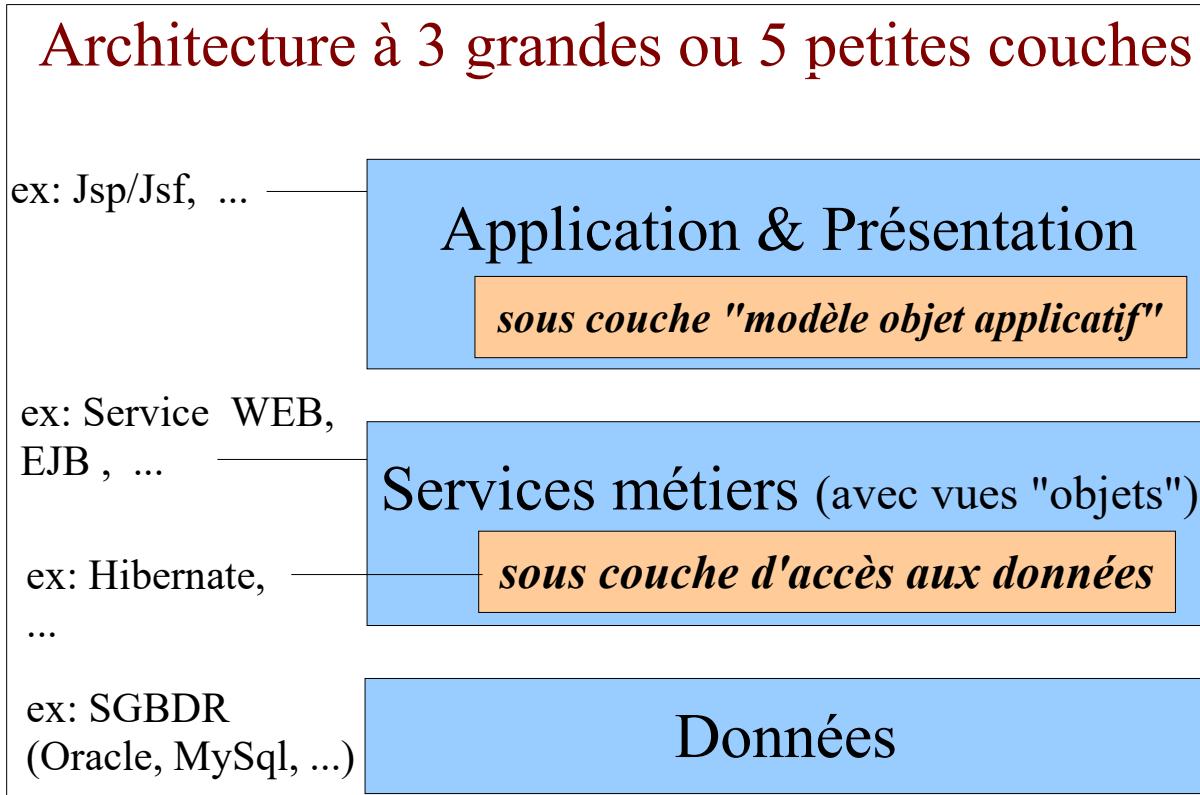
- Entités fondamentales (et persistantes)
- avec Data Access Object (D.A.O.) pour les gérer.

Le tiers "sources de données"



- Base de données (relationnelles, "XML" ou "objet")
- Moniteur transactionnel (ex: CICS, Tuxedo, ...)
- (Gros) système propriétaire (legacy system)
- ERP (Enterprise Resources Planning) ex:SAP
- Simples fichiers (.txt, .xml, ...) , ...

1.2. Couches logicielles classiques



1.3. Architectures logiques et physiques

1.3.a. *Architectures logiques*

Modèle d'architecture abstrait montrant clairement:

- les différentes parties logiques (bien identifiées)
- les rôles/fonctionnalités/responsabilités de chacune des parties

On raisonne en services rendus et pas encore avec des technologies et infrastructures précises.

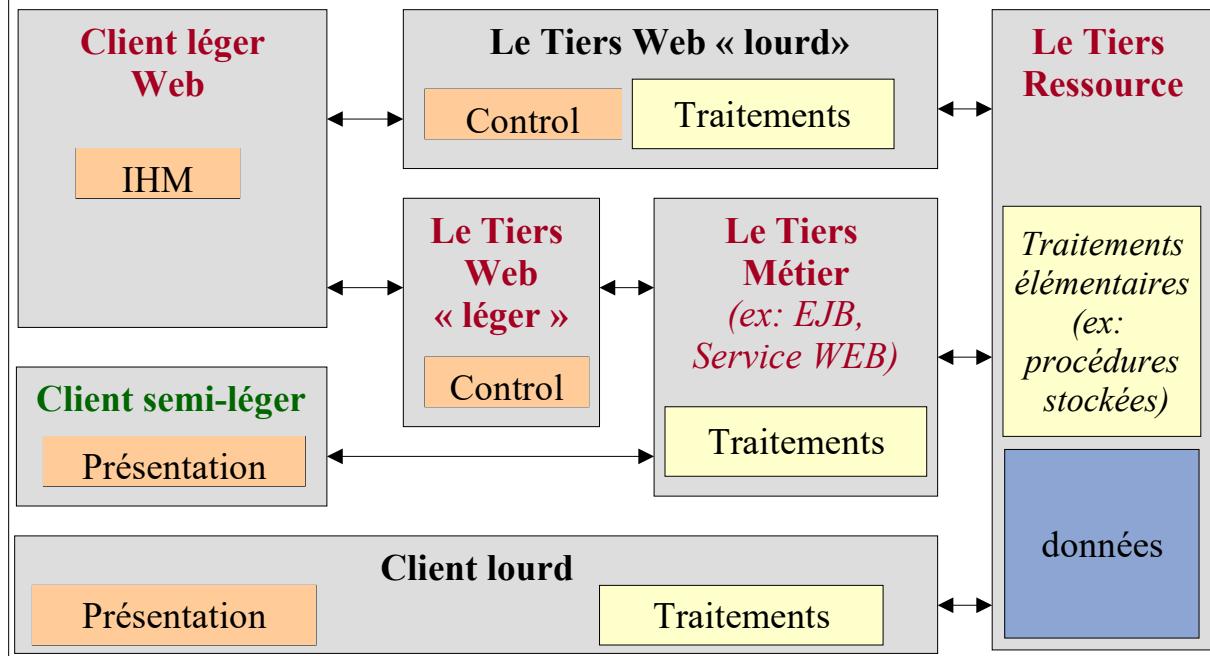
1.3.b. *Architectures physiques*

Modèle d'architecture concret montrant précisément:

- la distribution des parties logiques (préalablement identifiées/modélisées) au sein des différentes composantes physiques (machines / serveurs logiciels) du système informatique
- les technologies utilisées (API , protocoles ,)

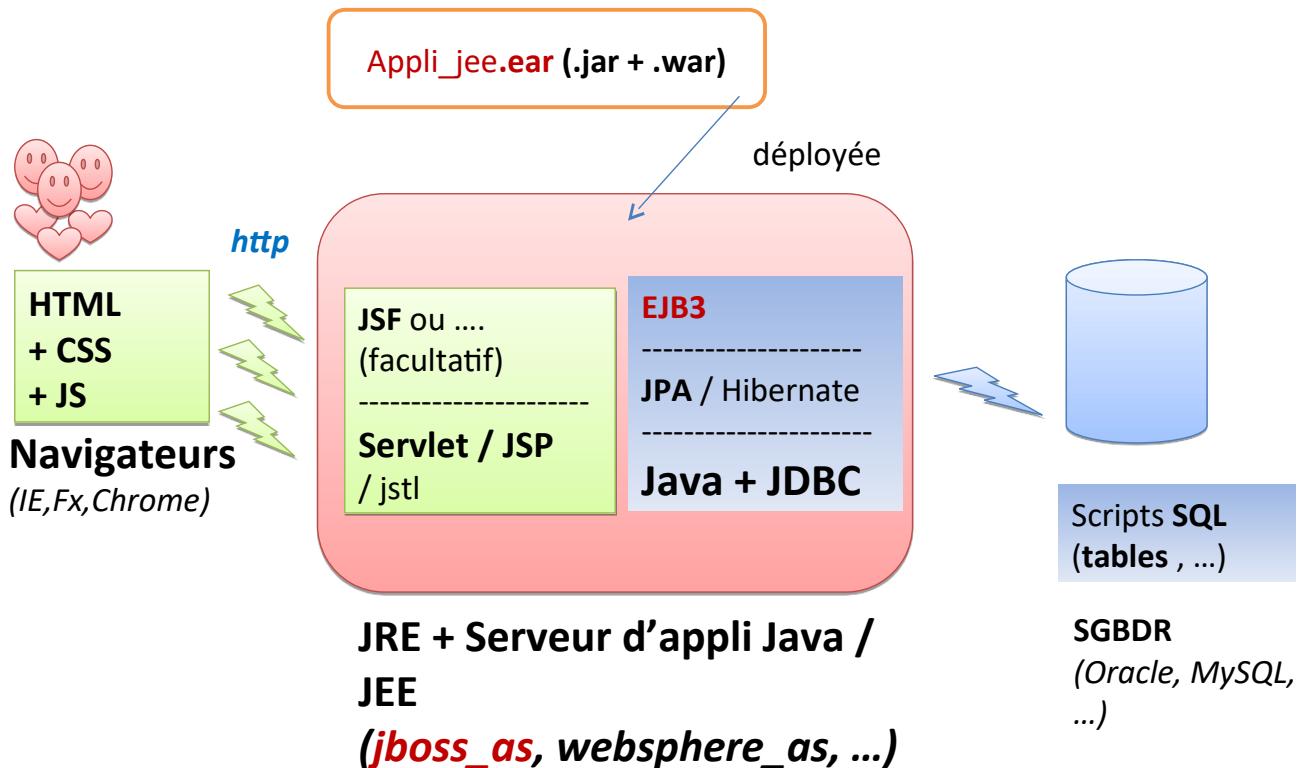
Il existe généralement plusieurs architectures physiques possibles pour transposer/projeter une certaine architecture logique.

Modèles d'architectures physiques



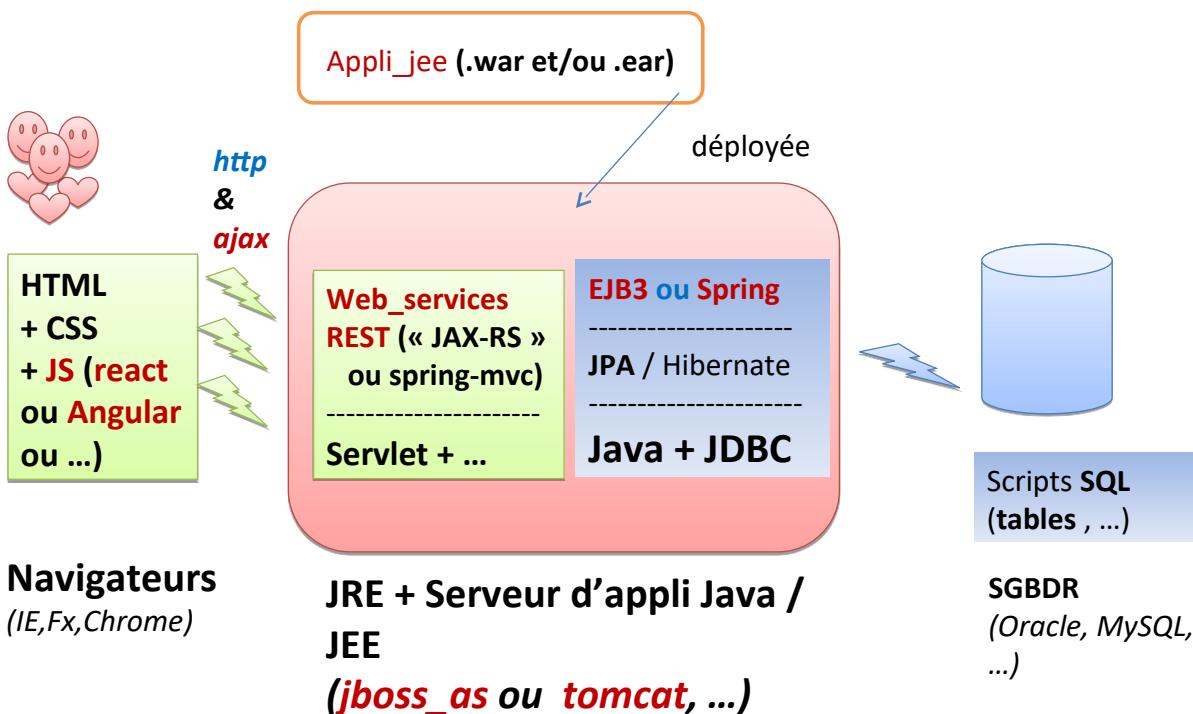
avec client léger java/web

Env exécution java/jee (v2)

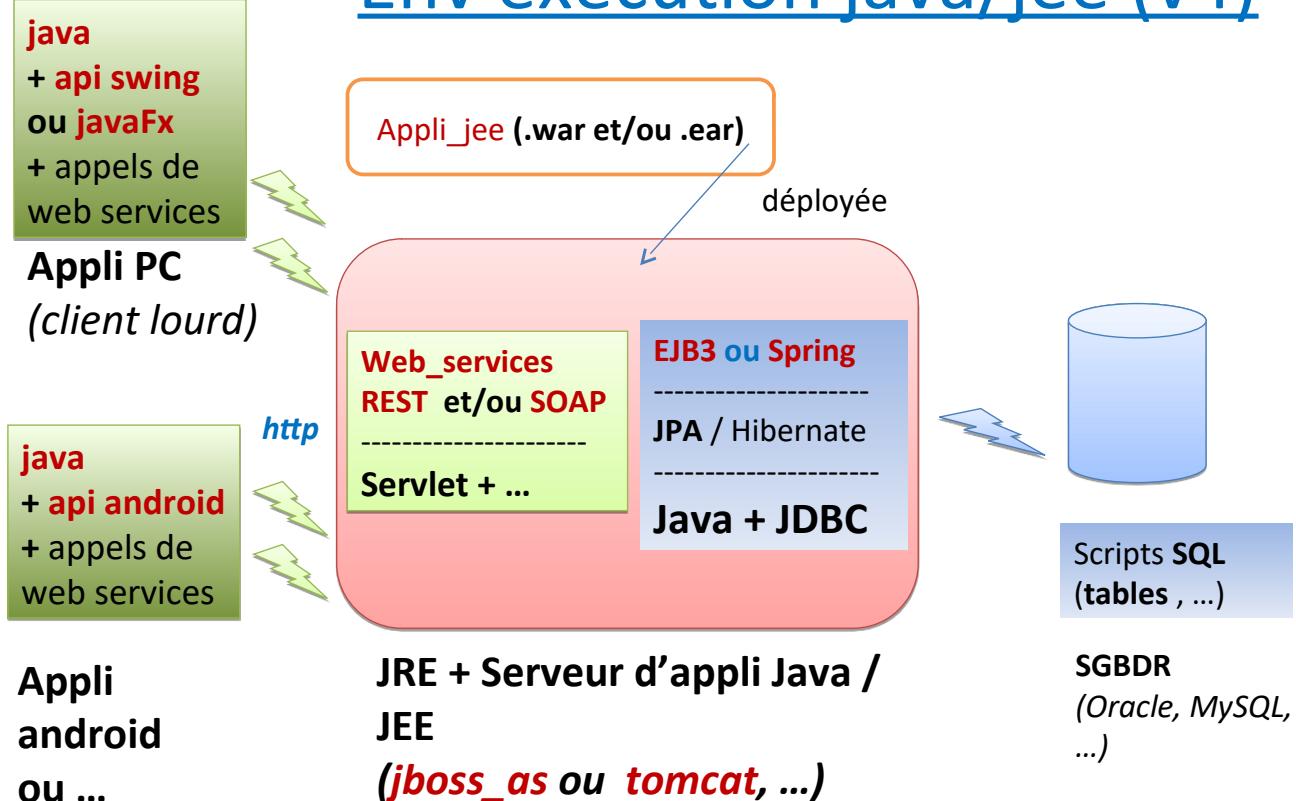


FrontEnd lourd et Api REST plus ou moins légère (selon complexité du métier)

Env exécution java/jee (v3)



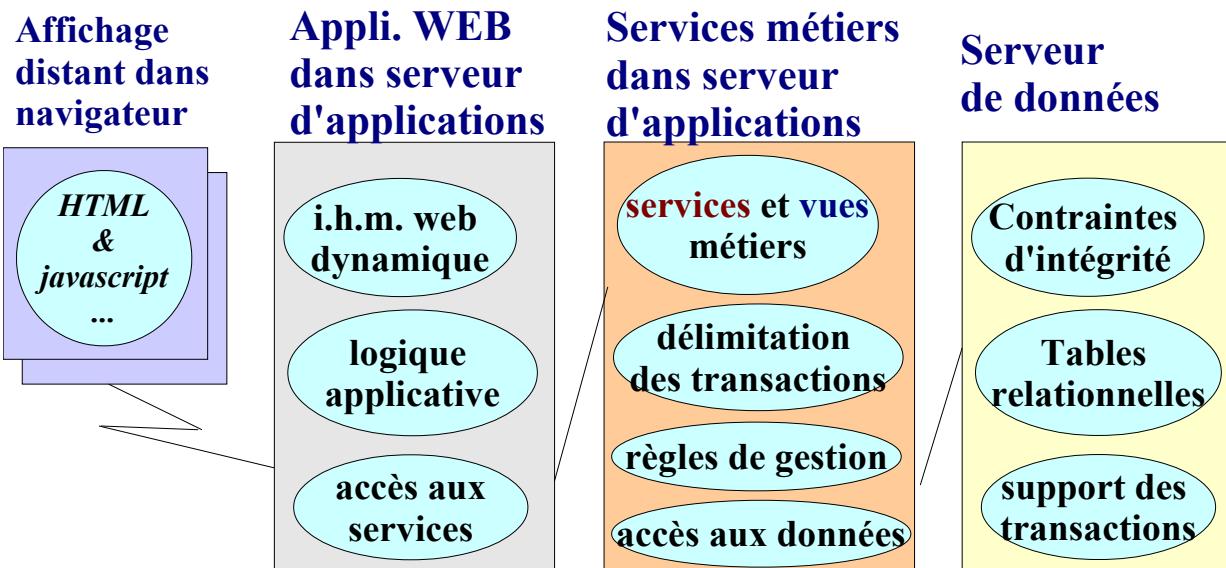
Env exécution java/jee (v4)



1.4. évolution des architectures (début 21eme siècle)

1.4.a. N-tiers JEE

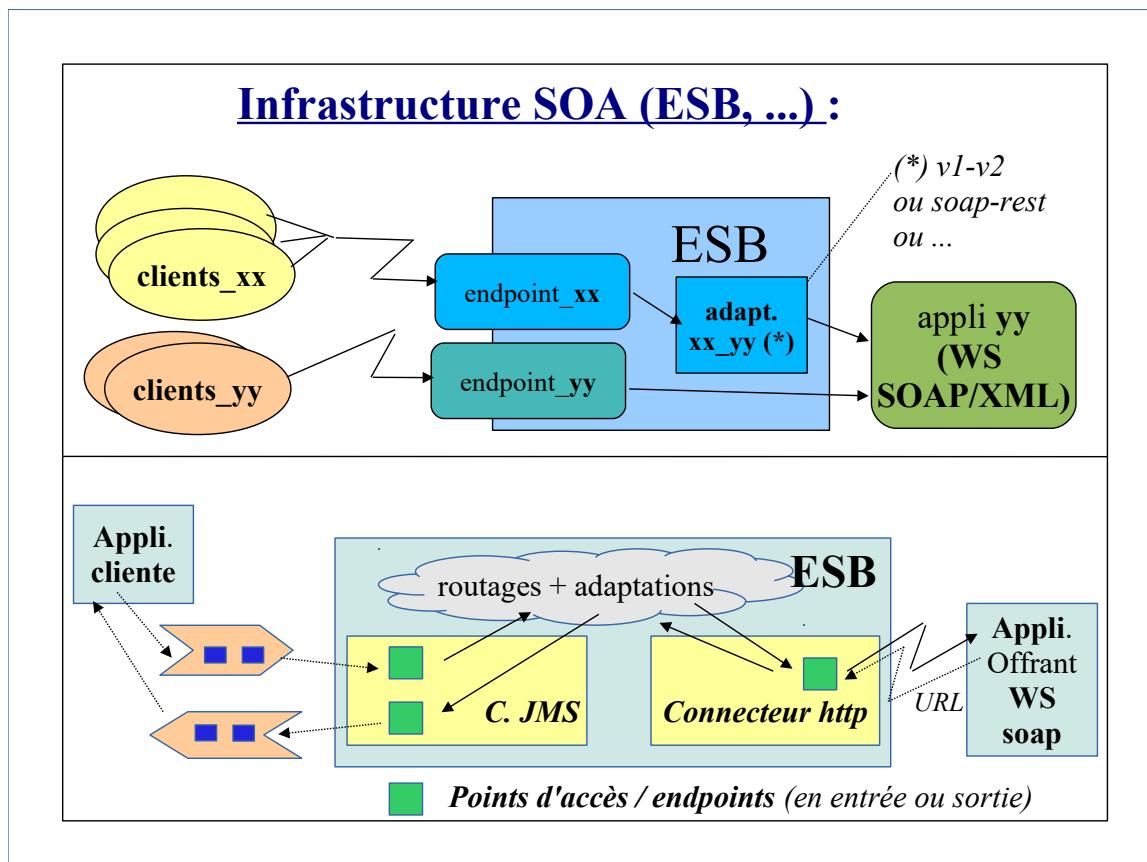
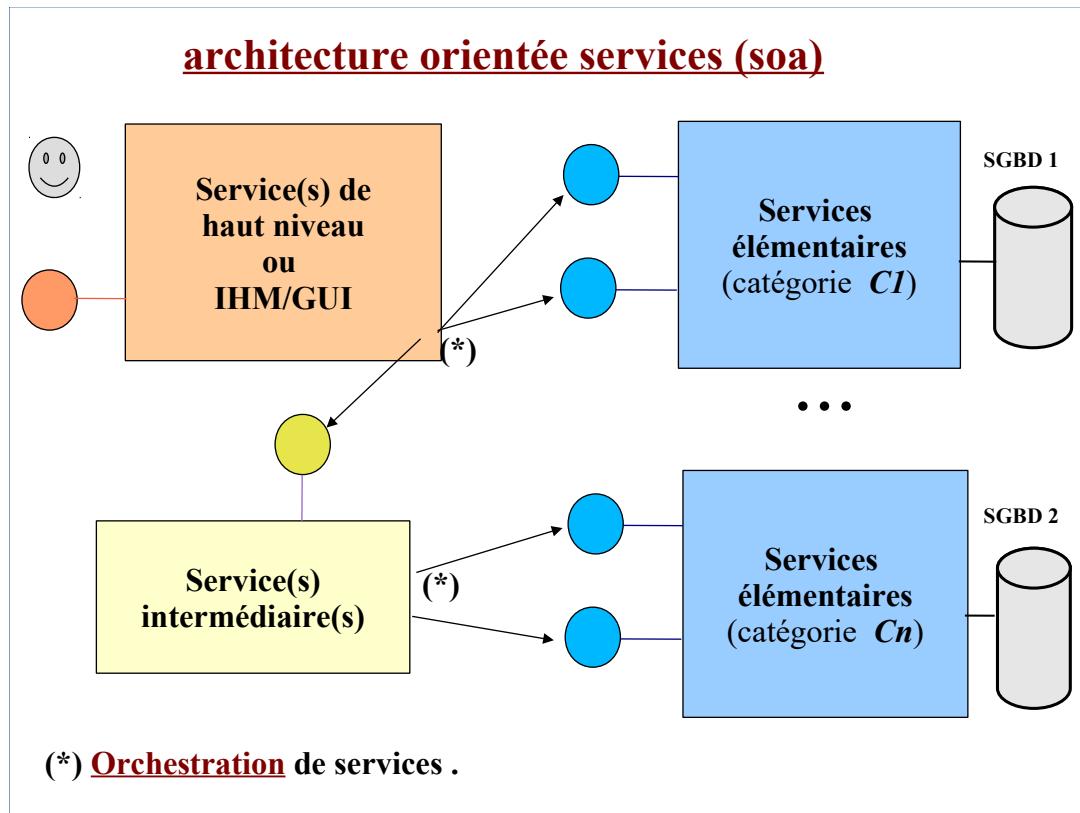
Architecture n-tiers (début années 2000)



**++ modularité , déploiements simplifiés , ihm correcte
- complexité de l'ensemble**

Tout début des années 2000 (age d'or de Java/JEE)

1.4.b. SOA (architecture orientée services) : vers 2010

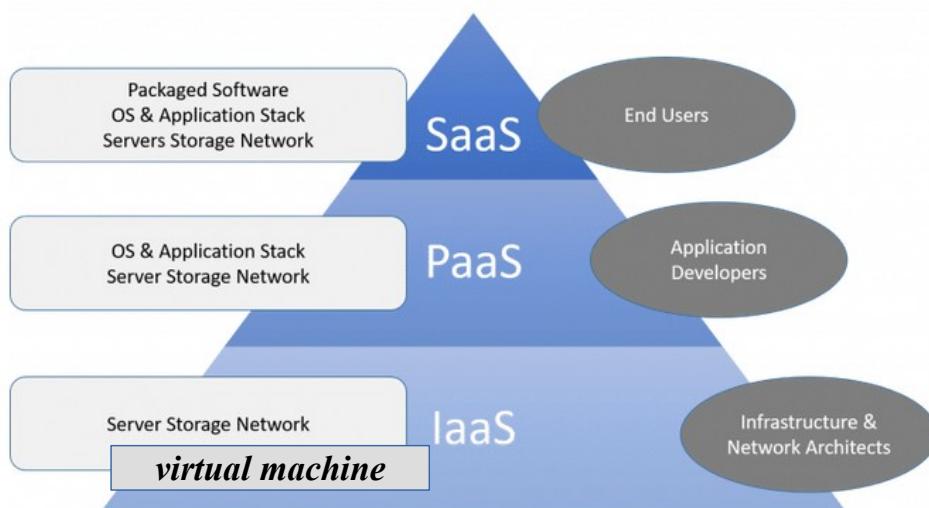


1.4.c. cloud-computing (essor rapide à partir de 2015)

Cloud computing (apports)



Cloud computing (hébergement , délégation)



Saas : Software As A Service

Paas : Platform As A Service

Iaas : Infrastructure As A Service

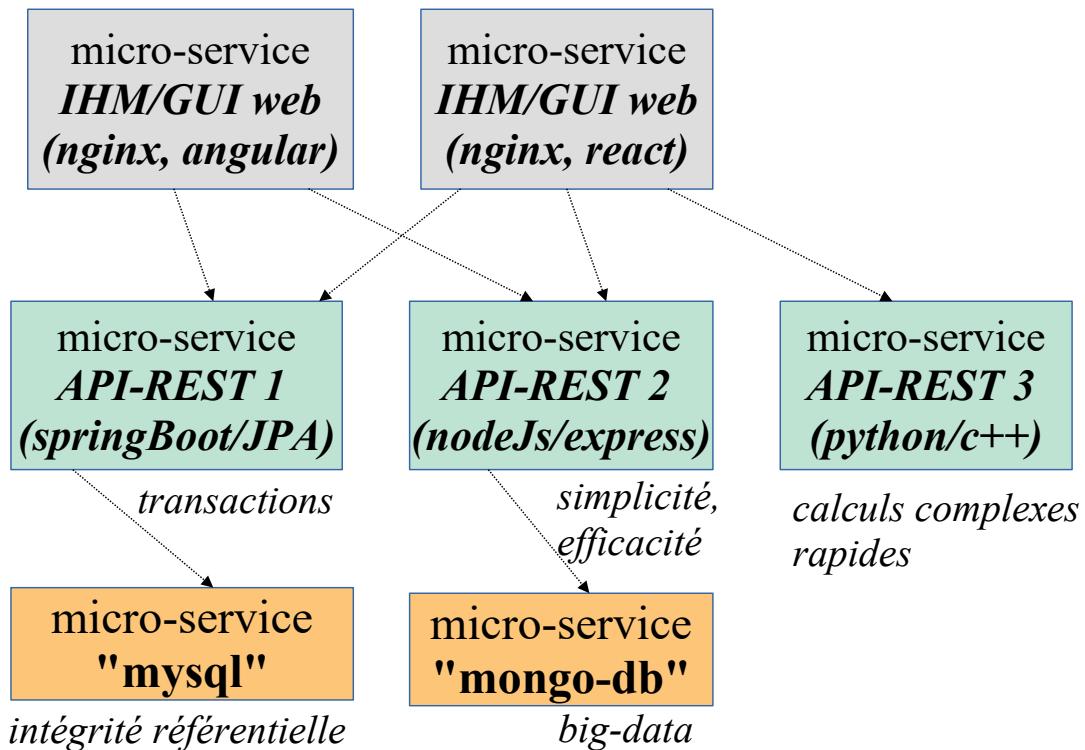
1.4.d. Architecture micro-services (vers 2020)

Architecture "micro-service" (MSA)

L'architecture "micro-services" peut être vue comme une évolution de l'architecture "SOA/orientée services" avec :

- *infrastructure logicielle séparant mieux les aspects "technique" et "fonctionnel"* (ApiGateway plutôt que ESB, si orchestrateurs "BPM" ou autres intermédiaires alors "oui mais léger").
- **alignement sur les technologies "cloud" (IaaS,PaaS) et "micro-conteneurs"** (infrastructure "scalable"/élastique (utilisation fréquente de Docker , Kubernetes , ...))
- *tolérance vis à vis des services inaccessibles* (pas de "point de panne simple" , cluster avec prise en compte de l'état des serveurs)
- alignement avec "WOA" (xml/soap --> json/REST/HTTP)
- *très peu d'inter-dépendances entre les micro-services* (idéalement "stateless" et *programmables avec technologies très variables* : Java/SpringBoot , NodeJs/Express , python,)

Architecture micro-services (avec micro-conteneurs)



2. Critères de qualité

2.1. Les principaux attributs de qualités

<i>Attributs de qualité</i>	<i>Caractéristiques</i>	<i>Mesures/précautions</i>
Disponibilité	<p>Application momentanément indisponible (changement de version, évolution, panne maintenance technique, sauvegarde,).</p> <p>taux dispo =</p> $\frac{\text{temps dispo}}{\text{temps dispo} + \text{temps indispo}}$	Cluster avec mécanisme "fail/over" , redondance , limiter les conséquences des inter-dépendances chaînées.
Souplesse/évolutivité ("Modifiabilité") (maintenabilité , extensibilité ,)	Application rigide (difficile à faire évoluer) ou bien souple (extensible/reconfigurable)	<p>Bien utiliser les concepts objets et l'architecture SOA.</p> <p>Bien dimensionner le nombre de couches logicielles (ni trop , ni pas assez).</p> <p>Bien identifier/modéliser les besoins fonctionnels (présents et futurs pré-sentis)</p> <p>Tenir compte du schéma directeur de l'urbanisation</p>
Performance	<p>"Bons ou mauvais temps de réponse ou de traitement" et</p> <p>"Nombre de réponses calculées/renvoyées par unité de temps" (débit /throughput) pour une bonne gestion de la charge (utilisateurs simultanés)</p>	<p>Mesures des temps d'exécutions (ihm , services , SQL , ...) et optimisations.</p> <p>Tests de charge (requêtes simultanées) .</p> <p>Optimisation du trafic réseau (ajax, ...).</p> <p>Utilisation de caches lorsque c'est possible.</p> <p>Surveillance régulière en production et redimensionnement des clusters.</p> <p>Séparation fonctionnelle pour parallélisme</p>
Fiabilité	Peu de pannes / plantages	Déetecter et éliminer les "fuites de mémoires" , bien gérer les exceptions , ...
Intégrité	Données toujours cohérentes ?	<p>Intégrité référentielle dans SGBDR , GDR/.... en SOA ,</p> <p>Tests des valeurs des données via dbUnit ,</p> <p>Utilisation de l'encapsulation (données</p>

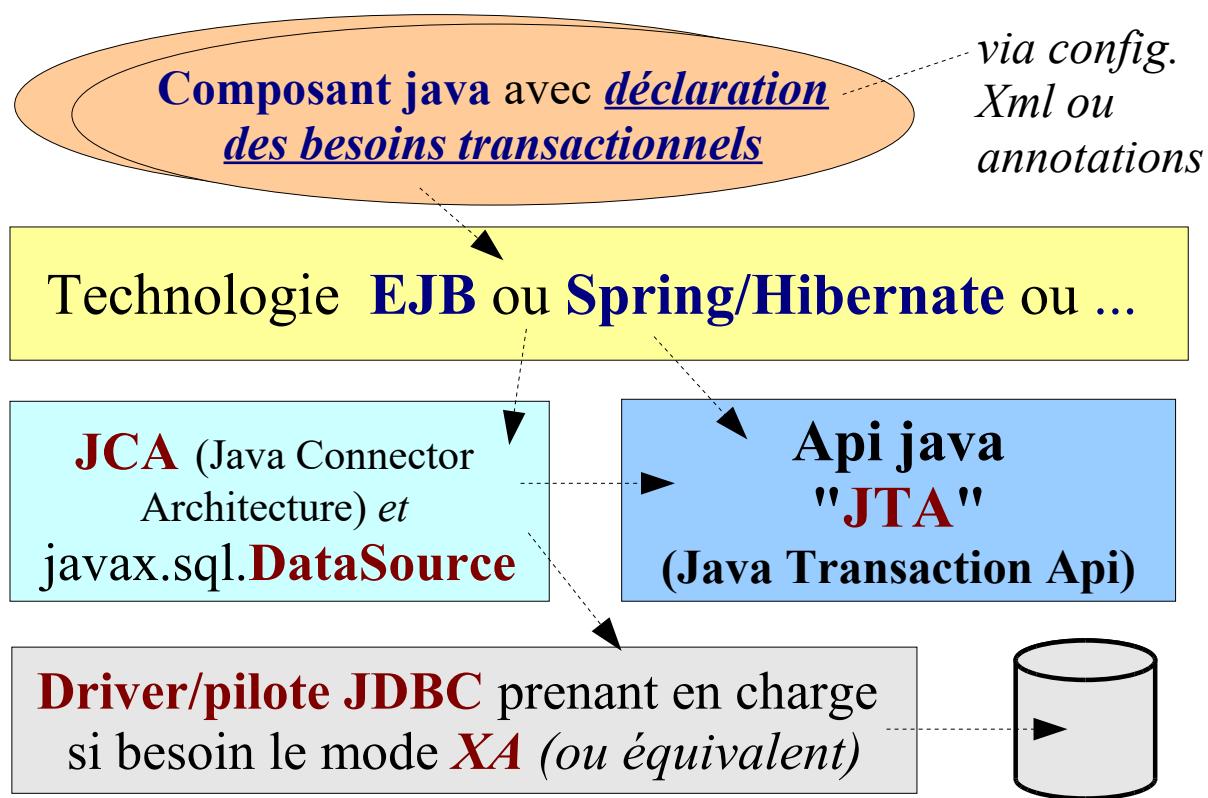
		internes privées bien protégées, ...).
Validité fonctionnelle (exactitude)	Selon cahier des charges (faire ce qui était demandé)	Tests unitaires pour chaque composants métiers (ex : service) et tests d'intégration (application complète avec interface graphique)
Sécurité	capacité du système à résister à l'attaque , et à continuer à offrir un service normal aux usagers légitimes	Pare-feu , DMZ , long mots de passes , cryptages , authentification forte , ...
Utilisabilité <i>(convivialité , ergonomie)</i>	Facilité d'apprentissage et d'utilisation du logiciel par les usagers	Soigner l'interface graphique : - intuitive et claire - efficace (raccourcis , ...) - non ambiguë (opération bien réalisée ou messages d'erreurs explicites, ...) - look agréable

2.2. Quelques autres attributs de qualité

<i>Attributs de qualité</i>	<i>Caractéristiques</i>	<i>Mesures/précautions</i>
Intégration simple (de niveau SI)	Intégrer dans des applications "œur de métier" des services offerts par des applications annexes (de supports)	Utilisation d'ESB / EAI et bonne modélisation transverse , bonne urbanisation.
Intégration simple (des différentes technologies complémentaires)	Permettre à des composants développés séparément d'interopérer correctement	Via interfaces normalisées , via éventuels adaptateurs ,
Interopérabilité	Import/export de documents, communications avec d'autres applications . Format compréhensible des messages échangés.	Utilisation de technologies "standard" et interopérables (ex : XML , HTTP) et/ou utilisation d'intermédiaires qui transforment les messages (ex : ESB)
Portabilité	Adaptabilité à différents environnements d'exécution (systèmes d'exploitations , serveurs d'applications , ...)	Utilisation de langage multi-plateformes (ex : " Java " ou ...).
réutilisabilité	de certains composants et/ou de certains services et/ou de l'architecture	Choisir des technologies ayant à priori une certaines pérennités. Etudier les "dénominateurs communs" entre différents besoins et décomposer/specialiser avec des choses génériques (facilement réutilisables) et d'autres choses plus spécifiques. Utiliser des "serveurs applicatifs ou

		frameworks" prédéfinis pour une réutilisabilité de l'architecture.
Testabilité (vérifiabilité) [<i>Testabilité = observabilité + contrôlabilité</i>]	Tests/vérifications/contrôles possibles , simples et déterministes	Prévoir systématiquement un test unitaire par composant logiciel important. Enregistrer pour re-jouer. Éviter les éléments incontrôlables (maîtriser les technologies et maîtriser la complexité du code) Incorporer quelques éléments de pilotage dans l'application (vérification ordre de marche, mesures/minitoring interne, ...).
Efficacité	Exploitation optimale des ressources (CPU, RAM,)	Mesurer , optimiser , re-mesurer , comparer, ...
Autonomie	Dépendances limitées vis à vis d'autres systèmes. → favorise une haute disponibilité et une évolution moins contrainte.	éviter les dépendances inutiles. Utiliser si possible des infrastructures "lights" (sans complexité/lourdeur inutile).
Composabilité , modularité	Pouvoir construire un système comme un assemblage de composants modulaires	Utilisation d'un framework d'intégration (ex : Spring ou CDI) basé sur de l'injection de dépendances.
Robustesse	capacité d'un logiciel à fonctionner lorsque l'on sort de ses spécifications.	Bien gérer les contrôles de saisies et les contraintes pour les valeurs possibles.
Coût (raisonnable?) de développement et fonctionnement		Chiffrages , estimations R.O.I.
<i>Durée de vie projetée</i>	"jetable" ou "à maintenir sur du long terme" ?	
<i>Marché cible /</i>		
...		
Intégrité conceptuelle	Cohérence par rapport au thème dominant quitte à abandonner/externaliser/déléguer certains éléments annexes	Bonne modélisation (ex : UML)
Complétude (et exactitude)	Répond bien à toutes les exigences	
Faisabilité	En tenant compte des ressources et des contraintes	Chifffrage , expérience , estimations .

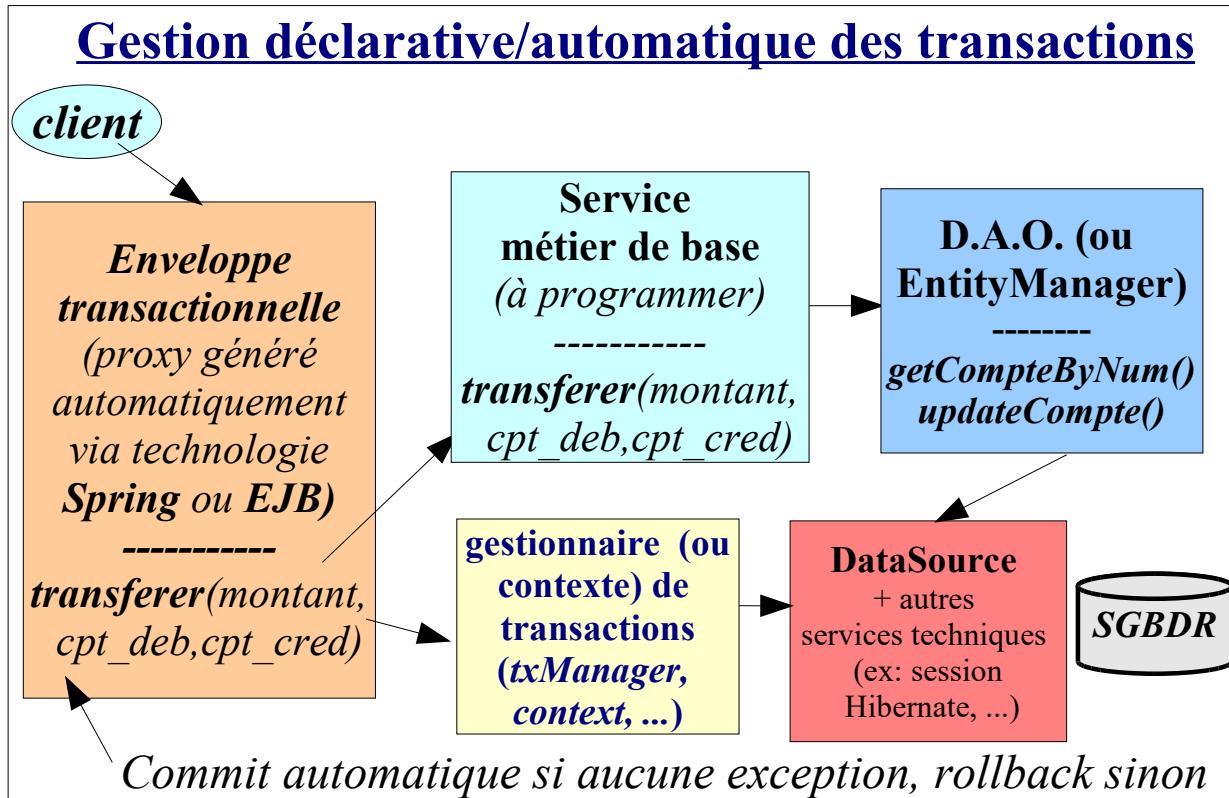
2.3. Infrastructure transactionnelle de JEE



Tout serveur JEE offre une infrastructure sérieuse pour bien gérer/piloter les transactions .

- L'Api JTA permet une bonne délimitation des transactions distribuées . Son utilisation directe nécessite beaucoup de ligne de code pour paramétriser le contexte transactionnel et pour explicitement déclencher les "commit" ou "rollback" .
- Les technologies de bas niveau "JCA , DataSource et JDBC/XA" servent à bien relayer les ordres de "commit" / "rollback" jusqu'à la source de données (SGBDR ou ...).
- Des technologies de haut niveau telles que EJB ou Spring/Hibernate permettent entre autres d'automatiser les transactions (commit implicite si aucune exception de remonte, rollback systématique sinon).

2.4. Gestion déclarative des transactions



Le code généré dans l'enveloppe transactionnelle est à peu près de cette teneur:

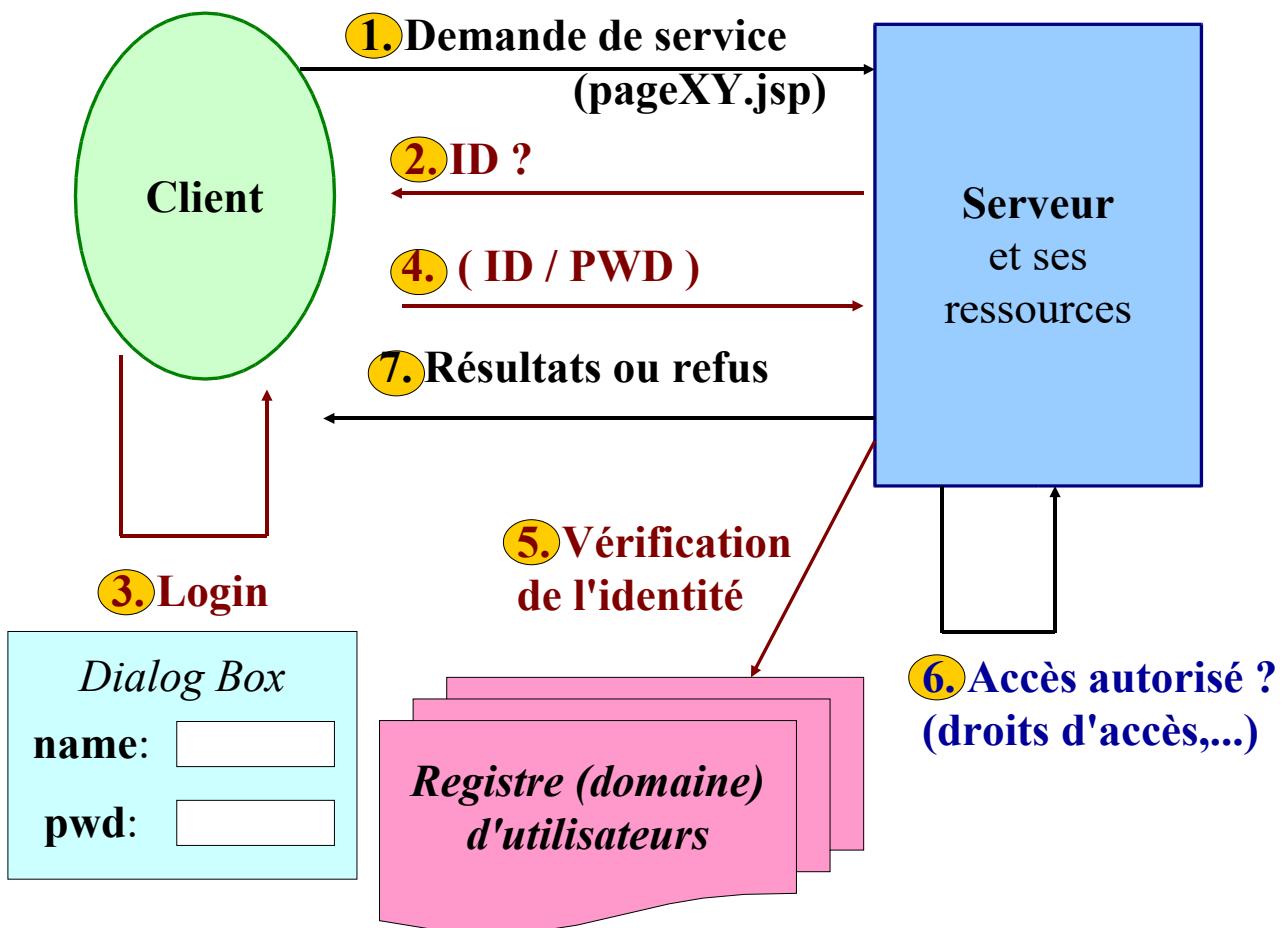
```
public void transferer(double montant, long num_cpt_deb, long num_cpt_cred){
    // initialisation (si nécessaire) de l'entityManager de JPA
    // selon existence dans le thread courant
    tx = ...beginTransaction(); // sauf si transaction (englobante) déjà en cours
    try{
        serviceDeBase.transferer(montant,num_cpt_deb,num_cpt_cred);
        tx.commit(); // ou ... si transaction (englobante) déjà en cours
    }
    catch(RuntimeException ex){    tx.rollback(); /* ou setRollbackOnly(); */ ... }
    catch(Exception e){    e.printStackTrace(); }
    finally{    // fermer si nécessaire EntityManager JPA
        // (si ouvert en début de cette méthode)
    }
}
```

2.5. Sécurité J2EE/JEE5

Sécurité J2EE/JEE5

Gérer la sécurité "J2EE/JEE5" consiste essentiellement à :

- ◆ **Préciser le ou les rôle(s) logique(s) requis pour pouvoir déclencher une certaine méthode sur un EJB ou pour activer certaines URL d'une application Web.**
→ *Travail généralement effectué par le développeur*
- ◆ **Authentifier l'utilisateur (UserName , PassWord).**
→ *Via technologie HTTPS ou JAAS ou ...*
- ◆ **Associer un ou plusieurs utilisateur(s) ou groupe(s) [d'un annuaire ldap ou ...] à chaque rôle logique.**
→ *Paramétrage et configuration liés au déploiement d'une application J2EE et à l'administration du serveur*



3. Principaux choix techniques (vers 2020)

3.1. Bases de données (relationnelles ou pas)

Points forts de mongoDB	Point faibles de mongoDB
<ul style="list-style-type: none"> Supporte de gros volumes Mode distribué (réPLICATION, sharding ,) Simple et efficace données au format JSON ... 	<ul style="list-style-type: none"> Moins de possibilités dans les requêtes qu'avec le traditionnel SQL Moins structuré , moins de contraintes d'intégrité ...

NB : si au sein d'une architecture micro-services alors un des micro-services peut utiliser une base relationnelle et un autre microservice peut utiliser une base noSQL telle que mongoDB .

3.2. Serveur d'application vs SpringBoot

Serveur d'application JEE (Jboss ou WebSphere)	Environnement SpringBoot
<ul style="list-style-type: none"> Support d'une grande marque (redHat , IBM , ...) respect du standard officiel JEE quelques dépendances vis à vis du serveur (config spécifique , ...) Tests quelquefois complexes 	<ul style="list-style-type: none"> Développement un peu plus simple et plus rapide (dao automatiques via SpringData , tests plus simples) Quelquefois plus de serveur et donc beaucoup de liberté . Simple à intégrer dans conteneur Docker

3.3. synchone vs asynchrone

A court/moyen terme (début 2020) :

- les environnements d'executions java/JEE sont plutôt synchrones et multi-threads et les API fondamentales aussi (Servlet , JDBC , JPA , transactions sur EJB , ...)
- Le concurrent nodeJs est de son coté complètement asynchrone mais la stack "transaction + ORM" manque un peu de robustesse et de standardisation
- Le style de code asynchrone est bien maîtrisé par les bons développeurs "javascript" (via Promise , async/await , RxJs , ...) mais les équivalents java sont encore très peu connus (CompletableFuture , RxJava , Reactor)

A plus long terme (2025 ou?????) :

- Le monde java va tenter d'intégrer petit à petit des mécanismes de plus en plus asynchrones
- L'écosystème spring est de ce coté précurseur (projet reactor , ...)

4. Comparaison avec .NET , NodeJs et Python

Plateforme technique (coté serveur)	Principales caractéristiques
Java/JEE	Fiable , standard existant et stabilisé depuis de nombreuses années. Performances correctes (à part au démarrage). Bien adapté aux applications de gestions ayant des besoins transactionnels (commit/rollback). Multi-plateforme (bien que souvent en prod sous linux)
.Net (Microsoft)	Pas bien multi-plateforme (malgré tentatives). Langage c# assez proche de java et donc performances assez comparables. Framework Web un peu moins au point que côté java car le monde .NET s'est surtout concentré sur le côté "application du côté client" / "desktop".
Python	Langage syntaxiquement très différents des autres (pas de { } mais des indentations) , performances assez faible si "python seul" mais bonnes performances en mode "python + langage C" . Python est particulièrement intéressant pour effectuer des tâches mathématiques (calculs, équations , ...) et scientifiques car le langage python est accompagné d'un énorme paquet de librairies mathématiques (calcul matriciel , calcul formel ,)
Javascript (nodeJs)	NodeJs (moteur d'interprétation javascript souvent côté serveur) offre de très bonnes performances (coeur léger consommant peu de mémoire , démarrage rapide, exécution efficace avec un seul thread javascript mais avec beaucoup de mécanismes asynchrones efficaces). La programmation modulaire est à court terme à double tranchant : d'un côté souplesse et liberté . De l'autre : ça part dans tous les sens et ça manque de standardisation. Pour l'instant le monde java/jee semble un peu plus robuste que nodeJs/express pour de grosses applications de gestions utilisant des bases relationnelles classiques (MySQL , Oracle, postgres, ...). Par contre NodeJs est tout à fait approprié pour développer des applications légères utilisant par exemple des bases de données "mongoDB" .

5. Tests unitaires et Test Driven Development

5.1. Présentation de JUnit 3,4,5

JUnit est un **framework** simple permettant d'effectuer des **tests** (unitaires , de non régression, ...) au cours d'un développement java . [Projet Open source ---> <http://junit.sourceforge.net/> , <http://junit.org>] . *JUnit est intégré au sein des IDE Eclipse et IntelliJ.*

JUnit existe en versions 3 , 4 et 5 avec des différences significatives d'une version à l'autre .

5.2. Présentation des anciennes versions 3 et 4

La très ancienne version 3 de JUnit n'utilisait pas d'annotations . Tout était basé sur un héritage (TestCase) et sur des conventions de nom sur les méthodes (setUp() , tearDown() , testXy()) . Depuis environ 2006/2007 , cette historique version 3 est devenue petit à petit obsolète (utilisée seulement dans les anciens projets).

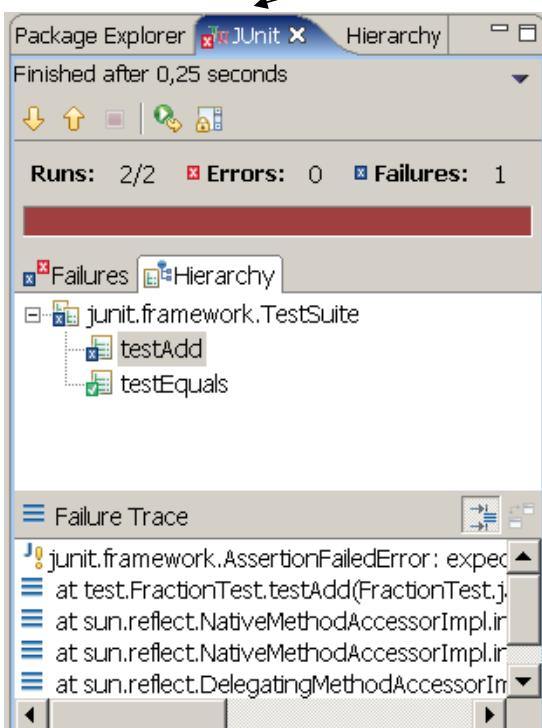
La version 4 a été massivement utilisée dans la majorité des projets java de 2007 à 2019 environ . La version 4 était basée sur le package org.junit (Test, Assert, ...) et les principales annotations étaient @Test , @Before , @After , @BeforeClass , @AfterClass .

5.3. Lancement des tests unitaires

Lancement des tests unitaires

Depuis l'IDE eclipse:

Run as ... / JUnit test



TestSuite en JUnit3
et lancement après une sélection de **package** en JUnit4 pour lancer d'un coup toute une série de tests.

Comptabilisations :

Error(s) : exceptions java non rattrapées.

Failure(s) : assertions non vérifiées.

VERT si aucune erreur.

Depuis "maven" :

coder des classes nommées "**TestXy**" ou "**XYTest**" dans **src/test/java** et lancement via **mvn test** ou autre.

5.4. Présentation de JUnit 5 ("jupiter")

La **version 5 de JUnit** a été entièrement restructurée et s'appuie sur certaines nouvelles fonctionnalités apportées par la **version 8 du langage java** (lambda expressions,) .

Principales différences entre JUnit4 et JUnit5 :

Junit 4	Junit 5
package org.junit	package org.junit.jupiter.api
Assert.assertTrue() , Assert.assertEquals(...)	Assertions.assertTrue() , Assertions.assertEquals(...)
@Before , @After	@BeforeEach , @AfterEach
@BeforeClass , @AfterClass (avec static)	@BeforeAll , @AfterAll(avec static)
...	...

5.5. Configuration maven pour JUnit 5 ("jupiter")

...

```

<properties>
    <junit.jupiter.version>5.4.2</junit.jupiter.version>
</properties>
<dependencies>
<dependency>
    <groupId>org.junit.jupiter</groupId> <artifactId>junit-jupiter-api</artifactId>
    <version>${junit.jupiter.version}</version> <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId> <artifactId>junit-jupiter-engine</artifactId>
    <version>${junit.jupiter.version}</version> <scope>test</scope>
</dependency>
</dependencies>
<build> <plugins>
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.0.0-M4</version> <!-- Need at least 2.22.0 to support JUnit 5 -->
</plugin>
</plugins> </build> ...

```

Eventuels compléments :

- | | |
|-----------------------------|---|
| junit-jupiter-params | <i>support des tests paramétrés avec JUnit Jupiter.</i> |
| junit-vintage-engine | <i>pour interpréter et exécuter des anciens tests codés en JUnit 3 ou 4</i> |
| junit-platform-.... | <i>pour intégration et lancement (console , maven , gradle ,)</i> |

5.6. Test Unitaire avec JUnit 5 (@Test, @BeforeEach, @AfterEach)

```

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
...
public class TestCalculatrice {
    private static Logger logger = LoggerFactory.getLogger(TestCalculatrice.class);
    private CalculatriceEx calculatriceEx; //à tester

```

```

@BeforeEach // @BeforeEach in jUnit5 , @Before in jUnit4
public void initCalculatriceEx() {
    this.calculatriceEx = new CalculatriceEx();
}

@Test
public void testSum() {
    calculatriceEx.pushVal(1.1);
    calculatriceEx.pushVal(2.2);
    calculatriceEx.pushVal(3.3);
    double somme = calculatriceEx.sum();
    logger.trace("somme(1.1, 2.2, 3.3)="+somme);
    //Assert.assert...() avec jUnit 4 et Assertions.assert...() avec jUnit5/jupiter
    Assertions.assertEquals( 6.6 , somme, 0.00000001);
}

@Test
public void testAverage() {
    calculatriceEx.pushVal(1.1);
    calculatriceEx.pushVal(1.5);
    double moyenne = calculatriceEx.average();
    logger.trace("moyenne(1.1, 1.5)="+moyenne);
    Assertions.assertEquals( 1.3 , moyenne, 0.00000001);
}
}

```

NB: Comme avec JUnit 4, par défaut JUnit 5 crée une nouvelle instance de la classe de test pour exécuter chaque méthode de test. (*@TestInstance(Lifecycle.PER_METHOD) par défaut*)

Assertions.assertEquals(*expectedValue, effectiveValue , delta*) ;
 Assertions.assertTrue(*booleanExpression*) ; Assertions.notNull(...)

NB :

import static org.junit.jupiter.api.Assertions.assertTrue;

permet d'écrire **assertTrue** (res==5) au lieu de **Assertions.assertTrue**(res==5)
 et ça peut aider à masquer la différence de préfixe entre JUnit 4 et JUnit5 .

5.7. Test Unitaire JUnit5 avec static (@BeforeAll , @BeforeAll)

```

...
import org.junit.jupiter.api.BeforeAll;
...
public class TestCalculatrice {
    private static SimpleCalculatrice calculatrice; //à tester

    @BeforeAll // @BeforeAll in jUnit5 , @BeforeClass in jUnit4
    public static void initCalculatrice() {
        calculatrice = new SimpleCalculatrice();
    }
}

```

```

@Test
public void testAddition() {
    double resAdd=calculatrice.addition(5.5, 6.6);
    logger.trace("addition(5.5,6.6)="+resAdd);
    Assertions.assertTrue(resAdd>= (12.1 - 0.00000001) &&
        resAdd <= (12.1 + 0.00000001));
}

@Test
public void testMultiplication() {
    double resMult=calculatrice.multiplication(2.0, 3.3);
    logger.trace("multiplication(2.0, 3.3)="+resMult);
    Assertions.assertEquals( 6.6 , resMult, 0.0000001);
}

}

```

5.8. Spécificités de JUnit5

```

@DisplayName("Ma classe de test JUnit5 que j'aime")
public class MyTest {

@Test
@DisplayName("Mon cas de test Xy qui va bien")
void Testxy() {
    // ...
}

```

NB: Les valeurs des @DisplayName seront affichées au sein des rapports d'exécution des tests

NB : contrairement à JUnit4, les méthodes préfixées par `@BeforeEach`, `@Test` , ... n'ont plus absolument besoin d'être `public` (protected ou bien la visibilité implicite par défaut de niveau package suffit) .

AssertAll() avec lambdas :

```

Assertions.assertAll("wrong Dimension",
    0 -> Assertions.assertTrue(elt.getWidth() == 400, "wrong width"),
    0 -> Assertions.assertTrue(elt.getHeight() == 500, "wrong height"));

```

vérifie que chacune des lambdas est ok (sans exception) .

```

Iterable<Integer> attendu = new ArrayList<>(Arrays.asList(1, 2, 3));
Iterable<Integer> actuel = new ArrayList<>(Arrays.asList(1, 2)); //manque 3
Assertions.assertIterableEquals(attendu, actuel); //echec

```

vérifie que les collections ont mêmes tailles et mêmes valeurs.

assertEquals() vérifie mêmes valeurs

assertSame() vérifie références sur même instance/objet .

AssertThrows avec lambda :

```
@Test
void verifierException() {
    String valeur = "123a";
    Assertions.assertThrows(NumberFormatException.class, () -> {
        Integer.valueOf(valeur);
    });
}
```

vérifie qu'une exception est bien levée (suite à erreur volontaire) et du bon type .

AssertTimeout avec lambda :

```
@Test
void verifierTimeout() {
    Assertions.assertTimeout(Duration.ofMillis(200), () -> {
        * traitement potentiellement trop lent *
        return "...";
    });
}
```

@Disabled("test temporairement désactivé")
sur classe ou méthode de test

@RepeatedTest(value = 3) est une variante de @Test permettant de lancer n fois un même test

5.9. Suppositions (pour exécuter ou pas un test selon le contexte) :

```
assumeTrue(System.getenv("OS").startsWith("Windows"));
```

//la suite du test ne sera exécutée que si os courant est Windows

//si le test n'est pas exécuté --> même comportement que si test vide --> ok/réussi/vert

//variante avec lambda exécutée qui si supposition ok :

```
assumingThat(System.getenv("OS").startsWith("Windows"), () -> {
```

```
assertTrue(new File("C:/Windows").exists(), "Répertoire Windows inexistant");
});
```

5.10. Tests imbriqués de JUnit5

```
public class MyTest {

    @BeforeEach
    void mainInit() {
        System.out.println("BeforeEach / first level");
    }

    @Nested
    class MonTestImbrique {
        @BeforeEach
        void subInit() {
            System.out.println("BeforeEach imbrique");
            valeur = 5;
        }

        @Test
        void simpleTestImbrique() {
            System.out.println("SimpleTest imbrique valeur=" + valeur);
            Assertions.assertEquals(5, valeur);
        }
    }
}
```

NB : En plaçant **@TestInstance(Lifecycle.PER_CLASS)** au dessus d'une classe de test Junit5 (imbriquée ou pas) , on a comme comportement le fait qu'une seule instance de la classe de test sera utilisée pour exécuter successivement toutes les méthodes de tests de la classe de test (contrairement au comportement par défaut PER_METHOD) .

NB2 : Le mode PER_CLASS permet aussi :

- que les méthodes annotées avec **@BeforeAll** et **@AfterAll** n'aient pas l'obligation d'être statiques
- d'utiliser les annotations **@BeforeAll** et **@AfterAll** sur des méthodes de classes annotées avec **@Nested**

5.11. Tests paramétrés via source/série de valeurs

Après avoir ajouté la dépendance nécessaire **junit-jupiter-params** , on peut écrire des méthodes de tests avec un paramètre qui sera alimenté avec une source/série de valeurs .

Un test paramétré sera ainsi lancé plusieurs fois (avec des arguments différents pour obtenir une certaine variété)

@ParameterizedTest

```
@ValueSource(ints = { 1, 2, 3 })
void testParametreAvecValueSource(int valeur) {
    assertEquals(valeur + valeur, valeur * 2);
}
```

@ValueSource(ints = { 1, 2, 3 }) ou @ValueSource(strings = { "un", "deux" }) ou ...

il existe également @EnumSource , @MethodSource , @CsvSource ,

```
import java.util.stream.Stream;
```

```
@ParameterizedTest
@MethodSource("fournirDonneesParametres")
void testTraiterSelonMethodSource(String element) {
    assertTrue(element.startsWith("elt"));
}

static Stream<String> fournirDonneesParametres() {
    return Stream.of( "elt1" , "elt2" );
}
```

5.12. Tests dynamiques / @TestFactory

```
...
import static org.junit.jupiter.api.Assertions.assertEquals;

import java.util.ArrayList;
import java.util.Collection;
import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.TestFactory;

class MonTestSimple {

    @TestFactory
    Collection<DynamicTest> dynamicTestsWithCollection() {
        Collection<DynamicTest> myDynamicTests = new ArrayList<>();
        for (int i = 1; i <= 5; i++) {
            int val = i;
            myDynamicTests.add(DynamicTest.dynamicTest("Ajout " + val + "+" + val,
                () -> assertEquals(val * 2, val + val)));
        }
        return myDynamicTests;
    }
}
```

--> à priori intéressant que si couplé avec une introspection dynamique (framework de tests).

5.13. Suite de Tests et Tags

@Tag("nomDeTag") de JUnit5 peut être placé sur une classe ou bien une méthode de test

Ces noms de tags (préalablement appelés catégories en Junit4) permettront d'effectuer ultérieurement des filtrages sur les tests à lancer ou pas .

Il est éventuellement possible de placer plusieurs tags complémentaires au dessus d'un même test (ex : @Tag("prod") @Tag("dev")) .

```
import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.runner.RunWith;
```

```
@RunWith(JUnitPlatform.class)
@SelectPackages({ "com.xyz.test.p1" }) //ne lancer que les tests de ce(s) package(s)
@IncludeTags({"prod","dev"}) //ne lancer que les tests qui ont ces tags
//il existe aussi @ExcludeTags ne pouvant pas être utilisé en même temps que @IncludeTags
public class MyTestSuite1
{
}
```

NB : par défaut , **@SelectPackages(...)** sélectionne tous les sous-packages également .

On peut éventuellement ajouter **@IncludePackages(...)** ou bien **@ExcludePackages(...)** pour filtrer les sous-packages à sélectionner.

On peut également ajouter **@IncludeClassNamePatterns({ "^.Simple\$" })** pour filtrer les noms des classes de Tests à lancer .

5.14. Bonnes pratiques "TDD" (Test Driven Developpement)

Utilités/objectifs des tests unitaires

- **Bien appréhender/formaliser le contrat technico/fonctionnel d'un composant logiciel** (en lisant le code d'un test bien écrit , on comprend le comportement attendu des opérations/méthodes du composant à tester).
- **Trouver les erreurs rapidement et simplifier la maintenance :**
Une fois le test unitaire écrit, on peut le relancer automatiquement sans effort des milliers de fois pour *vérifier l'absence de régression* et pour *localiser rapidement une erreur* en cas de problème .
- **Développer consciencieusement** (en testant naturellement l'absence de bug et le bon fonctionnement) au fur et a mesure de la programmation.
- S'assurer que tout le code écrit (et prévu pour être appelé/invoqué) soit **couvert** par un nombre suffisant de tests unitaires.
- Ne pas se contenter de la boutade "*tester c'est douter*" !

Stratégie habituelle de test (Test Driven Development)

- 1) **définir la structure du composant à tester** (exemple: *modèle UML* , *interface java* , ...) et un éventuel embryon d'implémentation.
- 2) **écrire la classe de test** (en s'appuyant sur une structure connue et définie du composant à tester mais sur une implémentation qui n'est pas encore opérationnelle) .
- 3) **lancer une première fois les tests unitaires** et y vérifier normalement que "sans code d'implémentation finalisé" les tests remontent bien des "échecs" .
- 4) **écrire le code d'implémentation du composant à tester.** Relancer les tests qui devraient normalement réussir.
- 5) coder et tester des **variantes** au niveau des tests (paramètres d'entrées volontairement erronés , vérification de remontée d'exception , ...)
- 6) poursuivre de manière incrémentale et itérative (nouvelle méthode,...)

Bonnes pratiques sur tests unitaires

- Tester essentiellement les méthodes publiques (pas les méthodes privées)
- Factoriser (lorsque c'est possible) le code de quelques tests (en appelant des sous méthodes de la classe de test , en héritant de classes utilitaires sur les Tests , ..)
- En cas de test qui ne passe plus , d'abord remettre en question le code du composant à tester , puis ensuite le code du test lui même (qui peut également être bogué ou bien trop simpliste).
- Utiliser éventuellement des "mocks" (composants en arrière plan simulés) pour de multiples bonnes raisons qui doivent cependant se justifier selon le contexte . Trop de "mocks" ralentissent quelquefois le développement et rendre le code des tests moins lisible .

6. Bénéfices de l'intégration continue

6.1. Objectifs de l'intégration continue

Sur un gros (ou moyen) projet, chaque développeur se concentre une une partie bien précise et génère des simples composants devant ultérieurement être assemblés entre eux pour produire l'application complète.

Sans automatisme, il faut alors manuellement:

- vérifier que les différents composants soient bien compatibles (mêmes versions, prévus pour s'interfacer entre eux)
- rassembler/packager les composants dans des modules exécutables (.exe, .dll , .jar , .ear, ...)
- déployer le tout sur un serveur d'application ou dans des conteneurs "dockers"
- lancer des tests globaux

Ce qui peut prendre beaucoup trop de temps !!!!!!

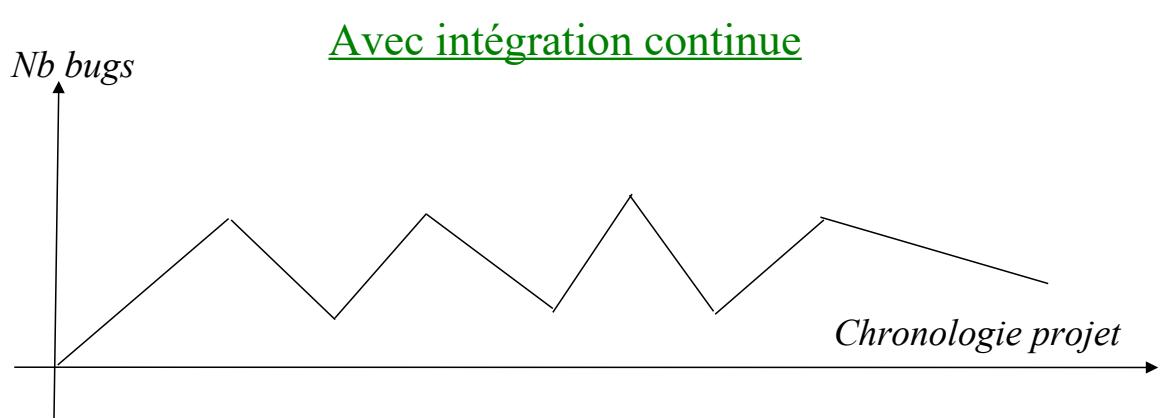
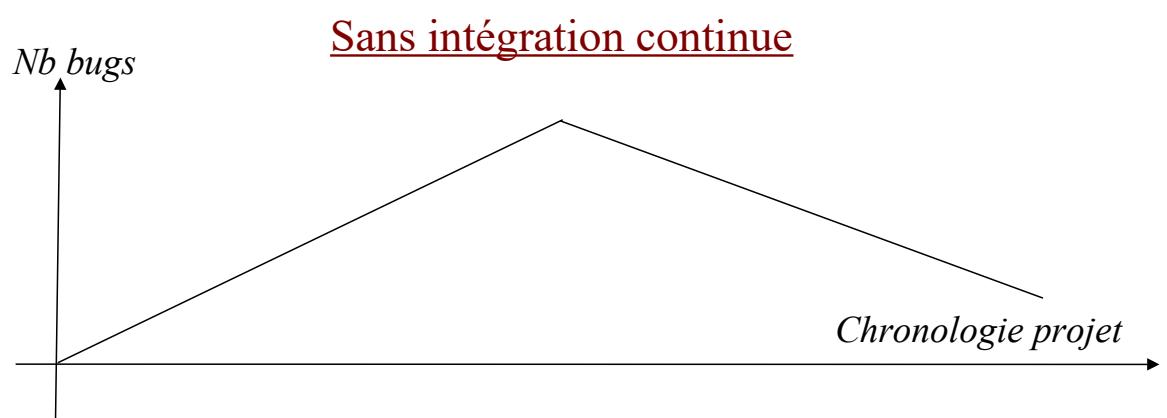
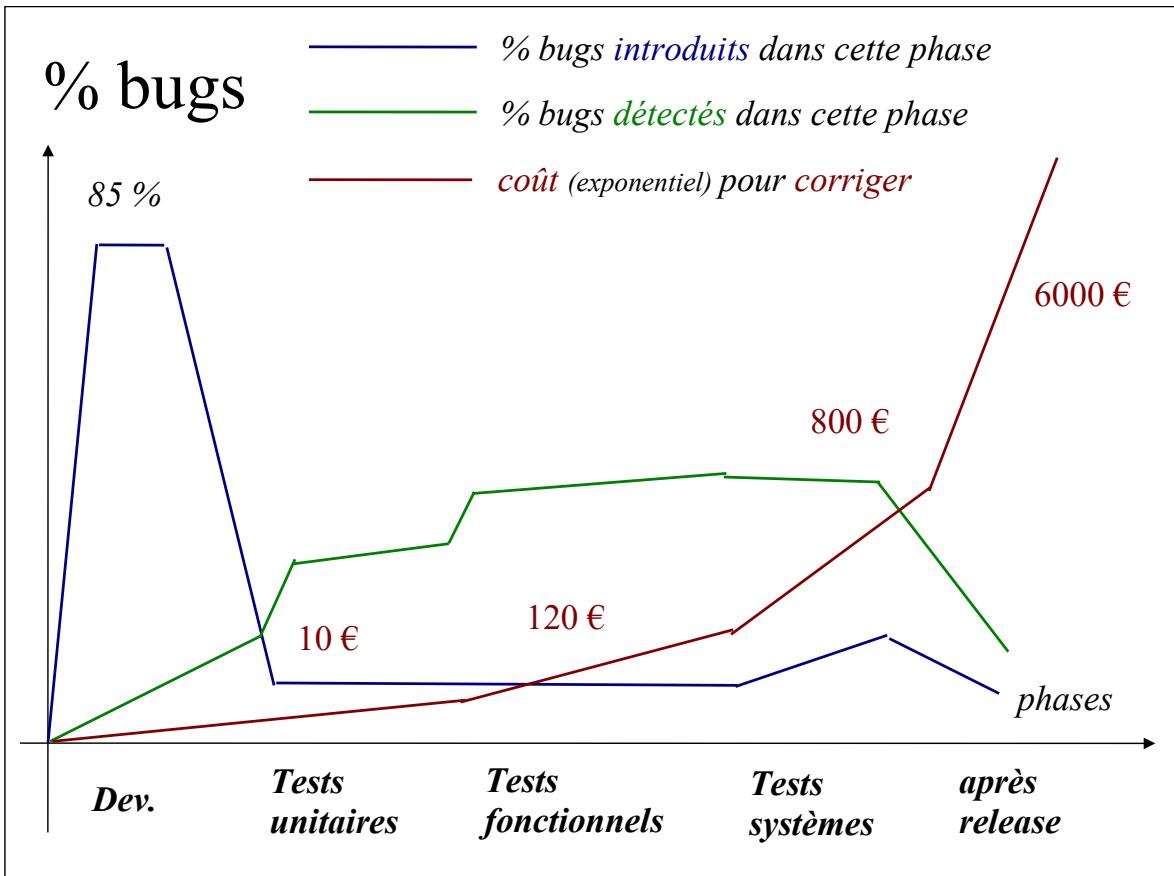
Pour rester concurrentielle , une SSII/ESN ou une maîtrise d'oeuvre interne doit s'appuyer sur un système automatisant la plupart des points précédents.

Un tel serveur système dit "d'intégration continue" va (*à peu près*):

- récupérer le code source du composant dans un répertoire neutre (indépendant de l'ide)
- recompiler ce code source (pour bien contrôler la version du compilateur utilisé)
- packager le code compilé d'un composant ou d'un module dans une archive adéquate (.jar , .war , .ear)
- déployer l'ensemble sur un environnement de tests (JVM , serveur d'application, docker,...)
- lancer des jeux de test qui ont été préalablement préparés
- remonter des messages et des statistiques
- historiser une nouvelle version du logiciel si les tests ont réussi

Dans le monde java, la plupart des environnements d'intégrations continues sont basés sur les technologies fondamentales suivantes:

- **ANT** (sorte de makefile en XML et donc indépendants de la plateforme)
- **MAVEN** (gestionnaire de projet indépendant de l'IDE)
- **GRADLE** (un peu plus moderne que maven mais moins utilisé pour l'instant)



Déetecter et corriger les bugs au plus tôt pour minimiser les risques et les coûts

En règle générale, plus une erreur est détectée tard , plus sa correction est coûteuse et d'autre part la plupart des bugs sont introduits durant la phase précoce de développement alors que leurs détections est reportée aux tests.

En automatisant le lancement très régulier et fréquent de toute une batterie de tests, l'intégration continue permet de détecter et corriger au plus tôt un grand nombre de bugs.

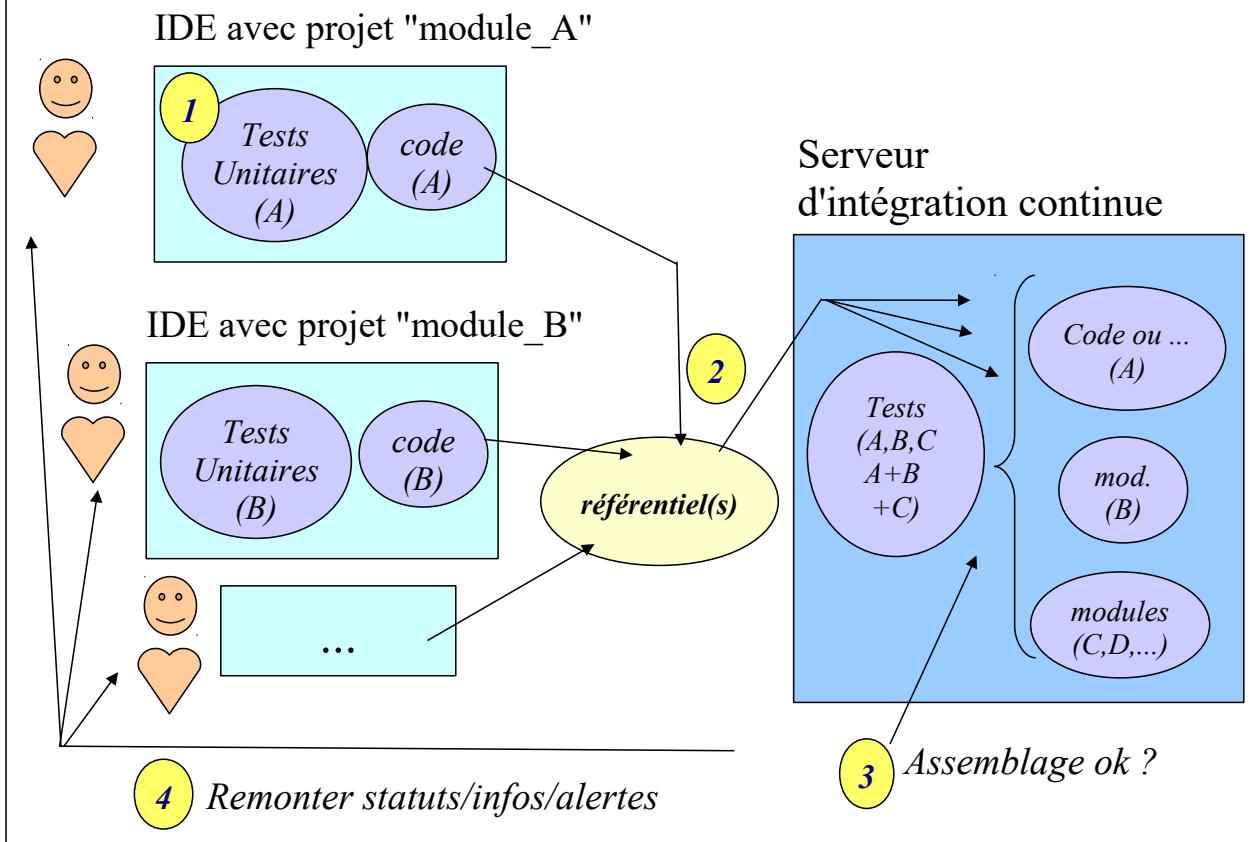
L'intégration continue est une des bonnes pratiques préconisées par les **méthodes agiles** dont XP (eXtreme Programming) .

L'**intégration continue** est décrit par son inventeur **Martin Fowler** comme:

* ... UNE PRATIQUE DE DEVELOPPEMENT LOGICIEL OU LES MEMBRES D'UNE EQUIPE INTEGRENTEUR LEUR TRAVAIL FREQUEMMENT, HABITUELLEMENT CHACUN AU MOINS UNE FOIS PAR JOUR - CE QUI ENTRAINE PLUSIEURS INTEGRATIONS PAR JOUR. CHAQUE INTEGRATION EST VALIDEE PAR UN 'BUILD' AUTOMATIQUE (CE QUI INCLUT LES TESTS) POUR DETECTER LES ERREURS D'INTEGRATION AUSSI VITE QUE POSSIBLE.
 CETTE APPROCHE PERMET DE REDUIRE SIGNIFICATIVEMENT LES PROBLEMES D'INTEGRATION ET PERMET A UNE EQUIPE DE DEVELOPPER DES LOGICIELS DE QUALITE PLUS RAPIDEMENT... *

* Extrait (traduit en français) du site <http://www.martinfowler.com/articles/continuousIntegration.html>

Intégration continue



6.2. Présentation et installation de Jenkins

Premiers pas avec Jenkins

Jenkins est actuellement un **logiciel d'intégration continue très en vogue** car il est très simple à configurer et à utiliser.

Installation de Jenkins :

Recopier **jenkins.war** dans **TOMCAT_HOME/webapps** (avec un éventuel Tomcat dédié à l'intégration continue configuré sur le port 8585 ou autre).

Etant donné que la configuration de jenkins ne nécessite pas de base de données relationnelle (mais de simples fichiers sur le disque dur), il n'y a rien d'autre à configurer lors de l'installation .



Url de la console "jenkins" :

<http://localhost:8585/jenkins>

Premier menu à activer :

Administrer Jenkins / Configurer le système

NB : il est également possible de démarrer une version récente de jenkins sans serveur tomcat via un script de ce genre :

startJenkins.bat

```
set JAVA_HOME=C:\Program Files\Java\jdk-11.0.12
set PATH="%JAVA_HOME%/bin";%PATH%
REM java -jar jenkins.jar -D"hudson.plugins.git.GitSCM.ALLOW_LOCAL_CHECKOUT=true"
java -jar jenkins.jar
```

et après un tel lancement l'url menant à la console jenkins sera simplement <http://localhost:8080>

6.3. Configuration nécessaire lors du premier démarrage

Lire le mot de passe temporaire à la console lors du premier démarrage (ex:
d1223a3ba2a44d079ecb7deec0625de8)
et reporter/recopier celui-ci dans la console de jenkins

Installer quelques plugins fondamentaux (ceux qui sont suggérés)

Configurer un compte principal (administrateur) pour les futurs démarrages :
par exemple *username=admin password=admin123*

6.4. Installation ou mise à jour de plugins pour Jenkins

Menu "tableau de bord" / "Administre Jenkins" / "Gestion des plugins"

6.5. Configuration élémentaire d'une tâche "jenkins"

Menu "tableau de bord" / "Nouveau item"
puis :

- donner un nom (ex : *jobXy*)
- choisir souvent "**projet free-style**" pour les cas simples/ordinaires
- **OK**

Dans la partie "gestion du code source", choisir généralement :

- **GIT**

et préciser l'url du référentiel git (par exemple <https://github.com/....repoXy>)

Attention: les versions récentes de Jenkins n'acceptent des URLs de type file:///c:/xx/yy qu'avec l'option -D"huson.plugins.git.GitSCM.ALLOW_LOCAL_CHECKOUT=true" à fixer au démarrage
et dans la partie "branch to build" on pourra par exemple choisir */master ou */main .

Dans la partie "build" , choisir généralement :

- **Invoquer les cibles Maven de haut niveau**

et préciser la "**cible (ou goal)**" maven à déclencher (ex : clean package)

NB: si le projet maven à construire est dans un sous répertoire du référentiel git (cas pas très conseillé mais admis), alors au niveau du build on peut préciser un chemin menant au pom.xml de type sous_rep1 ou bien sous_rep1/sous_sous_rep2 dans config avancée .

Sauvegarder assez rapidement ces configurations essentielles.

Les configurations secondaires annexes pourront être ajoutées ultérieurement

Lancement sur demande d'un "build" (associé à un job jenkins configuré)

The screenshot shows the Jenkins dashboard. On the left, there's a sidebar with icons for 'Lancer un build' (Launch build), 'Supprimer Maven project' (Delete Maven project), 'Configurer' (Configure), and 'Modules'. The main area displays a 'Historique des builds' (Build History) card. It shows one build entry: '#1' from '21 avr. 2015 14:03'. Below the card are two RSS feed links: 'RSS des builds' and 'RSS des échecs'.

Affichage des résultats via la console de jenkins

La logique de navigation/sélection de jenkins est la suivante :

**Jenkins (server) > Job (name/type/config) > number of instance
(with status/results)**

Exemple: Jenkins ▶ my-java-app1 ▶ #4

Après avoir sélectionné un des niveaux , on accède à un menu (coté gauche) pour :

- * créer/activer de nouveaux éléments
- * (re)configurer plus en détails l'élément sélectionné
- * afficher des détails sur l'élément sélectionné
- * ...

Concernant les résultats d'un build, la partie la plus intéressante est souvent "*sortie console*" :



Sortie de la console

```
-----
[INFO] BUILD SUCCESS
[INFO]
-----
[INFO] Total time: 24.003s
[INFO] Finished at: Tue Apr 21 14:51:42 CEST 2015
[INFO] Final Memory: 12M/32M
[INFO]
```

6.6. Différents types de "builds" (avec Jenkins)

Différents types de "builds"

Types de "build"	Commentaires/considérations
Local / privé	Lancé manuellement (et idéalement fréquemment) par le développeur (depuis son IDE). → <i>Permet de savoir si ses propres changements fonctionnent</i>
Intégration rapide (de jour)	Tests d'intégration rapides déclenchés après chaque commit ou bien régulièrement (ex : toutes les 20 minutes). Seuls les tests rapides (unitaires + intégrations) sont lancés → <i>Permet de savoir si l'assemblage des changements de tous les développeurs fonctionne .</i>
Intégration journalière poussée/sophistiquée à heure fixe (nightly build)	Tests sophistiqués (longs) , tests de performance , génération de documentation, de rapports , → <i>Permet de savoir si la dernière version produite du logiciel est en état de marche .</i>

Réglages de fréquence via une syntaxe "crontab" (par exemple dans Jenkins) :

Syntaxe "crontab" :

mm hh jj MM JJ [tâche]

mm représente les minutes (de 0 à 59)

hh représente l'heure (de 0 à 23)

jj représente le numéro du jour du mois (de 1 à 31)

MM représente le numéro du mois (de 1 à 12)

JJ représente le numéro du jour dans la semaine

(0 : dimanche , 1 : lundi , 6 : samedi , 7:dimanche)

Si, sur la même ligne, le « *nombre du jour du mois* » et le « *jour de la semaine* » sont renseignés, alors **cron** (ou) n'exécutera la *tâche* que quand ceux-ci coïncident .

.../...

Réglage de la fréquence des "builds"

Pour chaque valeur numérique (mm, hh, jj, MMM, JJJ) les notations possibles sont :

* : à chaque unité (0, 1, 2, 3, 4...)

5,8 : les unités 5 et 8

2-5 : les unités de 2 à 5 (2, 3, 4, 5)

*/3 : toutes les 3 unités (0, 3, 6, 9...)

10-20/3 : toutes les 3 unités, entre la dixième et la vingtième
(10, 13, 16, 19)

Et donc pour un nightly build d'intégration continue :

---> **0 3 * * 1-6** (tous les jours à 3h du matin
sauf les dimanches)

et pour un build rapide de jour :

→ ***/15 8-20 * * 1-5** (tous les jours sauf les week-ends ,
toutes les 15 minutes de 8h à 20h)

Lancement périodique (ex journalier) au niveau de Jenkins

Construire périodiquement ?

Planning ?

0 3 * * 1-6

Ce champ suit la syntaxe de cron (avec des différences mineures). Chaque ligne consiste en 5 champs séparés par des TABs ou des espaces :

 MINUTES HEURES JOURMOIS MOIS JOURSEMAINE

 MINUTES Les minutes dans une heure (0-59)

 HEURES Les heures dans une journée (0-23)

 JOURMOIS Le jour dans un mois (1-31)

 MOIS Le mois (1-12)

 JOURSEMAINE Le jour de la semaine (0-7) où 0 et 7 représentent le dimanche

dans cet exemple : tous les jours (de lundi à samedi) à 3h du matin. "nightly build"

Lancement sur détection périodique des changements (SVN/GIT)

- Scrutation de l'outil de gestion de version

Planning

*/15 8-20 * * 1-5

dans cet exemple : Jenkins vérifie toutes les 15 minutes si certains changements ont eu lieu au niveau du référentiel de code source (de lundi à vendredi et de 8h à 20h). En cas de changement détecté → lancement (potentiellement très fréquent) d'un build d'intégration.

Suppression des anciens builds (jenkins)

- Supprimer les anciens builds ()

Nombre de jours de conservation des builds

15

si non vide, les enregistrements de build seront conservés au maximum ce nombre de jours

Nombre maximum de builds à conserver

150

si non vide, pas plus de ce nombre de builds ne sera conservé

6.7. Exemple de pipeline Jenkins

```

pipeline {
    agent any

    stages {
        stage('SCM') {
            steps {
                // Get some code from a GitHub repository
                //git 'https://github.com/jglick/simple-maven-project-with-tests.git'
                git url : 'file:///C:/referentiels/appliSpringJavaWeb.git', branch : 'main'
            }
        }
        stage('Build') {
            steps {
                // Run Maven on a Unix agent.
                //sh "mvn -Dmaven.test.failure.ignore=true clean package"

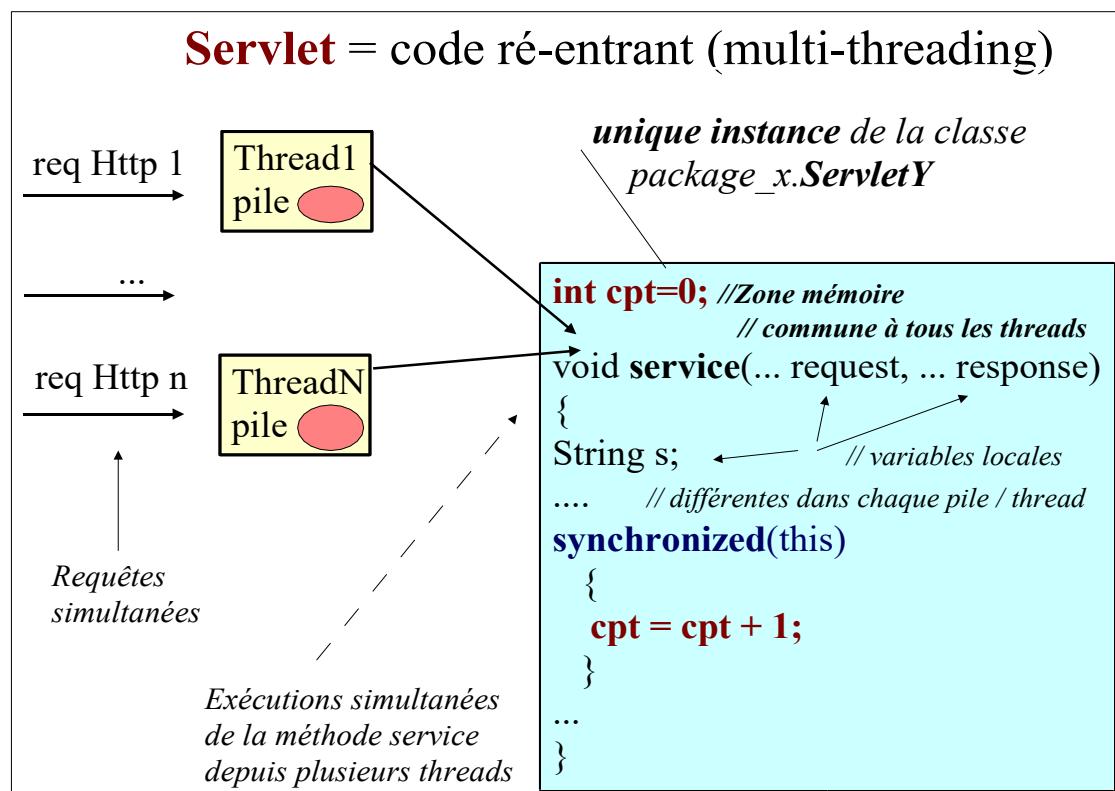
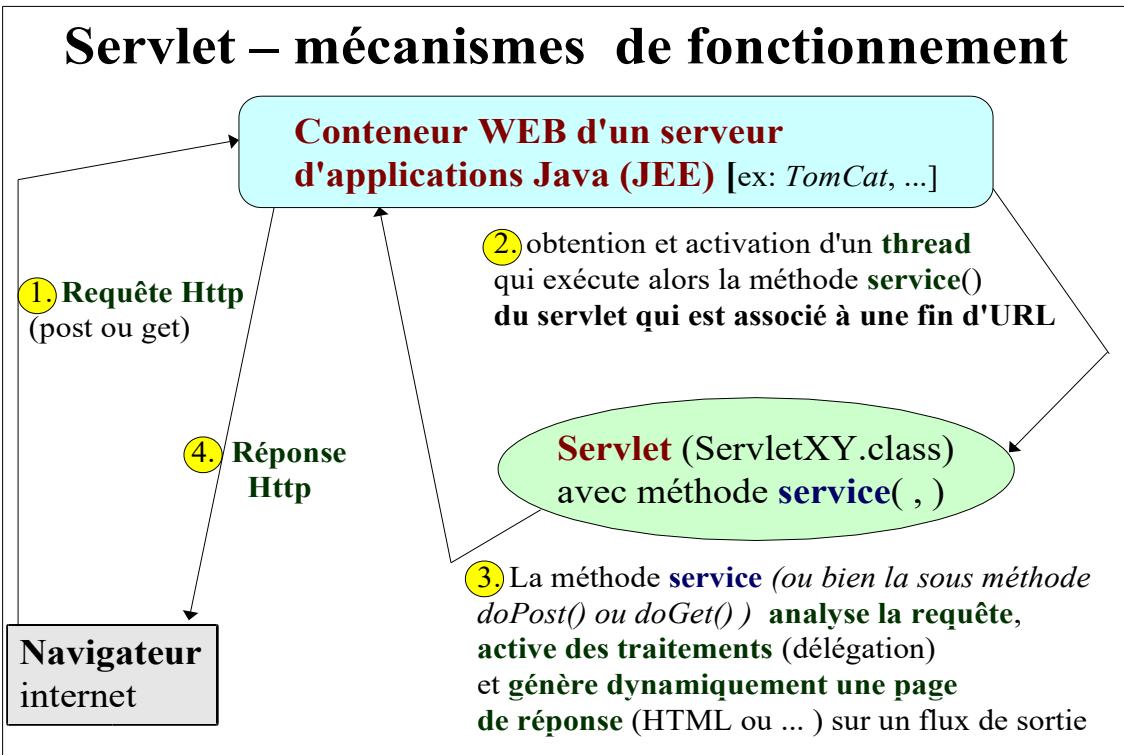
                // To run Maven on a Windows agent, use
                bat "mvn -Dmaven.test.failure.ignore=true clean package"
            }
        }
        post {
            // If Maven was able to run the tests, even if some of the test
            // failed, .....
            success {
                bat "mvn javadoc:javadoc"
                echo "OK, on pourrait faire autre chose ici"
            }
        }
    }
    stage('sonar scan') {
        steps {
            bat 'mvn sonar:sonar -Dsonar.host.url=http://localhost:9000
                 -Dsonar.login=admin -Dsonar.password=admin123'
        }
    }
    stage('prepa_docker') {
        steps {
            echo 'construction container docker possible (souvent sous linux)'
        }
    }
}

```

III - IHM Web

1. Les moteurs de Servlets

Un servlet est un **composant java** capable de générer dynamiquement des documents WEB (*html, xml, texte, image svg, ...*). Un *servlet s'exécute toujours côté serveur* au sein d'un "conteneur WEB".

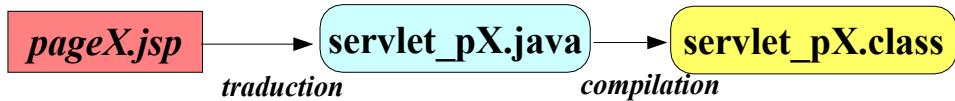


2. Java Server Pages

2.1. Pages JSP

Présentation des pages JSP

- page xml ou html comportant des portions (code java entre <% et %> ou bien balises spéciales) interprétées côté serveur.
- Les pages jsp sont essentiellement utilisées pour contrôler l'affichage des éléments.
- Les mécanismes internes de J2EE transforment la page Jsp en un servlet qui est ensuite automatiquement compilé avant l'exécution .



exemple

```

<% java.util.Date d = new java.util.Date(); %>
<html>
<body>
Date = <b> <%= d.toString() %> </b>
</body>
</html>
  
```

Les pages JSP peuvent assurer les mêmes fonctionnalités que les servlets.

Ils sont **beaucoup plus simples à écrire** (surtout en ce qui concerne l'encodage de l'affichage) .

2.2. TagLib (JSP) et JSTL

TagLib (pour pages JSP)

- Bibliothèque de balises spécifiques codées sous forme de classes java puis placées au niveau des pages JSP et interprétées coté serveur.
- permet de simplifier la syntaxe des pages JSP
- permet d'obtenir une syntaxe plus homogène (moins de mélange <% java %> , html).
- JSTL : Jsp Standard Tag Library
- Autres TagLibs classiques:
 - celles de struts (ex: <bean:write /> , ...)
 - celles de JSF (ex: <h:inputText .../> , ...)

Exemple (sans taglib):

```
<% java.util.Iterator it = listeProduits.iterator();
  while (it.hasNext()) {
    Produit prod = (Produit) it.next();
    %>
    <i> <%=prod.getLabel()%> </i> ,
    <b> <%=prod.getPrix()%> </b><br/>
<% } %>
```

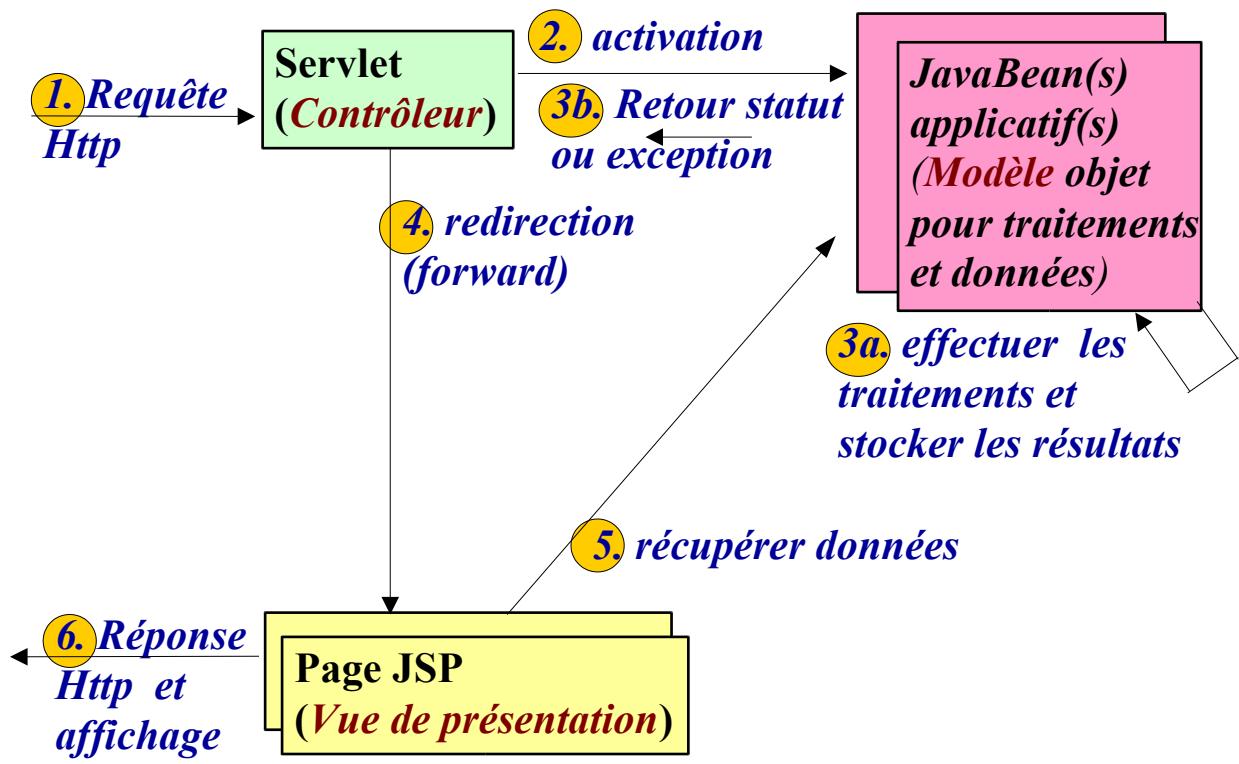
Exemple équivalent (avec <c:forEach> de JSTL) :

```
<c:forEach var="prod" items="${listeProduits}" >
  <i> <c:out value="${prod.label}" /> </i> ,
  <b> <c:out value="${prod.prix}" /> </b><br/>
</c:forEach>
```

3. Modèles MVC

MVC = Modèle - Vue - Contrôleur

MVC(2) : "Servlet + page JSP + JavaBean"

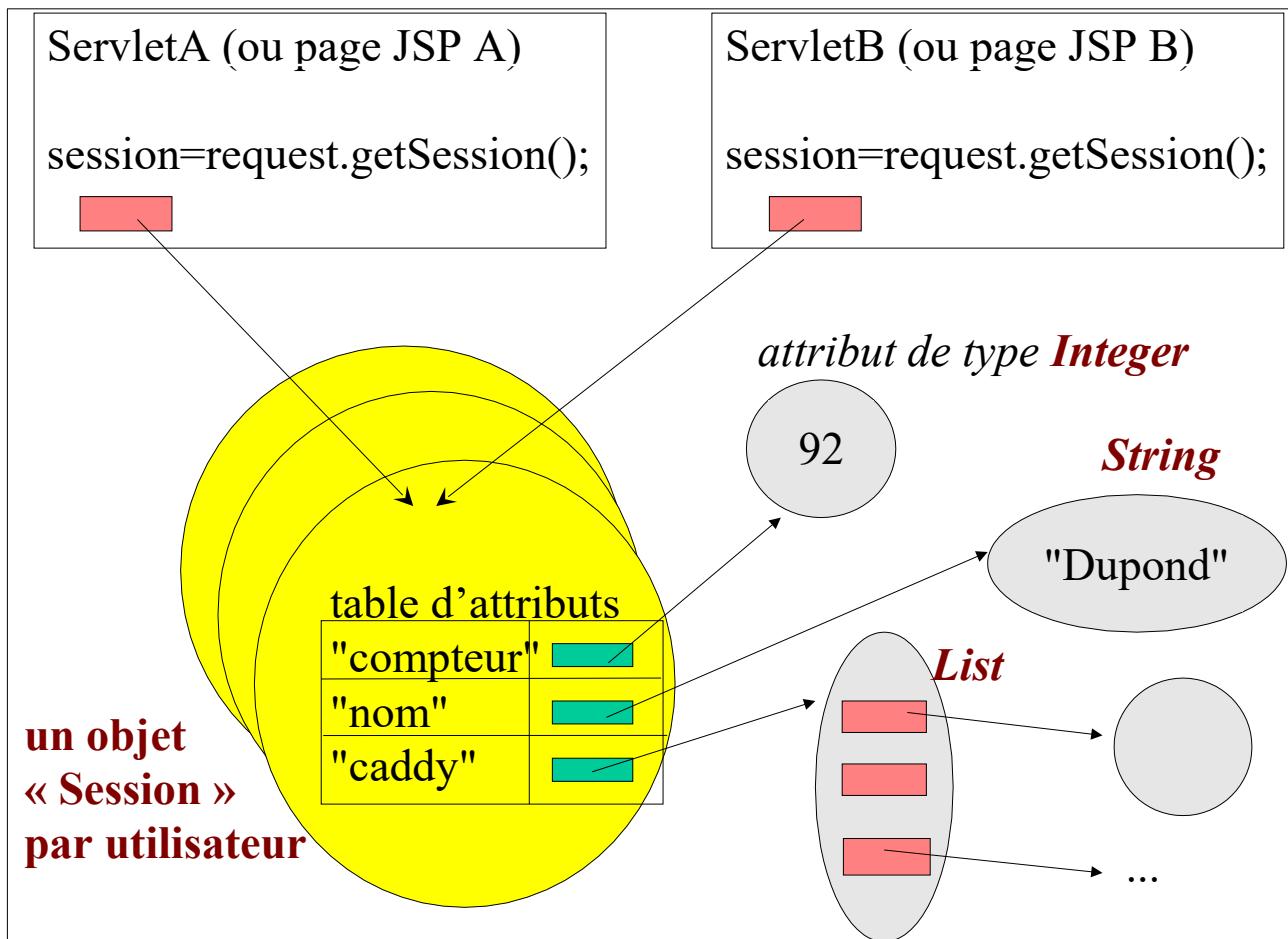


- Le **Servlet** joue le rôle de **contrôleur** : il reçoit une requête, lui applique un éventuel contrôle de saisie (**validation** quelquefois déléguée ou automatisée).
- Le contrôleur demande à un **JavaBean** d'**effectuer les traitements**. Le JavaBean effectue (ou bien *délegue*) les traitements (accès Jdbc, *EJB*,...) et récupère les résultats qu'il mémorise dans ses attributs (*ou dans d'éventuel(s) JavaBean(s) annexes de données*). Ce(s) **JavaBean(s)** joue(nt) le rôle de **Modèle**.
- Le contrôleur après avoir reçu un statut (ok/ko) ou une exception effectue une **redirection (forward)** vers une page JSP pour l'affichage du résultat ou vers une page JSP d'erreur.
- La **page JSP** (jouant le rôle de **Vue**) **récupère les données nécessaires à l'affichage auprès du JavaBean** et génère (met en forme) la page HTML à renvoyer.

4. Session HTTP

Une session HTTP est un **objet (propre à chaque utilisateur)** qui est :

- maintenu en mémoire dans le "conteneur Web"
- utilisé pour établir un lien entre les différentes requêtes émises successivement par un même utilisateur (login.jsp , menu.jsp , ... , commande.jsp).



Les mécanismes internes des serveurs JEE utilisent des cookies (ou à défaut des ré-écritures d'URL avec suffixes) pour véhiculer un identifiant d'objet session et pour ainsi pallier le fait que le protocole HTTP est sans état.

Remarques techniques:

- Un objet "session" est automatiquement détruit coté serveur au bout d'un certain temps d'inactivité de la part de l'utilisateur (*timeout de session réglable et de 15/20 minutes par défaut*).
 - Dans le cadre d'un fonctionnement en cluster (application s'exécutant sur plusieurs serveurs) , les contenus des objets sessions sont quelquefois répliqués pour anticiper des pannes .

5. De Struts à Java Server Faces

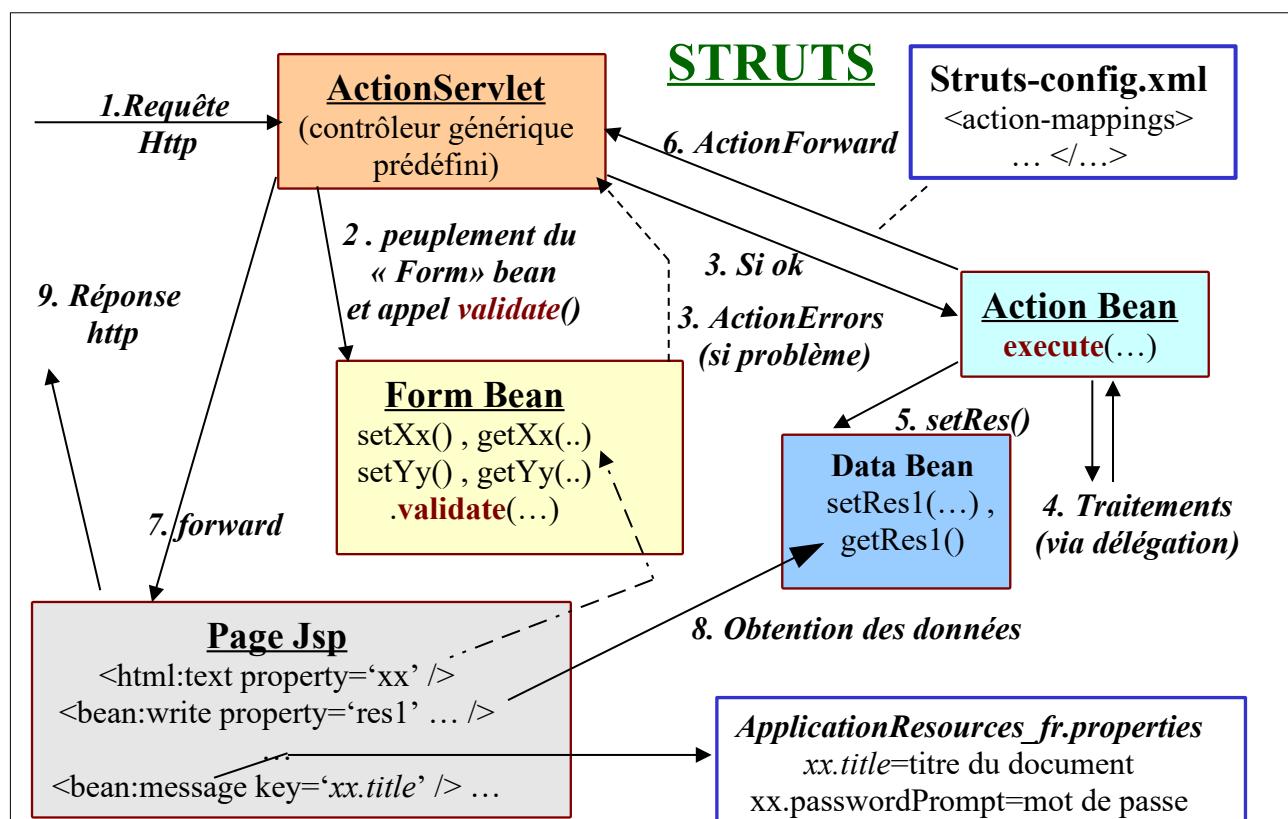
5.1. Struts 1

Créé au début des années 2000 , le framework "STRUTS" sert à automatiser les éléments du design

pattern "MVC2" .

L'architecture de STRUTS repose sur les éléments suivants :

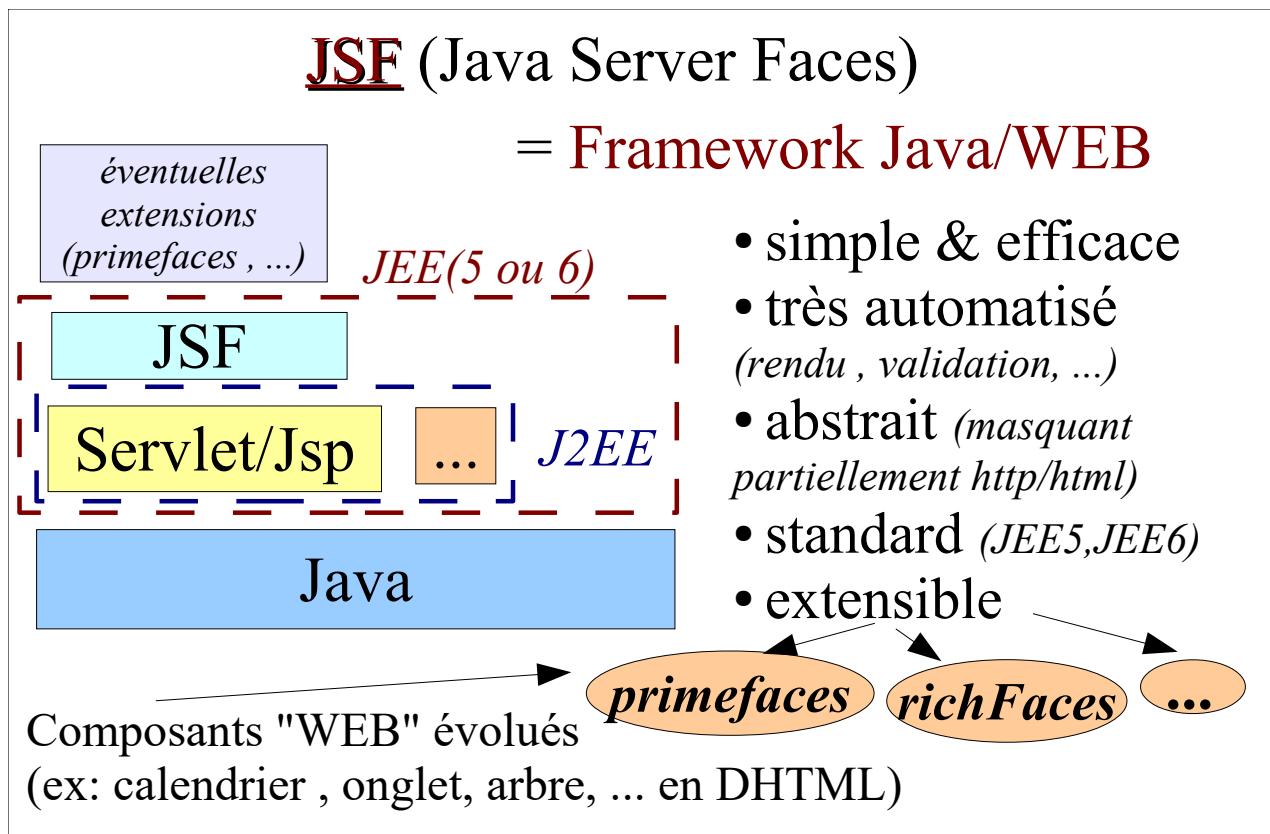
- Un **servlet prédefini** « **ActionServlet** » jouant le rôle de « **Super Contrôleur** » générique.
- Des « **java bean** » de type « **ActionForm Bean** » (à programmer) *faisant office de « firewall entre http et les actions internes* : un objet « **ActionForm Bean** » sert à gérer les paramètres http (stockage , relecture , vérification de saisies).
- Nb** : un objet « **ActionForm Bean** » est généralement utilisé pour **re-peupler un formulaire html avec les champs non erronés** (L'utilisateur n'a plus qu'à re-saisir les champs à problème).
- Des « **java bean** » de type **Action** (à programmer) et jouant le rôle de « **sous servlet** » : **un objet « action » déclenche des traitements** , peuple éventuellement un « **DataBean** » avec les résultats et **initialise un renvoi vers une vue** (page JSP) en utilisant les « **ActionMapping** » du fichier de configuration **struts_config.xml** .
- Des **pages JSP** dont la partie dynamique est en quasi totalité gérée via des balises prédefinies (TagLib **struts-html** et **struts-bean**).



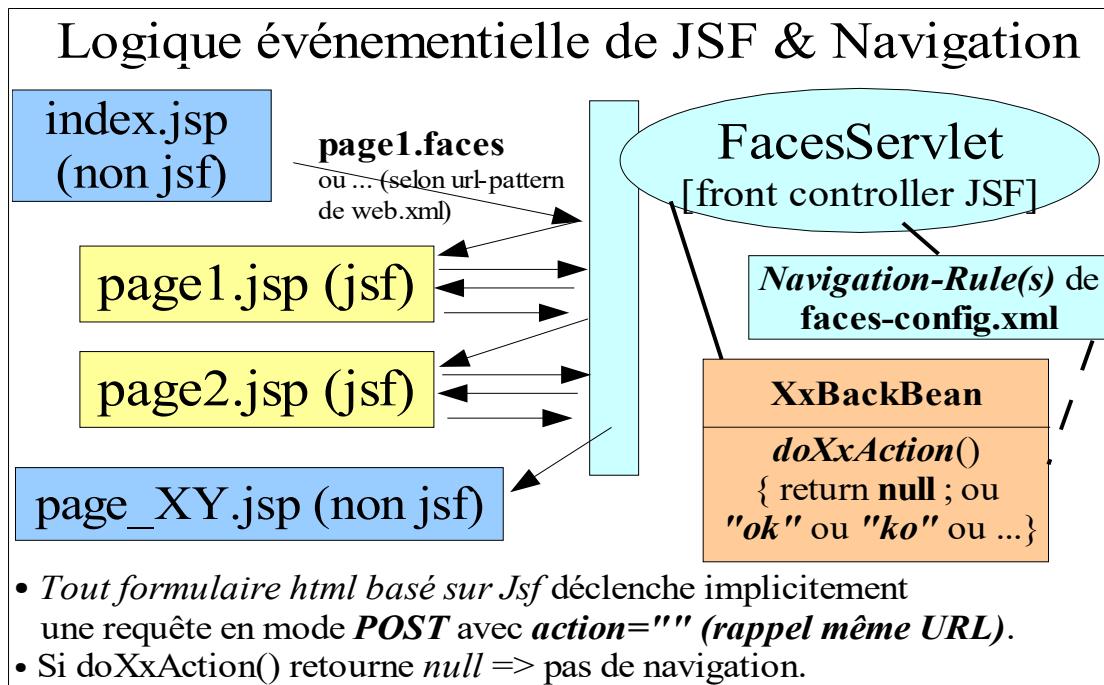
--> Bon framework devenant petit à petit "has been" mais encore utilisé sur de nombreuses applications à maintenir et/ou faire évoluer.

--> De nouveaux frameworks plus récents (ex: JSF) sont aujourd'hui plus élaborés tout en étant plus simples à configurer.

5.2. Framework JSF



5.2.a. Logique événementielle et navigation

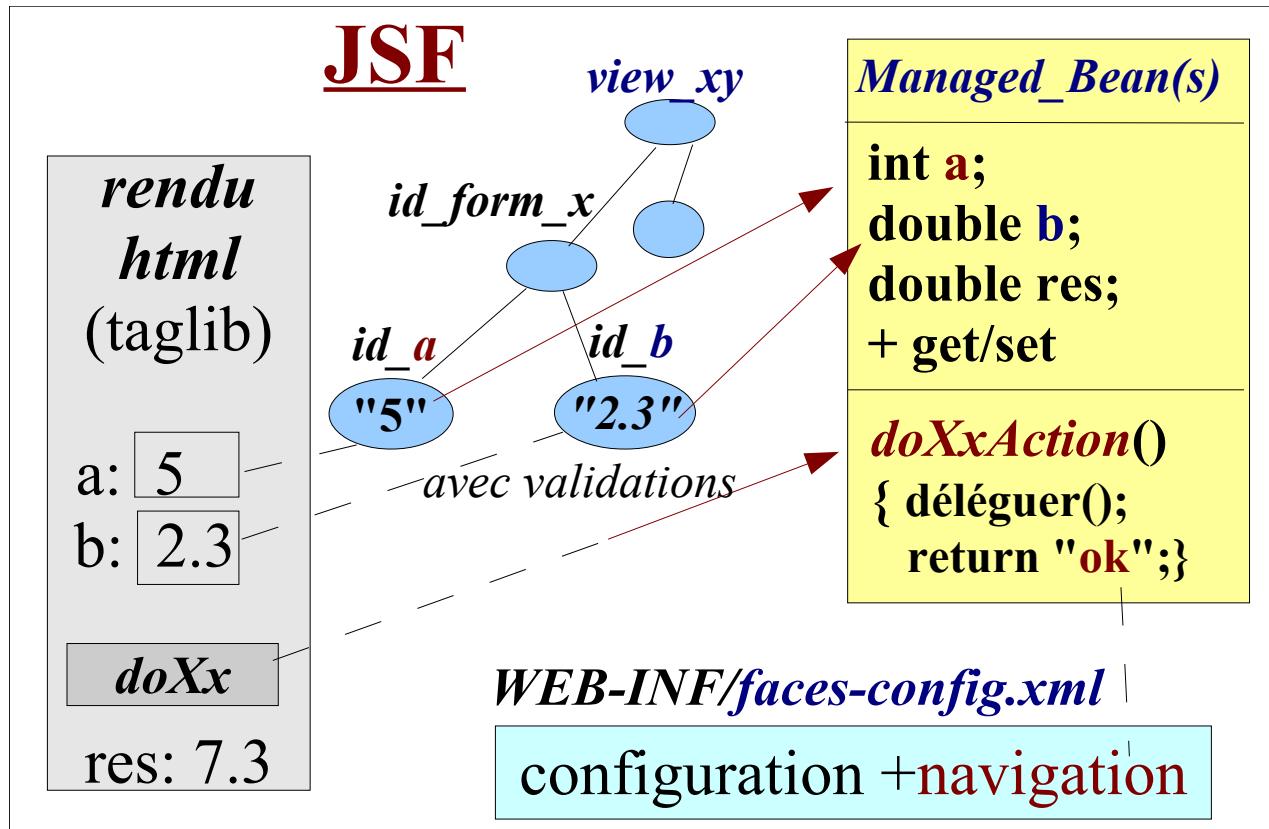


5.2.b. Quelques traits de JSF

JSF ==> accès aux composants via leurs noms , pas d'héritage imposé (contrairement à STRUTS)

JSF ==> fichier de configuration simple : ***faces-config.xml*** ou ***annotations*** avec ***JSF2***
 JSF ==> orienté "Drag & Drop" + propriétés + événements

5.2.c. Arbre de composants JSF (racine=view)



Un servlet prédéfini (`javax.faces.webapp.FacesServlet`) interne au framework JSF intercepte la requête http (`url = ".../faces/xxx.faces"`) pour effectuer les choses suivantes:

- Tenir compte du modèle de composants (ex: form html, zone de texte html , ...) décrit par les "taglib" de xxx.jsp pour mettre en place en mémoire un **arbre de composants**. Chacun de ces composants de bas niveau va ensuite être rempli avec l'une des données véhiculées par les paramètres de la requête http.
- En fonction des vérifications qui sont alors automatiquement déclenchées (valeurs numériques ?, valeurs comprises entre un minimum et un maximum ?, ...), des messages d'erreurs seront s'il le faut automatiquement générés et renvoyés au sein du formulaire de saisie qui réapparaît alors dans le navigateur (en même temps que les anciennes valeurs saisies).
- Si aucune erreur n'est détectée , un peuplement automatique d'un JavaBean de données/traitements (précisé par xxx.jsp ou xxx.xhtml) est alors déclenché puis un mécanisme événementiel prend ensuite le relais pour continuer les opérations.

5.2.d. Implémentations de JSF

L'api **JSF** (et sa documentation officielle) est à télécharger depuis le site de SUN.

Le package officiel de l'API JSF est **javax.faces** (javax.faces.event , javax.faces.component.html , ...).

Versions de JSF:

Versions de l'API JSF	Caractéristiques
1.0 (2004)	basé sur Servlet 2.3 & Jsp 1.2 (pour J2EE 1.3 ou 1.4 ou +)
1.1	comme 1.0 mais avec moins de bugs et avec de meilleures performances (pour J2EE 1.3 ou 1.4 ou +)
1.2	basé sur Servlet 2.5 & Jsp 2 (pour Java5 /JEE5) [nécessite Tomcat 6, ...] Ajouts de la version 1.2 : - unified expression language (EL commun à JSP2 & JSF] - ajax support , , fichier de config avec xsd (et plus de dtd)
2.0 , 2.1 , 2.x (depuis JEE6)	Avec annotations , navigations par défaut et intégration des " facelets " <i>Attention</i> : certaines nouveautés de JSF2 exigent ".xhtml" (".jsp") !

Implémentations:

Implémentations de JSF	Editeur	Caractéristiques
MyFaces	Jakarta/Apache	s'intègre facilement dans une appli. Java/web (pour Tomcat ou ...). http://myfaces.apache.org/
Jsf-Sun-RI	Sun	http://java.sun.com/j2ee/javaserverfaces/download.html
ibm-jsf	IBM	intégré à RAD6 , RAD7 et WebSphere 6.0 , 6.1

Extensions graphiques pour JSF:

composants graphiques JSF évolués (ex: calendrier , onglets , arbres , menus déroulants) [*rendu en Html + javaScript*]

Implémentations de JSF	Editeur	Caractéristiques
Tomahawk	Jakarta/Apache	<i>Historiquement première extension pour JSF (sans ajax)</i>
RichFaces 3	JBoss Labs	extension pour Ajax + composants graphiques évolués (intégration très simple dans JSF 1.2)
RichFaces 4 et 5	JBoss Labs	<i>Versions (complexes) de richfaces pour JSF2</i>
icesFaces	ICEsoft	Ajax + composants graphiques évolués
primeFaces	primefaces.org	Très bonne extension (simple) optimisée pour JSF2
		http://www.jsfmatrix.net/ = URL avec liste extensions

6. Wicket, Play,Struts2 ,Spring-MVC,GWT et les autres

Framework	Caractéristiques	Tendances
Wicket	Framework JEE/WEB plutôt orienté "composant"	Assez peu utilisé dans les années "2004-2010" et encore moins aujourd'hui
Play Framework	Framework JEE/WEB de type MVC ne s'appuyant volontairement pas sur l'api Servlet (langage : java ou scala) , convention plutôt que configuration	Inventé en 2007 et utilisé par très peu de développeurs
Struts2	<p>Framework MVC (anciennement "WebWork" puis renommé "Struts2"). Struts2 est très différent et mieux que Struts1 .</p> <p>Vis à vis du concurrent JSF , il semble moins évolué mais il offre l'avantage d'être moins gourmand en ressources (RAM, CPU, ...)</p>	<p>Assez bon framework mais la première version de 2006 était accompagnée de bugs et le concurrent JSF était à l'époque déjà bien en place.</p> <p>Struts 2 n'a pas été beaucoup utilisé d'autant plus qu'il se démarque très peu du concurrent Spring-MVC .</p>
Spring-MVC avec JSP ou bien Thymeleaf	<p>Framework MVC (très proche de Struts2 dans ses principes de fonctionnement). Spring-MVC est spécifique à l'écosystème Spring . Ce framework a la particularité de pouvoir s'adapter à plusieurs types de vues (ex : pages JSP , pages Thymeleaf , documents word ou pdf , ...) pouvant éventuellement cohabiter au sein d'une même application.</p>	<p>Ce framework est utilisé par un nombre non négligeables de développeurs "Spring".</p> <p>Bien que "correct" , ce framework est un peu plus complexe que JSF (mais un peu plus performant car moins gourmand en ressources).</p> <p>Aujourd'hui dans les années "2020", le framework Spring-MVC est de plus en plus rarement utilisé pour générer des pages HTML mais plutôt utilisé pour générer des réponses "JSON" (api REST)</p>
GWT (Google Web Toolkit)	<p>GWT est un "toolkit" (boîte à outils java) plutôt qu'un véritable framework. GWT consiste à coder une interface graphique en java . Puis celle-ci est ensuite automatiquement transformée en code javascript interprété coté navigateur .</p>	<p>Né en 2006, GWT a eu son heure de gloire durant la période "2008-2014" dite "Web 2.0 / ajax" .</p> <p>Aux alentours de 2015 environ, l'entreprise "Google" a laissé tomber le framework GWT pour se concentrer sur l'alternative "Angular" en pur javascript/typescript . Aujourd'hui GWT est encore maintenu par quelques développeurs "open-source" passionnés ...</p>
Autres ...	Ça existe

Liens pour approfondir "Wicket" :

https://en.wikipedia.org/wiki/Apache_Wicket
<https://wicket.apache.org/>

Liens pour approfondir "Play Framework" :

https://en.wikipedia.org/wiki/Play_Framework
<https://www.playframework.com/>
<https://linsolas.developpez.com/articles/java/play/guide/?page=page2>

Liens pour approfondir "Struts2" :

<https://struts.apache.org/>
<https://tahe.developpez.com/java/struts2/?page=page2>
<https://www.javatpoint.com/struts-2-tutorial>

Liens pour approfondir "Spring-MVC" et "Thymeleaf" :

<https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html>
<https://objis.com/tutoriel-spring-n9-introduction-spring-mvc/>
<https://www.javatpoint.com/spring-mvc-tutorial>

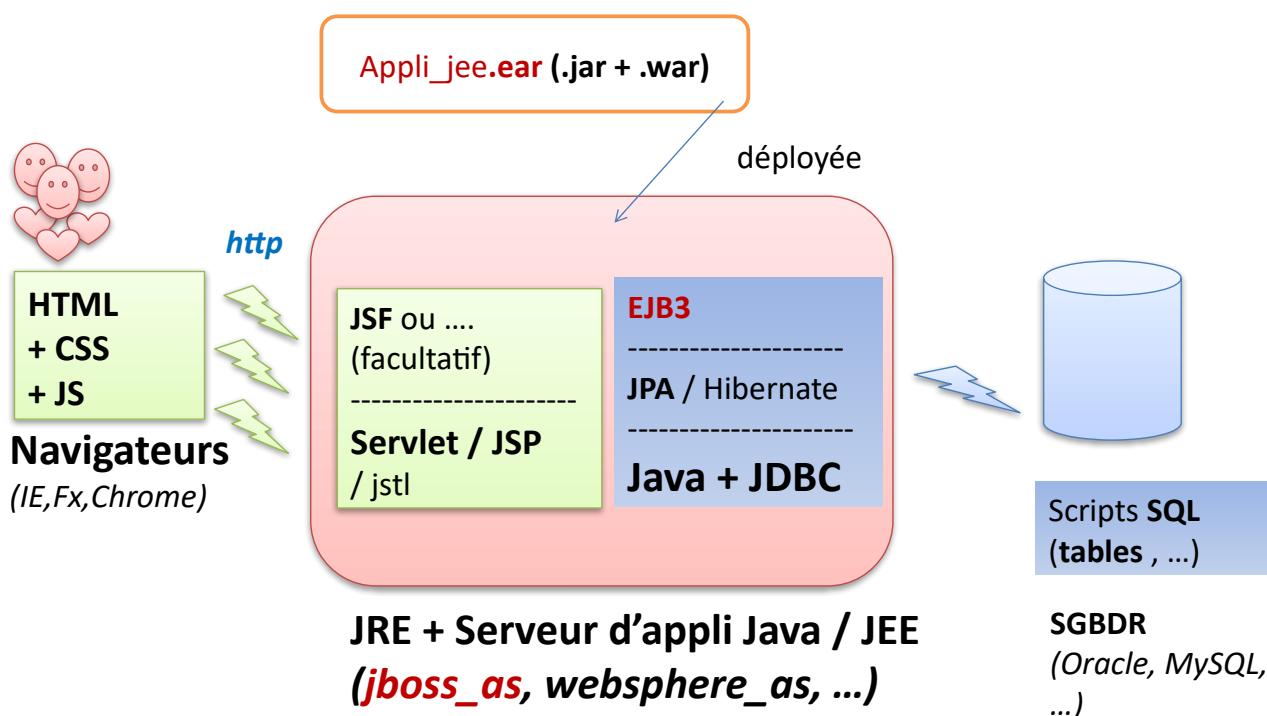
<https://www.thymeleaf.org/>
<https://fr.wikipedia.org/wiki/Thymeleaf>

Liens pour approfondir "GWT" :

<https://www.gwtproject.org/>
https://en.wikipedia.org/wiki/Google_Web_Toolkit
<https://www.jmdoudoux.fr/java/dej/chap-gwt.htm>

7. Java web coté serveurs

Env exécution java/jee



Quasiment toutes les pages HTML sont générées par du code java coté serveur.

Ceci a un coût élevé (en terme de consommation de ressources CPU/réseaux/mémoire) car :

- besoin de générer fréquemment de nombreuses pages (si beaucoup d'utilisateurs connectés)
- beaucoup de transferts HTTP (requêtes à analyser, réponses HTML à fabriquer)
- besoin de maintenir coté serveur de nombreuses Sessions utilisateurs (1 objet HttpSession par utilisateur connecté + avec quelquefois plein de petits objets reliés à la session)

Avantages :

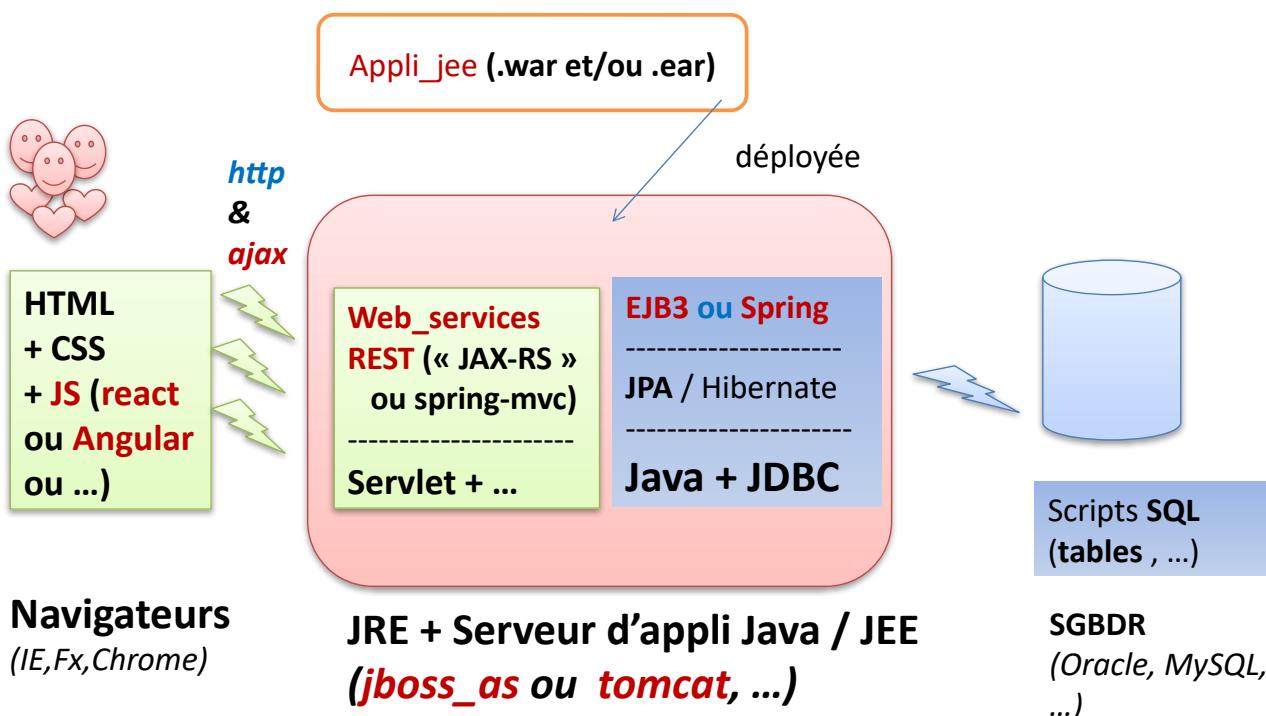
- codage et sécurisation un peu plus simple (tout se fait au même niveau : dans le code java coté serveur)
- plus fiable à l'ancienne époque (2000-2010) où le code javascript était interprété différemment selon les navigateurs

Inconvénients :

- **performances très moyenne** (beaucoup de ressources nécessaires, besoin de serveurs en "cluster" pour supporter la charge induite si nombreux utilisateurs simultanés).
- **temps de réaction mauvais** (ça réagit quelquefois lentement si le serveur est trop sollicité ou bien si les pages HTML sont complexes et longues à construire ou bien réactualiser).

8. Intégration Ajax et Single Page Applications

Frontend js + Backend javaEE



Aucune page HTML n'est générée par du code java côté serveur.

Une page unique *index.html* est accompagnée d'un gros bloc de code javascript qui va :

- **afficher une sous page ou une autre**
- **appeler des web-services REST côté serveur de manière à envoyer des données saisies à stocker en base de données ou bien récupérer des données au format JSON que l'on a envie d'afficher**

NB: On parle en terme de "**SPA = Single Page Application**" pour désigner ce comportement côté navigateur / frontEnd .

Ceci a un coût bien plus faible côté serveur (en terme de consommation de ressources CPU/réseaux/mémoire) car :

- les traitements IHM (saisies, affichages, contrôles) sont effectués par plein de navigateurs en parallèle lorsque plein d'utilisateurs travaillent de manière simultanée)
- moins de transferts HTTP (réponses JSON bien plus légères que réponses HTML)
- plus besoin de maintenir de session côté serveur (les sessions utilisateurs sont maintenant gérées en javascript côté navigateur)

Avantages :

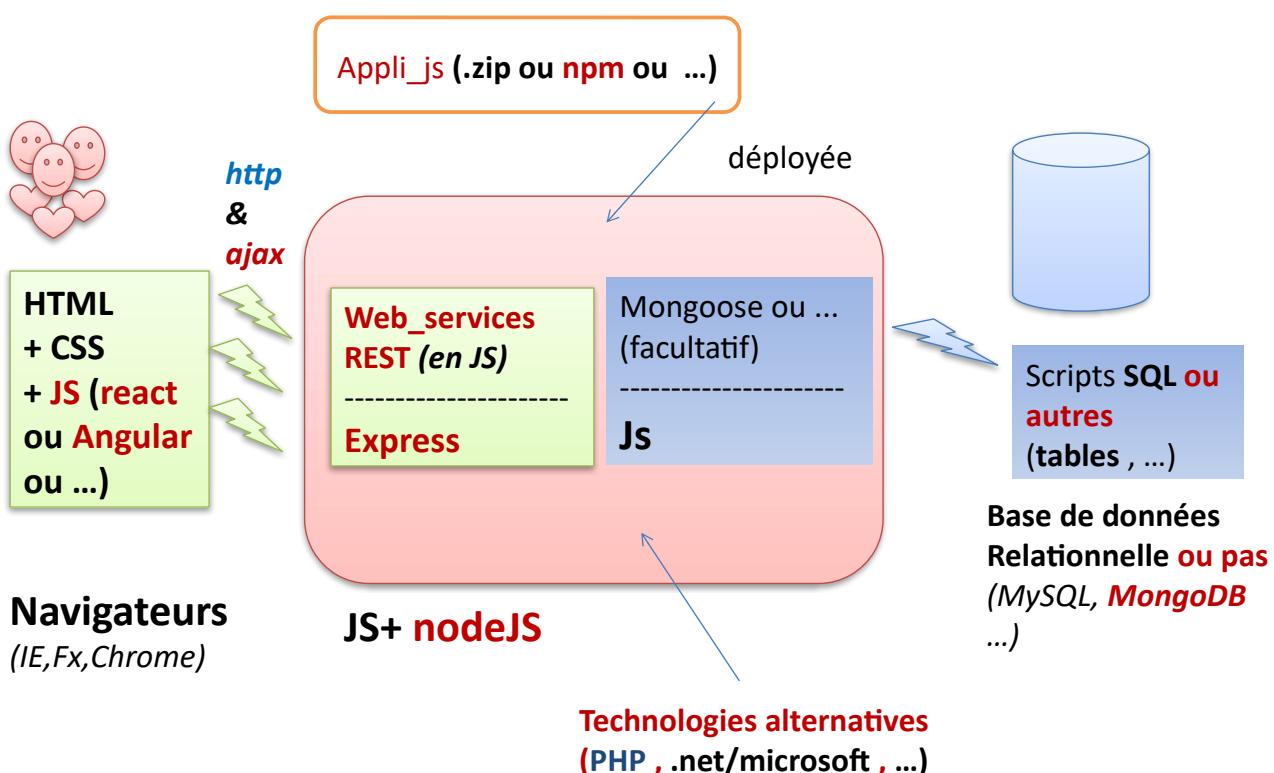
- **bien meilleures performances** (surtout si plein d'utilisateurs en simultané)
- **bien meilleur réactivité** aux événements (ça réagit beaucoup plus vite)

Inconvénients :

- **2 parties séparées ("frontEnd javascript" et "backend java") à coder séparément puis à assembler d'une manière ou d'une autre .**
- **Sécurisation et déploiement** un peu **plus complexe** (besoin de serveurs annexes : Api-Gateway , serveur d'authentification OAuth2/OIDC , ...)

Concurrent sérieux de java côté serveur : nodeJS :

Env exécution sans java



8.1. Appels de WS REST (HTTP) depuis js/ajax

Avec ajax , ça va briller!

AJAX est l'acronyme d'*Asynchronous JavaScript And XML*, mais ça peut être utilisé avec *Json* .

Cadre des appels

Lorsqu'une requête http est initiée depuis du code javascript s'exécutant dans le contexte d'une page html , on dit que l'on effectue un appel "ajax" .

Le sigle Ajax correspondant à peu près à "asynchronous javascript activation framework" indique :

- le déclenchement non bloquant d'une requête http
- l'enregistrement d'une fonction "callback" qui sera automatiquement appelée en différé lorsque la réponse reviendra

NB : l'appel non bloquant peut être considéré comme asynchrone mais le protocole HTTP est un protocole de transport synchrone (avec timeout si la réponse ne revient pas dans un délai raisonnable).

Techniquement, un appel ajax s'effectue en s'appuyant sur un objet technique "**XmLHttpRequesT**" fourni par tous les navigateurs pas trop anciens .

La partie Xml de XMLHttpRequest tient au fait qu'historiquement les premiers webServices normalisés (SOAP) étaient au format Xml. Bien que le terme "XmlHttpRequest" n'ai pas été changé pour des raisons de compatibilité ascendante du code javascript, il est possible de déclencher n'importe quelle requête HTTP depuis XHR , y compris des requêtes au format "JSON" .

Pour simplifier la syntaxe d'un appel ajax on peut éventuellement s'appuyer sur des librairies "javascript" complémentaires ("jquery" , "fetch" , "RxJs" , ...) .

Le principe des appels "ajax" sert essentiellement à déclencher une requête HTTP suite à un événement utilisateur en vue d'obtenir des données permettant de réactualiser une partie de la page HTML courante . La page courante n'est pas entièrement remplacée par une autre. Seule une sous partie de celle ci est ajustée par code js/DOM (Document Object Model).

Certaines applications , dites "SPA: Single Page Application" sont entièrement bâties sur ce principe . Au lieu de switcher de pages html on switch de sous pages (<div ...>....</div>) .

XHR (XmlHttpRequest) disponible dans tout navigateur actuel

Exemple basique :

```
function makeAjaxRequest(callback) {
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4 && (xhr.status == 200 || xhr.status == 0)) {
            callback(xhr.responseText);
        }
    };
    xhr.open("GET", "handlingData.php", true);
    xhr.send(null);
}

function readData(sData) {
    if (sData!=null) {
        alert("good response");
        // + display data with DOM
    } else {
        alert("empty response");
    }
}
makeAjaxRequest(readData);
```

variations en mode "POST" :

```
xhr.open("POST", "handlingData.php", true);
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");

xhr.send("param1=xx&param2=yy");
```

```
xhr.open("POST", "/json-handler");
xhr.setRequestHeader("Content-Type", "application/json");
xhr.send(JSON.stringify({prenom:"Jean", nom:"Bon"}));
```

Exemple plus élaboré :

my_ajax_util.js

```
//subfunction with errCallback as optional callback :
function registerCallbacks(xhr,callback,errCallback) {
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4){
            if((xhr.status == 200 || xhr.status == 0)) {
                callback(xhr.responseText);
            }
            else {
                if(errCallback)
                    errCallback(xhr.responseText);
            }
        }
    };
}

function makeAjaxGetRequest(url,callback,errCallback) {
    var xhr = new XMLHttpRequest();
    registerCallbacks(xhr,callback,errCallback);
    xhr.open("GET", url, true);  xhr.send(null);
}

function makeAjaxDeleteRequest(url,callback,errCallback) {
    var xhr = new XMLHttpRequest();
    registerCallbacks(xhr,callback,errCallback);
    xhr.open("DELETE", url, true);  xhr.send(null);
}

function makeAjaxPostRequest(url,jsonData,callback,errCallback) {
    var xhr = new XMLHttpRequest();
    registerCallbacks(xhr,callback,errCallback);
    xhr.open("POST", url, true);
    xhr.setRequestHeader("Content-Type", "application/json");
    xhr.send(jsonData);
}

function makeAjaxPutRequest(url,jsonData,callback,errCallback) {
    var xhr = new XMLHttpRequest();
    registerCallbacks(xhr,callback,errCallback);
    xhr.open("PUT", url, true);
    xhr.setRequestHeader("Content-Type", "application/json");
    xhr.send(jsonData);
}
```

appelAjax.js

```

var traiterReponse = function (response){
    //response ici au format "json string"
    var zoneResultat = document.getElementById("spanRes");
    var jsDevise = JSON.parse(response);
    zoneResultat.innerHTML=jsDevise.change; //ou .rate
}

function onSearchDevise(){
    var zoneSaisieCode = document.getElementById("txtCodeDevise");
    var codeDevise = zoneSaisieCode.value;
    console.log("codeDevise="+codeDevise);
    var urlWsGet= "./devise-api/public/devise/"+codeDevise;
    makeAjaxGetRequest(urlWsGet,traiterReponse); //non bloquant (asynchrone)
    //....
}

```

appelAjax.html

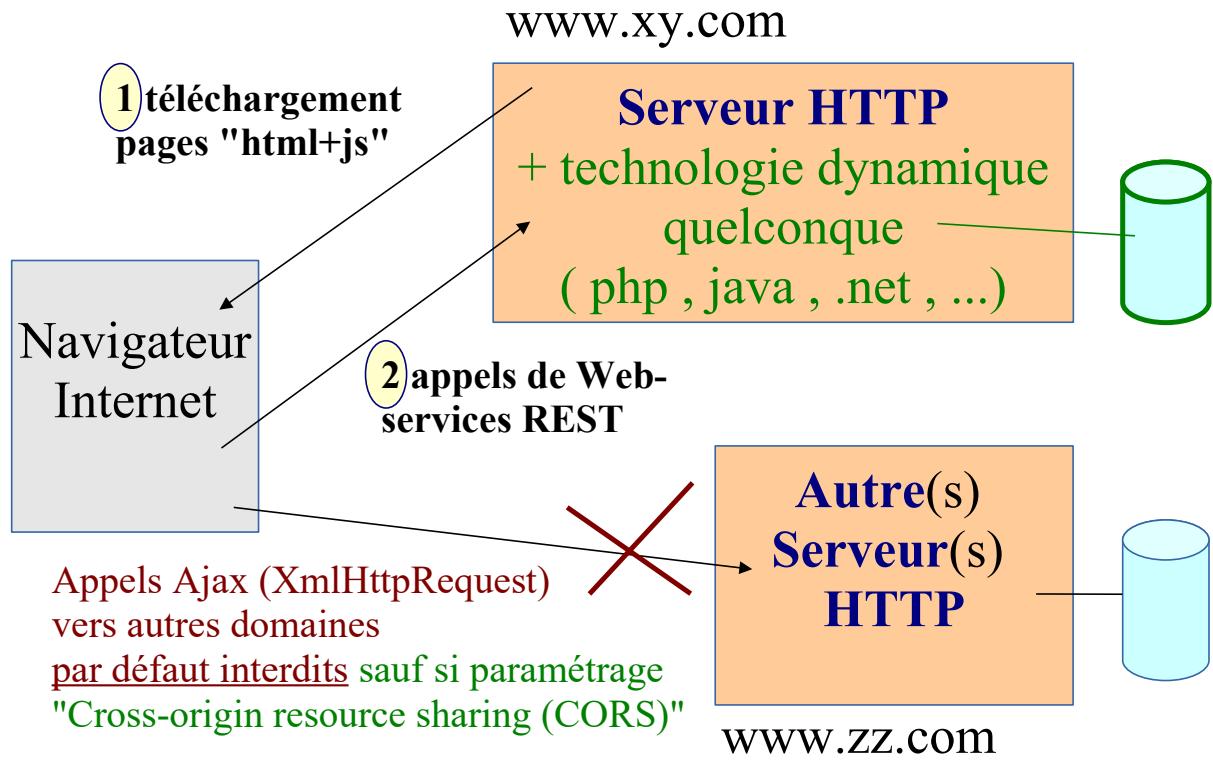
```

<html>
<head>
    <meta charset="UTF-8">
    <title>appelAjax</title>
    <script src="js/my_ajax_util.js"></script>
    <script src="js/appelAjax.js"></script>
</head>
<body>
    codeDevise :<input type="text" id="txtCodeDevise"/> <br/>
    <input type="button" id="btnSearch" onclick="onSearchDevise()" value="rechercher devise"/> <br/>
    Devise : <span id="spanRes"></span>
</body>
</html>

```

8.2. Limitations Ajax sans CORS

Cadre des appels "html/js/ajax" vers services REST



8.3. CORS (Cross Origin Resource Sharing)

CORS=Cross Origin Resource Sharing

CORS est une **norme du W3C** qui précise certains **champs** à placer dans une **entête HTTP** qui serviront à échanger entre le navigateur et le serveur des informations qui serviront à décider si une requête sera ou pas acceptée.

(utile si domaines différents) , dans requête simple ou bien dans pré-échange préliminaire quelquefois déclenché en plus :

Au sein d'une requête "demande autorisation" envoyée du client vers le serveur :

Origin: <http://www.xy.com>

Dans la "réponse à demande d'autorisation" renvoyée par le serveur :

Access-Control-Allow-Origin: <http://www.xy.com>

Ou bien

Access-Control-Allow-Origin: * (si public)

→ requête acceptée

Si absence de "Access-Control-Allow-Origin :" ou bien valeur différente
---> requête refusée

Paramétrage "CORS" avec Spring-mvc

```
import org.springframework.web.bind.annotation.CrossOrigin;
...
@RestController
@CrossOrigin(origins = "*")
//@CrossOrigin(origins = { "http://localhost:4200" ,
//                         "http://www.partenaire-particulier.com" })
@RequestMapping(value="/rest/products" , headers="Accept=application/json")
public class ProductCtrl {...}
```

et

@Configuration

```

public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

...
@Override
protected void configure(final HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/", "/favicon.ico", "/**/*.{png,jpg,gif,svg}")
        .antMatchers("/devise-api/public/**").permitAll()
        .antMatchers("/devise-api/private/**").authenticated()
        .and().cors() //enable CORS (avec @CrossOrigin sur class @RestController)
        .and().csrf().disable()
        .exceptionHandling().authenticationEntryPoint(unauthorizedHandler)
        .and()
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        .and()
        .addFilterBefore(jwtAuthenticationFilter,
                         UsernamePasswordAuthenticationFilter.class);
}
...
}

```

IV - Clients Java

1. AWT, Swing et SWT

AWT seul	Java 1.0 et 1.1 (1995-1998)	IHM basique basée sur le dénominateur commun de windows , max et unix et avec différence de look selon plateforme
AWT + SWING	À partir de java 1.2 (1999)	SWING est un complément important pour AWT qui apporte de nouveaux composants et qui en s'appuyant moins sur l'os natif permet d'obtenir une interface graphique avec à peu près le même look sur différentes plateformes
SWT	Depuis 2003 environ	Autre paquet de composants graphiques "java" développé par IBM et principalement utilisé pour le code interne de l'IDE ECLIPSE. SWT s'appuie plus sur l'os natif → look un peu affecté mais moins gourmand en ressources que SWING et un peu plus rapide.

1.1. Exemple AWT

```

import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class MyApp {

    public static void main(String[] args) {
        Frame frame = new Frame("Application");

        frame.add(new Label("Hello!"));
        frame.setSize(500, 500);
        frame.setLocationRelativeTo(null); // Centers the window

        frame.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                frame.dispose(); // Releases native screen resources
            }
        });

        frame.setVisible(true);
    }
}

```

1.2. Exemple SWING

```
// Hello.java (Java SE 8)
import javax.swing.*;

public class Hello extends JFrame {
    public Hello() {
        super("hello");
        this.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        this.add(new JLabel("Hello, world!"));
        this.pack();
        this.setVisible(true);
    }

    public static void main(final String[] args) {
        SwingUtilities.invokeLater(Hello::new);
    }
}
```

1.3. Exemple SWT

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;

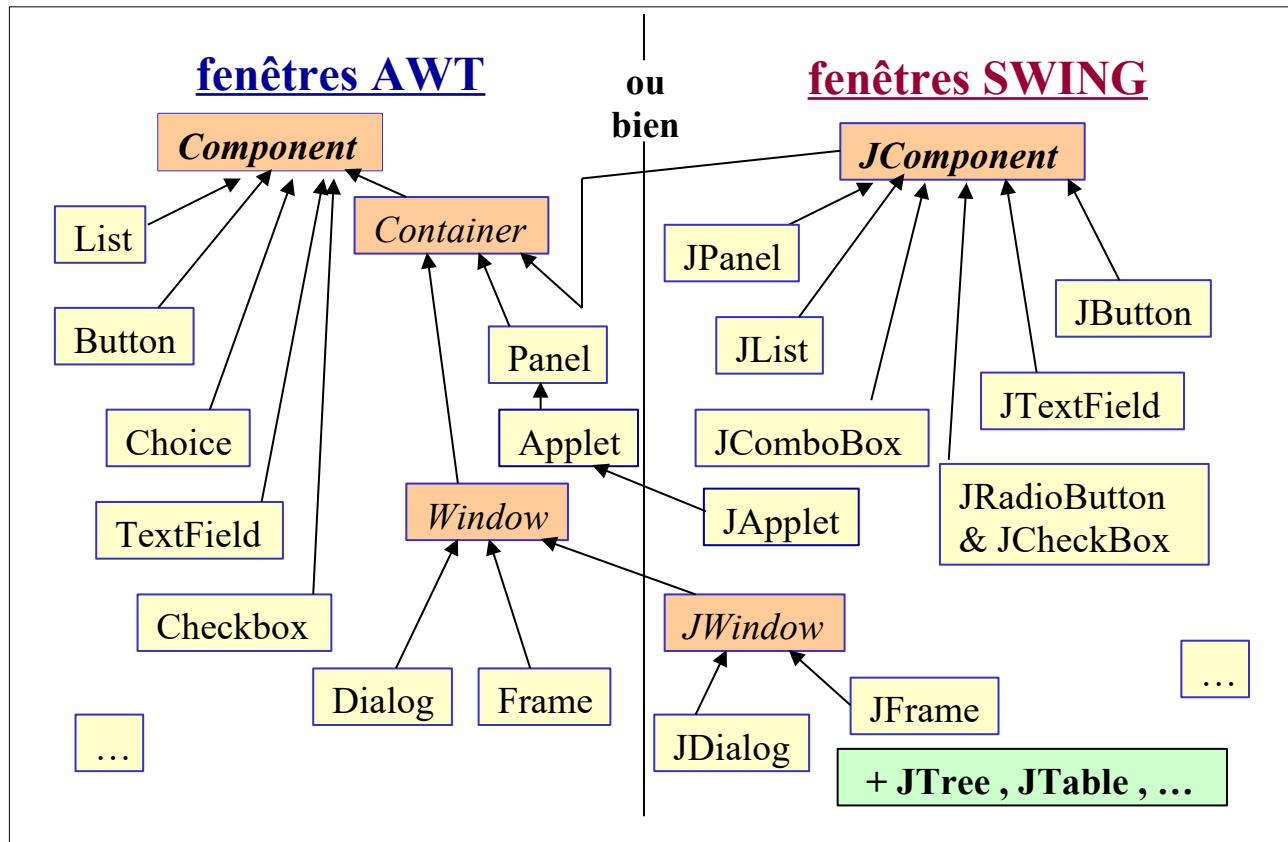
public class HelloWorld
{
    public static void main (String[] args)
    {
        Display display = new Display();
        Shell shell = new Shell(display);
        Label label = new Label(shell, SWT.NONE);
        label.setText("Hello World");
        label.pack();
        shell.pack();
        shell.open();
        while (!shell.isDisposed())
        {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

1.4. Eléments de base sur AWT/SWING et événements

1.4.a. Gestion des fenêtres et des contrôles

Pour gérer les fenêtres , il existe (au choix) deux api (relativement proches et parallèles):

- Les classes (de `java.awt`) héritant de `Component`. Il s'agit d'une encapsulation JAVA des api natives du système d'exploitation.
- Les classes (de `javax.swing`) héritant de `JComponent` (fenêtre 100 % java).



1.4.b. Composant de l'interface graphique (Component)

La classe abstraite `java.awt.Component` hérite directement de `Object` et correspond à un composant quelconque de l'interface graphique au sens large (fenêtre, contrôle,...). Parmi les méthodes de cette classe on retiendra principalement les suivantes:

<code>getParent()</code>	retourne le Container contenant l'objet
<code>setBackground(color)</code>	Fixe la couleur de fond
<code>setForeground(color)</code>	Fixe la couleur d'avant plan (texte, traits,...)
<code>setFont(fonte)</code>	Fixe la police de caractères
<code>setVisible(booléen)</code>	cache ou rend visible l'objet graphique
<code>setEnabled(booléen)</code>	active ou inactive (grise) l'objet
<code> setLocation(x,y), setSize(w,h)</code>	déplace, redimensionne l'objet
<code>requestFocus()</code>	demande le focus sur cet objet
<code>bounds()</code>	retourne la position et dimension de l'objet sous la forme d'un objet Rectangle (coordonnée)

Dans l'arbre d'héritage on trouvera directement sous la classe Component une série de

classes correspondant aux contrôles (boutons poussoirs, zones d'éditions , ...) plus une classe abstraite nommée `java.awt.Container` .

1.4.c. Fenêtres "Container" et sous fenêtres

La classe `java.awt.Container` correspond à un Conteneur (fenêtre pouvant contenir d'autres sous fenêtres). Ses principales méthodes sont les suivantes:

<code>add(component)</code>	Ajoute un composant (contrôle, panel, ...)
<code>remove(component)</code>	Enlève un sous objet
<code>countComponents()</code>	Retourne le nombre de composants
<code>setLayout(layoutMgr)</code>	Associe un gestionnaire de répartition au conteneur
<code>locate(x,y)</code>	Retourne une référence sur le Component en (x,y)

- Les classes `java.awt.Panel` et `javax.swing.JPanel` correspondent à un "sous-paneau" . Il s'agit d'un élément généralement invisible qui permet de gérer finement la disposition des contrôles dans une fenêtre en jouant le rôle de **Conteneur intermédiaire**: un objet Panel est toujours contenu dans un autre conteneur et contient à son tour d'autres contrôles ou sous conteneurs.
- Les classes "Fenêtre" (`java.awt.Window` et `javax.swing.JFrame`) correspondent à une fenêtre applicative (sans bordure , ni menu) qui comporte deux sous classes importantes: `java.awt.Dialog` (boîte de dialogue) et `java.awt.Frame` (fenêtre cadre ou fenêtre principale).
- Les classes `java.awt.Frame` et `javax.swing.JFrame` correspondent à une fenêtre principale qui peut avoir un titre, une barre de menu, une bordure permettant de la redimensionner , un icône et un curseur spécifique. Cette classe est très souvent sous classée car elle correspond au point central d'une application graphique. Les principales méthodes de **Frame** sont:

<code>setTitle(titre)</code>	Change le titre
<code>setMenuBar(barreDeMenu)</code>	Associe un menu à la fenêtre
<code>setResizable(monBoolean)</code>	true / false

1.4.d. Les Contrôles (composants graphiques élémentaires)

Libellé (étiquette)

```
lbl = new javax.swing.JLabel("Nom :");
```

Champ de saisie simple (une seule ligne)

```
champ = new javax.swing.JTextField();
champ.setText("Valeur par défaut");
String texteSaisi = champ.getText();
champ.select(2 /*start*/ ,6 /*end*/);
String texteSelectionne = champ.getSelectedText();
```

Bouton poussoir

```
jbtnOk = new javax.swing.JButton("Ok");
jbtnOk.setText("OK");
```

"Case à cocher" ou "bouton radio" accompagné de son libellé.

En version swing:

2 classes différentes:

- **JCheckBox** pour les cases à cocher (non exclusives).
- **JRadioButton** pour les boutons "radio" (exclusifs entre eux).

Paramétrage de l'exclusivité entre différentes options:

```
ButtonGroup bg = new ButtonGroup(); // objet invisible
bg.add(jRadioButtonMarie);
bg.add(jRadioButtonCelibataire);
```

Liste déroulante pour choisir un élément parmi n.

En version swing:

```
combo = new javax.swing.JComboBox();
combo.addItem("rouge");
combo.addItem("vert"); combo.addItem("bleu");

combo.setSelectedItem("vert"); //ou combo.setSelectedIndex(1);

nbElts = combo.getItemCount();
int selIndex = combo.getSelectedIndex();
String chaineChoisie = combo.getSelectedItem();
```

Liste d'éléments (avec sélection multiple possible)

En version swing:

```
Vector vectSel = new Vector();
JList jListSel = new JList();
...
vectSel.addElement(uneChose);
...
jListSel.setListData(vectSel);
...
int tabSelIndex[] = jListSel.getSelectedIndices();
```

Zone de saisie multi-lignes

En version swing:

classe	fonctionnalités
JTextArea	zone de texte simple à plusieurs lignes (light).

JTextPane	éditeur de texte sophistiqué (mise en forme possible des caractères).
JEditorPane	Zone de texte sophistiquée (text/plain ou text/html)

Boite de dialogue

- javax.swing.JDialog est la classe générique des boites de dialogue en version SWING.
- Un choix de nom de fichier s'effectue avec la classe **JFileChooser** :

```
JFileChooser chooser = new JFileChooser();
int returnVal = chooser.showOpenDialog(parent);
if(returnVal == JFileChooser.APPROVE_OPTION)
{
    fileName= chooser.getSelectedFile().getName();
}
```

Menu

JMenuBar, JMenu, JMenuItem (*à imbriquer et à attacher à la fenêtre principale*)
JPopupMenu ==> menu contextuel (sur click droit)

1.4.e. Fonctionnalités générales des composants SWING élémentaires:

.setToolTipText("Chaine InfoBulle")	précise le texte de la bulle d'aide
.setVisible(true/false)	montre ou cache le composant
.setEnabled(false/true)	grise ou dégrise le composant
.setOpaque(false)	précise la "non transparence" du fond

Nb: ces méthodes proviennent de **JComponent**

1.4.f. Gestion simple du scrolling (JScrollPane)

```
JScrollPane jScrollPaneXXX = new JScrollPane();
...
jScrollPaneXXX.setViewportView(jUneChose); // chose = liste, panel ,arbre, ...
```

1.4.g. Boîte de message et Prompt

La classe **JOptionPane** comporte quelques **méthodes statiques** qu'il suffit d'appeler directement (en préfixant par le nom de la classe) et qui permettent d'afficher des boîtes de messages et des invites pour saisir des valeurs:
(*NB: p=this ou null ou fenêtre parente*)

<i>méthode statique</i>	<i>fonctionnalité</i>
JOptionPane. showMessageDialog(p,"Bienvenue");	Affiche une boîte de message

<pre>name= JOptionPane.showInputDialog(p,"Quel est votre nom"); JOptionPane.showConfirmDialog(p,"voulez vous ...?") => renvoie une valeur du genre OK_OPTION ou YES_OPTION</pre>	Prompt Demande de confirmation (Yes,No,Cancel)
---	--

Paramètres optionnels de ces méthodes statiques:

title : titre de la mini boîte de dialogue.

optionType: YES_NO_OPTION ou OK_CANCEL_OPTION ou ...

messageType: WARNING_MESSAGE ou ERROR_MESSAGE ou QUESTION_MESSAGE ou INFORMATION_MESSAGE.

1.4.h. Onglets (JTabbedPane)

La classe **JTabbedPane** correspond à un conteneur de type "Série d'onglets".

Chaque onglet sera programmé sous la forme d'un élément héritant de **JPanel**.



Le simple fait d'incorporer plusieurs panneaux (JPanel) dans un conteneur de type "JTabbedPane" permet d'obtenir le fameux Look à onglets.

Chaque onglet est associé à un nom ou libellé (sous forme de chaîne de caractères):

```
JTabbedPane jSerieOnglets = new JTabbedPane();
...
jSerieOnglets.addTab("Onglet1", jPanelOnglet1);
jSerieOnglets.addTab("Onglet2", jPanelOnglet2);
```

1.4.i. Gestionnaire de répartition (LayoutManager)

Un gestionnaire de répartition (défini par l'interface **LayoutManager**) est un **objet invisible** qui sera **associé à un conteneur** et qui servira à disposer au mieux les contrôles qui seront ultérieurement placés dans le conteneur.

Le programmeur a simplement besoin d'installer (de façon facultative) le gestionnaire de répartition.

→ conteneur.setLayout(layoutManager);

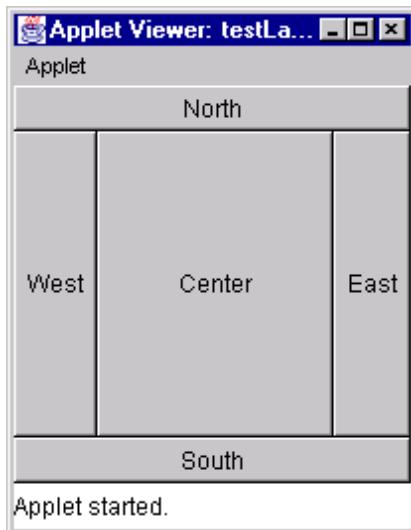
Celui-ci sera alors automatiquement utilisé par la suite.

Il faudra tout de même préciser certains paramètres lors de l'ajout d'un contrôle dans le conteneur → exemple: conteneur.add("South",contrôle);

JAVA dispose de quelques gestionnaires de répartition prédéfinis:

BorderLayout	<i>Arrange les contrôles sur les bords et au centre du conteneur</i> en fonction des indications "South", "North", "East", "West", "Center". Les paramètres facultatifs <i>hgap</i> et <i>vgap</i> du constructeur permettent de préciser l'espacement entre deux composants adjacents.
FlowLayout	Arrange les composants en ligne de gauche à droite. Fait entrer autant de composants que possible sur la ligne

<i>(par défaut)</i>	courante avant de passer à la prochaine. Au sein d'une ligne les choses sont alignées en fonction du paramètre facultatif align du constructeur. FlowLayout.CENTER , .LEFT , .RIGHT
GridLayout, CardLayout, GridBagLayout, autres gestionnaires (complexes)...



1.4.j. Positionnement absolu (sans LayoutManager)

```
this.*getContentPane().*/setLayout( null );
jTextField1.setBounds( new Rectangle(126, 8, 63, 21) );
```

1.4.k. Classe **Graphics** – pour Dessiner (lignes, rectangles, ...)

Pour dessiner quoi que ce soit dans une fenêtre ou dans un autre périphérique graphique (imprimante, mémoire) , il faut passer par la classe Graphics.

Cette classe est à mettre en parallèle avec la notion de "Graphic Context (gc)" de X-Window ou bien encore avec le "Device Context (DC)" de Win32.

La classe `java.awt.Graphics` permet de préciser comment les choses seront dessinées (Coordonnées , fontes , couleurs, modes des tracés, styles des hachures, ...).

C'est aussi au travers d'elle que seront invoquées les méthodes permettant de dessiner telles que `.drawLine()` ou `fillRect()`.

Dans la pratique, on ne peut pas directement créer une nouvelle instance de la classe `Graphics` qui est d'ailleurs abstraite mais on procédera de la façon suivante:

- En manipulant l'instance "graphics" qui nous est fournie par la méthode `paint()` permettant de (re)dessiner le contenu de la fenêtre qui a besoin d'être rafraîchie.
- En obtenant l'instance en invoquant la méthode `getGraphics()` des classes `Component` ou `Image`.

```
public abstract class java.awt.Graphics extends java.lang.Object {
    // Constructors: protected Graphics();
    public abstract void clearRect(int x, int y, int width, int height);
    public abstract void clipRect(int x, int y, int width, int height);
    public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle);
    public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer);
    ...
    public abstract void drawLine(int x1, int y1, int x2, int y2);
    public abstract void drawOval(int x, int y, int width, int height);
    public abstract void drawPolygon(int xPoints[], int yPoints[], int nPoints);
    public void drawRect(int x, int y, int width, int height);
    public abstract void drawRoundRect(int x, int y, int width, int height,
                                      int arcWidth, int arcHeight);
    public abstract void drawString(String str, int x, int y);
    public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle);
    public abstract void fillOval(int x, int y, int width, int height);
    public abstract void fillPolygon(int xPoints[], int yPoints[], int nPoints);
    public abstract void fillRect(int x, int y, int width, int height);
    public abstract void fillRoundRect(int x, int y, int width, int height,
                                      int arcWidth, int arcHeight);
    ...
    public abstract void setColor(Color c);
    public abstract void setFont(Font font);
    public abstract void setPaintMode();
    public abstract void setXORMode(Color c1);
    public abstract void translate(int x, int y);
}
```

1.4.l. Gestion des événements

Le principe de fonctionnement du modèle événementiel en vigueur depuis le jdk 1.1 **est basé sur le modèle "fournisseur / abonnés"**:

L'événement sera envoyé à tous les objets qui auront préalablement marqué leurs intérêts vis à vis de celui-ci via une procédure d'enregistrement.

Au sein de ce modèle événementiel, il faut considérer plusieurs entités :

- L'objet qui génère l'événement (fenêtre, contrôle, ...) (**source d'événement**)
- Un (ou plusieurs) objet(s) qui vont traiter (gérer) l'événement (**listener**).
 - + un éventuel objet *intermédiaire* appelé **adaptateur**
- Un objet (**event**) qui va véhiculer les détails (données) de l'événement.

Dans le cas le plus simple une fenêtre parente (englobante) pourra gérer le rôle de listener.

1.4.m. Notion d'abonnement

Le fait de s'abonner se code de la façon suivante:

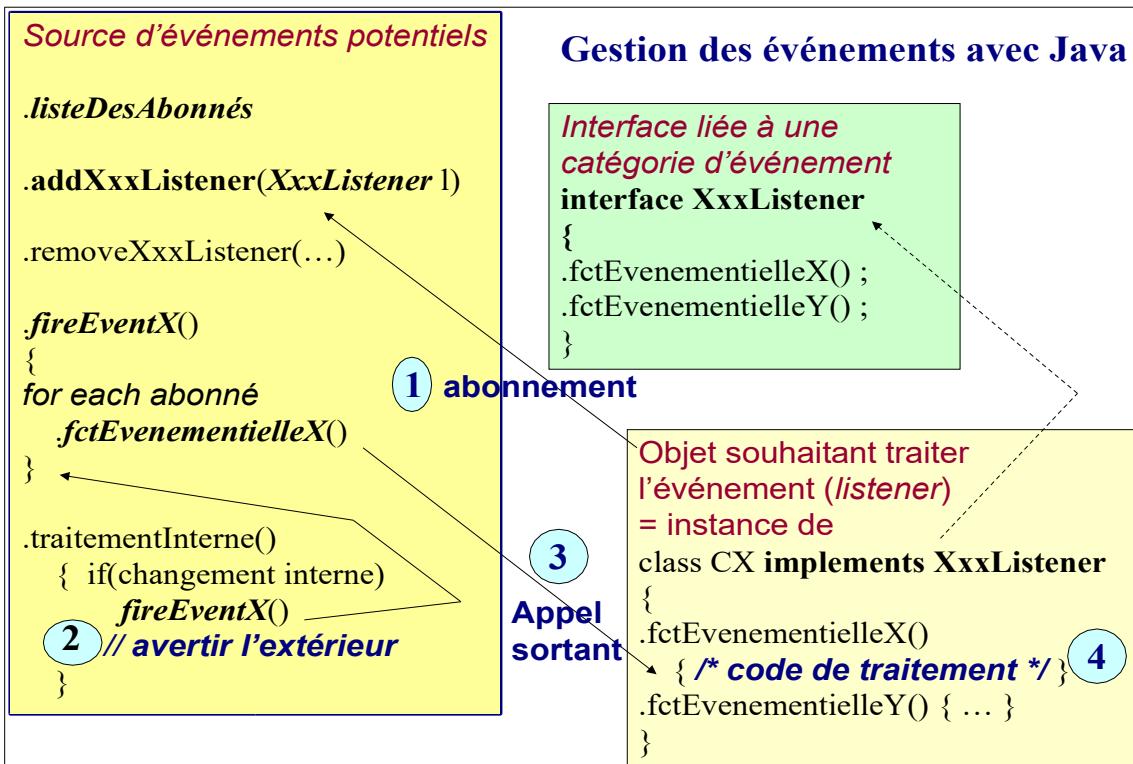
```
composantSource.addXXXListener(objetQuiSouhaiteGererEvenement);
```

```
composantSource.addXXXListener(deuxièmeAbonné);
```

>>> XXX correspond ici à une **catégorie d'événements**. <<<

Pour chacune de ces catégories (XXX), il existe:

- une méthode pour s'abonner: **addXXXListener()**
- une **interface XXXListener** que doit implémenter l'objet qui s'abonne et qui souhaite traiter l'événement.
- une sous classe d'événement **XXXEvent** héritant de la classe abstraite **java.awt.AWTEvent** dérivant elle même de **java.util.EventObject**.



Exemple:

Si l'on veut traiter au niveau d'une instance de la classe Delegue le click sur un bouton

poussoir, il faudra alors coder les choses de la façon suivante:

```
import java.awt.*; import javax.swing.*;
import java.awt.event.*; // pour accéder à l'interface XXXListener

class PetiteFenetre extends JFrame{
    public PetiteFenetre()
    {
        Delegue unDelegue = new Delegue();
        JButton monBouton = new JButton("Touch Me");
        monBouton.addActionListener(unDelegue); // abonnement
        this.getContentPane().add(monBouton);
    }
    public static void main(String [] argv)
    {
        JFrame f = new PetiteFenetre();
        f.pack();
        f.setVisible(true);
    }
}
```

```
class Delegue implements ActionListener
{
    ...
    /* il faut coder toutes les méthodes de l'interface ActionListener
       → soit une seule méthode ici (cas le plus simple) */
    public void actionPerformed(ActionEvent e)
    {
        // traitement (gestion) du message:
        System.exit(0);
    }
}
```

Dans la plupart des interfaces graphiques(IHM), l'entité délégué qui traite l'événement est très fortement lié à l'objet fenêtre , Panneau, ou boîte de dialogue qui contient directement le composant source qui sera à la source de l'événement (ex: Bouton poussoir).

NB1: Un composant graphique peut très bien gérer lui même l'événement reçu:
(implements XXXListener et addXXXListener(this);)

NB2: Si c'est la fenêtre parent (qui s'abonne et) qui gère les événements provenant de ses fenêtres filles, il faudra alors distinguer celles-ci via la source de l'événement:

```
Object origine=e.getSource();
if(origine==sousComposant1)
{
    ...
}
else if(origine== sousComposant2) ...
```

La plupart des générateurs de code ont généralement recours à l'utilisation des classes imbriquées:

```
class MaFenetre .... // classe "conteneur graphique"
{
...
public void initialiser()
{
    ...
jButtonOK.addActionListener(new
    /*début du code de la classe imbriquée*/ java.awt.event.ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        jButtonOKActionPerformed(e);
    } /* fin du code de la classe imbriquée */
});
...
}
...
void jButtonOKActionPerformed(ActionEvent e)
{
    /* traitement de l'événement */
}
```

Interfaces liées aux catégories de messages (ayant chacune une méthode d'abonnement addXXXListener et un type d'événement XXXEvent qui leur correspondent) :

<i>Interface / catégorie d'événements</i>	<i>types de composants sources</i>
ActionListener	Button (éventuellement JRadioButton, JCheckBox, ...)
MouseListener	Panel, Canvas , ... (éventuellement JList, ...)
KeyListener	Zone de saisie (==> codez en priorité keyReleased)
ItemListener	Boutons radios , cases à cocher , zones de listes, ...
...	...

<i>Interface</i>	<i>méthodes à implémenter (void)</i>
ActionListener	actionPerformed(ActionEvent)
AdjustmentListener	adjustmentValueChanged(AjustmentEvent)
ComponentListener	componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent)
ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)
ItemListener	itemStateChanged(ItemEvent)
KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
MouseListener	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent)

	mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
TextListener	textValueChanged(TextEvent)
WindowListener	windowActivated(WindowEvent) windowClosed(WindowEvent) windowClosing(WindowEvent) windowDesactivated(WindowEvent) windowDeiconified(WindowEvent) windowIconified(WindowEvent) windowOpened(WindowEvent)

Exemple très classique (nouvelle sélection dans une liste déroulante):

```
jComboBoxTypeBase.addItemListener(new java.awt.event.ItemListener()
{
    public void itemStateChanged(ItemEvent e) {
        jComboBoxTypeBase_itemStateChanged(e);
    }
});

void jComboBoxTypeBase_itemStateChanged(ItemEvent e) {
    int index=jComboBoxTypeBase.getSelectedIndex();
    ...
}
```

1.4.n. Adaptateurs

NB: L'API de JAVA fournit des *adaptateurs par défaut* qui ne font rien (code vide) sous la forme de *classes abstraites* (**MouseAdapter**, **KeyAdapter**, **WindowAdapter**, ...). Il suffit alors de dériver une ces classes pour ensuite avoir la possibilité de ne coder (redéfinir) qu'une seule des méthodes de l'interface XXXListener correspondante:

```
class MonAdaptateur extends MouseAdapter {
    public void mouseEntered(MouseEvent e) { ... }
}
```

Exemples très classiques:

```
jTreeDep.addMouseListener(new java.awt.event.MouseAdapter() {
    public void mouseClicked(MouseEvent e) {
        jTreeDep_mouseClicked(e);
    }
});

...
```

```
jTextFieldAnneeMini.addKeyListener(new java.awt.event.KeyAdapter() {
    public void keyReleased(KeyEvent e) {
        jTextFieldAnneeMini_keyReleased(e);
    }
});

...
```

2. JavaFX

JavaFx est une bibliothèque de composants graphiques beaucoup plus modernes que SWING/AWT.

Une version mature de JavaFx a été introduite dans Java 8 (2014).

JavaFx a été enlevé de JAVA_SE à partir de java 11 de façon à

- gagner en légèreté .

- séparer clairement "JavaFx" en tant que complément facultatif (seulement utile pour quelques applications).

JavaFx est aujourd'hui utilisé pour développer en java des interfaces graphiques sophistiquées et est entre autre utilisé par la NASA .

Il existe un projet **JavaFXPorts** consistant à utiliser javaFx pour développer des applications mobiles (pour Andoid , pour IOS/iphone)

2.1. Structures fondamentales de javaFx

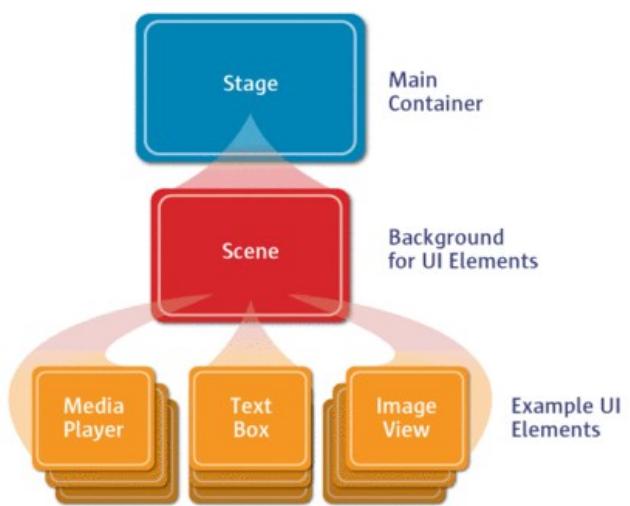
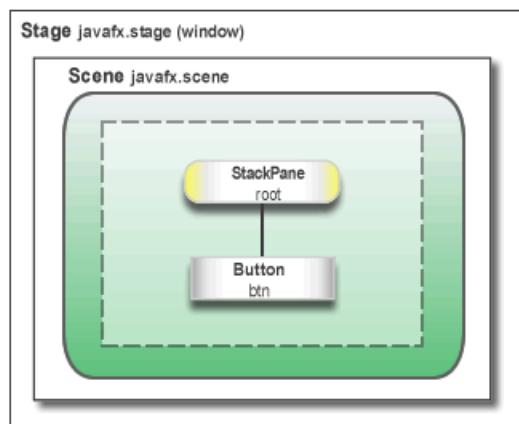


Image provenant de : <http://www.oracle.com>



Le niveau principale "**Stage**" correspond généralement à une **fenêtre** .

Comme dans un **théâtre** , on peut présenter sucessivement plusieurs **scènes** .

Chaque **scène** présentable/interchangeable est constituée d'une **arborescence de composants visuels** (**panneaux** , **boutons** , ...) .

2.2. Exemple élémentaire javaFx

```

package my_java_fx_app;

import javafx.application.Application; import javafx.stage.Stage;
import javafx.event.ActionEvent; import javafx.scene.Scene;
import javafx.scene.control.Button; import javafx.scene.control.Label;
import javafx.scene.layout.Vbox; import javafx.geometry.Pos;

public class HelloWorldJavaFx8App extends Application {

    @Override
    public void start(Stage primaryStage) {
        Button btn = new Button();
        btn.setText("Say 'Hello World'");
        Label labelMessage = new Label();

        btn.setOnAction((ActionEvent event)->{labelMessage.setText("Hello World!");});

        VBox root = new VBox(10); // Create the VBox layout with a 10px spacing
        root.setAlignment(Pos.CENTER);
        root.getChildren().add(btn);
        root.getChildren().add(labelMessage);

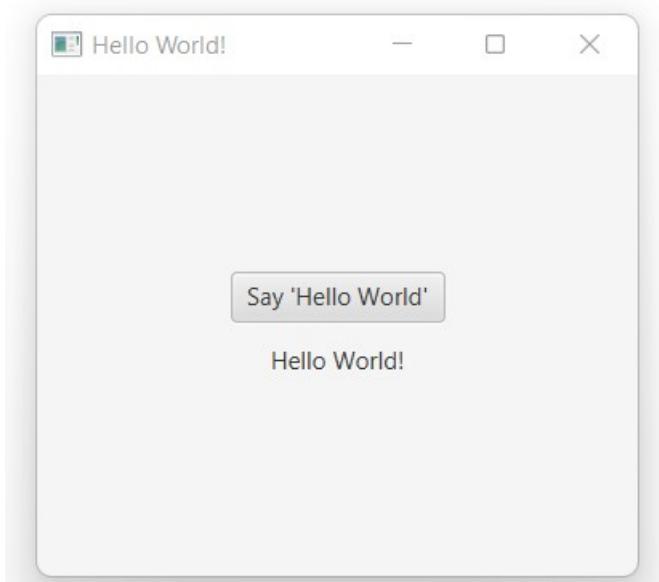
        Scene scene = new Scene(root, 300, 250);

        primaryStage.setTitle("Hello World!");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}

```

mvn clean javafx:run



Configuration maven :**pom.xml**

```

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>tp</groupId>
    <artifactId>myJavaFx8App</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <description>exemple application javaFx - jdk 8 or 11 or 17</description>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <java.version>11</java.version>
        <javafx.version>11.0.2</javafx.version>
        <javafx.maven.plugin.version>0.0.8</javafx.maven.plugin.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.openjfx</groupId>
            <artifactId>javafx-controls</artifactId>
            <version>${javafx.version}</version>
        </dependency>

        <dependency>
            <groupId>org.openjfx</groupId>
            <artifactId>javafx-fxml</artifactId>
            <version>${javafx.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.10.1</version>
                <configuration>
                    <release>${java.version}</release>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.openjfx</groupId>
                <artifactId>javafx-maven-plugin</artifactId>
                <version>${javafx.maven.plugin.version}</version>
                <configuration>
                    <mainClass>my_java_fx_app.HelloWorldJavaFx8App</mainClass>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Ceci permet la lancer **mvn clean javafx:run**

2.3. définitions .fxml et contrôleurs associés :

La structure d'une interface graphique **javaFx** peut être définie dans un fichier xml spécial d'extension **.fxml** qu'il est possible de coder directement en mode texte.

On peut trouver des plugins javaFx pour certains IDE (ex : eclipse) de manière à aider à générer ces fichiers via des drag & drop et des palettes .

En général un fichier .fxml par scène plus un fichier .fxml principal (mainLayout) .
Ces fichiers peuvent être placés dans un répertoire **fxml** placé dans **src/main/resources** .

Chaque fichier **.fxml** décrit clairement **la structure arborescente des composants** .

On a besoin d'associer à chaque fichier .fxml un fichier java (appelé contrôleur) qui va décrire le comportement des composants (réactions aux événements) .

La classe **FXMLLoader** (souvent utilisée au sein de la classe principale) permet de charger en mémoire un assemblage exécutable constitué d'un fichier .fxml et d'un contrôleur associé.

Exemple partiel :

FXML/RootLayout.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.input.*?>
<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane prefHeight="400.0" prefWidth="600.0" stylesheets="@DarkTheme.css"
xmlns:fx="http://javafx.com/fxml" fx:controller="my_java_fx_app.view.RootLayoutController">
    <top>
        <MenuBar BorderPane.alignment="CENTER">
            <menus>
                <Menu text="File">
                    <items>
                        <MenuItem onAction="#handleExit" text="Exit" />
                    </items>
                </Menu>
                <Menu text="Scenes">
                    <items>
                        <MenuItem onAction="#handleDisplayDrawingScene" text="drawing" />
                        <!-- <MenuItem onAction="#handleDisplayOtherScene" text="other" /> -->
                    </items>
                </Menu>
                <Menu text="Help">
                    <items>
                        <MenuItem onAction="#handleAbout" text="About" />
                    </items>
                </Menu>
            </menus>
        </MenuBar>
    </top>
    <!-- <center> variable mainContent / switch here </center> -->
    <bottom>
        <HBox spacing="6.0">
            <Label fx:id="lblMessage" />
        </HBox>
    </bottom>
</BorderPane>
```

```
</HBox>
</bottom>
</BorderPane>
```

fxml/DrawingSceneLayout.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.layout.HBox?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.canvas.Canvas?>
<?import javafx.scene.control.ColorPicker?>
<?import javafx.scene.control.ComboBox?>

<BorderPane prefHeight="363.0" prefWidth="693.0"
stylesheets="@{MyTheme.css" styleClass="drawing-scene-background"
xmlns="http://javafx.com/javafx/8.0.40" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="my_java_fx_app.view.DrawingSceneController">
<top>
<HBox spacing="4.0" >
<Label text="shape:"/><ComboBox fx:id="comboBoxShape"/>
<Label text="color:"/> <ColorPicker fx:id="colorPickerLine" />
<Label text="fill:"/> <ColorPicker fx:id="colorPickerFill" />
<Button text="undo" onAction="#handleUndo" /><Button text="clear" onAction="#handleClearAll" />
</HBox>
</top>
<center>
<AnchorPane fx:id="drawingPane" styleClass="drawing-area-background"/>
</center>
<bottom>
<HBox spacing="6.0" alignment="BASELINE_RIGHT">
<Label text="coords: "><Label fx:id="lblCoords" />
</HBox>
</bottom>
</BorderPane>
```

RootLayoutController.java

```
package my_java_fx_app.view;

import javafx.fxml.FXML;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.Label;
import my_java_fx_app.MyJavaFx8App;

public class RootLayoutController {

    // Reference to the main application:
    private MyJavaFx8App mainApp;

    public void setMainApp(MyJavaFx8App mainApp) {
        this.mainApp = mainApp;
    }

    @FXML //with fx:id="lblMessage" in RootLayout.fxml
    public Label lblMessage;
```

```

@FXML
private void handleAbout() {
    Alert alert = new Alert(AlertType.INFORMATION);
    alert.setTitle("MyJavaFx8App");
    alert.setHeaderText("About");
    alert.setContentText("Author: Didier Defrance");
    alert.showAndWait();
    lblMessage.setText("javaFx app");
}

@FXML
private void handleExit() {
    //System.out.println("exit ...");
    System.exit(0);
}

//...

@FXML
private void handleDisplayDrawingScene() {
    lblMessage.setText("drawing javaFx , SVG, ...");
    mainApp.showDrawingScene();
}

```

MyJavaFx8App.java

```

package my_java_fx_app;

import java.io.IOException;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;
import my_java_fx_app.view.RootLayoutController;

public class MyJavaFx8App extends Application {

    private BorderPane rootLayout=null;
    private Pane drawingPane=null;

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {

        // Load root layout from fxml file.
        FXMLLoader loader = new FXMLLoader();
        loader.setLocation(MyJavaFx8App.class
            .getResource("/fxml/RootLayout.fxml"));
        try {
            rootLayout = (BorderPane) loader.load();
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Give the controller access to the main app.
        RootLayoutController controller = loader.getController(); //fx:controller attribute in RootLayout.fxml
    }
}

```

```

controller.setMainApp(this);

// StackPane root = new StackPane(); Scene scene = new Scene(root, 500, 400);
Scene scene = new Scene(rootLayout); //size in RootLayout.fxml
primaryStage.setTitle("My javaFx App / java8");
primaryStage.setScene(scene);
primaryStage.show();

showDrawingScene();
}

private Pane loadFXMLPane(String fxmlPath) {
    Pane pane=null;
    try {
        FXMLLoader loader = new FXMLLoader();
        loader.setLocation(this.getClass().getResource(fxmlPath));
        pane = (Pane) loader.load();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return pane;
}

public void showDrawingScene() {
    if(drawingPane==null){
        drawingPane = loadFXMLPane("/FXML/DrawingSceneLayout.fxml");
    }
    rootLayout.setCenter(drawingPane);
}
}

```

DrawingSceneController.java

```

package my_java_fx_app.view;

import javafx.beans.property.*;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.fxml.FXML;
import javafx.scene.control.*;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.*;

public class DrawingSceneController {

    @FXML
    private Label lblCoords;

    @FXML
    private ColorPicker colorPickerLine;

    private ObjectProperty<Color> lineColorProperty = new SimpleObjectProperty<Color>() ;

    @FXML
    private ColorPicker colorPickerFill;

    private ObjectProperty<Color> fillColorProperty = new SimpleObjectProperty<Color>() ;

    @FXML

```

```

private ComboBox<String> comboBoxShape;

private ObservableList<String> shapeTypes =
    FXCollections.observableArrayList(
        "LINE",
        "RECTANGLE",
        "CIRCLE"
    );

private StringProperty selectedShapeTypeProperty = new SimpleStringProperty() ;

private Shape currentShape=null; //javafx.scene.shape.Shape

@FXML
private Pane drawingPane;

@FXML
private void handleUndo() {
    if(currentShape!=null){
        drawingPane.getChildren().remove(currentShape);
        currentShape=null;
    }
}

@FXML
private void handleClearAll() {
    drawingPane.getChildren().clear();
    currentShape=null;
}

/**
 * Initializes the controller class. This method is automatically called
 * after the fxml file has been loaded.
 */
@FXML
private void initialize() {
    comboBoxShape.setItems(shapeTypes);
    selectedShapeTypeProperty.bindBidirectional(comboBoxShape.valueProperty());
    selectedShapeTypeProperty.setValue("LINE");//by default

    lineColorProperty.bindBidirectional(colorPickerLine.valueProperty());
    lineColorProperty.setValue(Color.BLACK);//by default

    fillColorProperty.bindBidirectional(colorPickerFill.valueProperty());
    fillColorProperty.setValue(Color.WHITE);//by default

    lblCoords.setText("0,0");

    drawingPane.setOnMousePressed((MouseEvent me)->{
        lblCoords.setText("x="+me.getX()+" ,y="+me.getY());
        initShape(me.getX(),me.getY());
    });
}

drawingPane.setOnMouseReleased((MouseEvent me)->{
    lblCoords.setText("x="+me.getX()+" ,y="+me.getY());
    updateShapeSize(me.getX(),me.getY());
});

drawingPane.setOnMouseDragged((MouseEvent me)->{
    lblCoords.setText("x="+me.getX()+" ,y="+me.getY());
    updateShapeSize(me.getX(),me.getY());
});

```

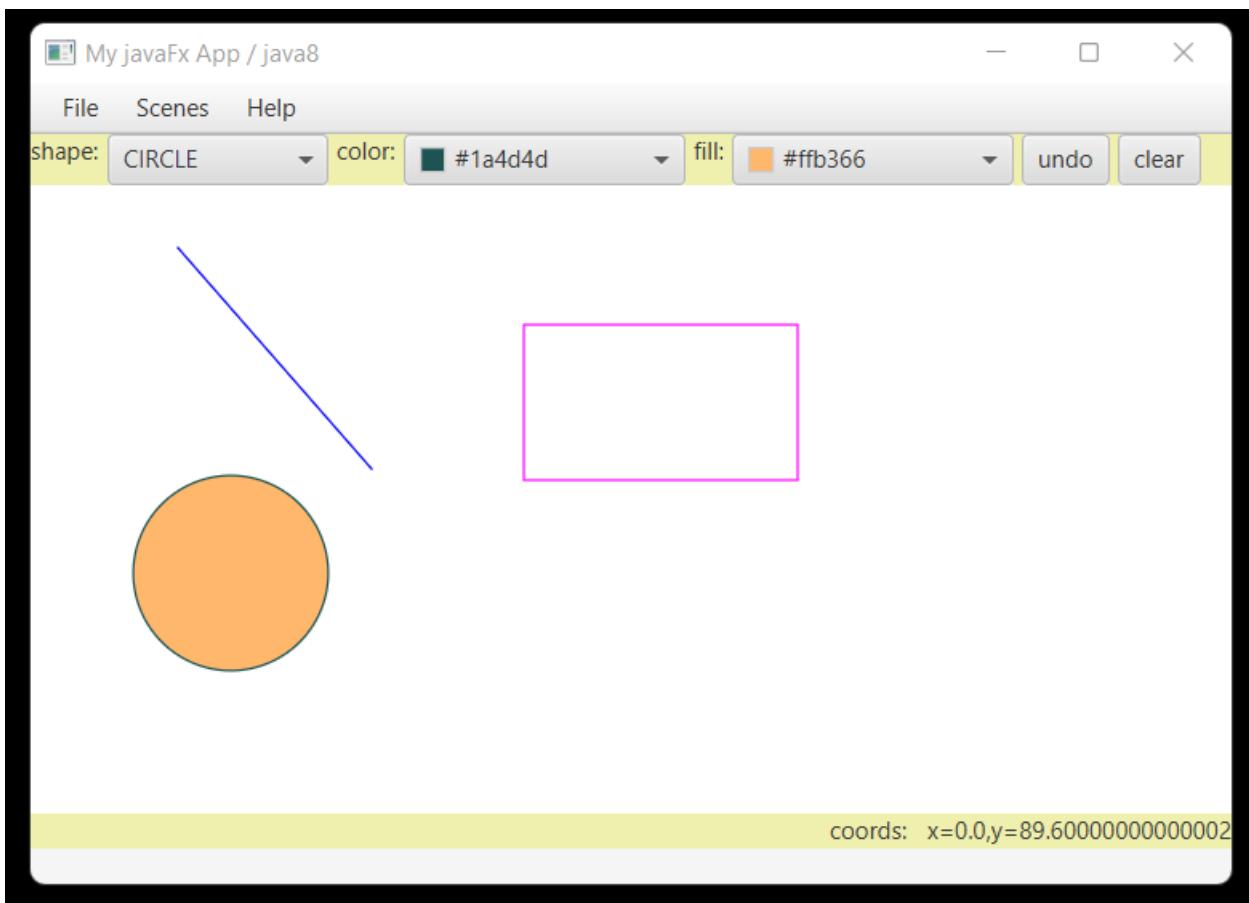
```
        }

    );

    drawingPane.setOnMouseClicked((MouseEvent me)->{
        lblCoords.setText("x="+me.getX()+" ,y="+me.getY());
    });
}

private void initShape(double x,double y){
    switch(this.selectedShapeTypeProperty.get()){
        case "LINE":
            Line line= new Line();
            this.currentShape = line;
            line.setStartX(x);
            line.setStartY(y);
            line.setEndX(x);
            line.setEndY(y);
            break;
        case "RECTANGLE":
            Rectangle rectangle = new Rectangle();
            this.currentShape =rectangle;
            rectangle.setX(x);
            rectangle.setY(y);
            rectangle.setWidth(0);
            rectangle.setHeight(0);
            break;
        case "CIRCLE":
            Circle circle = new Circle();
            this.currentShape = circle;
            circle.setCenterX(x);
            circle.setCenterY(y);
            circle.setRadius(0);
            break;
    }
    this.currentShape.setFill(this.fillColorProperty.get());
    this.currentShape.setStroke(this.lineColorProperty.get());
    this.currentShape.setStrokeWidth(1);
    drawingPane.getChildren().add(currentShape);
}

private void updateShapeSize(double x,double y){
    switch(this.selectedShapeTypeProperty.get()){
        case "LINE":
            Line line= (Line) this.currentShape;
            line.setEndX(x);
            line.setEndY(y);
            break;
        case "RECTANGLE":
            Rectangle rectangle = (Rectangle)this.currentShape;
            rectangle.setWidth(x-rectangle.getX());
            rectangle.setHeight(y-rectangle.getY());
            break;
        case "CIRCLE":
            Circle circle = (Circle)this.currentShape;
            circle.setRadius(Math.sqrt( Math.pow(x-circle.getCenterX(),2)
                + Math.pow(y-circle.getCenterY(),2)));
            break;
    }
}
```



2.4. Autres fonctionnalités/spécificités de javaFx :

Au sein d'un contrôleur , on peut partir d'un simple POJO/javabean de ce genre

```
Person person = new Person("alex" , "Therieur");
person.setEmail("alex.therieur@la-bas.fr");
person.setPhoneNumber("0102030405");
```

et ensuite coder et associer un model javaFx :

```
PersonFxModel personFxModel = new PersonFxModel(); //petite classe à coder en utilisant
//javafx.beans.property.SimpleStringProperty;
personFxModel.setPerson(person);
```

Ceci permet ensuite d'établir **un binding bi-directionnel automatique** entre **un champ de saisie et une propriété (en mémoire)** de l'objet java person (*comme le fait le framework angular*) :

```
txtFieldFirstName.textProperty().bindBidirectional(personFxModel.getFirstNameProperty());
lblPersonData.setText(personFxModel.getPerson().toString());
```

3. Android

3.1. Univers "Android"



Le système d'exploitation "Android" (pour smartphones et tablettes) est basé sur OHA (**Open Handset Alliance**) débuté en 2007 et "**open source**" .

Android est devenu en 2010 , le premier "OS" pour les " téléphones **mobiles**" .

Il ne faut débourser que 25 dollars pour avoir le droit de publier des applications sur "**Android Play Store**" .

Le système d'exploitation Android est techniquement basé sur un **noyau linux** et une machine virtuelle java optimisée (**Dalvik**) ou bien **ART** (*Android RunTime*) depuis la V5 . ART est rétro-compatibile "dalvik" . Le byte-code java est recompilé pour être encore plus efficace.

Une application "Android" ressemble un peu à un module OSGi : elle peut être installée , lancée , stoppée , désinstallée indépendamment des autres applications .

3.2. Développement d'application android (présentation)

Le développement d'application androïd s'effectue en langage java en s'appuyant sur une bibliothèque de librairies spécifiques : "Android SDK".

Un environnement de développement (Eclipse ou bien "Android Studio") est très utile pour :

- écrire le code de l'application (en respectant le format "android")
- gérer des ressources "android"
- packager le code de l'application dans un format compréhensible par la machine virtuelle android
- effectuer des tests avec un "émulateur Android".

3.2.a. Android SDK

"Android Software Development Kit" est une boîte à outils formée de :

- bibliothèque (API) androïd (dans une des versions existantes)
- programmes utilitaires

...

3.2.b. Android Studio (basé sur IntelliJ et gradle)

La plate-forme de développement officielle pour androïd était initialement basée sur eclipse (et il est encore possible d'utiliser des plugins "eclipse" pour Android).

Depuis 2015, Google a maintenant choisi l'IDE "**Android Studio**" comme **plate-forme officielle de développement pour les applications Android**.

Cet IDE est en interne basé sur IntelliJ et gradle (un peu plus récent que "maven") .

Installation :

- Android Studio nécessite préalablement l'installation d'un **jdk** (idéalement 64bits) sous linux ou windows.
- La procédure d'installation de "**Android Studio**" installe en même temps "**Android SDK**" .

3.3. Activité "androïd"

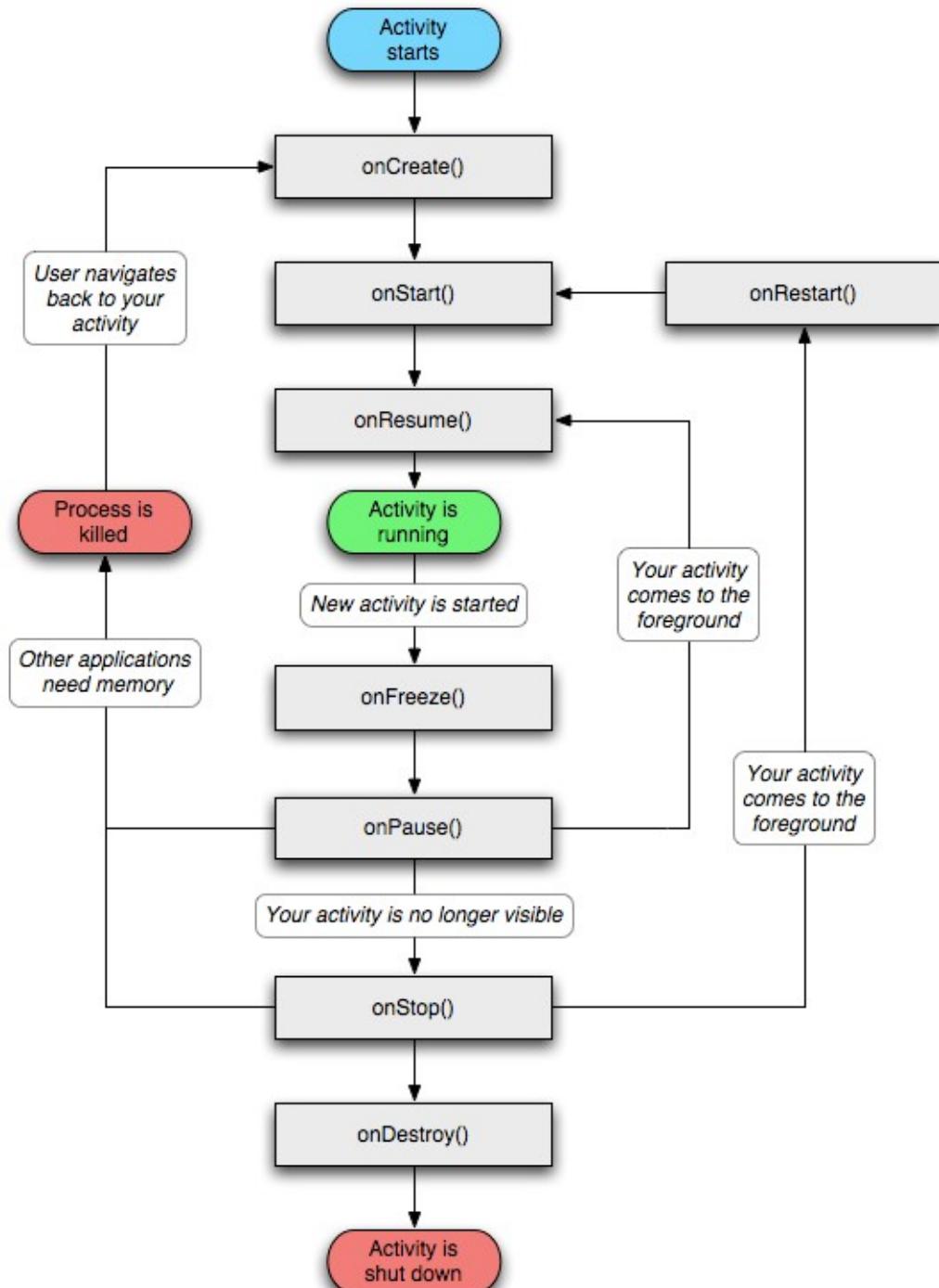
Une **activité androïd** correspond à peu près à un **contexte_applicatif plus une fenêtre principale (occupant assez souvent tout l'écran)** .

Une application simple possède une seule activité.

Lorsque plusieurs applications fonctionnent en même temps , le système androïd gère une **pile**

d'activités (l'activité en sommet de pile est celle qui est visible et qui a la plus grande priorité).

Cycle de vie d'une activité android :



Etats possibles d'une activité :

"active" ou "running"	visible (au sommet de la pile) – cette activité a le "focus"
"paused" (suspendue)	partiellement visible seulement (temporairement suspendue par une autre activité plus prioritaire)
"stopped"	invisible , peut éventuellement être entièrement supprimé si plus de place en mémoire

Toute activité hérite de `android.app.Activity` (héritant elle même de `android.content.Context`) .

Une activité moderne pourra hériter de `android.support.v7.app.AppCompatActivity` qui est une version améliorée (par héritage) de `Activity` .

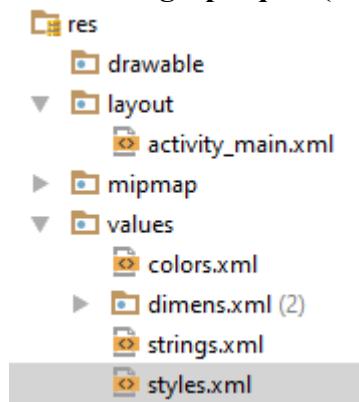
Activité "Hello World" :

```
package tp.myfirstandroidapp;

import android.content.Context;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //setContentView(R.layout.activity_main);
        TextView text = new TextView(this);
        text.setText("Hello world");
        setContentView(text);
    }
}
```

En plus du code java (activité , ...) , une application android est généralement constituée de **ressources graphiques (icônes , thèmes , layout , ...)** souvent au format `xml` ou `png` ou autres .



Toute **application android** doit être décrite par un **fichier de configuration** de ce type :

manifests/AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="tp.myfirstandroidapp">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
```

```

    android:label="@string/app_name"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
<activity android:name=".MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
</application>
</manifest>

```

Tout le code d'une application android doit être packagée en un **fichier ".apk"** (au format ZIP) par exemple via des scripts "**gradle**" de ce type :

MyFirstAndroidApp/**build.gradle**

```

buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.1.0'
    }
}

allprojects {
    repositories {
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}

```

MyFirstAndroidApp/**local.properties**

```
sdk.dir=C:\\\\Prog\\\\Android\\\\sdk
```

MyFirstAndroidApp/**settings.gradle**

```
include ':app'
```

MyFirstAndroidApp/**app/build.gradle**

```

apply plugin: 'com.android.application'

android {
    compileSdkVersion 23
    buildToolsVersion "23.0.3"

    defaultConfig {
        applicationId "tp.myfirstandroidapp"
    }
}

```

```

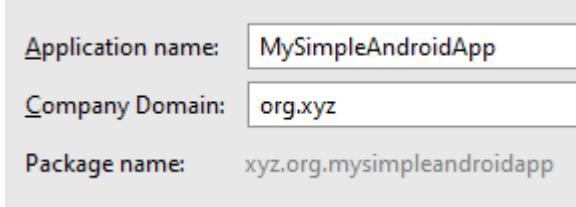
minSdkVersion 16
targetSdkVersion 23
versionCode 1
versionName "1.0"
}
buildTypes {
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
    }
}
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    testCompile 'junit:junit:4.12'
    compile 'com.android.support:appcompat-v7:23.4.0'
}

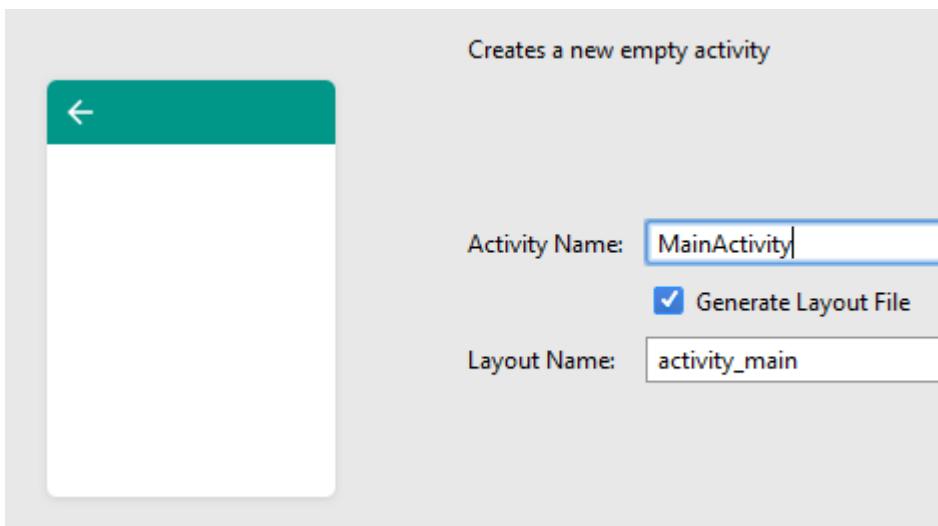
```

Mémento pratique et synthétique avec "Android Studio" :

Création d'une nouvelle application :

File / new project ...



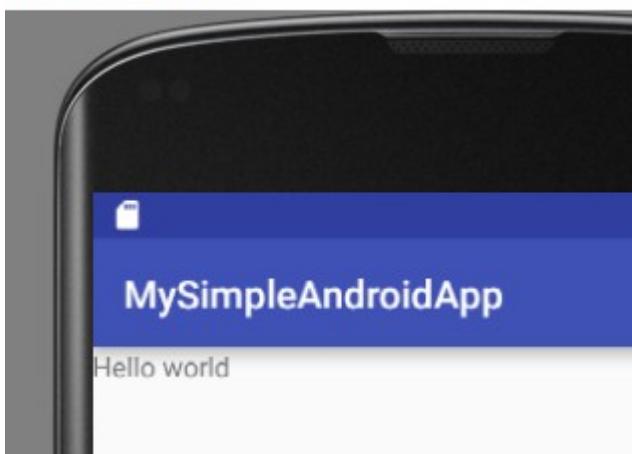


Lancement d'un test :

Menu **Run / run App ...** ou bien icône

... choisir (si besoin) l'émulateur (ex : Nexus 4 ou 5) – Android Virtual Device .

5554:Nexus_4_API_23



3.4. Les ressources (xml , png, ...)

Une ressource est :

- pas en java
- statique (pas dynamique)

Rangement/classement standardisé des ressources :

res/drawable	Dessins et images (png, jpg , git , svg , ...)
res/layout	Dispositions des vues (en XML)
res/menu	Paramétrages (XML) des menus
res/raw	Données brutes (binaires)
res/values	Chaînes de caractères, couleurs , styles, ...

NB : Les répertoires exacts des ressources peuvent éventuellement être organisés (en plusieurs variantes) de la manière suivante :

`res/<type_de_ressource>[<-quantificateur 1><-quantificateur 2>...<-quantificateur N>]`

exemples :

res/drawable-small (pour petits écrans)

res/drawable-large (pour grands écrans)

res/values-fr

Principaux "quantificateurs" de ressources :

<i>quantificateurs</i>	<i>exemples</i>	<i>priorités</i>
Langue et régions	-en , -fr	2
Taille écran	-small , -normal , -large (tablettes) , -xlarge	3
Orientation de l'écran	-port (portrait) -land (landscape/ paysage)	5
Résolution de l'écran	-ldpi (environ 120) -mdpi (160) -hdpi (240) -xhdpi (>=320) -nodpi	8
....		

En se plaçant dans le répertoire "res" , on peut créer un nouveau fichier de ressource via l'assistant "**new / android resource file**" de android studio .

res/values/strings.xml

```
<resources>
    <string name="app_name">MyFirstAndroidApp</string>
    <string name="hello_msg">Hello world</string>
    <string name="welcome_msg">Bonjour %1$s, vous avez %2$d ans.</string>
</resources>
```

Depuis le code java d'une activité , la récupération d'une valeur de ressource s'effectue via la classe "**R**" de la façon suivante :

R.typeRessource.ID_Ressource

Exemples :

```
setContentView(R.layout.activity_main);
text.setText(R.string.hello_msg);
```

Avec auto-complétion automatique lors de la saisie du code !!!

```
Resources res = getResources(); //android.content.res.Resources;
// didier se mettra dans %1 ($s = string) et 45 ira dans %2 ($d=integer)
String chaine = res.getString(R.string.welcome_msg, "didier", 45);
TextView vue = (TextView) findViewById(R.id.vueMsg);
vue.setText(chaine);
```

NB : depuis une autre ressource XML (ex : partie d'un layout) une référence de ressource est exprimée via une syntaxe commençant par le caractère '@' .

@typeRessource/ID_Ressource

exemple :

```
android:layout_alignStart="@+id/textView1"
"@string/hello_msg"
```

Couleurs et styles :

res/values/colors.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#3F51B5</color>
    <color name="colorPrimaryDark">#303F9F</color>
    <color name="colorAccent">#FF4081</color>
</resources>
```

res/values/styles.xml

```
<resources>
    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
        <!--
            <item name="android:textSize">20sp</item>
            <item name="android:textColor">#FF0000</item>
        -->
    </style>
</resources>
```

→ Une ressource de type "style" pourra être référencée par **style="@style/AppTheme"** au sein de <RelativeLayout .../> ou autres "layouts" .

Autre type de ressource (avancée) : animations

à ranger dans res/anim .

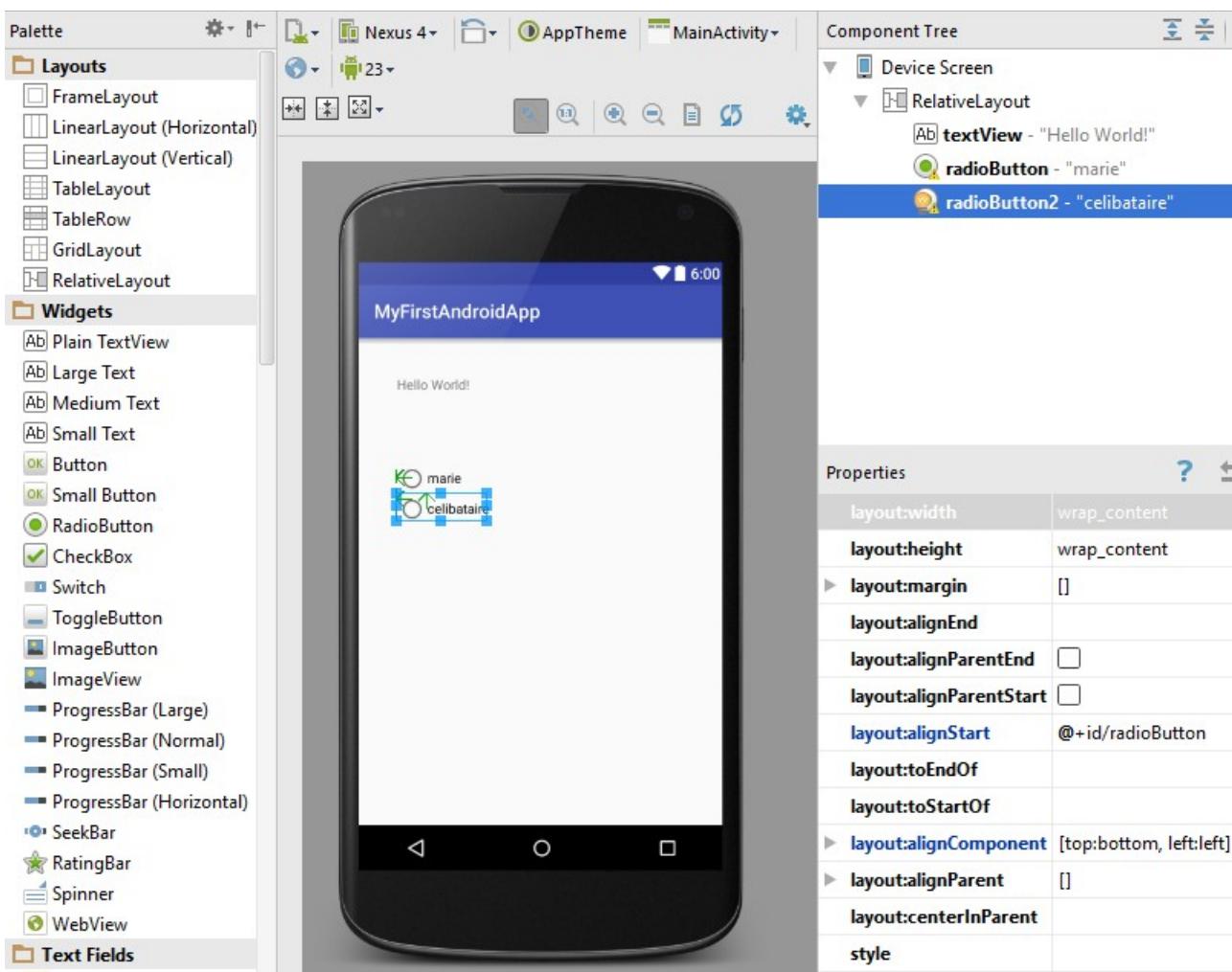
→ pour cas "pointu" seulement , à développer ultérieurement (si le temps le permet).

3.5. Layouts

3.5.a. *Principe et syntaxe XML (layout android)*

Un fichier **res/layout/xlyout.xml** correspond à un encodage XML d'une disposition d'éléments graphiques (zones de textes , labels ,) .

Depuis l'IDE "**Android Studio**" , on peut facilement générer un tel fichier en mode "**wysiswyg**" :



Exemple (xml) partiel :

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="tp.myfirstandroidapp.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        android:layout_marginTop="25dp"
        android:layout_marginLeft="24dp"
        android:layout_marginStart="24dp"
        android:layout_alignParentTop="true"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true" />
```

```

    android:id="@+id/textView" />

<RadioButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="marie"
    android:id="@+id/radioButtonMarie"
    android:layout_below="@+id/textView"
    android:layout_alignLeft="@+id/textView"
    android:layout_alignStart="@+id/textView"
    android:layout_marginTop="74dp"
    android:checked="false" />
...
</RelativeLayout>
```

Utilisation au sein d'une activité java :

```

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        TextView text = (TextView) findViewById(R.id.textView);
        text.setText(R.string.hello_msg);
        RadioButton rbMarie = (RadioButton) findViewById(R.id.radioButtonMarie);
        rbMarie.setChecked(true); //cocher "marie" par défaut
    }
}
```

Arborescence Des Layouts :**View**

ViewGroup (pouvant contenir de multiples "View" /
design pattern "composite")
RelativeLayout , ...Layout
TextView , ... (autres "widget")

3.5.b. Principaux types de "Layout"

Type de Layout	Organisation des éléments	Exemples ou ...
----------------	---------------------------	-----------------

LinearLayout	<p>Alignement en ligne ou verticalement selon orientation = "horizontal" ou "vertical"</p> <p>Paramètres facultatifs des éléments <u>internes</u> :</p> <p>layout_width et layout_height = "fill_parent" or "wrap_content"</p> <p>layout_weight="1" ou "2" , layout_gravity="top , bottom , center , ..."</p>	
RelativeLayout	Placement en relatif par rapport au parent ou bien par rapport à un autre élément (ex : le précédent) .	
TableLayout et TableRow	<p>Placer les éléments comme dans un tableau.</p> <p><u>Paramètres facultatifs des éléments internes</u> :</p> <p>layout_column = "0" ou "1" ou ...</p> <p>layout_span="1" ou "2" , layout_gravity="right , left , center "</p>	
FrameLayout	<p>Affiche un seul élément à la fois (à l'image d'un jeu de cartes).</p> <p>Il faut alors mettre à jour le champ "visibility" des éléments.</p>	
GridLayout	Grille (x,y)	

Type de "container"	Organisation des éléments	Exemples ou ...
ScrollView	Scrolling vertical (et/ou) horizontal	
ListView	
GridView	...	
VideoView	...	

Les "Layouts" peuvent évidemment être **imbriqués** les uns dans les autres .

3.6. Widgets simples

3.6.a. Label ("TextView")

```
textView.setText(R.string.textView);
textView.setTextSize(8);
textView.setTextColor(0x112233); //0x112233 en java , #112233 en valeur texte
```

3.6.b. Zone de saisie ("EditText")

EditText hérite de **TextView** et ajoute le comportement "éeditable" .

```
editText.setHint(R.string.editText);
```

//éventuel mode "multi-line" / textarea :

```
editText.setInputType(InputType.TYPE_TEXT_FLAG_MULTI_LINE);
editText.setLines(5);
```



3.6.c. Bouton poussoir (Button)

Button hérite de **TextView** et ajoute le comportement "cliquable" .

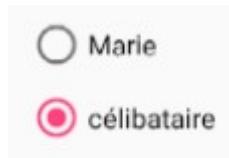
3.6.d. Cases à cocher (non exclusives) / CheckBox



La classe "CheckBox" hérite de "Button".

```
checkBox.setText(R.string.checkBoxXy);
checkBox.setChecked(true) ;
if(checkBox.isChecked())
```

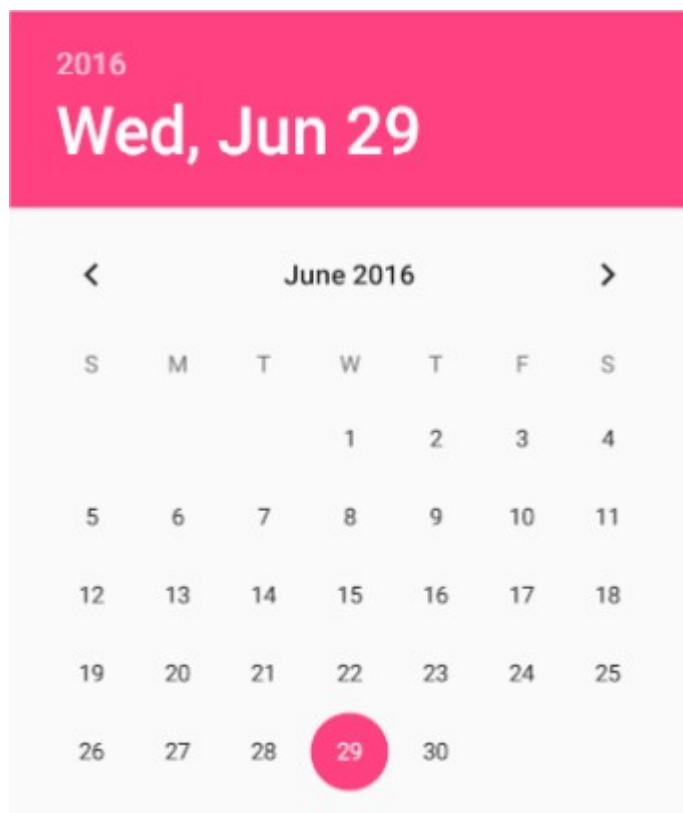
3.6.e. Boutons "radio" / RadioButton , RadioGroup



La classe "RadioButton" hérite de "Button".

Gestion de l'exclusivité via le conteneur "RadioGroup" :

*Les boutons radios exclusifs doivent être insérés dans le même RadioGroup !!!
(en général dès le layout Xml).*

3.6.f. DatePicker (choix de date)

3.7. Gestion des événements "android"

3.7.a. Gestion élémentaire d'événement

```

...
public class MainActivity extends AppCompatActivity {
    TextView textViewMsg;
    RadioButton rbMarie, rbCelib;
    private View.OnClickListener clickListenerRadioBoutons = new View.OnClickListener() {
        public void onClick(View v) {
            textViewMsg.setText("click on id=" + v.getId());
            if(v instanceof RadioButton){
                textViewMsg.append(" text=" + ((RadioButton) v).getText());
            }
        }
    };
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        textViewMsg = (TextView) findViewById(R.id.textView);
        textViewMsg.setText(R.string.hello_msg);
        rbMarie = (RadioButton) findViewById(R.id.radioButtonMarie);
        rbMarie.setOnClickListener(clickListenerRadioBoutons);
        rbCelib = (RadioButton) findViewById(R.id.radioButtonCelibataire);
        rbCelib.setOnClickListener(clickListenerRadioBoutons);
    }
}

```

3.7.b. Principaux événements

Enregistrement du "listener"	Méthode événementielle à coder (public)
view.setOnClickListener(...)	void onClick (View v){ //réagir au click }
view.setOnTouchListener(...)	boolean onTouch (View v, MotionEvent event) { /* Réagir au toucher */ return true; //événement entièrement traité }
view.setOnKeyListener(...)	boolean onKey (View v, int keyCode , KeyEvent event) { /* Réagir à un appui sur une touche */ return true or false ; }
editText.addTextChangedListener(...)	3 méthodes de l'interface TextWatcher

```
private TextWatcher textWatcher = new TextWatcher() {
    @Override
    public void onTextChanged(CharSequence s, int start, int before, int count) {
        // faire éventuellement quelque-chose au moment où le texte change
    }

    @Override
    public void beforeTextChanged(CharSequence s, int start, int count, int after) {
        // faire éventuellement quelque-chose avant que le changement de texte soit pris en compte
    }

    @Override
    public void afterTextChanged(Editable s) {
        // faire éventuellement quelque-chose après que le changement de texte a été pris en compte
    }
};
```

4. Déploiement Java Web Start (de java 5 à java 8)

De 1995 à 2003 (jdk 1.0 à 1.4)	Applet java	Un applet est une petite application java (en awt/swing) qui une fois téléchargé s'exécute dans un navigateur web (avec des extensions)
De 2004 à 2017) (java 5 à 8)	Java Web Start	Un lien hypertexte spécial placé dans une page HTML permet de (télécharger si besoin) puis lancer une application java (souvent basée sur AWT/SWING) au dehors du navigateur et qui sera gérée par une machine virtuelle java devant être installée sur l'ordinateur.
Déconseillé à partir de java 9 et plus maintenu à partir de java 11	Plus rien	<p>La fonctionnalité "Java Web Start" a été retirée. On peut tout de même (sur une page HTML) proposer un téléchargement d'un installeur d'application java (ex : LibreOffice)</p> <p>-----</p> <p>Autre alternative possible (depuis 1999) : appli java côté serveur qui génère des fichiers ".SVG" à télécharger et afficher dans un navigateur</p> <p>-----</p> <p>Autre alternative depuis HTML5 (201x) : code java générant HTML accompagné de code javascript pilotant l'api "canvas" et/ou "chart-js" .</p>

NB : La tentative "**Applet java**" de la fin des années 1990 a plus ou moins échoué car :

- machine virtuelle java longue à démarrer (performances moyennes , conso mémoire , ..)
- problème de sécurité (fonctionnalités très limitées sauf si ajout de certificats complexes)
- problème de compatibilité avec différents navigateurs (IE, netscape , ...)

Java Web Start a ensuite été proposé en tant que planB de java 5 à java 8.

Comme exemple concret d'application quelquefois démarrée (selon version) via java web start on peut citer "**pronote**" qui est une application souvent proposée par les collèges et les lycées et qui sert à visualiser les notes des élèves sur internet (la partie graphique de pronote permet d'afficher des diagrammes : évolutions des notes , comparaison vis à vis de la moyenne de la classe , ...).

4.1. Mise en œuvre de java web start

Une page html peut comporter un lien hypertexte du genre:

```
<a href="MyApp.jnlp">Launch My Application</a>
```

L'utilisateur pourra ainsi d'un simple click déclencher une **requête http** dont l'url se termine par l'extension "**.jnlp**" .

Si le serveur web (ex: apache) est bien configuré il va associer l'extension **.jnlp** au type MIME **application/x-java-jnlp-file** .

[Apache nécessite à cet effet la ligne **application/x-java-jnlp-file JNLP** dans le fichier de configuration **.mime.types**]

Le fichier **jnlp** est un cas particulier de fichier xml ayant la forme suivante:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- JNLP File for JAVA WEB START -->
<jnlp
  spec="1.0+"
  codebase="http://www.yyy.com/siteA/apps"
  href="xxx.jnlp">
  <information>
    <title>xxx Application</title>
    <vendor>YYY</vendor>
    <homepage href="docs/help.html"/>
    <description>xxx Application</description>
    <description kind="short">blabla.</description>
    <icon href="images/xxx.jpg"/>
    <icon kind="splash" href="images/splash.gif"/>
    <offline-allowed/>
  </information>
  <security>
    <!-- <all-permissions/> -->
  </security>
  <resources>
    <j2se version="1.3"/>
    <jar href="lib/xxx.jar"/>
  </resources>
  <application-desc main-class="XxxApp">
    <!-- <argument>arg1</argument>
        <argument>arg2</argument> -->
  </application-desc>
</jnlp>
```

Il faut évidemment veiller à ce tous les éléments nécessaires au fonctionnement de l'application soient accessibles sur le serveur web aux endroits spécifiés par le fichier jnlp :

- Par rapport au **codebase** précisé ("http://..." ou "[file:///c:/tp/...](#)") on doit essentiellement trouver un fichier **.jar** dont le chemin relatif est exprimé dans l'attribut **href** de la sous balise **<jar>** de la balise **<resources>** .
- Ce fichier jar doit comporter les classes de l'application dont celle qui comporte la fonction principale **main()** et dont le nom est précisée par l'attribut **main-class** de **<application-desc>** .
- ...

4.2. Acquisition des permissions selon l'accord de l'utilisateur

JNLP signifie Java Network Launching Protocol .

Outre la technologie Java Web Start , **JNLP** consiste également en une **Api** permettant d'obtenir certaines permissions avec l'accord de l'utilisateur (via des boîtes de dialogue lancées

automatiquement).

4.2.a. Demander l'autorisation de lire un fichier

```
import javax.jnlp.*;
...
FileOpenService fos;

try {
    fos = (FileOpenService)
        ServiceManager.lookup("javax.jnlp.FileOpenService");
} catch (UnavailableServiceException e) {
    fos = null;
}

if (fos != null) {
    try {
        // demander à l'utilisateur de sélectionner un
        // fichier via ce service:
        FileContents fc = fos.openFileDialog(null, null);
        /* FileContents [] fcs = fos.openMultiFileDialog(null, null); */
        InputStream fluxLecture = fc.getInputStream();
        n=fc.getLength(); ...
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

4.2.b. Demander l'autorisation d'écrire dans un fichier

```
FileSaveService fss;

try {
    fss = (FileSaveService)
        ServiceManager.lookup("javax.jnlp.FileSaveService");
} catch (UnavailableServiceException e) {
    fss = null;
}

if (fss != null) {
    try {
        ...
        // demander à l'utilisateur de sauvegarder un
        // fichier via ce service:
        FileContents resFc =
            fss.saveAsFileDialog(null, null, fc);
        if(resFc==null) System.err.println("opération annulée");
        ...
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

4.2.c. Présentation des principaux services de JNLP

FileOpenService	Lire les données d'un fichier sur le système de fichier local (poste client)
FileSaveService	Sauvegarder un fichier localement
BasicService	Permet de récupérer via <code>getCodeBase()</code> l'url du serveur web à partir duquel le téléchargement a été effectué (tel que par exemple: " <code>http://www.xxx.com/siteyyy</code> ").
PersistentService	Etre autorisé à stocker des données sur le poste du client , sachant que ces données sont identifiées par des url (comme les cookies): <code>URL uri = basicServ.getCodeBase();</code> <code>persistentServ.create(uri,tailleMax);</code> <code>FileContents fc = persistentServ.get(uri);</code>
PrintService	Permettre une impression coté client
ClipboardService	Etre autorisé à lire ou écrire dans le presse papier.

Nb:

- Tous ces services sont utiles lorsque l'application téléchargée (via java web start) n'a par défaut que très peu de permissions.
- La classe **FileContents** représente un **fichier local** (avec son nom et ses données) . La principale restriction est que l'on ne peut pas accroître facilement la taille du fichier.

5. Quelques alternatives "non-java"

Pour de developpement d'applications mobiles graphiques :

- **PWA** (Progressive Web App) par exemple avec javascript/typescript/**angular**
- **application mobile hybride** (javavascript/typescript avec **Cordova/ionic**).
- ...

Pour de developpement d'applications "desktop" graphiques :

- **npm/nodeJs/electron**
- ...

Pour de developpement de graphiques (courbes , diagrammes,) :

- **api javascript "canvas" + api "chartJs" ou autres**
- *génération de fichier SVG*

V - Persistence

1. JDBC

1.1. Présentation de JDBC

JDBC signifie ***Java DataBase Connectivity*** .

Il s'agit d'une **API standard** et de bas niveau **permettant à un programme Java de s'interfacer** avec **une base de données relationnelle** quelconque (Oracle, DB2, Informix, ...) .

L'api **JDBC** correspond au package **java.sql** .

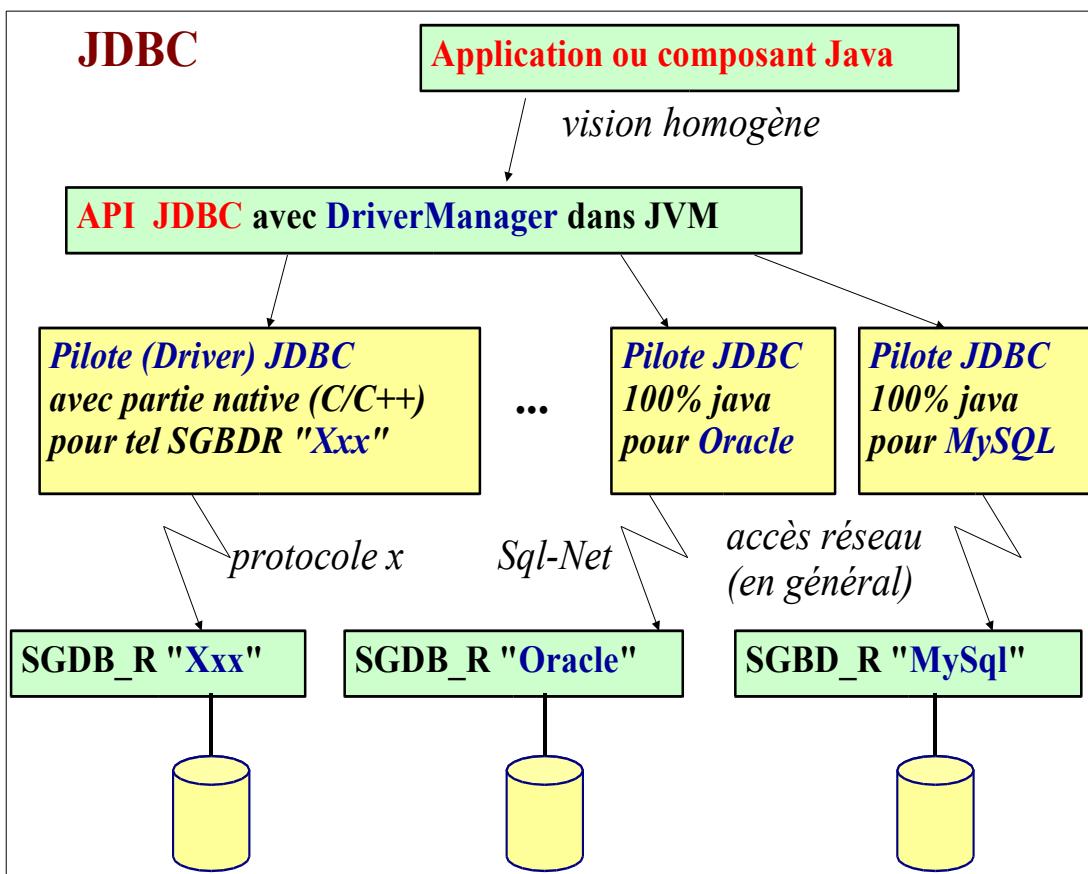
javax.sql.DataSource est une extension standard permettant de gérer **des sources de données pré-paramétrées** et souvent associées à des **Pools de connexions**.

Principales fonctionnalités de l'api JDBC :

- * Effectuer une connexions vers une base de données
- * Lancer des ordres SQL "Insert into, Delete From , Update, ..."
- * Récupérer des enregistrements suite à une requête "Select ... From"
- * Récupérer des informations sur la structure d'une base (MetaData)
- * Déclencher des procédures stockées,

Présentation de JDBC

- JDBC signifie ***Java DataBase Connectivity*** .
- Il s'agit d'une **API standard** et de bas niveau **permettant à un programme Java de s'interfacer** avec **une base de données relationnelle** quelconque (Oracle, DB2, Informix, ...) .
- L'api **JDBC** correspond au package **java.sql** .
- **javax.sql.DataSource** est une extension standard permettant de gérer les **Pools de connexions**.
- Principales fonctionnalités de l'api JDBC :
 - * Effectuer une connexions vers une base de données
 - * Lancer des ordres SQL "Insert into, Delete From , Update, ..."
 - * Récupérer des enregistrements suite à une requête "Select ... From"
 - * Récupérer des informations sur la structure d'une base (MetaData)
 - * Déclencher des procédures stockées
 - * ...



Quel que soit le type de driver JDBC , celui-ci est lié à un type de SGBD (Oracle, DB2, Sybase, Informix, ...).

En règle générale, les éditeurs de SGBD-R (Oracle , IBM, ...) proposent un télé-chargement gratuit des drivers JDBC nécessaires pour se connecter à leurs produits.

La solution idéale est un pilote JDBC 100% Java donc indépendant de toute plate-forme.

1.2. Paramétrage et établissement d'une connexion

Paramétrages à effectuer avec JDBC

* Le **CLASSPATH** doit comporter le ".jar" ou ".zip" contenant le code Java du Driver JDBC que l'on souhaite utiliser.

* **Choix du Driver JDBC à charger en mémoire :**

`Class.forName(chNomClasseDuDriver);`

→ "com.mysql.jdbc.Driver" pour **MySQL**

→ "oracle.jdbc.driver.OracleDriver" pour **Oracle 8i**

* **Url désignant la base de donnée vers laquelle on souhaite établir une connexion :**

Connection cn = DriverManager.getConnection(

chUrlDB, [userName],[password]);

→ "jdbc:mysql://xxxhost/xxxxdb" pour **MySQL**

→ "jdbc:oracle:thin:@xxxhost:1521:myDB" pour **Oracle 8i**

1.3. Driver/Pilote JDBC et URL de connexion

SGBD-R	<i>archive contenant le code java du driver JDBC</i>	<i>classe du pilote JDBC</i>	<i>URL</i>
Oracle	...\\ora81\\jdbc\\lib\\classe12.zip ou ojdbc14.jar	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@localhost:1521:myDB
DB2	...\\SQLLIB\\java\\db2java.zip		
MySQL	...\\mysql\\lib\\mysql-connector-java-xxxx-bin.jar	com.mysql.jdbc.Driver	jdbc:mysql://localhost/xxxxdb
Access	dans runtime JVM Win32	sun.jdbc.odbc.JdbcOdbcDriver	jdbc:odbc:dsnxxx
...			

1.4. Pool de connexions et DataSource JDBC

Pool de connexions vers SGBDR

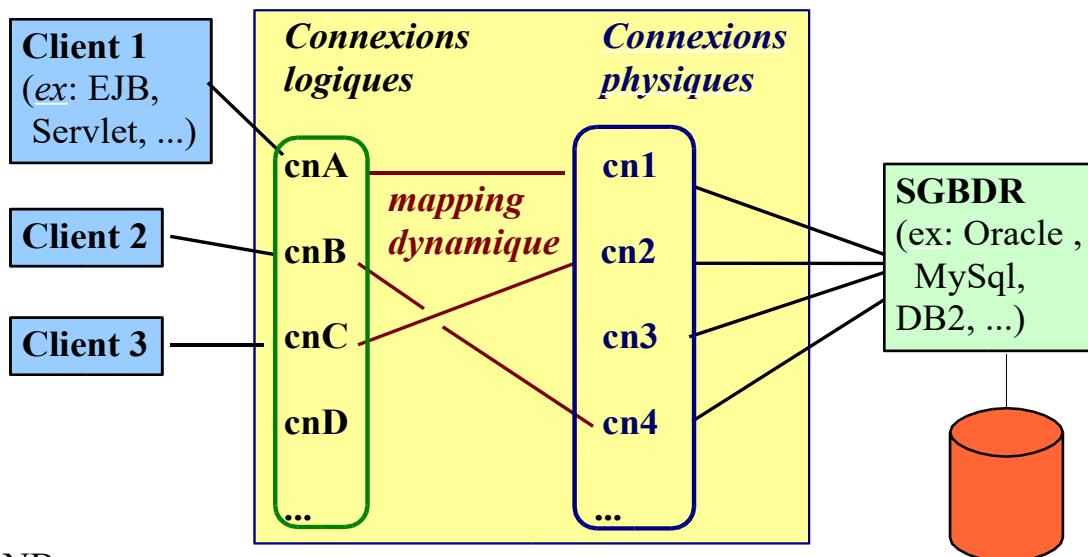
Rôles (utilités) des pools de connexions:

Recycler et *partager* (par différentes attributions successives) un ensemble de connexions physiques vers un certain SGBDR.

Ceci permet d'éviter les 2 écueils suivants:

- Ouvrir, fermer et ré-ouvrir , ... des connexions vers le SGBDR (opérations longues répétées → mauvaises performances).
- Utiliser simultanément une même connexion pour effectuer de multiples traitements → mauvaise gestion des concurrences d'accès et des transactions (joyeux mélanges)

Pool de connexions



NB:

Dès d'un client ferme une connexion logique , la connexion physique associée est considérée comme libre et peut alors être recyclée de façon à ce qu'un autre client puisse obtenir une nouvelle connexion logique.

Remarque: Etant donné qu'une connexion libérée (via close) par un composant n'est pas vraiment fermée mais peut être tout de suite réutilisée par un autre composant, chaque traitement (à l'intérieur d'une méthode d'un composant) doit:

- demander une connexion disponible dans le pool
- l'utiliser brièvement
- rapidement la libérer

Vue du pool par le client java - **DataSource**

Un client java voit un pool de connexions JDBC comme un objet de type `javax.sql.DataSource`.

L'accès à cette source de données découle d'une **recherche JNDI** à partir d'un nom convenu (à paramétrer):

```
InitialContext ic = new InitialContext();
String dsName="java:comp/env/jdbc/dsBaseX"
DataSource ds = (DataSource) ic.lookup(dsName);
```

L'objet *DataSource* permet alors de **récupérer de nouvelles connexions logiques**:

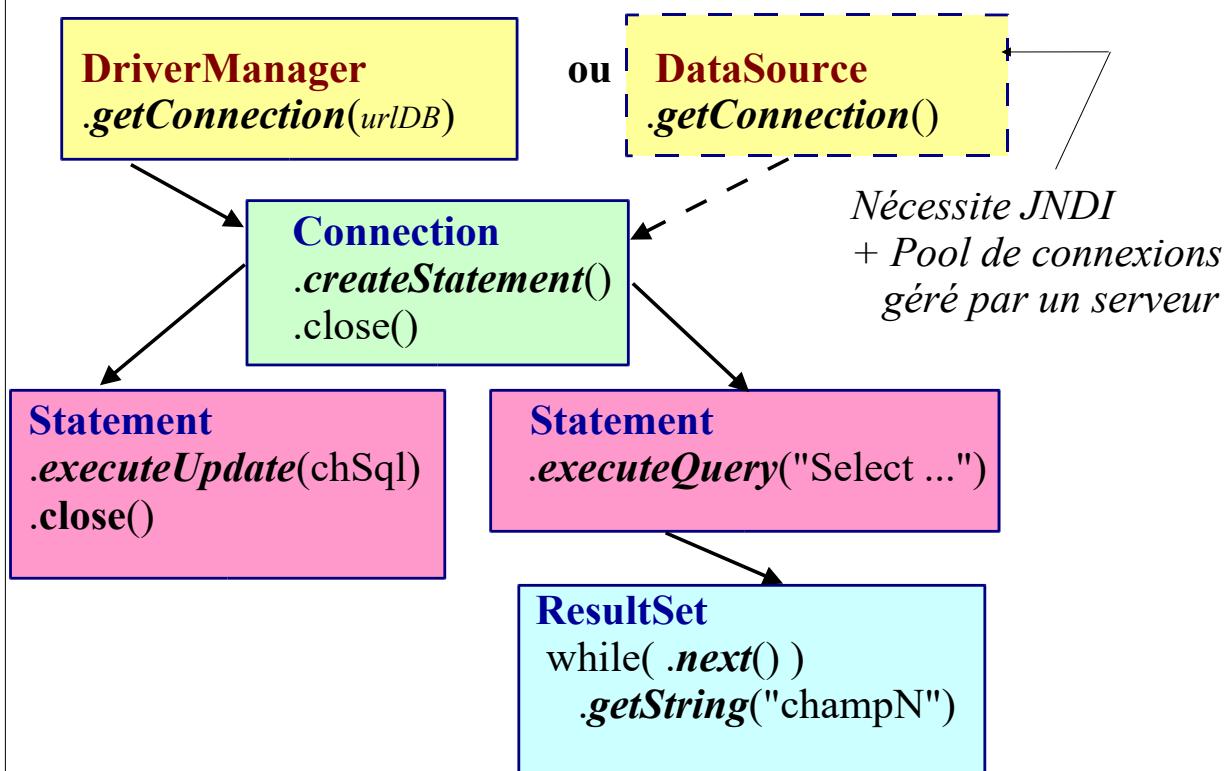
```
Connection cn = ds.getConnection();

// ... utilisation classique d'une connexion JDBC ...

cn.close(); // fermeture de la connexion logique
```

1.5. Utilisation de JDBC pour lancer des requêtes SQL

Principaux types d'objets de l'api JDBC



Exemple de code jdbc:

```
// chargement en mémoire de la classe de pilote/driver jdbc:  
Class.forName("com.mysql.jdbc.Driver");  
...  
// Ouverture de la connexion:  
String chUrl_db ="jdbc:mysql://localhost/test"; // version MySql  
Connection cn = DriverManager.getConnection(chUrl_db,user,password);  
  
Statement statement=cn.createStatement();  
String chOrdreSql = "DELETE FROM telephone WHERE nom='dupond' ";  
int nbLignesAffectees= statement.executeUpdate(chOrdreSql);  
statement.close();  
...  
statement=cn.createStatement();  
String chRequeteSql = "SELECT * FROM telephone WHERE numero=0102030405 ";  
ResultSet rs = statement.executeQuery(chRequeteSql);  
while(rs.next())  
{  
    String nom = rs.getString("NOM"); // récupérer la valeur de la colonne "NOM"  
    String numero = rs.getString("NUMERO"); ...  
}  
rs.close(); statement.close(); cn.close();  
// + traitement des exceptions (SQLException , ...)
```

1.6. Gestion des transactions (tout ou rien)

```
connexion.setAutoCommit(false); // true par défaut  
  
// quelques ordres SQL (Mises à jour , Insertions , Suppressions)  
  
if(...)  
    connexion.commit(); // pour valider Toutes les mј.  
else  
    connexion.rollback(); // pour annuler toutes les mј depuis le dernier commit/rollback
```

1.7. Préparer et lancer n fois un ordre Sql paramétrable

```
PreparedStatement pstmt = cn.prepareStatement( "Insert Into Table2 Values (? , ?) " );  
for(int i=0;i<3;i++)  
{  
    pstmt.setString(1,"Valeur"+i /*valeur du champ1 (texte)* /);
```

```
pstmt.setInt(2,i /*valeur du champ2 (numérique)*/ );  
pstmt.executeUpdate();  
}  
...
```

1.8. Autres fonctionnalités importantes de JDBC

- Accès à la structure de la base (**MetaData**)
- Appels de procédures stockées
- isolations transactionnelles
- Récupérer la valeur d'une clef auto-incrémentée
- ...

2. Problématique "O.R.M."

2.1. Objectif & contraintes:

L'objectif principal d'une technologie de **mapping objet/Relationnel** est d'établir une **correspondance relativement transparente et efficace** entre :

un ensemble d'objets en mémoire

et

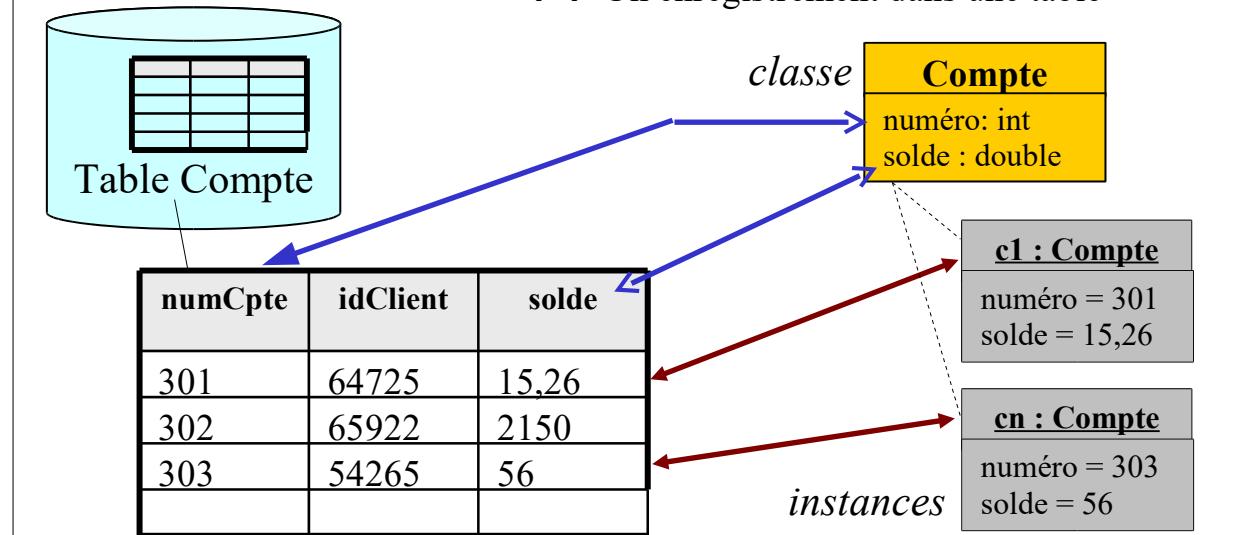
un ensemble d'enregistrements d'une base relationnelle.

O.R.M. (*Mapping Objet-Relationnel*)

Entité persistante = vue orientée objet d'un enregistrement

En gros (à peu près): Une instance (@Entity)

↔ Un enregistrement dans une table



Une telle technologie doit permettre à **un programme orienté objet de ne voir que des objets** dont certains sont des **objets persistants**. *Le code SQL est en très grande partie caché* car les objets persistants sont automatiquement pris en charge par la technologie "O.R.M."

Autrement dit, **le code SQL n'est plus (ou très peu) dans le code "java" mais est généré automatiquement à partir d'une configuration de mapping (fichiers XML, annotations, ...)**.

Contraintes : pour être **exploitable**, une **technologie "O.R.M."** se doit d'être :

- simple (à configurer et à utiliser) et **intuitive**
- portable (possibilité de l'intégrer facilement dans différents serveurs d'application)
- efficace (bonnes performances, fiable , ...)
- capable de gérer les **transactions** ou bien de participer à une transaction déjà initiée
- ...

2.2. Eléments techniques devant être bien gérés

Au delà des simples associations élémentaires du type

"1 enregistrement d'une table <---> 1 instance d'une classe" ,

une **technologie "O.R.M." sophistiquée** doit savoir gérer certaines **correspondances évoluées** :

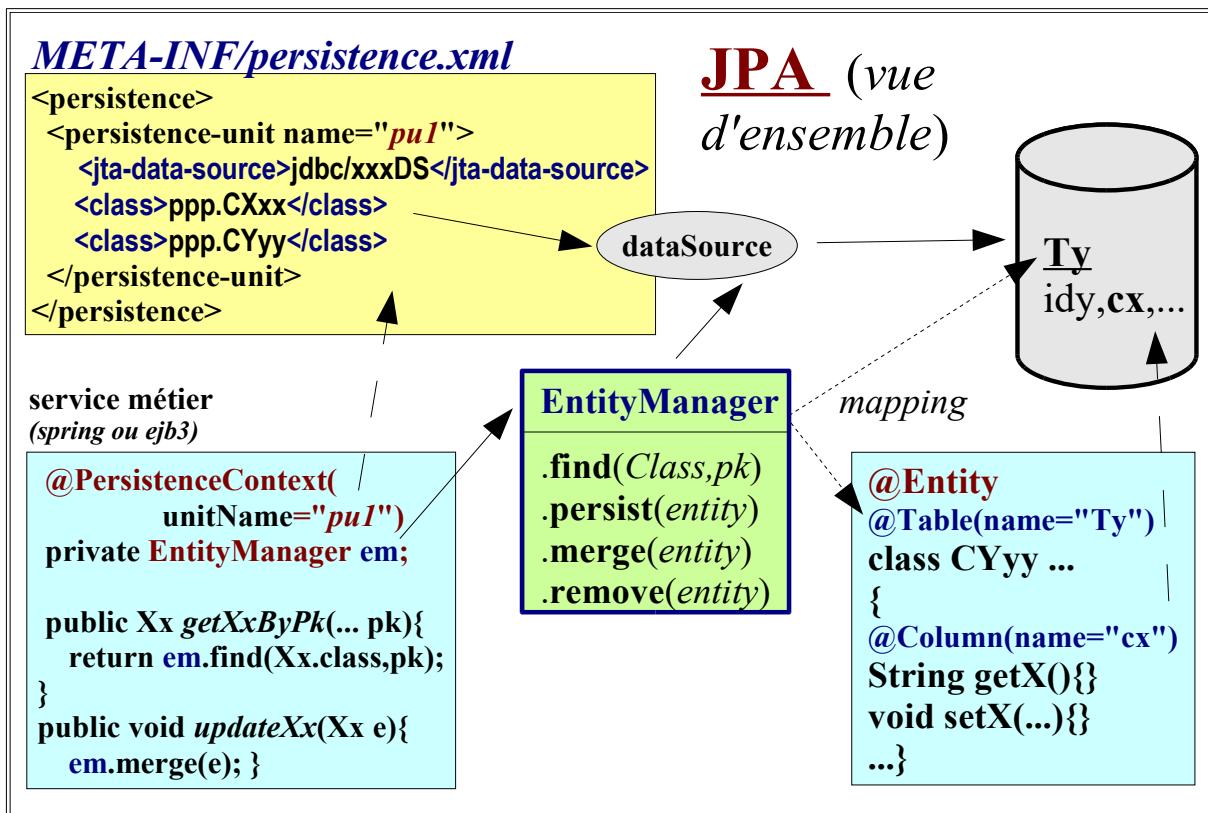
- Associations "1-1" , "1-n" , "n-n" (jointures entre tables <---> relations entre objets)
- Objets "valeur" (sous parties d'enregistrements , tables secondaires , ..)
- Généralisation/Héritage/Polymorphisme (<---> schéma relationnel ?)
- ...

Une technologie "O.R.M." doit sur ces différents points être **flexible et efficace** .

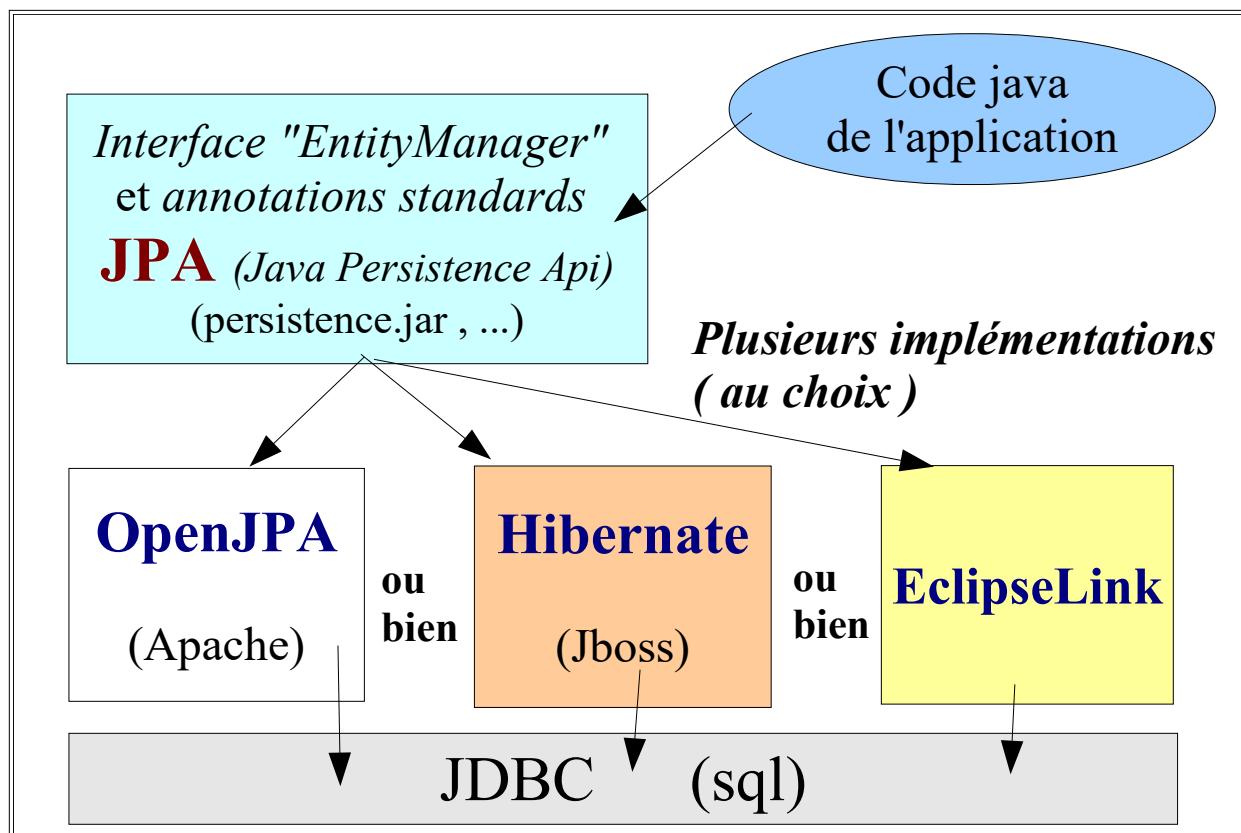
3. JPA et JPA 2 (Java Persistence Api)

JPA (Java Persistence Api) et **@Entity**

- API O.R.M. officielle de Java EE
- Configuration basée sur des **annotations** (*avec configuration xml possible : orm.xml*).
- Utilisable dans les contextes suivants:
 - dans **module d'EJB3**
 - dans **module Spring**
 - de façon **autonome** (java sans serveur)
- Plusieurs **implémentations** internes disponibles (*OpenJPA* , *Hibernate* depuis version 3.2 ,)



3.1. Plusieurs implémentations disponibles pour JPA



Selon le contexte, l'implémentation de JPA sera libre ou bien quasi imposée par le serveur d'applications (ex : JBoss Application Serveur utilise en interne Hibernate pour implémenter JPA).

3.2. Exemple d'entité persistante (@Entity)

```

@Entity
@Table(name = "T_CUSTOMER")
public class Customer implements Serializable {

    @Id
private Long number;

    @Column(name = "customer_name")
private String name;
private Address address;

    @OneToMany(...)
private List<Order> orders = new ArrayList<Order>();

    @ManyToMany(...)
private List<PhoneNumber> phones = new ArrayList<PhoneNumber>();

    public Customer() {} // No-arg constructor

    public Long getNumber() {return number;} // pk
    public void setNumber(Long number) {this.number = number;}

    public String getName() {return name;}
    public void setName(String name) {this.name = name;}
    public Address getAddress() {return address;}
    public void setAddress(Address address) {this.address = address;}

    public List<Order> getOrders() {return orders;}
    public void setOrders(List<Order> orders) {
        this.orders = orders;
    }
    public List<PhoneNumber> getPhones() {return phones;}
    public void setPhones(List<PhoneNumber> phones) {
        this.phones = phones;
    }

    // Business method to add a phone number to the customer
    public void addPhone(PhoneNumber phone) {
        this.getPhones().add(phone);
        // Update the phone entity instance to refer to this customer
        phone.addCustomer(this);
    }
}

```

Dans l'exemple précédent :

- l'annotation **@Entity** permet de marquer la classe Customer comme une classe d'objets persistants
- l'annotation **@Table(name="T_CUSTOMER")** permet d'associer la classe java "Customer" à la table relationnelle "T_CUSTOMER".

- l'annotation `@Column(name="customer_name")` permet d'associer l'attribut "name" de la classe "Customer" à la colonne "customer_name" de la table relationnelle.
- l'annotation `@Id` permet de marquer le champ "number" comme **identifiant de l'entité (et comme clef primaire de la table)**.
- les autres annotations (`@OneToMany` , ...) seront approfondies dans un chapitre ultérieur.

NB: Lorsqu'une annotation (`@Column` ou ...) n'est pas présente , les noms sont censés coïncider (sachant que les minuscules et majuscules ont quelquefois de l'importance du côté SQL/relationnel selon le système d'exploitation hôte (ex: linux)).

Remarques importantes:

- Les annotations paramétrant le mapping objet/relationnel (`@Id` , `@Column`, `@OneToMany`,) peuvent soit être placées au dessus de l'attribut privé soit être placées au dessus de la méthode en "get".
 [au dessus des "private" --> plus lisible car dans le haut de la classe]
 [au dessus des "get" --> plus clair et/ou plus fonctionnel en cas d'héritage]
- **Attention:** il faut absolument garder un style homogène (si quelques annotations au dessus des "privates" et quelques annotations au dessus des "getXxx()" --> ça ne fonctionne pas !!! sauf si annotation spéciale `@Access` de JPA 2 avec *AccessType.FIELD* ou *PROPERTY*)
- Les annotations paramétrant des injections de dépendances (`@Resource` , `@PersistenceContext` , ...) peuvent soit être placées au dessus de l'attribut privé soit être placées au dessus de la méthode en "set".
- Même si une propriété (avec private +get/set) n'est marquée par aucune annotation , elle sera tout de même prise en compte lors du mapping objet-relationnel.
- Pour éventuellement (si nécessaire) désactiver le mapping d'une propriété d'une classe persistante, il faut utiliser l'annotation "`@Transient`" (au dessus du "private" ou du "getter" de la propriété).
- `@Enumerated(EnumType.STRING)` permet de stocker la valeur d'une énumération sous forme de chaîne de caractères dans une colonne d'une table relationnelle.
- `@Temporal(TemporalType.DATE ou TemporalType.TIME ou TemporalType.TIMESTAMP)` permet de préciser la valeur significative d'une date java (java.util.Date) .
- ...

3.3. Différents états - objet potentiellement persistant

Etats objets	Caractéristiques
transient	Nouvel objet créé en mémoire, pris en charge par la JVM Java mais pas encore contrôlé par une session Hibernate ou bien l'entityManager de JPA (le mapping objet/relationnel n'a jamais été activé). un tel objet n'a quelquefois pas encore de clef primaire (elle sera souvent attribuée plus tard lors d'un appel à
(proche état	

Etats objets	Caractéristiques
détaché	entityManager.persist() ou session.save()
persistant	objet actuellement sous le contrôle d'une session Hibernate ou de l'entityManager de JPA (<i>avant session.close() ou entityManager.close()</i>). Le mapping objet/relationnel est alors actif et une synchronisation (mémoire ==> base de données) est alors automatiquement déclenchée suite à une mise à jour (changement d'une valeur d'un attribut).
détaché	<p>objet qui n'est plus sous le contrôle d'une session Hibernate ou de l'entityManager de JPA (<i>après session.close() ou entityManager.close()</i>). Les valeurs en mémoire sont conservées mais ne seront plus mises à jour (mapping objet/relationnel désactivé).</p> <p>Un objet détaché pourra éventuellement être ré-attaché à une session Hibernate (lors d'un update ou save_or_update() sur une session hibernate ou lors d'un appel à merge() sur l'entityManager de JPA) et ses éventuelles nouvelles valeurs pourront alors être synchronisées dans la base de données.</p>

3.4. Cycle de vie d'un objet JPA/Hibernate

transient ==> **persistent** (==> ...)

(via **new**) (via **session.save()** ou
 entityManager.persist())

insert into

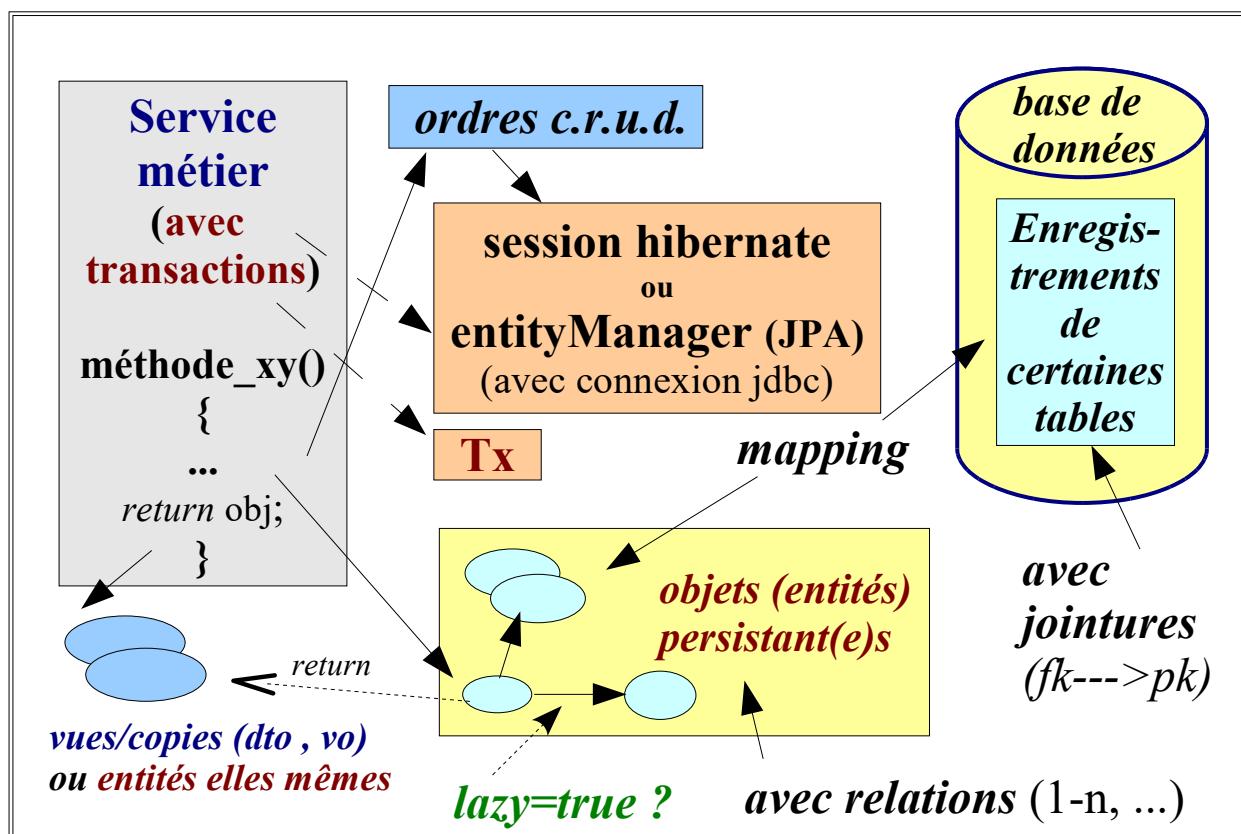
ou bien (après hibernation prolongée):

persistent	==>	détaché	(==> persistent)
(via query)		(via session.close() ou entityManager.close())	(via session.update() , ... ou entityManager.merge())
select			update set ...

3.5. Synchronisation automatique dans l'état persistant

Synchronisation automatique dans l'état persistant (.update() ou .persist() non obligatoire), mais .save() ou .merge() obligatoire pour un objet détaché.

3.6. objet persistant et architecture n-tiers



Remarques importantes:

- La technologie O.R.M. JPA ou hibernate sert essentiellement à remonter (et gérer) en mémoire un ensemble d'objets Java en tous points synchronisés avec les enregistrements d'une base de données relationnelle.
- Pour obtenir de bonnes performances (`lazy=true`) sans pour autant trop compliquer le code de l'application, il faut considérer les objets "entités persistantes" comme une structure d'ensemble orientée objet virtuellement liée à l'ensemble des données de la base (sachant que les mécanismes internes de JPA/Hibernate remontent les données en mémoire qu'en fonction des accès réellement effectués sur la structure objet [`lazy=true`]).
- La couche métier appelante (généralement développée avec Spring ou des EJB) doit souvent retourner des valeurs vers la couche présentation (IHM). Ces valeurs sont soit des références directes sur les entités persistantes elles mêmes ou bien des copies partielles sous formes de vues métiers (objets sérialisables / D.T.O. / V.O.).
- Lorsque JPA/Hibernate est intégré dans Spring ou les EJB, les transactions sont automatiquement gérées par le conteneur et le code est alors significativement simplifié.

3.7. Principales méthodes JPA / EntityManager et Query

Principales méthodes de l'objet EntityManager:

méthodes	traitements
.find(class,pk)	Recherche en base et retourne un objet ayant les classe java et clef primaire indiquées
.createQuery(req_jpql)	Créer une requête JPQL et retourne cet objet
.refresh(obj)	Met à jour les valeurs d'un objet en mémoire en fonction de celles actuellement présentes dans la base de données (reload) .
.persist(obj_détaché) ou bien assez inutilement .persist(obj_déjà_persistant)	Rend persistant un objet détaché : Sauvegarde les valeurs d'un nouvel objet dans la base de données (<i>insert into ...</i>) et retourne une exception de type <i>EntityExistsException</i> si l'entité existe déjà . La clef primaire est quelquefois automatiquement calculée lors de cette opération
.merge(obj_détaché) ou bien assez inutilement .merge(obj_déjà_dans_contexte_persistence)	Sauvegarde ou bien met à jour (<i>update ...</i>) les valeurs d'un objet dans la base de données. Cette méthode s'appelle <i>.merge()</i> car l'entité passé en argument peut quelquefois être utilisée pour remplacer les valeurs d'une ancienne version (avec la même clef primaire) déjà présente dans le contexte de persistance.
.remove(obj)	Supprime l'objet (delete ... from where dans la base de données)
.flush() déclenché indirectement via .commit()	Synchronise l'état de la base de données à partir des nouvelles valeurs des entités persistantes qui ont été modifiées en mémoire.

manager.**persist()** de **JPA** correspond à peu près au session.**save()** de **Hibernate**

manager.**find()** de **JPA** correspond à peu près au session.**get()** de **Hibernate**

manager.**remove()** de **JPA** correspond à peu près au session.**delete()** de **Hibernate**

manager.**merge()** de **JPA** correspond à peu près au session.**save_or_update()** de **Hibernate**

NB:

entityManager.getTransaction().commit() déclenche automatiquement *entityManager.flush()*

.clear() vide le contexte de persistance [tout passe à l'état détaché] sans effectuer de *.flush()*

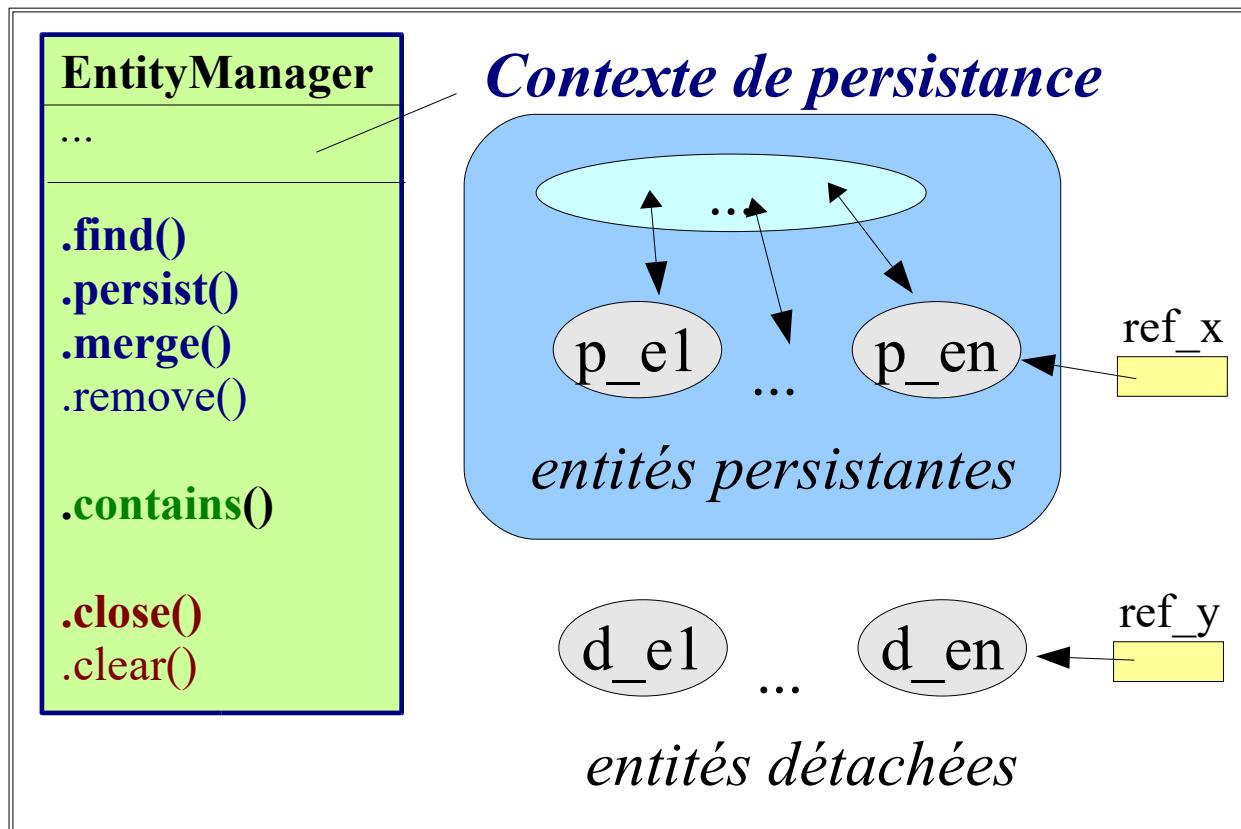
Principales méthodes de l'objet Query:

méthodes	traitements
.setParameter(paramName,paramValue)	met à jour un paramètre de la requête jpql
.getSingleResult()	récupère l'unique objet d'une requête simple.
.getResultList()	retourne sous forme d'objet "java.util.List" la liste des objets retournés en résultat de la requête JPQL.
.executeUpdate()	exécute une requête autre qu'un select
.setMaxResults()	fixe le nombre maxi d'éléments à récupérer

Exemple :

```
return
entityManager.createQuery("select c from Client as c where c.age >=:ageMini", Client.class)
    .setParameter("ageMini",18)
    .getResultList();
```

3.8. Contexte de persistance et proxy-ing



3.9. Relations (1-1, n-1, 1-n et n-n)

Annotations JPA	Significations
@OneToOne	1-1 (fk [unique] ---> pk) [<i><many-to-one unique='true'></i> de hibernate]
@OneToMany	1-n (collection avec clef [pk <----fk=clef])
@ManyToOne	n-1 (fk ----> pk)
@ManyToMany	n-n (fk1 ----> (k1,k2) ---> pk2)

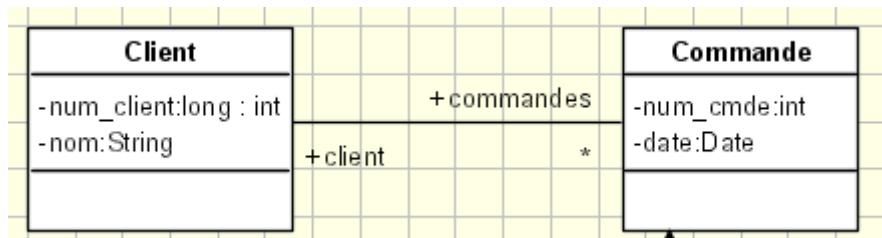
NB: les annotations **@OneToOne** , **@OneToMany** et **@ManyToMany** peuvent éventuellement comporter un attribut "*mappedBy*" dont la valeur correspond à la *propriété qui sert à établir une correspondance inverse (et secondaire)* au sein d'une **relation bidirectionnelle** .

NB: les mises à jour des relations effectuées uniquement du côté secondaire d'une relation bidirectionnelle ne seront pas automatiquement sauvegardées (tant que le côté principal de la relation n'aura pas été explicitement réajusté).

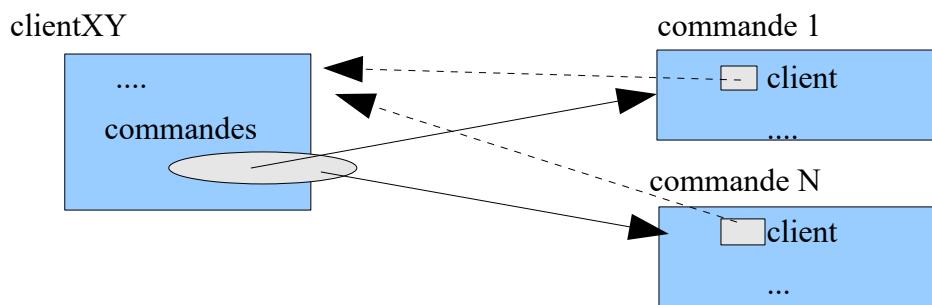
L'annotation **@JoinColumn** permet d'indiquer (*via name="..."*) le **nom de la colonne correspondant à la clef étrangère** dans la base de données.

3.9.a. Relations 1-n et n-1 (entité-entités)

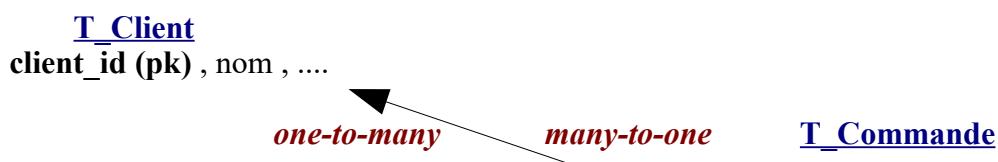
UML:



Java:



Base de données:



num_cmde (pk) , **id_client (fk)**, date,

Annotations JPA:

```
@Entity
@Table(name = "T_Commande")
public class Commande
{
    @ManyToOne
    @JoinColumn(name = "id_client") //fk (nom de colonne dans la table)
    private Client client;

    ...
    public Client getClient() { return client; }
    public void setClient(Client client) { this.client = client; }
}
```

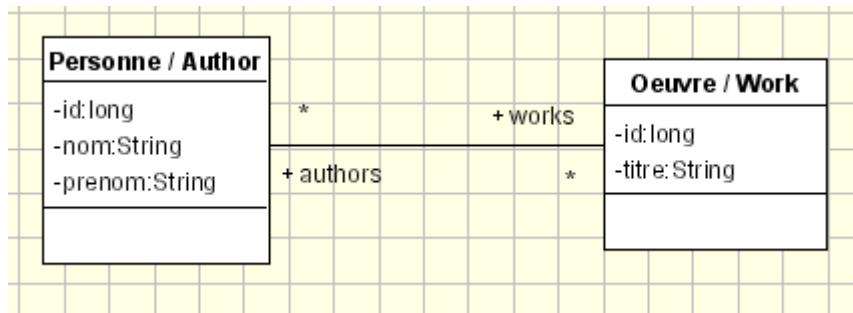
```
@Entity
@Table(name = "T_Client")
public class Client
{
    //NB : "client" (valeur de mappedBy) = nom (java) de la relation inverse
    @OneToMany(fetch=FetchType.LAZY,mappedBy="client")
    private List<Commande> commandes;

    ...
    public List<Commande> getCommandes() { return commandes; }
    public void setCommandes(List<Commande> cmdes) { this.commandes = cmdes; }

    public void addCommande(Commande c) {
        if (commandes == null) commandes = new ArrayList<Commande>();
        commandes.add(c);
    }
}
```

3.9.b. Relation n-n ordinaire

UML:



Mapping relationnel ==> Table intermédiaire

T_Author
auth_id (pk) , nom , prenom ,

Author_Work
author_id (fk) , **work_id(fk)**
many-to-many

T**Work**
w_id (pk), titre, ...

Java:

- > des collections dans les 2 sens
- avec un sens principal et un sens secondaire (avec mappedBy).

JPA:

du coté "principal" de la relation (coté où les mises à jour seront "persistées"):

```

@Entity
public class Work
{
    @ManyToMany( fetch = FetchType.LAZY)
    @JoinTable(name = "Author_Work",
               joinColumns = {@JoinColumn(name = "work_id")},
               inverseJoinColumns = {@JoinColumn(name = "author_id")})
    private List<Author> authors;

    ...
    public List<Author> getAuthors()  {
        return authors;
    }
    public void setAuthors(List<Author> authors)  {      this. authors = authors;  }
    ..
}

```

NB: l'attribut **joinColumns** (de @JoinTable) correspond à la **clef étrangère pointant vers l'entité courante** et l'attribut **inverseJoinColumns** correspond à la **clef étrangère pointant vers les éléments de l'autre côté de la relation (ceux qui seront rangés dans la collection paramétrée)**.

3.10. JPA et Héritage

Stratégie "une table par hiérarchie de classes"
(avec *colonne discriminante*)

Table "Base_et_dérivés"

<i>id</i>	<i>typeDerivé</i> (discriminateur)	<i>pC.</i>	<i>p1a</i>	<i>p1b</i>	<i>p2</i>	<i>p3</i>
1	D1	xx	aa	bb	Null	Null
2	D2	yy	Null	Null	22	Null
3	D3	zz	Null	Null	Null	33

Colonne discriminante (*typeDérivé*) Null (N/A)

Une seule grande table permet d'héberger les instances de toute une hiérarchie de classe.
Pour distinguer les instances des différentes sous classes , on utilise une propriété discriminante (à telle valeur correspond telle sous classe).

Stratégie "table principale"
+ une table (de compléments) par classe fille"

Table "Base"

<i>id</i>	<i>pC.</i>
1	XX
2	YY
3	ZZ

Table "Compl_D1"

<i>id</i>	<i>p1a</i>	<i>p1b</i>
1	aa	bb

Compl_D2

<i>id</i>	<i>p2</i>
2	22

Compl_D3

<i>id</i>	<i>p3</i>
3	33

"Join inheritance"

3.11. Api "Criteria" de JPA 2

3.11.a. Apports et inconvénients de l'API "Criteria"

L'api "Criteria" (qui est apparue avec la version 2 de JPA) correspond à une syntaxe alternative à JPQL pour exprimer et déclencher des requêtes retournant des entités persistantes "JPA".

Contrairement à JPQL , l'api des criteria ne s'appuie plus sur un dérivé de SQL mais s'appuie sur une série d'**expression de critères de recherche qui sont entièrement construits via des méthodes "java"** .

L'api "Criteria" est plus "orientée objet" et plus "fortement typée" que JPQL.

Le principal avantage réside dans **la détection de certaines erreurs dès la phase de compilation.**

A l'inverse, la plupart des erreurs dans l'expression d'une requête JPQL ne sont en général détectées que lors de l'exécution des tests unitaires.

Du coté négatif , l'**API "Criteria" de JPA n'est syntaxiquement pas très simple** (voir franchement complexe) et il faut un certain temps pour s'y habituer (**le code est moins "lisible" que JPQL au premier abord**) .

3.11.b. Exemple basique (pour la syntaxe et le déclenchement)

```
public List<Devise> getAllDevise() {
    /*return this.entityManager.createQuery(
        "select d from Devise d", Devise.class).getResultList();*/
    
    //construction:
    CriteriaBuilder cb = entityManager.getCriteriaBuilder();
    CriteriaQuery<Devise> criteriaQuery = cb.createQuery(Devise.class);

    //préparation (expression des critères de recherche)
    Root<Devise> deviseRoot = criteriaQuery.from(Devise.class);
    criteriaQuery.select(deviseRoot);

    //déclenchement (assez semblable) à celui d'une requête JPQL :
    return this.entityManager.createQuery(criteriaQuery).getResultList();
}
```

Attention : ça se complique très vite avec jointures , prédictats , modèles ,

3.12. Approches top-down , down-top , ...

Approches	Principes
top-down	On part d'une structure orientée objet (éventuellement modélisée via UML)

	considérée comme de haut niveau et l'on génère automatiquement des tables avec une structure compatible
down-top	On part d'une structure relationnelle existante (éventuellement modélisée via un MCD Merise) considérée comme de bas niveau et l'on génère automatiquement des classes d'entités persistantes avec une structure compatible
meet-in-the-middle	On se débrouille pour mapper au cas par cas une structure objet et une structure relationnelle (existantes toutes les 2).

3.12.a. Configuration JPA 2.2 pour approche top-down

```
# si l'option spring.jpa.hibernate.ddl-auto=create est activée, toutes les
# tables nécessaires seront re-crées automatiquement à chaque démarrage
# à vide et en fonction de la structure des classes java (@Entity)
spring.jpa.hibernate.ddl-auto=create
```

```
#tables créés automatiquement au démarrage et fichier sql déclenché automatiquement :
...javax.persistence.schema-generation.database.action=drop-and-create
...javax.persistence.sql-load-script-source=META-INF/data.sql
```

```
#fichiers sql générés (pour consultation) mais pas déclenchés:
...javax.persistence.schema-generation.create-source=metadata
...javax.persistence.schema-generation.scripts.action=drop-and-create
...javax.persistence.schema-generation.scripts.create-target=src/main/script/create.sql
...javax.persistence.schema-generation.scripts.drop-target=src/main/script/drop.sql
```

3.12.b. Outils pour approche down-top

- assistants eclipse "**jpa-tools**"

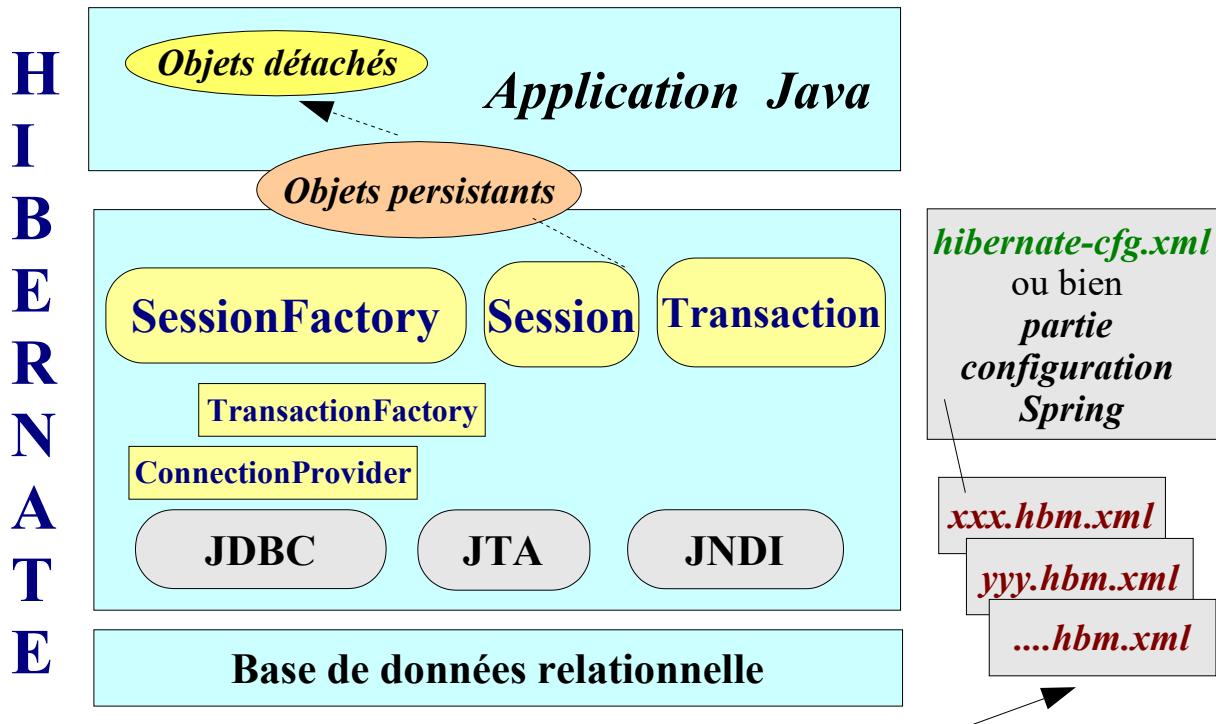
Mode opératoire à un peu adapter selon la version exacte d'eclipse (3.4 , 3.5 ou 3.6)

- 1) Paramétrier une connexion vers une base de données existante au niveau d'éclipse
(menu "Windows / **Show View** / Other ... / connectivity / **DataSourceExplorer** / New .).
Paramétrier à ce niveau l'accès à la base de données (chemin du ".jar" du driver JDBC , url , ...).
Tester la connexion SQL via l'explorateur de base de données (menu "Windows / Show View // **DataSourceExplorer** / / connect).
- 2) Créer un nouveau **projet JPA** (ou bien ajouter la **facette "JPA"** à un projet java existant)
- 3) depuis le projet JPA déclencher le menu contextuel "**JPA Tools ...**" / "**Generate Entities**"
choisir la connexion SQL , le schéma de la base de données , les tables à prendre en compte et quelques options (package java , ...)
==> résultats = classes d'entités persistantes avec des annotations JPA (@Entity , @Table , @Id , @OneToOne ,).

4. Hibernate et les ORM

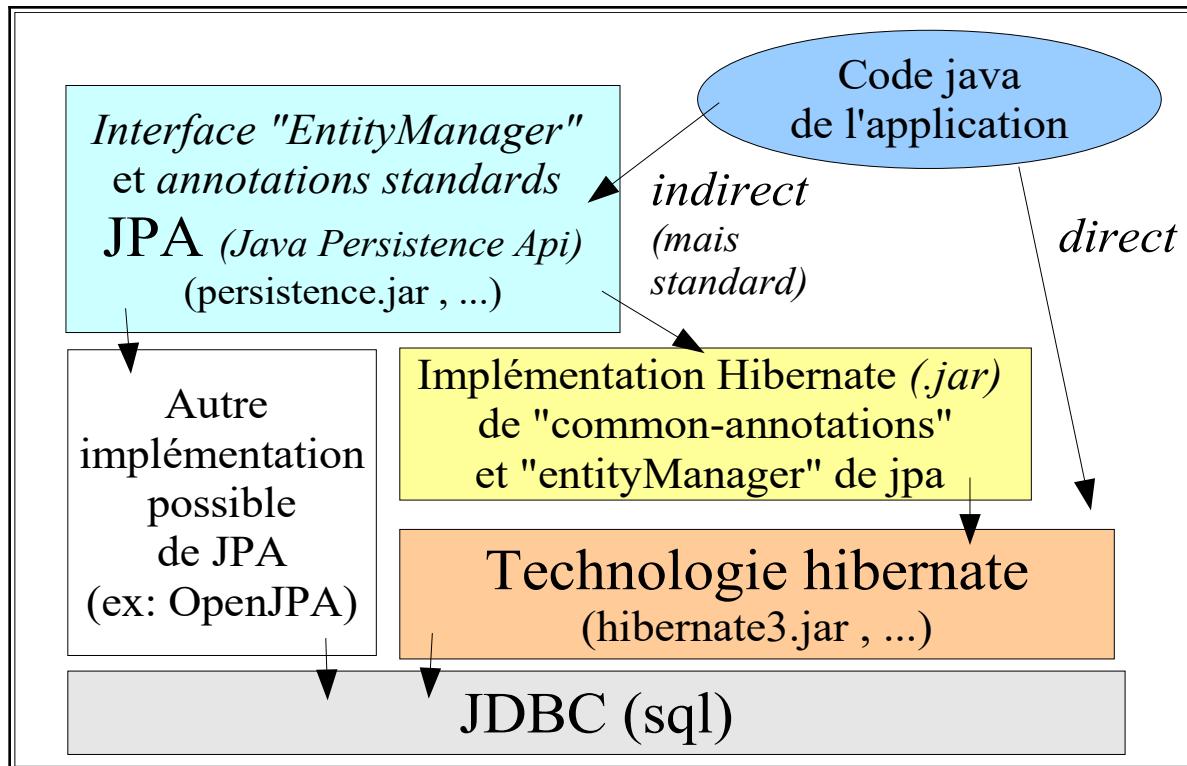
4.1. Hibernate

Hibernate est une technologie java "open source" spécialisée dans l'implémentation du mapping "objet-relationnel".



Configuration d'un "mapping" entre table(s) et objet(s) java

Depuis la version 3.2, la technologie Hibernate peut éventuellement être utilisée en tant que sous couche d'implémentation de l'api officielle "JPA" du monde java .



5. Détails d'une couche de persistance

5.1. D.A.O. (Data Access Object)

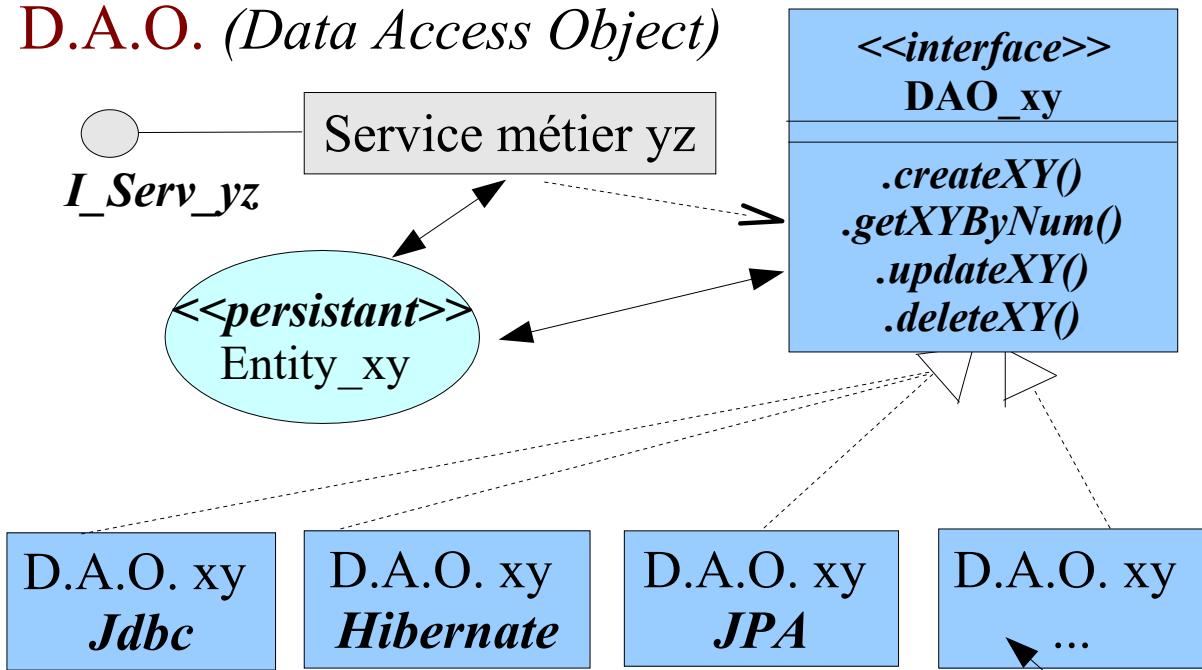
5.1.a. Principe

La couche de persistance est souvent en bonne partie matérialisée par des objets spécialisés dans l'accès aux données (les D.A.O. = Data Access Object).

Un composant de type **DAO** est généralement structuré en :

- une partie "**interface abstraite**" répertoriant tout un tas de méthodes CRUD (Create, Retreive, Update, Delete)
- une "**implémentation concrète**" (idéalement interchangeable) basée sur une technologie donnée (ex: JDBC, Hibernate, JPA,).

D.A.O. (*Data Access Object*)



Éventuel fichiers xml pour stocker des petits volumes de données (avec différentes technologies xml : dom , jaxb2 ,)

5.1.b. DAO génériques ou DAO Spécifiques

Selon les types d'entités manipulées par un DAO, on peut ranger les DAO en deux grandes classes:

- les **DAO génériques** (fonctionnant sur n'importe quel type d'entité ["Object" en java]).
- les **DAO spécifiques** (spécialisés dans la persistance d'un type précis d'entité --> exemple: CompteDAO avec méthodes "getCompteByNum()", deleteCompte(),)

NB: Les objets "session" d'hibernate et l'objet "entity manager" de JPA peuvent être considérés comme des DAO génériques .

6. Bases SQL et NoSQL

6.1. Rappel (ACID)

A.C.I.D. ==> Atomicity , Consistency , Isolation , Durability

- L' **Atomicité** désigne le comportement "**tout ou rien**" (Le tout vu en tant qu'élément unique et atomique doit soit réussir , soit échouer). Il n'y a pas de demi-mesure.
- La **Consistance** d'une transaction désigne le fait que **les différentes opérations doivent laisser le système dans un état stable et cohérent**.
- Le concept d' **Isolation** signifie ici que **2 transactions concurrentes n'interfèrent pas**

- entre elles (Points critiques: résultats intermédiaires et opérations annulées).
- La Durabilité indique que les résultats d'une transaction doivent absolument être mémorisés de façon durable (sur un support physique) de façon à survivre suite à une éventuelle défaillance (un fichier de Log peut également être très utile).

6.2. Bases de données SQL (relationnelles) / SGBDR

- Forte consistance (intégrité garantie)
- Transactionnelles (ACID)
- Très matures / actuellement traditionnelles (adoptions faciles/habituelles)

mais pas très scalables en écriture du fait des contraintes "ACID" : détérioration des performances lorsque beaucoup de charge (utilisateurs simultanés) et de volume (big data).

6.3. Bases NoSQL (Not only SQL)

NoSQL désigne une catégorie de SGBD qui n'est plus fondée sur l'architecture classique des bases relationnelles. L'unité logique n'y est plus la table et les données ne sont en général pas manipulées avec SQL.

Origine/Historique :

À l'origine, servant à manipuler des bases de données géantes pour des sites web de très grande audience tels que [Google](#), [Amazon.com](#), [Facebook](#) ou [eBay1](#), le NoSQL s'est aussi étendu par le bas après 2010. Il renonce aux fonctionnalités classiques des [SGBD relationnels](#) au profit de la simplicité. Les performances restent bonnes avec la montée en charge ([scalabilité](#)) en multipliant simplement le nombre de serveurs, solution raisonnable avec la baisse des coûts, en particulier si les revenus croissent en même temps que l'activité.

Les systèmes géants sont les premiers concernés : énorme quantité de données, structuration relationnelle faible (ou de moindre importance que la capacité d'accès très rapide, quitte à multiplier les serveurs).

Un modèle typique en NoSQL est le système clé-valeur, avec une base de données pouvant se résumer topologiquement à un simple [tableau associatif](#) unidimensionnel avec des millions — voire des milliards — d'entrées. Parmi les applications typiques, on retrouve des analyses temps-réel, statistiques, du stockage de logs (journaux), etc.

En 2009 à San Francisco des développeurs de logiciels NoSQL qui, selon le magazine [Computerworld](#), « racontent comment ils ont renversé la tyrannie des coûteux et lents SGBD relationnels par des moyens plus simples et plus rapides de manipuler des données ». Selon Jon Travis, un des présentateurs de la conférence, « les SGBD relationnels offrent trop, alors que les produits NoSQL offrent exactement ce dont vous avez besoin ».

Beaucoup de produits NoSQL peuvent manipuler de très grandes quantités de données (centaines de [téraoctets](#)) et offrent une [évolutivité à la charge](#). Ces produits NoSQL ne sont pas toujours décrits comme des SGBD, mais quelquefois comme des logiciels de *stockage* de données.

Théorie :

Un système de gestion de base de données (SGBD) permet de réaliser des transactions atomiques, cohérentes, isolées, et durables (ACID).

Les capacités ACID garantissent que si plusieurs utilisateurs font de manière simultanée des modifications des données, toutes les modifications vont être prises en compte, dans un ordre précis et maîtrisé de manière à avoir un résultat cohérent (intégrité des données) avec l'historique des modifications faites par chacun. La mise en œuvre stricte des capacités ACID entraîne des coûts logiciels importants et un niveau de performance moindre à infrastructure matérielle équivalente.

Les SGBD d'annuaires ont servi de modèle en permettant de lever certaines de ces contraintes en fonction de l'usage, en particulier dans les cas où la grande majorité des accès aux bases de données consistent en lectures sans modification (dans ce cas seule la propriété de persistance importe).

Pour faire face à des volumes importants de données, accédés de différents endroits du monde, il faut pouvoir répliquer ces données sur différentes machines physiques, c'est ce que l'on appelle un environnement distribué. Le théorème CAP démontre qu'il n'est pas possible d'assurer des transactions totalement ACID dans un environnement distribué.

Les bases de données NoSQL répondent aussi au théorème du CAP d'Eric Brewer qui est plus adapté aux systèmes distribués. Ce théorème énonce que tout système distribué peut répondre aux contraintes suivantes:

- **Cohérence** : tous les noeuds du système voient exactement les mêmes données au même moment
- **Haute disponibilité (Availability)** : en cas de panne, les données restent accessibles
- **Tolérance au Partitionnement** : le système peut être partitionné

Mais le théorème du CAP précise aussi que seulement deux de ces trois contraintes peuvent être respectées en même temps.

D'autres caractéristiques communes aux différentes bases de données NoSQL peuvent être citées tel que le partitionnement horizontal sur plusieurs noeuds, la réPLICATION des données, les schémas dynamiques ou l'absence de schéma.

Le protocole PAXOS est très efficace pour la lecture dans un environnement distribué, beaucoup moins pour l'écriture / modification et il ne supporte pas les transactions ACID.

Les solutions du marché implémentent ce protocole en ajoutant leurs techniques propres pour limiter les conséquences de l'impossibilité d'ACID lors des écritures et mises à jour de données.

Problèmes de l'atomicité :

- une opération atomique ne permet pas des accès entrelacés
- une partie de la base de données est verrouillée lors de l'écriture

==> En se passant de l'atomicité, on a plus besoin de verrouillage .

Problèmes de la consistance :

- toutes les transactions doivent être totalement commitées ou annulées
- tous les membres d'un cluster doivent avoir les mêmes données (synchronisations immédiates sur tous les autres noeuds)

- respect des schémas

==> En se passant de la consistance immédiate, on a plus besoin d'avoir exactement et immédiatement les mêmes données sur tous les serveurs, on synchronise lorsque l'on peut (avec un léger différé).

Problèmes de la durabilité:

- avant de répondre au client , on doit être sûr que les données ont été écrites sur le disque.

==> En se passant de la durabilité , on n'a plus besoin d'attendre la confirmation de l'écriture.

NB : chaque technologie NoSQL gère à sa façon les aspects précédents (atomicité , consistance , durabilité) . On ne se passe évidemment pas de tout ça .

C'est essentiellement l'aspect "synchronisation non immédiate" qui est mis en avant .

6.4. Types de bases de données NoSQL :

Les solutions NoSQL existantes peuvent être regroupées en 4 grandes familles.

- **Clé / valeur** : Ce modèle peut être assimilé à une hashmap distribuée. Les données sont, donc, simplement représentées par un couple clé/valeur. La valeur peut être une simple chaîne de caractères, un objet sérialisé... Cette absence de structure ou de typage ont un impact important sur le requêtage. En effet, toute l'intelligence portée auparavant par les requêtes SQL devra être portée par l'applicatif qui interroge la BD. Néanmoins, la communication avec la BD se résumera aux opérations PUT, GET et DELETE. Les solutions les plus connues sont Redis, Riak et Voldemort créé par LinkedIn.
- **Orienté colonne** : Ce modèle ressemble à première vue à une table dans un SGBDR à la différence qu'avec une BD NoSQL orientée colonne, le nombre de colonnes est dynamique. En effet, dans une table relationnelle, le nombre de colonnes est fixé dès la création du schéma de la table et ce nombre reste le même pour tous les enregistrements dans cette table. Par contre, avec ce modèle, le nombre de colonnes peut varier d'un enregistrement à un autre ce qui évite de retrouver des colonnes ayant des valeurs NULL. Comme solutions, on retrouve principalement HBase (implémentation Open Source du modèle BigTable publié par Google) ainsi que Cassandra (projet Apache qui respecte l'architecture distribuée de Dynamo d'Amazon et le modèle BigTable de Google).
- **Orienté document** : Ce modèle se base sur le paradigme clé valeur. La valeur, dans ce cas, est un document de type JSON ou XML. L'avantage est de pouvoir récupérer, via une seule clé, un ensemble d'informations structurées de manière hiérarchique. La même opération dans le monde relationnel impliquerait plusieurs jointures. Pour ce modèle, les implémentations les plus populaires sont CouchDB d'Apache, RavenDB (destiné aux plateformes .NET/Windows avec la possibilité d'interrogation via LINQ) et MongoDB.
- **Orienté graphe** : Ce modèle de représentation des données se base sur la théorie des graphes. Il s'appuie sur la notion de noeuds, de relations et de propriétés/attributs qui leur sont rattachées. Ce modèle facilite la représentation du monde réel, ce qui le rend adapté au traitement des données des réseaux sociaux. La principale solution est Neo4J.

Les deux mouvements “orienté colonne” et “orienté document” découlent bien du système clé valeur et c'est la nature ou la structure de la valeur qui diffère.

Le marché :

Dans le marché des SGBD *NoSQL* se trouvent [Cassandra](#), [MongoDB](#), [Voldemort](#), [CouchDB](#) et [SimpleDB](#). Selon Oracle, le « battage » autour de ces produits vient du fait qu'ils sont impliqués dans de grands sites web tels que [Facebook](#), [LinkedIn](#) ou [Amazon.com](#). C'est un marché jeune, encore sans réel leader (en 2011). Le marché évolue rapidement et les comparatifs de produits y sont rapidement dépassés. Lors d'un sondage réalisé en 2010 auprès de professionnels de l'informatique, 44 % des sondés répondaient encore qu'ils n'ont jamais entendu parler de NoSQL.

6.5. SQL vs NoSQL

Les SGBD relationnels sont largement répandus dans les entreprises. Dimensionnés pour une quantité d'informations et un nombre d'utilisateurs typiques d'une entreprise, ils ont pour fonction principale le [traitement de transactions](#).

Ils montrent cependant leurs limites lorsqu'ils sont utilisés dans un périmètre plus large, tel qu'un site web populaire, en répartition de charge(load balancing), fréquenté par des millions de visiteurs dans le monde entier : les SGBD relationnels exigeraient alors des logiciels et des ordinateurs coûteux ainsi que des compétences en optimisation peu répandues.

Ce segment de marché est de ce fait occupé par les logiciels *NoSQL*, conçus spécifiquement pour un usage de type [Internet](#). Ces produits abandonnent la représentation matricielle de l'information et le langage de commande SQL en échange d'une simplicité, d'une performance et surtout d'une [scalabilité](#) accrues. La complexité de mise en œuvre du traitement des transactions a été réduite dans le but d'obtenir des services plus simples et plus spécialisés.

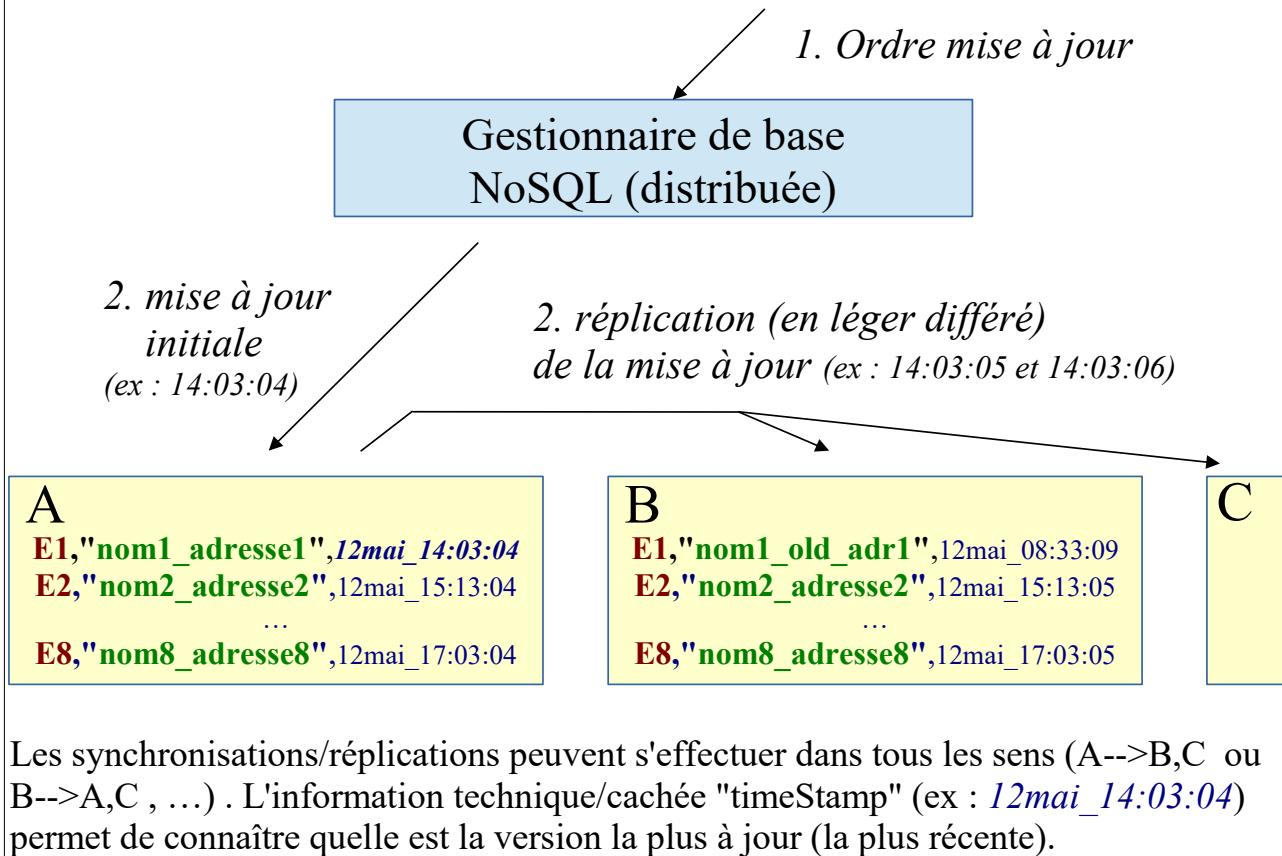
Les deux domaines, bases relationnelles et NoSQL, répondant à des besoins et des contraintes différentes, coexistent souvent dans les architectures logicielles de certains domaines métiers.

Choix selon questions suivantes:

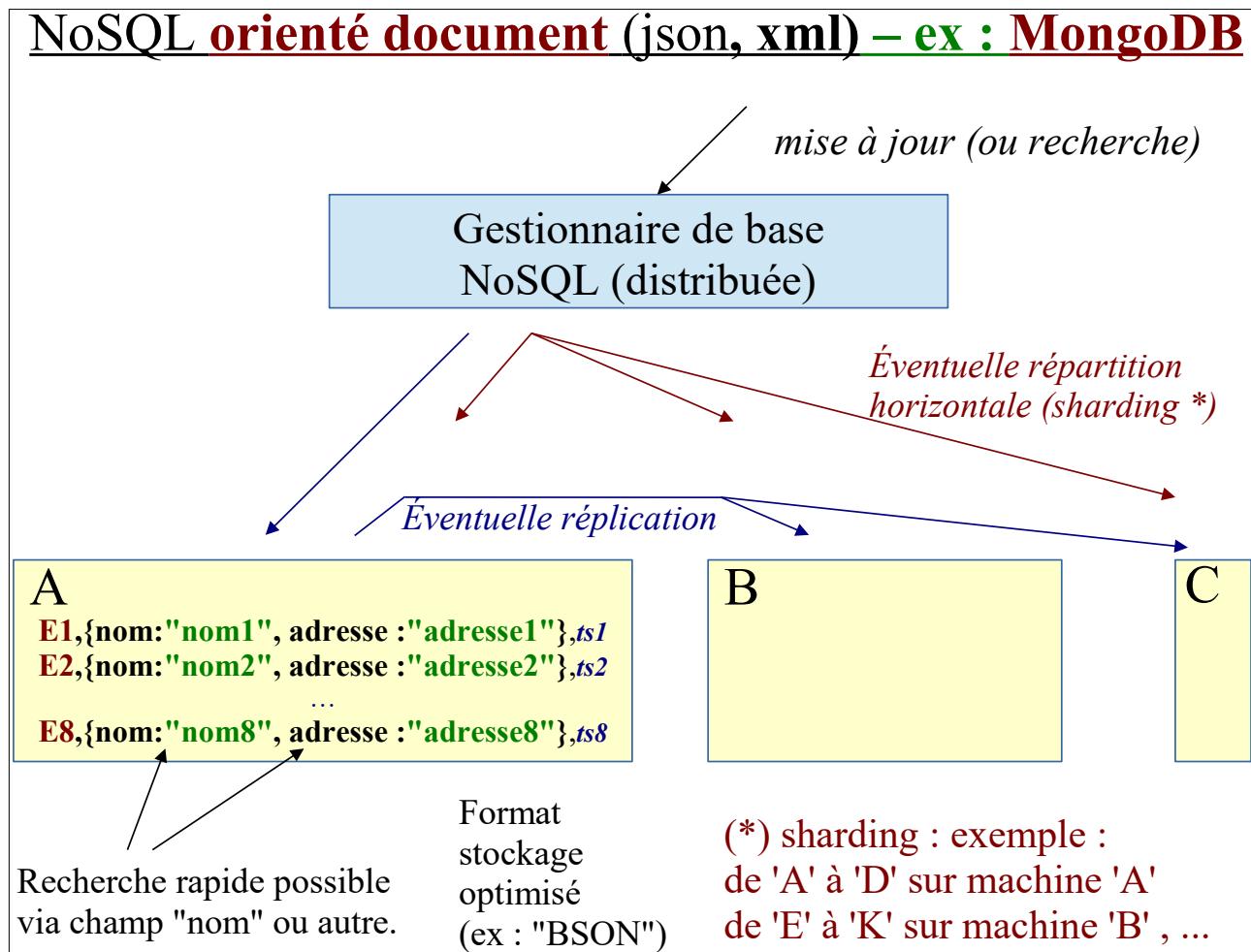
- Comment veut-on structurer nos données ?
- Combien de données à manipuler ?
- Ratio "lecture/écriture" ?

6.6. Basic NoSQL (clef,valeurs)

Basic NoSQL (**key**, **value**, *timeStamp*)

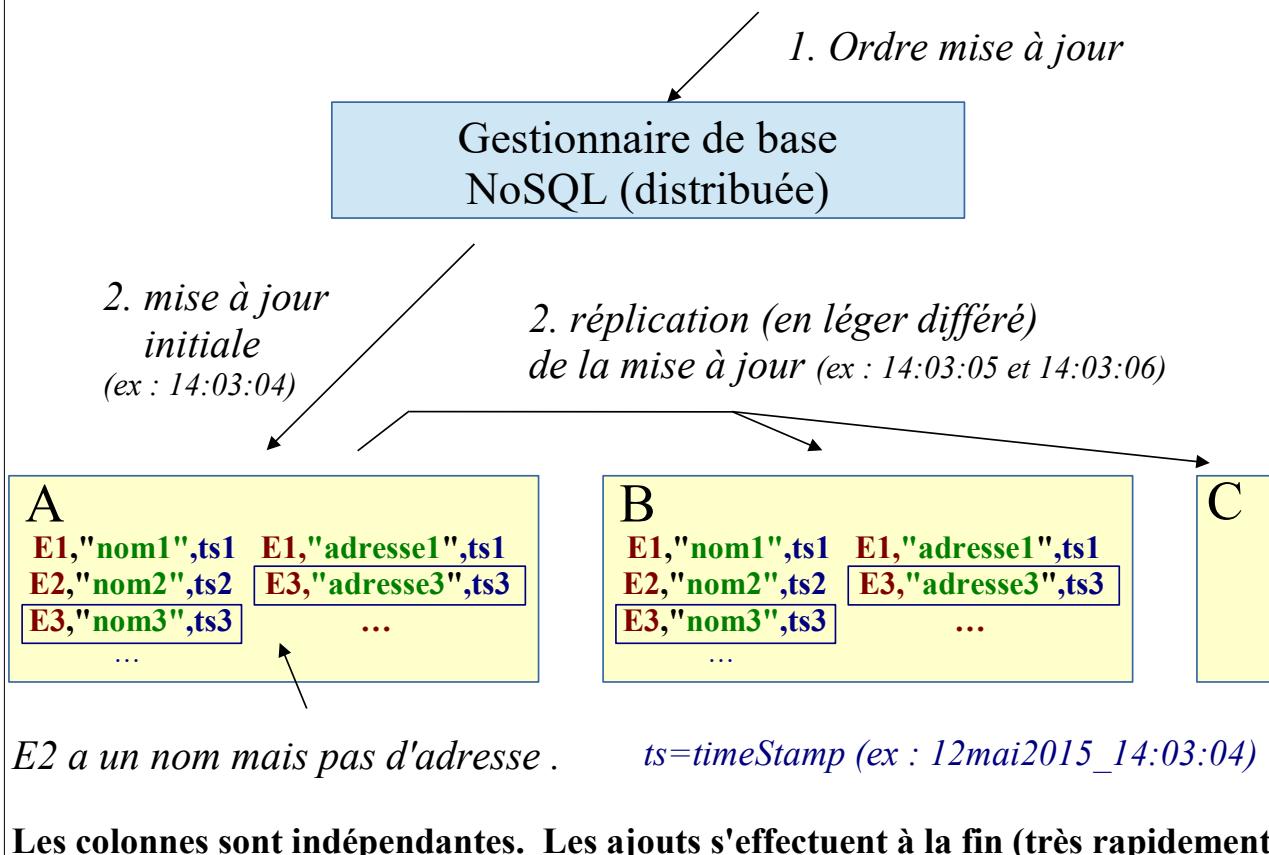


6.7. Orienté document



6.8. Orienté colonnes

NoSQL orienté colonnes (ex : "cassandra")



VI - Communication

1. SOAP Web services avec JAX-WS

2 grands types de services WEB: **SOAP/XML** et **REST/HTTP**

WS-* (SOAP / XML)

- "Payload" systématiquement en **XML** (*sauf pièces attachées / HTTP*)
- Enveloppe **SOAP** en XML (*header facultatif pour extensions*)
- **Protocole de transport au choix (HTTP, JMS, ...)**
- Sémantique quelconque (*appels méthodes*), **description WSDL**
- Plutôt orienté Middleware SOA (**arrière plan**)

REST (HTTP)

- "Payload" au choix (**XML , HTML , JSON, ...**)
- Pas d'enveloppe imposée
- **Protocole de transport = toujours HTTP.**
- Sémantique "**CRUD**" (*modes http PUT,GET,POST,DELETE*)
- Plutôt orienté IHM Web/Web2 (**avant plan**)

Points clefs des Web services "SOAP"

Le **format "xml rigoureux"** des requêtes/réponses (définis par ".xsd" , ".wsdl") permet de retraiter sans aucune ambiguïté les messages "soap" au niveau certains services intermédiaires (dans ESB ou ...). Certains automatismes génériques sont applicables .

Fortement typés (xsd:string , xsd:double) les web-services "soap" conviennent très bien à des appels et implémentations au sein de langages fortement typés (ex : "c++" , "c#" , "java" , "...").

A l'inverse , des langages faiblement typés tels que "php" ou "js" sont moins appropriés pour soap (appels cependant faisables)

La **relative complexité et la verbosité "xml" des messages "soap"** engendrent des appels moins immédiats (*mode http "post" avec enveloppe à préparer*) et des **performances moyennes**.

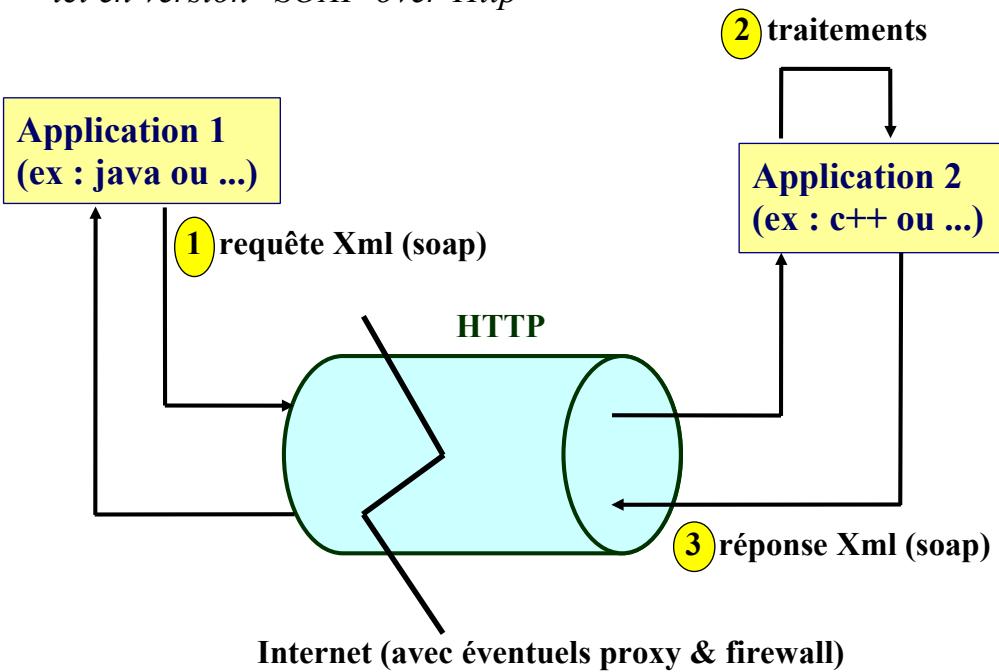
Soap peut être utilisé en mode "envoi de document" mais c'est rare.

Les messages "soap" peuvent être véhiculés par "jms" mais c'est rare.

1.1. Protocole SOAP

SOAP (Simple Object Access Protocol)

ici en version "SOAP-over-Http"



Enveloppe SOAP (ici véhiculée via HTTP)

message http (requête / réponse)

```
POST /SoapEndUrl HTTP/1.1
Host: www.yyy.com
Content-type: text/xml
Content-Length: 524
SOAPAction:
```

```
<SOAP-ENV:Envelope
... xmlns:SOAP-ENV=
'http://schemas.xml.org/soap/envelope/'>
  <SOAP-ENV:Header>
  ...
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
  ...
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Entête Http

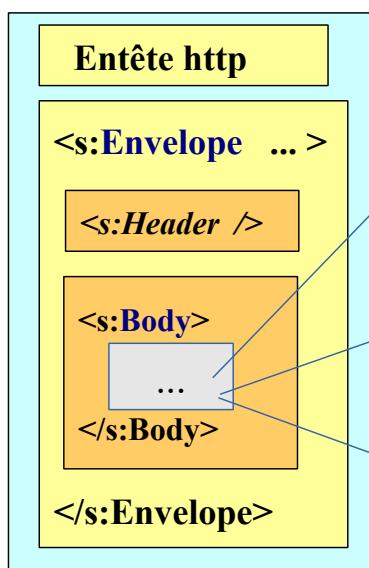
Enveloppe SOAP
(Corps du message Http au format **text/xml**)

Soap Header
facultatif

Soap Body

Contenu du corps de l'enveloppe "SOAP"

message http
(avec contenu SOAP)



requête
<methodeXy>
<p1>val1</p1>
<p2>val2</p2>
</methodeXy>

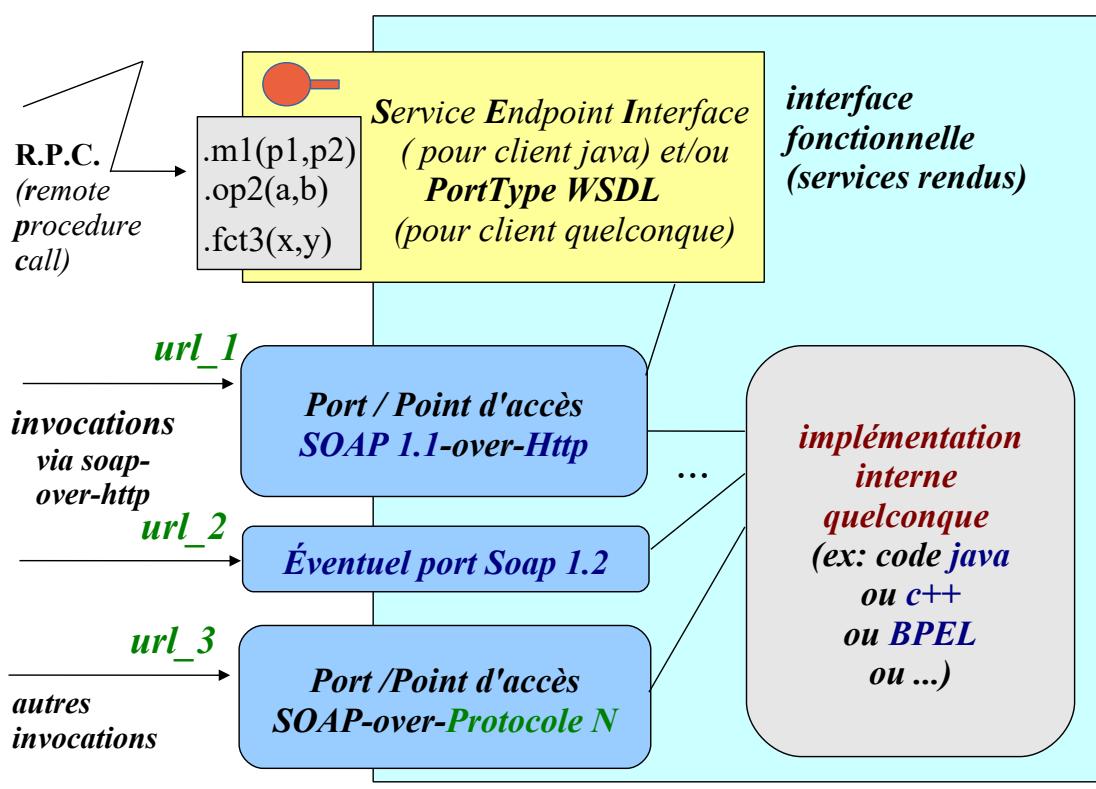
réponse

<methodeXyResponse>
<return>val_resultat</return>
</methodeXyResponse>

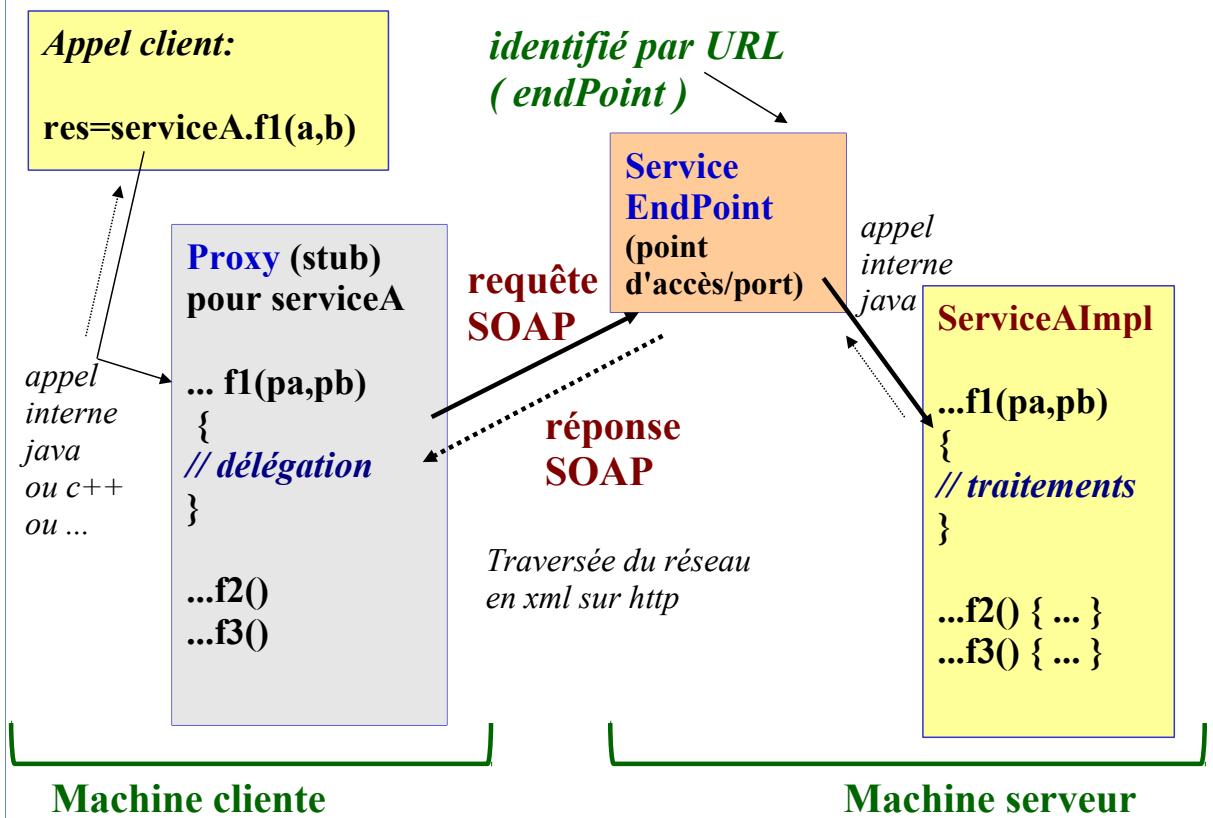
Exception (fault)

<s:fault>
<faultcode>soap:Server</faultcode>
<faultstring>msg_error</faultstring>
</s:fault>

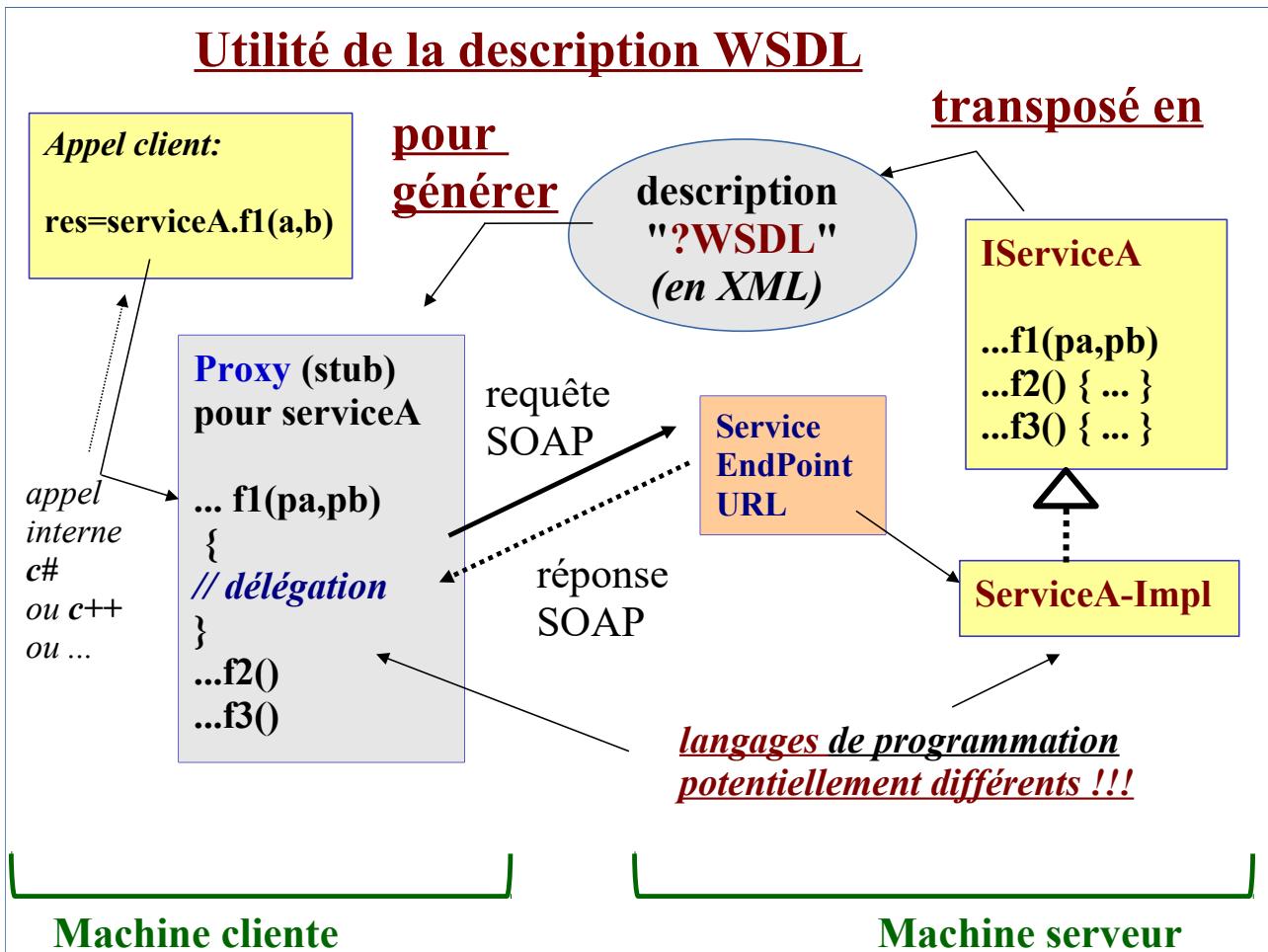
Anatomie d'un Service Web "SOAP"



Proxy et Point d'accès / Port avec SOAP



1.2. WSDL (Web Service Description Language)



Exemple :

Code coté serveur du *service web* en C++ (*gSoap*)

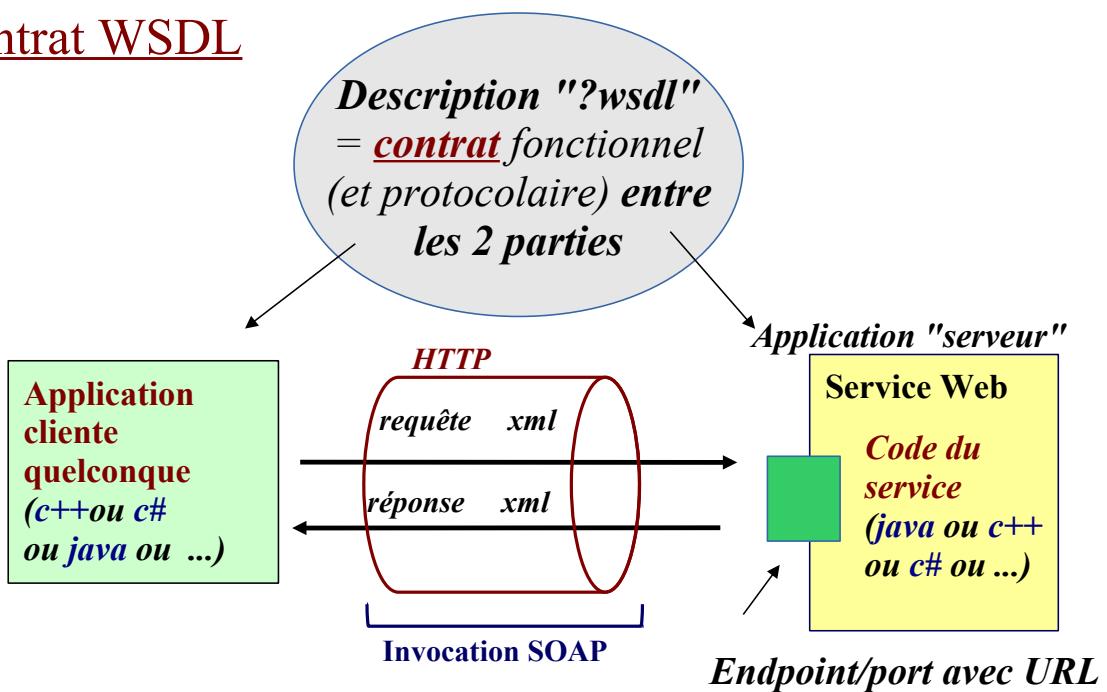
- génération du fichier WSDL via soapcpp2

- génération via **WSDL2Java** (ou **wsimport**)

Description WSDL (pour WS Soap)

- **WSDL = Web Service Description Language**
- C'est une **description XML de la structure d'un service** soap accessible qui est volontairement **indépendante des langages de programmation** (*c++ ou java ou c# ou ...*).
- *L'essentiel d'une description WSDL peut être vu comme une transposition xml d'une interface java (liste d'opérations avec des types précis de paramètres en entrée et en retour).*
- *Un Fichier WSDL est souvent généré automatiquement côté serveur , publié au bout d'une URL , téléchargé , puis analysé par un programme du genre wsdl2Java ou wsimport pour générer un "proxy" côté client*

Contrat WSDL



Par convention, si l'url d'invocation soap d'un service publié est <http://hostName/xxx/yyy>

alors l'URL menant à la description WSDL est

<http://hostName/xxx/yyy?wsdl> ← ?wsdl en plus

1.3. Api java JAX-WS

Présentation de l'api JAX-WS

JAX-WS signifie Java Api for Xml Web Services

JAX-WS est déjà intégré dans le jdk >=1.6

Principales caractéristiques :

- paramétrage par annotations (ex: **@WebService , @WebParam**)
- utilisation interne de **JAXB2** (et ses annotations **@Xml...**)

JAX-WS est dédié au services "SOAP" avec description WSDL.

Une partie de JAX-WS est dédiée à WS-S (WS-Security).

La technologie "CXF" complémente le jdk sur certains points (sécurité , intercepteurs , intégration webApp , intégration spring, ...)

1.3.a. Mise en oeuvre de JAX-WS (coté serveur)

Paramétrage d'une interface de service web

```
//@WebService(targetNamespace="http://services.myapp.xy.com/")
@WebService
public interface GestionComptes {
    public Compte getCompteByNum(
        @WebParam(name="numCpt")long numCpt)
        throws MyServiceException;
    public List<Stat> getStats(
        @WebParam(name="annee") int annee);
    public void transferer(
        @WebParam(name="montant")double montant,
        @WebParam(name="numCptDeb")long numCptDeb,
        @WebParam(name="numCptCred")long numCptCred)
        throws MyServiceException;
}
```

Paramétrage d'une classe d'implémentation

```
/*@WebService(targetNamespace="http://impl.services.myapp.xy.com/",
endpointInterface="com.xy.myapp.services.impl.GestionComptes")*/
@WebService(endpointInterface=
    "com.xy.myapp.services.impl.GestionComptes")
public class GestionComptesImpl implements GestionComptes {
... // code d'implémentation java habituel (R.A.S.)
}
```

Paramétrage d'une classe de données (en entrée ou sortie)

```
@XmlType(namespace="http://data.myapp.xy.com/")
@XmlRootElement(name="stat")
public class Stat {
    private int num_mois; //de 1 a 12
    private double ventes; //+get/set
    ... }
```

1.3.b.

Principaux paramétrages JAX-WS:

NB: les noms associés au service (**namespace**, **PortType**, **ServiceName**) sont par défaut basés sur les noms des éléments java (package , nom_interface et/ou classe d'implémentation) et peuvent éventuellement être précisés/redéfinis via certains attributs facultatifs de l'annotation **@WebService** .

Namespace par défaut: "**http://**" + nom_package_java_à_l_envers + "**/**"

NB: de façon à ce que les noms des paramètres des méthodes d'une interface java soient bien retranscrits dans un fichier WSDL, on peut les préciser via **@WebParam(name="paramName")**.

```
public int addition(int a,int b);
    ==> addition(xsd:int arg0,xsd:int arg1) dans WSDL
public int addition(@WebParam(name="a") int a,
                    @WebParam(name="b") int b)
    ==> addition(xsd:int a ,xsd:int b) dans WSDL
```

L'annotation facultative **@WebResult** permet entre autre de spécifier le nom de la valeur de retour (dans SOAP et WSDL) .
Par défaut le nom du résultat correspond a "**return**" .

Implémentation sous forme d'EJB3 (pour serveur JEE5+)

- Combiner les annotations précédentes (@WebService ,) avec l'annotation **@Stateless** au niveau de la classe d'implémentation d'un EJB3 Session sans état.
- Déployer le tout selon les spécifications JEE (.jar , .ear) au sein d'un serveur d'application comportant un conteneur d'EJB3

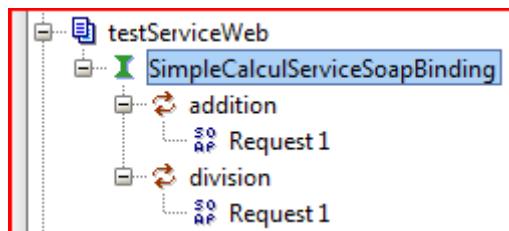
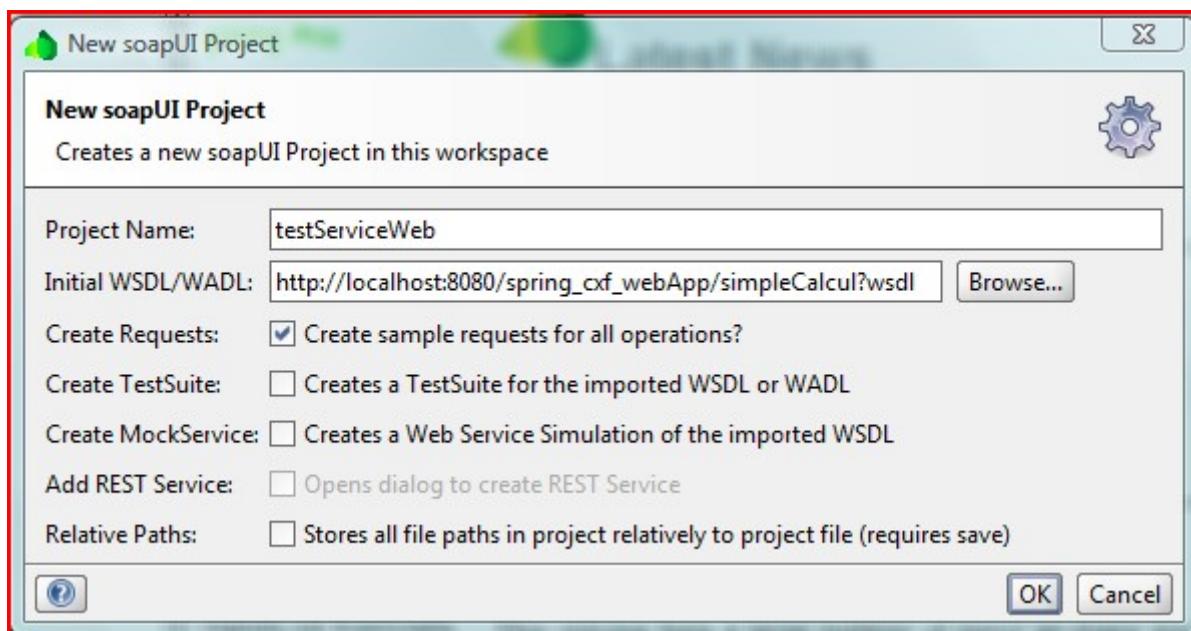
Repérer l'url du service WEB et de la description WSDL (**au cas par cas selon serveur**)
[ex: <http://localhost:8080/xyz/XXXBean?wsdl>]

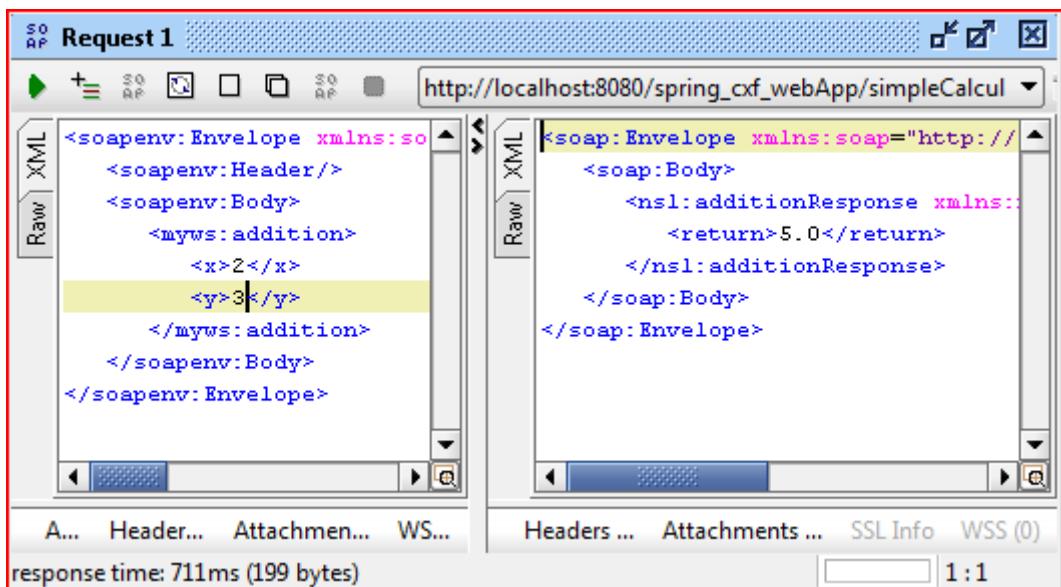
1.4. Tests via SOAPUI

==> pour tester rapidement un service Web , le plus simple est d'utiliser le programme utilitaire "soapui" (dont une version gratuite est disponible) .

Mode opératoire:

- Télécharger et installer soapui .
- Lancer le produit via le script ".bat" du sous répertoire bin.
- Nouveau projet (de test) avec nom à choisir.
- Préciser l'URL (se terminant généralement par "**?wsdl**") de la description du service à invoquer.
- Sélectionner une des méthodes pour la tester en mode requête / réponse.
- Renseigner quelques valeurs au sein d'une requête.
- Déclencher l'invocation de la méthode via SOAP
- Observer la réponse (ou le message d'erreur)





1.5. JAX-Ws côté client

```

private static void testCalculateurServiceWithoutWsImport() {

    QName SERVICE_NAME = new QName("http://myws/", "CalculateurService");
    QName PORT_NAME = new QName("http://myws/", "CalculateurServicePort");

    // en précisant une URL WSDL connue et accessible
    String wdlUrl =
        "http://localhost:8080/spring_cxf_webApp/services/calculateur?wsdl";

    URL wsdlDocumentLocation=null;
    try {wsdlDocumentLocation = new URL(wdlUrl);
    } catch (MalformedURLException e) {      e.printStackTrace();}

    //avec import javax.xml.ws.Service;
    Service service = Service.create(wsdlDocumentLocation, SERVICE_NAME);

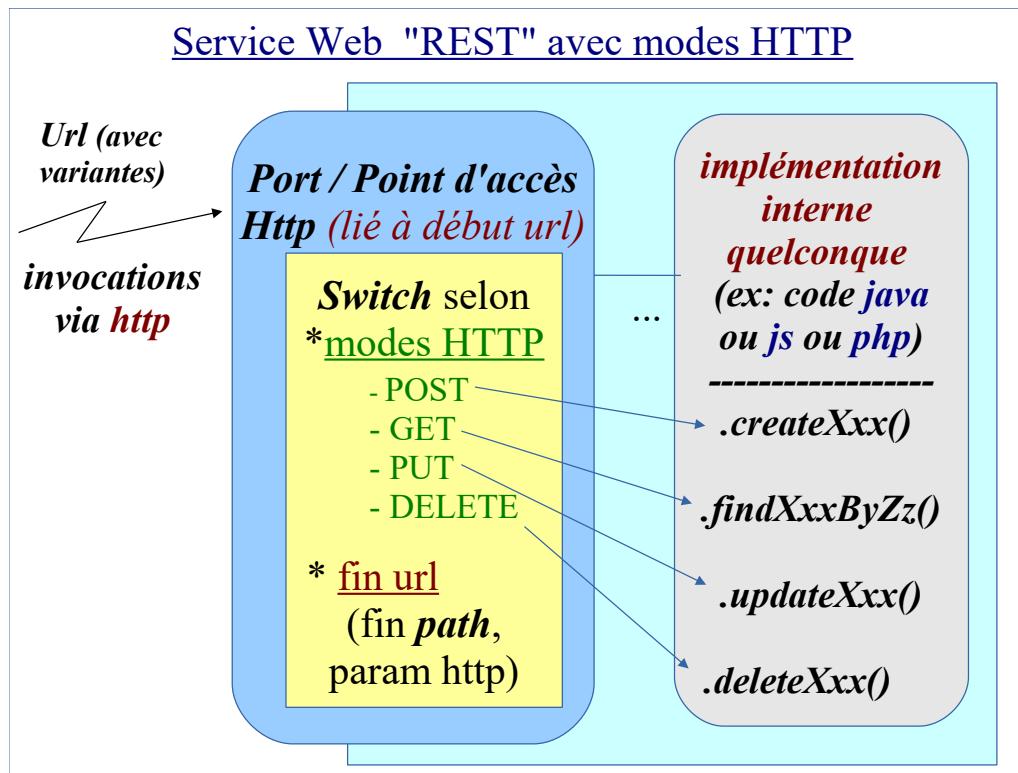
    Calculateur caculateurWSProxy = (Calculateur)
        service.getPort(PORT_NAME, Calculateur.class);

    double tauxTvaReduitPct =
        caculateurWSProxy.getTauxTva("TauxReduit");
    System.out.println("TauxReduit=" + tauxTvaReduitPct);
}

```

2. REST Services avec JAX-RS

2.1. Caractéristiques clefs des web-services "REST" / "HTTP"



Points clefs des Web services "REST"

Retournant des données dans un format quelconque ("XML", "JSON" et éventuellement "txt" ou "html") les web-services "REST" offrent des résultats qui nécessitent généralement peu de re-traitements pour être mis en forme au sein d'une IHM web.

Le format "**au cas par cas**" des données retournées par les services REST permet peu d'automatisme(s) sur les niveaux intermédiaires.

Souvent associés au format "**JSON**" les web-services "REST" conviennent parfaitement à des appels (ou implémentations) au sein du langage javascript .

La **relative simplicité** des URLs d'invocation des services "REST" permet des appels plus immédiats (*un simple href="..." suffit en mode GET pour les recherches de données*) .

La **compacité/simplicité** des messages "JSON" souvent associés à "REST" permet d'obtenir d'assez bonnes performances .

2.2. Fondamentaux sur Web Services "R.E.S.T."

REST = style d'architecture (conventions)

REST est l'acronyme de **R**epresentational **E**State **S**Transfert.

C'est un **style d'architecture** qui a été décrit par *Roy Thomas Fielding* dans sa thèse «*Architectural Styles and the Design of Network-based Software Architectures*».

L'information de base, dans une architecture REST, est appelée **ressource**.

Toute information (à sémantique stable) qui peut être nommée est une ressource: un article , une photo, une personne , un service ou n'importe quel concept.

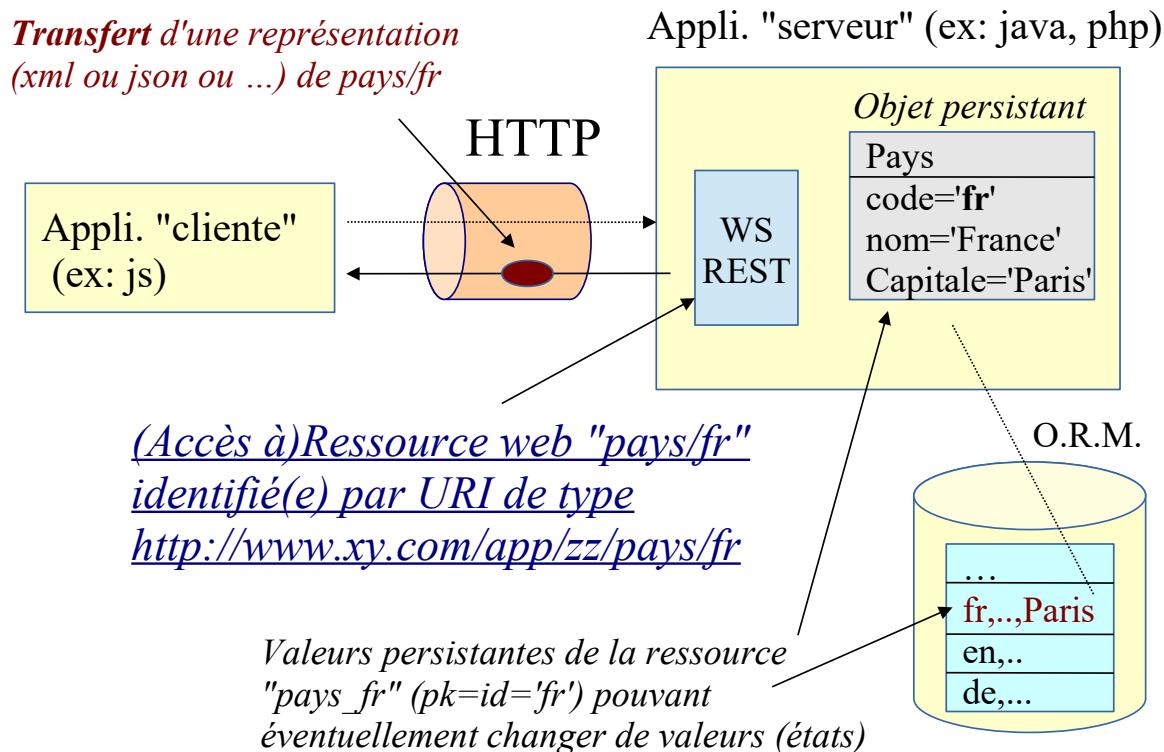
Une ressource est identifiée par un **identificateur de ressource**. Sur le web ces identificateurs sont les **URI** (Uniform Resource Identifier).

NB: dans la plupart des cas, une ressource REST correspond indirectement à un enregistrement en base (avec la *clef primaire* comme partie finale de l'uri "identifiant").

Les composants de l'architecture REST manipulent ces ressources en **transférant à travers le réseau** (via HTTP) des **représentations de ces ressources**.

Sur le web, on trouve aujourd'hui le plus souvent des représentations au format **HTML , XML ou JSON**.

REST : transferts de représentations de ressources



Format JSON (JSON = *JavaScript Object Notation*)

Les 2 principales caractéristiques

de JSON sont :

- Le principe de clé / valeur (map)
- L'organisation des données sous forme de tableau

```
[  
  {  
    "nom": "article a",  
    "prix": 3.05,  
    "disponible": false,  
    "descriptif": "article1"  
  },  
  {  
    "nom": "article b",  
    "prix": 13.05,  
    "disponible": true,  
    "descriptif": null  
  }  
]
```

Les types de données valables sont :

- tableau
- objet
- chaîne de caractères
- valeur numérique (entier, double)
- booléen (true/false)
- null

une liste d'articles

une personne

```
{  
  "nom": "xxxx",  
  "prenom": "yyyy",  
  "age": 25  
}
```

REST et méthodes HTTP (verbes)

Les méthodes HTTP sont utilisées pour indiquer la sémantique des actions demandées :

- **GET** : lecture/recherche d'information
- **POST** : envoi d'information
- **PUT** : mise à jour d'information
- **DELETE** : suppression d'information

Par exemple, pour récupérer la liste des adhérents d'un club, on peut effectuer une requête de type **GET** vers la ressource **http://monsite.com/adherents**

Pour obtenir que les adhérents ayant plus de 20 ans, la requête devient **http://monsite.com/adherents?ageMinimum=20**

Pour supprimer numéro 4, on peut employer une requête de type **DELETE** telle que **http://monsite.com/adherents/4**

Pour envoyer des informations, on utilise **POST** ou **PUT** en passant les informations dans le corps (invisible) du message HTTP avec comme URL celle de la ressource web que l'on veut créer ou mettre à jour.

2.3. Statuts HTTP (code d'erreur ou ...)

Catégories de code/statut HTTP :

1xx	Information (rare)
2xx (ex : 200)	Succès
3xx	Redirection
4xx	Erreur du client
5xx (ex : 500)	Erreur du serveur

Principaux codes/statuts en cas de succès ou de redirection:

200 , OK	Requête traitée avec succès. La réponse selon méthode de requête utilisée
201 , Created (rare)	Requête traitée avec succès et création d'un document.
204 , No Content (rare)	Requête traitée avec succès mais pas d'information à renvoyer.
301 , Moved Permanently	Document déplacé de façon permanente
304 , Not Modified	Document non modifié depuis la dernière requête

Principaux codes d'erreurs :

400 , Bad Request	La syntaxe de la requête est erronée (ex : invalid argument)
401 , Unauthorized	Une authentification est nécessaire pour accéder à la ressource.
403 , Forbidden	authentification effectuée mais manque de droits d'accès (selon rôles, ...)
404 , Not Found	Ressource non trouvée.
409 , Conflict	La requête ne peut être traitée en l'état actuel.
...	<i>liste complète sur</i> Liste_des_codes_HTTP">https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP
500 , Internal Server Error	Erreur interne (vague) du serveur (ex ; bug , exception , ...).
501, Not Implemented	Fonctionnalité réclamée non supportée par le serveur
503 , Service Unavailable	Service temporairement indisponible ou en maintenance.

2.4. Safe and idempotent REST API

Une Api "Rest" désigne un ensemble de Web-services liés à un certain domaine fonctionnel (ex : gestion des stocks ou facturation ou ...)

Un appel "HTTP" vers une api-rest est dit "*safe*" s'il n'engendre pas de modifications du coté des ressources du serveur ("*safe*" = "*readonly*") .

En mathématique , une fonction est dite "idempotente" si plusieurs appels successifs avec les mêmes paramètres retournent toujours le même résultat.

Au niveau d'une api-rest , une invocation HTTP (ex : *GET* , *PUT* ou *DELETE*) est dite "idempotente" si plusieurs appels successifs avec les mêmes paramètres engendrent un même état résultat" au niveau du serveur .

Mais la réponse HTTP peut cependant varier .

Exemple: premier appel à "delete xyz/567" --> return "200/OK"

et second appel à "delete xyz/567"--> return 404 / notFound

mais dans les 2 cas , la ressource de type "xyz" et d'id=567 est censée ne plus exister .

Le DELETE est donc généralement considéré comme idempotent .

	safe	idempotent
GET (et HEAD,OPTIONS)	y	y
PUT	n	y
DELETE	n	y
POST	n	n

Intérêt de l'impotence comportementale du coté serveur :

Une application cliente doit souvent passer par des intermédiaires pour véhiculer une requête HTTP jusqu'au serveur . Certains mécanismes intermédiaires considèrent "internet / http" comme pas fiable à 100 % et vont quelquefois effectuer plusieurs retransmissions d'une requête si la première tentative échoue . il vaut mieux donc que le serveur se comporte de manière idempotente dans un maximum de cas .

Bien que le vocabulaire "*idempotency*" ne soit pas du tout approprié , il est tout de même conseillé de retourner des réponses HTTP dans un format assez homogène vers le client pour que celui-ci soit simple à programmer (pas trop de if ... else ...)

Dans tous les cas , bien documenter "comportements & réponses" d'une apit rest .

2.5. API java pour REST (JAX-RS)

JAX-RS est une API java officielle/normalisée (depuis les spécifications **JEE 6**) qui permet de mettre en œuvre des services REST en JAVA :

Une classe java avec annotations JAX-RS sera exposée comme web service REST.

JAX-RS 1 se limitait à l'implémentation serveur "java".

JAX-RS 2 (JEE 7) propose maintenant une api pour les appels (du coté client "java")

Pour implémenter les services REST, on utilise principalement les annotations JAX-RS suivantes:

- **@Path** : définit le chemin (fin d'URL) de la ressource. Cette annotation se place sur la classe et/ou sur la méthode implémentant le service.
- **@GET, @PUT, @POST, @DELETE** : définit le mode HTTP (selon logique CRUD)
- **@Produces** spécifie le ou les Types MIME de la réponse du service
- **@Consumes** : spécifie le ou les Types MIME acceptés en entré du service

Les principales implémentations de JAX-RS sont :

- **Jersey**, implémentation de référence de SUN/Oracle (bien : simple/léger et efficace)
- **Resteasy**, l'implémentation interne à jboss (bien)
- **CXF** (gérant à la fois "SOAP" et "REST" , remplacer si besoin la sous couche "jettison" par "jackson" pour mieux générer du "json")
- **Restlet** (???)

2.6. Code typique d'une classe java (avec annotations de JAX-RS)

Exemple JAX-RS au format JSON :

```
import java.util.List;

import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
...
/*
 * classe java du WS REST lié aux clients (personnes)
 */
@Path("client") //avant dernière partie de l'URL
@Produces("application/json") //conv. automatique réponse java en réponse json

public class ClientRest {
//@EJB ne fonctionne pas ici ,
```

```

// il faut utiliser @Inject (plus moderne de CDI) à la place
@Inject // @Inject de l'api CDI (Container Dependency Injection)
// ne fonctionne que si
// le fichier WEB-INF/beans.xml est présent dans l'application.
private IServiceClient serviceClient;

@Path("/{idClient}") // dernière partie de l'URL
@GET // GET pour lecture , recherche unique par id/clef primaire
//URL = http://localhost:8080/myappWeb/services/rest/client/1
public Client rechercherClientQuiVaBien(
    @PathParam("idClient") Long numClient) {
    return serviceClient.rechercherInfosClient(numClient);
}

/* DEVINETTE :
Soap et Rest sont sur un bateau.
Soap glisse sur un savon et tombe à l'eau.
Qui est qui reste ?
*/
@Path("") // dernière partie de l'URL
// (éventuellement vide si rien de plus)
@POST // POST pour "ajout" ou "modif",
//URL = http://localhost:8080/myappWeb/services/rest/client
//avec en entrée { "prenom": "jean" , "nom" : "Bon" , ...}
@Consumes("application/json")
public Client postClient(Client cli) {
    System.out.println("requête recue: " + cli.toString());
    //en entrée cli.numClient est quelquefois à null (ou pas renseigné)
    cli = serviceClient.saveOrUpdateClient(cli);
    //en retour cli.numClient est quelquefois auto incrémenté
    return cli;
}

@Path("")
@GET // recherche multiple via critère(s) de recherche
//URL=http://localhost:8080/myappWeb/services/rest/client?nom=Therieur
public List<Client> rechercherClients(@QueryParam("nom") String nom) {
    if(nom!=null) {
        return serviceClient.rechercherListeClientsParNom(nom);
    }else {
        return serviceClient.rechercherTousLesClients();
    }
}
}

```

2.7. Configuration de JAX-RS intégrée à un projet JEE6/CDI

Lorsque JAX-RS est utilisé au sein d'un projet JEE6/CDI (par exemple avec ou sans "EJB3" pour JBoss7 ou bien pour Tomcat EE), on peut configurer la partie intermédiaire des URL "rest" et les classes java à prendre en charge via une configuration ressemblant à la suivante :

```
package tp.web.rest; //ou autre
import java.util.HashSet; import java.util.Set;
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

//url: http://localhost:8080/myWebApp/services/rest + @Path() java
@ApplicationPath("/services/rest")
public class MyRestApplicationConfig extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        final Set<Class<?>> classes = new HashSet<Class<?>>();
        classes.add(XyServiceRest.class);
        classes.add(ServiceRest2.class);
        return classes;
    }
}
```

Ceci fonctionne avec des classes java (ex: XyServiceRest) utilisant "**@Inject**" et avec **beans.xml** présent dans WEB-INF .

Variante basée sur une découverte automatique des classes de WS-Rest (comportant **@Path("...")**) :

```
@ApplicationPath("/rest")//partie du milieu des URLs après http://localhost:8080/appliJee-web
// et avant les valeurs de @Path() des classes java
public class MyRestApplicationConfig extends Application {
//rien (découverte automatique via un scan automatique des packages java de l'application)
}
```

Autre variante (avec **getSingletons()** à préparer) :

```
public class MyRestApplicationConfig extends Application {
    @Override
    public Set<Object> getSingletons() {
        final Set<Object> singletons = new HashSet<Object>();
        singletons.add(myRestImplSingleton);
        // myRestImplSingleton peut être instancié via un new , une fabrique ou autre
        // possibilité de mixer cela avec CDI ("beans.xml" , @Named , @Inject , ....)
        return singletons;
    }
}
```

3. JNDI

Présentation de JNDI

API Java ...

JNDI = Java Naming & Directory Interface

*Permettant
d'accéder à ...*

*Ex: Liste de
composants (EJB, ...)
et de ressources
techniques (pool, ...)*

*Ex: annuaire LDAP
(employés,)*

- **Des services de noms** : liste d'associations (bindings) entre des ressources et des noms logiques.
- **Des services d'annuaire (Directory)** : chaque entrée de l'annuaire comporte plusieurs caractéristiques (ex: nom, email , ... pour une Personne) et il est possible d'effectuer des recherches portant sur ces différents critères.

Recherche (via JNDI) de ressources enregistrées avec des noms logiques

Serveur de noms (souvent intégré ...)

Liste d'associations (bindings):

```
"nom1" , refObj1  
"nomN" , refObjN
```

2 Accès au
serveur de nom et
recherche via JNDI

Client

```
...  
ic=new InitialContext();  
ressource = ic.lookup("nomN");  
...  
ressource.fctX();
```

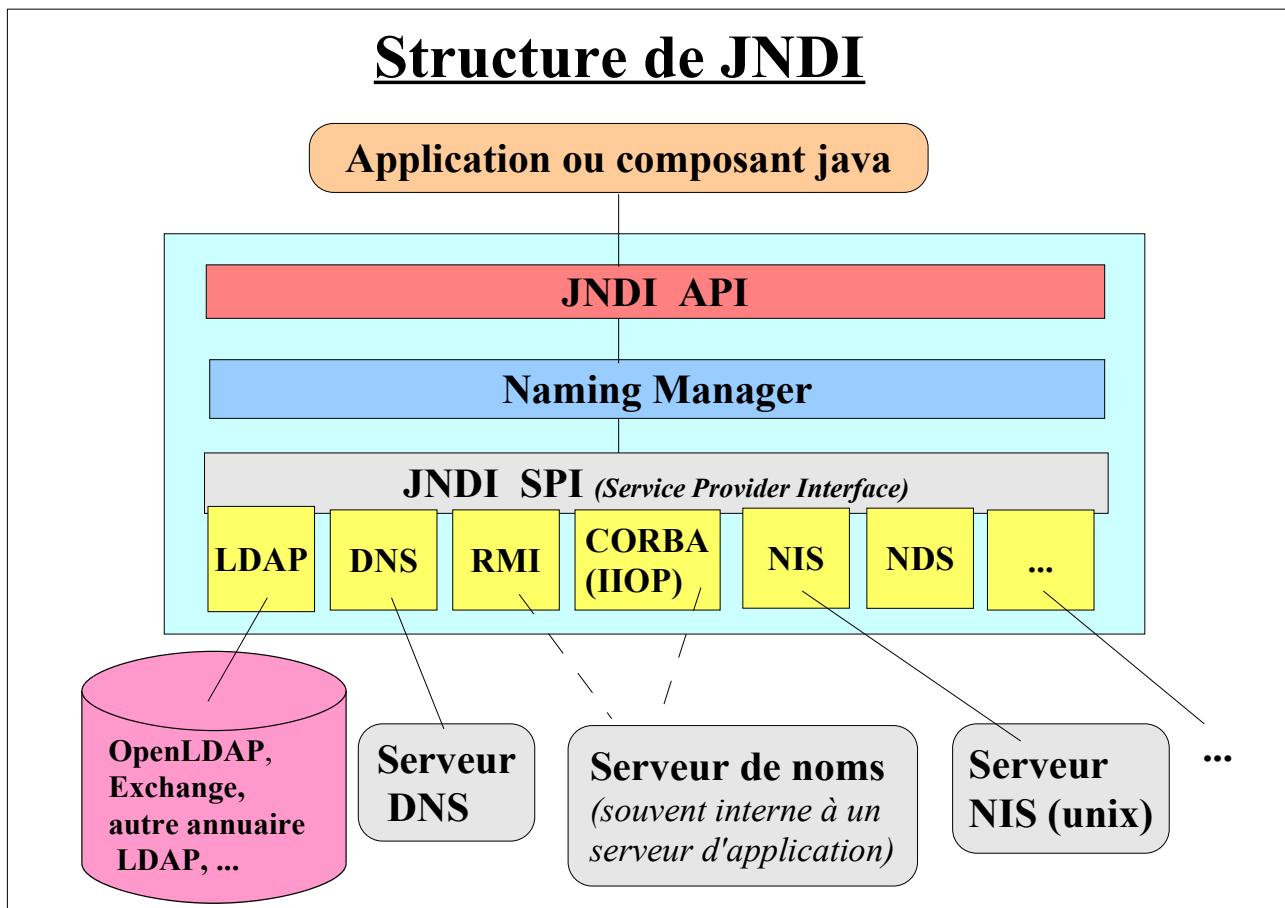
3 utilisation
de la ressource

1 Accès au serveur de nom
et enregistrement
de la ressource via JNDI

Serveur de ressources

(ex: Pool de connexions, objet RMI, ...)

```
...  
objRessource=new CRessource();  
ic=new InitialContext();  
ic.rebind("nomN", objRessource);  
...
```



Paramétrage du « Initial Context » de JNDI

De façon à préciser les *protocoles & adresses des serveurs de noms* que les couches basses de JNDI doivent utiliser, il faut renseigner quelques *propriétés systèmes*:

Paramétrage généralement effectué au sein de fichiers ".properties" :

(ex1: jndi.properties [pour rmi-over-iiop / tnameserv]):

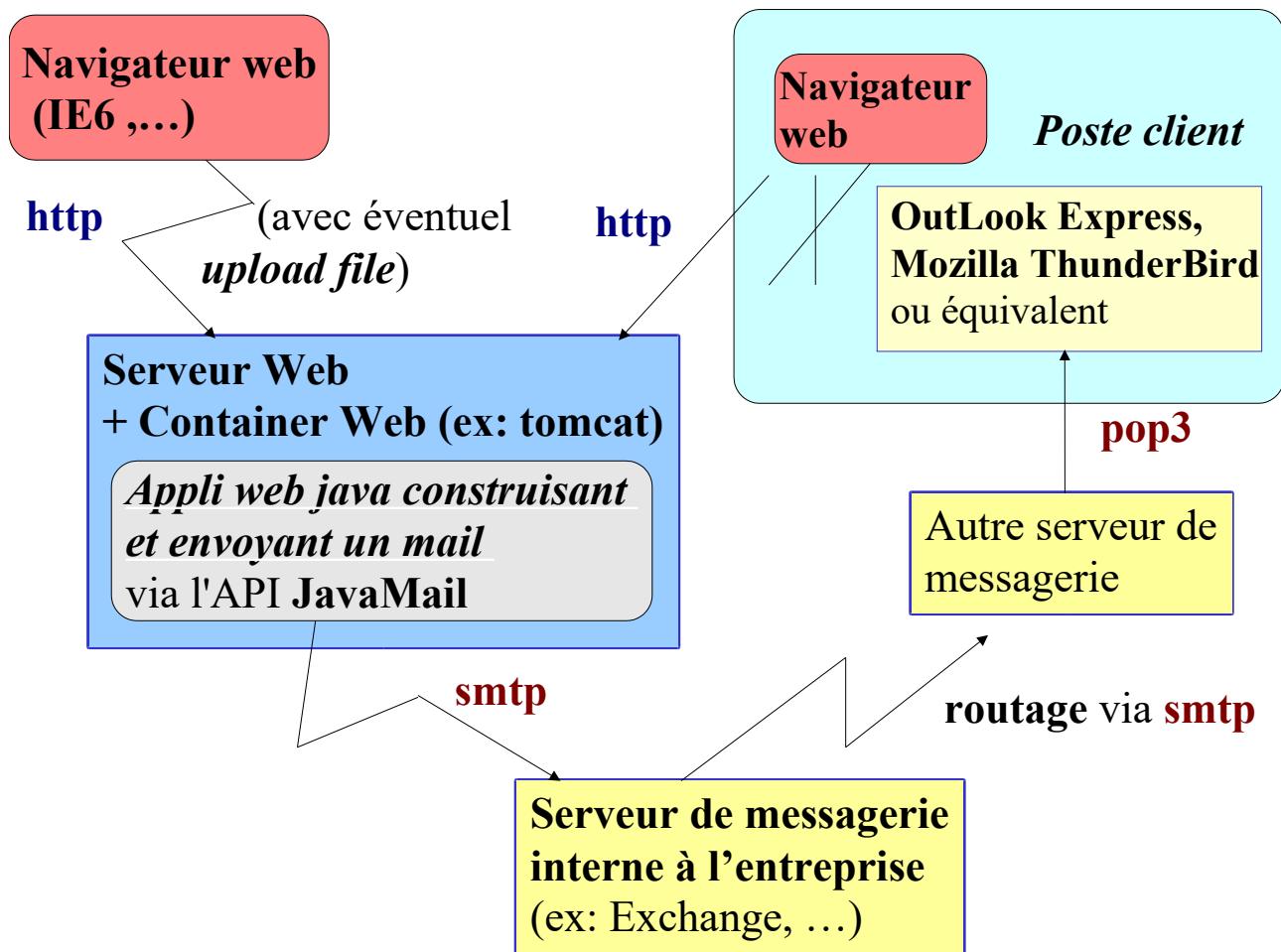
```
java.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
java.naming.provider.url=iiop://servnom:900
```

(ex2: JbossClientJndi.properties):

```
java.naming.factory.initial=
    org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming.client
```

4. L'api javax.mail

Pour envoyer (ou consulter) un mail depuis un programme java, on peut utiliser l'api standard **javax.mail** (à télécharger via l'url <http://java.sun.com/products/javamail>).



Exemple : Séquence d'envoi d'un mail (sans pièce jointe)

```

import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;

String host = "localhost"; //"mydomain.com"
String from = "didier@mydomain.com";
String to = "destinataire@xxxxxxx.com";
Properties props = System.getProperties();
props.put("mail.smtp.host", host);
Session session = Session.getDefaultInstance(props, null);

MimeMessage message = new MimeMessage(session);
message.setFrom(new InternetAddress(from));
message.addRecipient(Message.RecipientType.TO, new InternetAddress(to));
message.setSubject("Hello JavaMail");
message.setText("Welcome to JavaMail");

Transport.send(message);
  
```

5. Messaging asynchrone avec JMS

5.1. Présentation de JMS

JMS (*Java Message Service*)

JMS est une **API** permettant de faire **dialoguer des applications** de façon **asynchrone**.

Architecture associée: **MOM** (Message Oriented MiddleWare).

NB: *JMS n'est qu'une API qui sert à accéder à un véritable fournisseur de Files de messages* (ex: **MQSeries/WebSphere_MQ** d'IBM , **ActiveMQ** d'apache, ...)

Dans la terminologie JMS, les Clients JMS sont des programmes Java qui envoient et reçoivent des messages dans/depuis une file (**message queue**).

Une file de message sera gérée par un "Provider JMS" .

Les clients utiliseront **JNDI** pour accéder à une file.

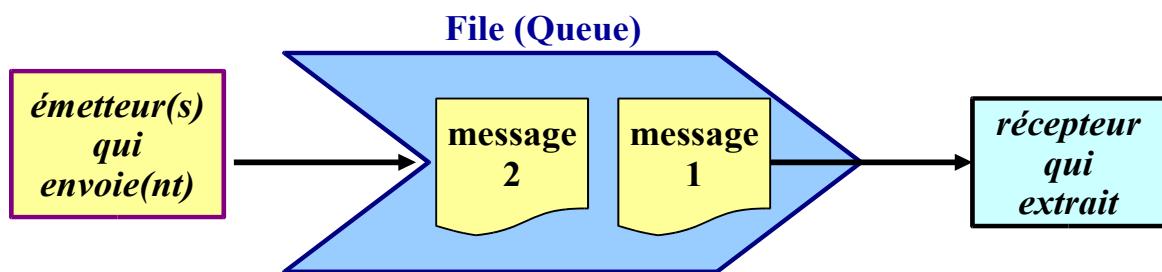
L'objet **ConnectionFactory** sera utilisé pour établir une connexion avec une file.

L'objet **Destination** (*File* ou *Topic*) sert à préciser la destination d'un message que l'on envoi ou bien la source d'un message que l'on souhaite récupérer.

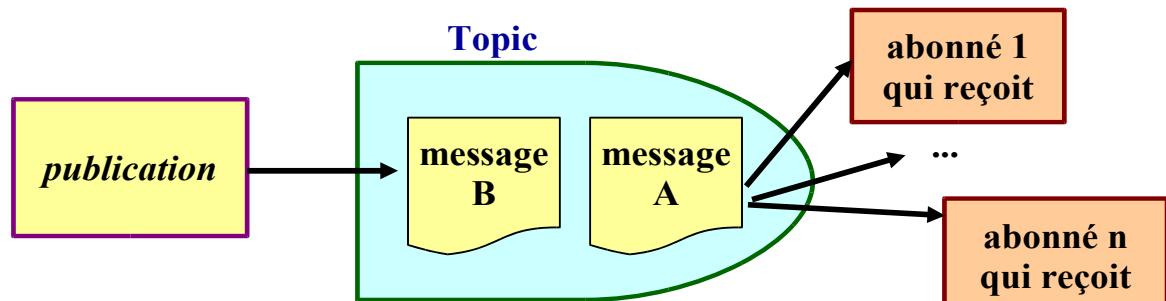
JMS permet de mettre en oeuvre les 2 modèles suivants:

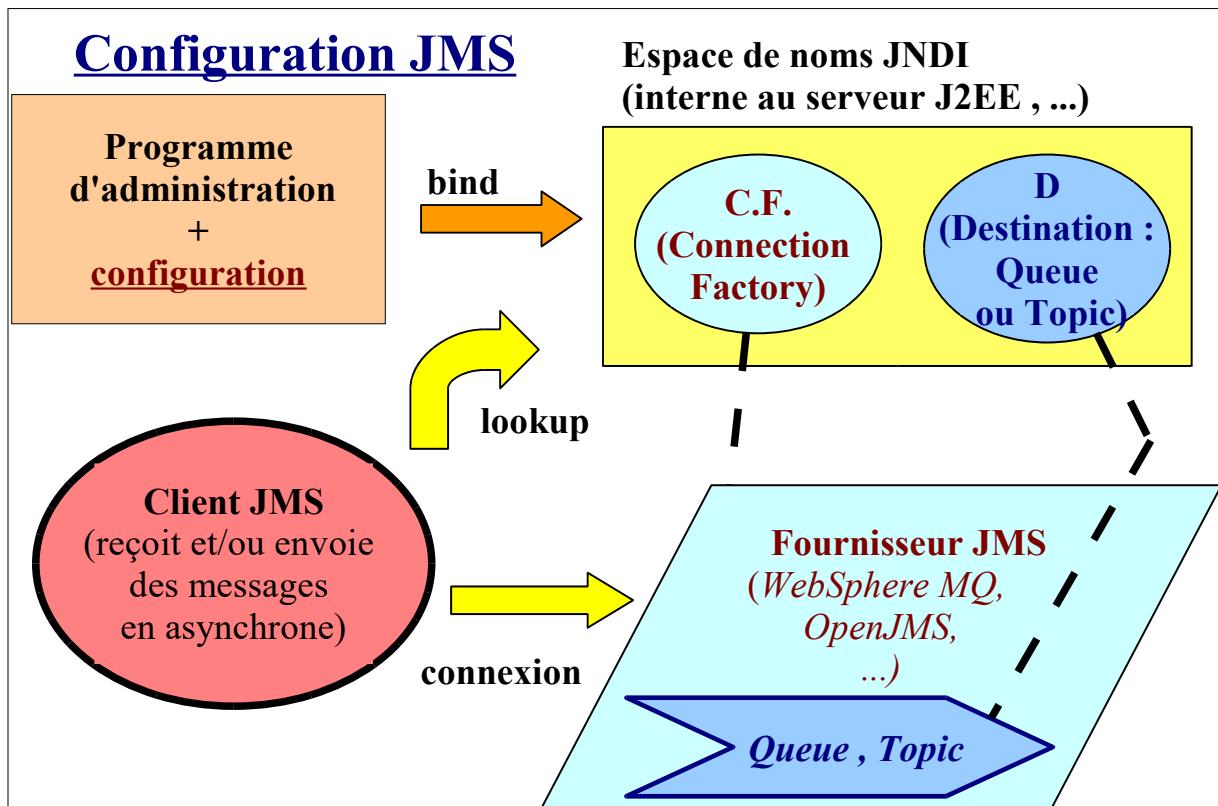
- **PTP** (Point To Point)
- **Pub/Sub** (Published & Subscribe) .../...

JMS Queue : *Point To Point*



JMS Topic : *Publish / Subscribe*





5.2. JMS 1.1

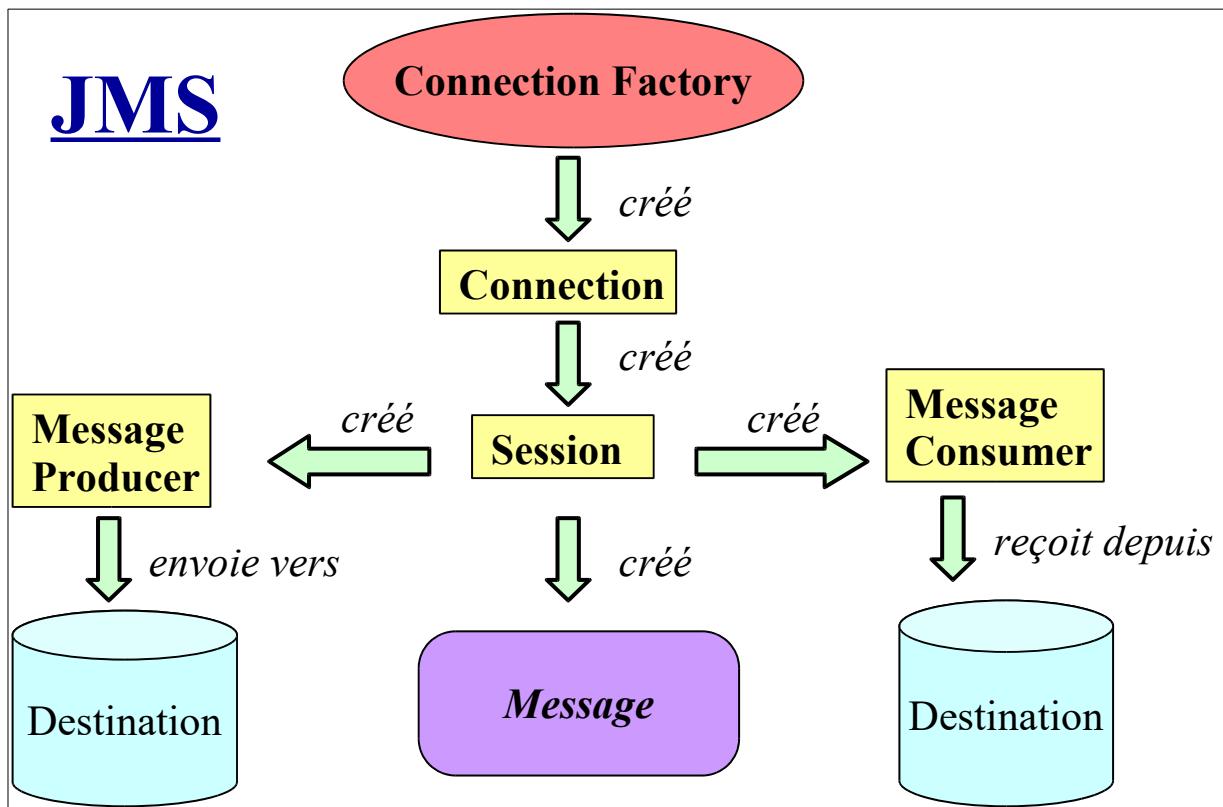
2 modèles de destination :

- **PTP** (Point To Point) - *File de message spécifique à une entité (comparable à une boîte à lettres).*
- **Pub/Sub** (Published & Subscribe) - *Les clients et les serveurs utilisent une même file logique dont le contenu est organisé de façon hiérarchique. On publiera des messages au niveau d'un certain noeud et ceux qui se sont inscrits (abonnés) vis à vis de ce noeud recevront alors ces messages.*

Le tableau ci-dessous résume les différentes **interfaces** utilisées au niveau de l'api JMS:

JSM (Interface générique)	PTP Domain	Pub/Sub Domain
ConnectionFactory (mt)	QueueConnectionFactory	TopicConnectionFactory
Connection (mt)	QueueConnection	TopicConnection
Destination (mt)	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver,QueueBrowser	TopicSubscriber

(mt) : multi-threading support.



5.3. Exemples (fragments) de code JMS 1.1

5.3.a. Obtention de l'objet ConnectionFactory via JNDI:

```

QueueConnectionFactory queueConnectionFactory;
Context messaging = new InitialContext();
queueConnectionFactory = (QueueConnectionFactory)
messaging.lookup("QueueConnectionFactory");
  
```

ou bien

```

TopicConnectionFactory topicConnectionFactory;
Context messaging = new InitialContext();
topicConnectionFactory = (TopicConnectionFactory)
messaging.lookup("TopicConnectionFactory");
  
```

5.3.b. Obtention d'une file de message via JNDI:

```

Queue stockQueue;
stockQueue = (Queue) messaging.lookup("StockQueue");
  
```

ou bien

```

Topic stockTopic;
stockTopic = (Topic) messaging.lookup("StockTopic");
  
```

5.3.c. Création d'un objet Connection via l'usine:

```
QueueConnection queueConnection;
queueConnection = queueConnectionFactory.createQueueConnection();
ou bien
```

```
TopicConnection topicConnection;
topicConnection = topicConnectionFactory.createTopicConnection();
```

5.3.d. Création d'une Session à partir de l'objet Connexion:

```
QueueSession session;
session = queueConnection.createQueueSession(false /*trasact. */,
                                              Session.AUTO_ACKNOWLEDGE);
```

ou bien

```
TopicSession session;
session = topicConnection.createTopicSession(false,
                                              Session.CLIENT_ACKNOWLEDGE);
```

5.3.e. Obtention de l'objet "MessageProducer" pour envois

```
QueueSender sender;
sender = session.createSender(queue);
ou bien
```

```
TopicPublisher publisher;
publisher = session.createPublisher(stockTopic);
```

5.3.f. Obtention de l'objet "MessageConsumer" pour réceptions

```
QueueReceiver receiver;
receiver = session.createReceiver(queue);
ou bien
```

```
TopicSubscriber subscriber;
subscriber = session.createSubscriber(stockTopic);
et
```

```
StockListener myListener;
subscriber.setMessageListener(myListener);
avec
```

```
public class StockListener implements javax.jms.MessageListener {
    void onMessage(Message message) {
        // unpack and handle the messages we receive.
    }
}
```

5.3.g. Déclencher le début possible des réceptions de messages:

`queueConnection.start(); ou bien topicConnection.start();`

5.3.h. Crédation de messages

Création d'un message binaire:

```
byte[] stockData; // stock information as a byte array
BytesMessage message;
message = session.createByteMessage();
message.writeBytes(stockData);
```

Création d'un message en mode texte:

```
String stockData; // stock information as a String
TextMessage message;
message = session.createTextMessage();
message.setText(stockData);
```

Création d'un message structuré (avec différents champs nommés):

```
MapMessage message;
message = session.createMapMessage();
message.setString("Name", stockName); message.setDouble("Value", stockValue);
message.setLong("Time", stockTime); message.setDouble("Diff", stockDiff);
message.setString("Info", stockInfo);
```

Création d'un message à données à lire séquentiellement:

```
StreamMessage message;
message = session.createStreamMessage();
message.writeString(stockName); ...message.writeDouble(stockValue);
message.writeLong(stockTime); ...
```

Création d'un message contenant les valeurs d'un objet Java (Sérialisation):

```
ObjectMessage message = session.createObjectMessage();
message.setObject(stockObject);
```

5.3.i. Envoi et réception

Envoi et réception d'un message en mode PTP:

```
sender.send(message);
et
StreamMessage stockMessage = (StreamMessage) receiver.receive();
```

Publication et réception d'un message en mode Pub/Sub:

```
publisher.publish(message);
et
appel automatique de la méthode OnMessage() de l'abonné
```

5.3.j. Extraction des valeurs d'un message

Extraction des valeurs d'un message binaire:

```
byte[] stockData; // stock information as a byte array
int length;
length = message.readBytes(stockData);
```

Extractions des valeurs d'un message texte:

```
String stockData;
stockData = message.getText();
```

Extractions des valeurs d'un message structuré:

```
stockName = message.getString("Name");
stockValue = message.getDouble("Value");
stockTime = message.getLong("Time");
stockDiff = message.getDouble("Diff");
stockInfo = message.getString("Info");
```

Extractions des valeurs d'un message à valeurs séquentielles:

```
stockName = message.readString(); stockValue = message.readDouble();
stockTime = message.readLong(); stockDiff = message.readDouble();
stockInfo = message.readString();
```

Extractions des valeurs d'un message objet:

```
stockObject = message.getObject();
```

5.4. Champs des entêtes d'un message JMS

Champ de l'entête	signification	fixé (affecté) par
JMSDestination	File de destination	méthode send()
JMSDeliveryMode	PERSISTENT ou NON PERSISTENT	méthode send()
JMSExpiration	0 : pas d'expiration. sinon <i>n ms</i> à vivre.	méthode send()
JMSPriority	priorité de 0 à 9 (0-4: normal) (5-9: high)	méthode send()
JMSMessageID	<i>ID:xxx</i> identifiant du message	méthode send()

JMSTimestamp	estampillage de temps	méthode send()
JMSCorrelationID	identifiant de la requête associée à la réponse	Client
JMSReplyTo	File où il faut placer la réponse.	Client
JMSType	selon le contexte , catégorie , ...	Client
JMSRedelivered	si réception multiple d'un même message	Provider

Le champ **JMSReplyTo** peut comporter le nom d'une file (éventuellement temporaire) que l'émetteur de la requête a préalablement créé pour récupérer la réponse..

5.5. Liste des principaux "Provider JMS"

La liste des principales implémentations disponibles se trouve au bout de l'url suivante:
<http://java.sun.com/products/jms/vendors.html>

Parmi les implémentations les plus connues, on peut citer:

- **MQSeries** (→ renommé **WebSphere MQ**) d' IBM
- Services "JMS" intégrés dans les serveurs J2EE (JBoss, WebLogic, WebSphere).

Produits (FreeWare - OpenSource):

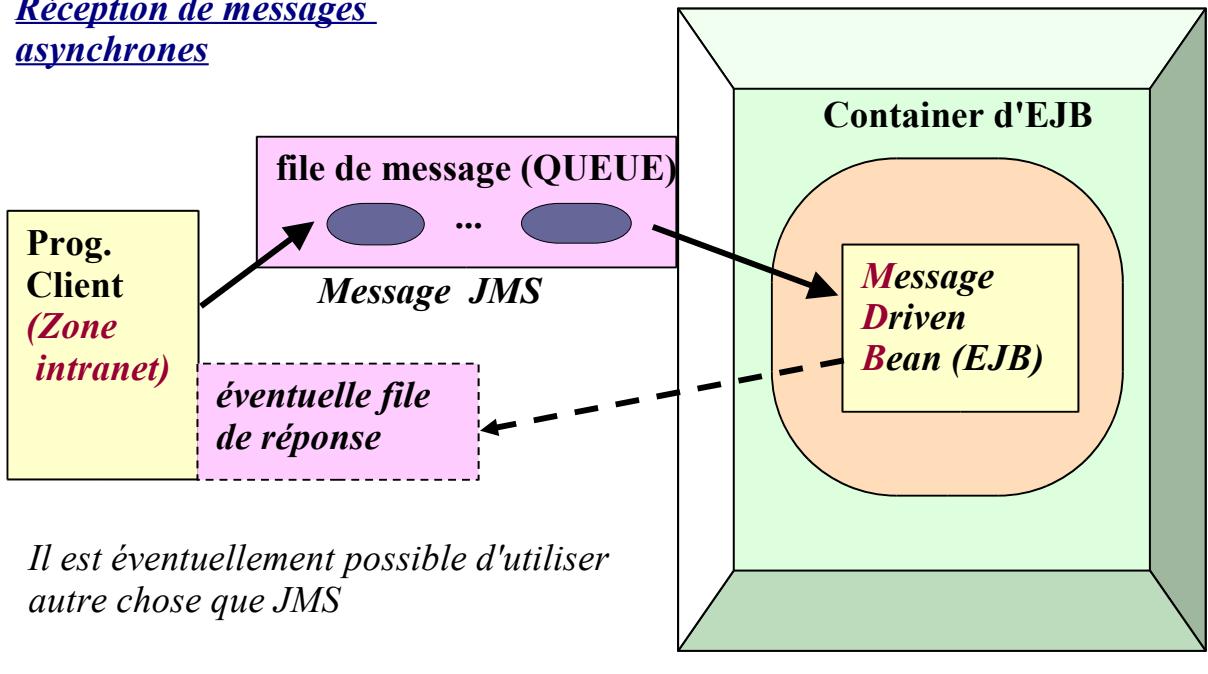
- **OpenJMS**
- **ActiveMQ** (*d'Apache Software*)
- **Joram** de OW2
- ...

5.6. EJB3 de type MDB

EJB piloté par message JMS asynchrone

Réception de messages asynchrones

Serveur d'application J2EE



5.7. JMS2 (version simplifiée de JMS 1.1)

De 2002 à 2013 , JMS 1.1 a été utilisé par un énorme paquet de projets java et les développeurs se sont habitués à utiliser cette API .

En 2013, Une version simplifiée de JMS appelée JMS2 a été proposée .

Cependant cette version complémentaire n'est qu'un supplément facultatif intéressant qui ne remplace pas les anciens éléments de JMS 1.1 (qui continuent à être fournis et implémentés) .

Détails accessibles sur <https://www.oracle.com/technical-resources/articles/java/jms20.html> .

Principales simplifications :

Trois nouvelles interfaces: **JMSPContext**, **JMSProducer**, and **JMSConsumer**:

- **JMSPContext** remplace les anciens objets séparés **Connection** et **Session** objects en un seul objet
- **JMSProducer** est un remplacement plus léger pour **MessageProducer** utilisant "method chaining" alias "builder pattern".
- **JMSConsumer** est un remplacement plus léger pour **MessageConsumer** avec le même style de code (enchainement d'appels de méthodes)

Exemples de simplification :

Envoyer un message en mode texte :

1.1	<pre>public void sendMessageJMS11(ConnectionFactory connectionFactory, Queue queueString text) { try { Connection connection = connectionFactory.createConnection(); try { Session session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE); MessageProducer messageProducer = session.createProducer(queue); TextMessage textMessage = session.createTextMessage(text); messageProducer.send(textMessage); } finally { connection.close(); } } catch (JMSException ex) { // handle exception (details omitted) } }</pre>
2.0	<pre>public void sendMessageJMS20(ConnectionFactory connectionFactory, Queue queue, String text) { try (JMSPContext context = connectionFactory.createContext();) { context.createProducer().send(queue, text); } catch (JMSRuntimeException ex) { // handle exception (details omitted) } }</pre>

NB: Cette api simplifiée est apparue très tardivement et n'est pas beaucoup utilisée par les développeurs habitués à utiliser JMS 1.1.

6. JNI et JNA

Pour appeler à partir d'un bloc de code java du code natif écrit en c/c++ , deux technologies sont possibles :

- la première technologie s'appelle **JNI** (Java Native Interface). Elle est fournie par défaut par le JDK et nécessite de manipuler un langage natif pour effectuer les appels aux fonctions natives ;
- la seconde technologie s'appelle **JNA** (Java Native Access). C'est une [API](#) tierce qui offre l'avantage de s'abstraire de la couche native.

6.1. Aperçu sur JNI

Dès les premières versions, le langage JAVA donne la possibilité d'appeler; à partir de certains morceaux de code JAVA, des ***fonctions (méthodes) écrites en C ou C++***.

Ces méthodes sont dites natives et doivent être déclarées comme telles en utilisant le mot clé **native**.

Nb: Il ne faut surtout pas abuser des méthodes natives car elles ne sont pas indépendantes de la plate-forme (Linux ou Windows, ...).

6.1.a. Déclaration et appel d'une méthode native

```
public class MaClasseJava
{
    int x = 10;
    private native int aff();
    public static void main ( String [] argv)
    {
        // Chargement de la librairie dynamique
        // contenant le code compilé de la méthode native:
        System.loadLibrary("MaLibrairieNative");

        MaClasseJava instance = new MaClasseJava();
        instance.aff();
    }
}
```

6.1.b. Implémentation d'une méthode native

La fonction doit être écrite en C. Il faut pour cela, connaître le prototype exact de la fonction. Ce prototype est particulier dans le sens où il doit s'interfacer avec un objet Java dont l'ensemble des attributs doit être vu depuis le C sous la forme d'une structure.

On obtient ce prototype en utilisant l'outil **javah** suivi du nom de la classe qui va générer automatiquement un fichier d'entête (.h) contenant le prototype de la fonction C. Il faut avoir préalablement compilé le source Java avec l'outil **javac**.

Il ne reste alors plus qu'à écrire le corps de la fonction.

javah nompaquet.Nomclasse

Exemple de fichier header généré par javah (version JDK 1.2 / JNI) :

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class p1_MaClasseJava */

#ifndef _Included_p1_MaClasseJava
#define _Included_p1_MaClasseJava
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:    p1_MaClasseJava
 * Method:   aff
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_p1_MaClasseJava_aff
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

code de la méthode native:

```
#include <stdio.h>
#include "p1_MaClasseJava.h"

JNIEXPORT jint JNICALL Java_p1_MaClasseJava_aff
  (JNIEnv * env, jobject obj)
{
jclass cls = env->GetObjectClass(obj);
jfieldID champX=env->GetFieldID(cls,"x" /*field name */,"I" /*signature*/);
jint x = env->GetIntField(obj,champX);
printf(" La valeur de l'attribut x est %ld \n", x);
return 0;
}
```

6.1.c. Etablir le lien entre JAVA et le code C:

Le code C doit être compilé sous la forme d'une librairie dynamique (.dll sous windows , so sous unix) qui sera chargée au début de l'exécution du programme Java.

Nb: Dans l'environnement Windows, une DLL peut être fabriquée à partir du produit Visual C++ (projet de type DLL sans MFC) ou bien CodeBloc ou autre .

6.2. Exemple JNA

Code disponible sur projet *exJna* du référentiel https://github.com/didier-mycontrib/java_8_9_1x

Code natif en langage c dans src/native (ou ailleurs) :

basiclib.h

```
void display(char* ch);
int addition(int a, int b);
```

basiclib.c

```
#include "basiclib.h"
#include <stdlib.h>
#include <stdio.h>

void display(char* ch) {
    printf("%s", ch);
}

int addition(int a, int b){
    return a+b;
}
```

build_basiclib.bat

```
REM on windows , GCC may be a subpart of codeblock
REM https://www.codeblocks.org/downloads/binaries/
```

```
set GCC_PATH=C:\Program Files\CodeBlocks\MinGW\bin
set PATH=%GCC_PATH%;%PATH%
```

```
cd %~dp0
```

```
gcc -c basiclib.c
gcc -shared basiclib.o -o basiclib.dll
copy basiclib.dll ..\..\basiclib.dll
```

→ ceci permet de construire une librairie native partagée (.dll sous windows ou .so sous linux)

Code d'une application java/jna :

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>tp</groupId>
<artifactId>exJna</artifactId>
<version>0.0.1-SNAPSHOT</version>
<properties>
    <java.version>11</java.version>
```

```

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<!-- windows/preferences/general/workspace / UTF8 avec eclipse coherent -->
</properties>

<dependencies>

<dependency>
    <groupId>net.java.dev.jna</groupId>
    <artifactId>jna</artifactId>
    <version>5.12.1</version>
</dependency>

</dependencies>

<build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.10.1</version>
            <configuration>
                <source>${java.version}</source>
                <target>${java.version}</target>
            </configuration>
        </plugin>
    </plugins>
</build>
</project>

```

tp.CLibrary.java

```

package tp;

import com.sun.jna.Native;
import com.sun.jna.Library;
import com.sun.jna.Platform;

public interface CLibrary extends Library {
    CLibrary INSTANCE = (CLibrary) Native.load((Platform.isWindows() ? "msvert" : "c"),
                                                CLibrary.class);

    void printf(String format, Object... args);
}

```

tp.CBasicLib.java

```

package tp;

import com.sun.jna.Native;
import com.sun.jna.Library;

public interface CBasicLib extends Library {
    CBasicLib INSTANCE = (CBasicLib) Native.load("basiclib", CBasicLib.class);
    //NB: basiclib.dll (on windows) or basiclib.so (on linux)
    //may be found in root of java project directory (with eclipse) or ...
    void display(String g); //from basiclib
    int addition(int a, int b); //from basiclib
}

```

tp.ExJnaApp.java

```
package tp;

public class ExJnaApp {

    public static void main(String[] args) {
        CLibrary.INSTANCE.printf("Hello world via native printf\n");

        try {
            CBasicLib.INSTANCE.display("Hello world via native display\n");
            int res = CBasicLib.INSTANCE.addition(5,6);
            System.out.println("res of native addition(5,6) = " +res);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

→
res of native addition(5,6) = 11
Hello world via native printf
Hello world via native display

VII - EJB et Serveurs d'applications

1. Versions des principales API de JEE / Jakarta EE

de JEE 1.0 à JEE 1.4 (avant 2005) : pas d'annotation , config. XML de dingues !!!

JavaEE 5 et JavaEE 6 : avec annotations et "SUN"

JavaEE 7 : "ORACLE" et "<http://xmlns.jcp.org>"

JavaEE 8 (2018) : "ORACLE"

Jakarta EE 8 (2019) : "Eclipse Foundation"

Jakarta EE 9 (2020): javax. to jakarta.* namespace change.*

En règle générale , un serveur d'une version N supporte du code standard (et des configurations "standards") en versions N-1 et N-2 .

Par contre un ancien code complémentaire (non standard) de l'époque N-1 est rarement parfaitement supporté par la version N (ex : extension richfaces3 pour JSF1 / JEE5 qui n'est plus supporté par JSF2 de JEE6).

1.1. Variantes "Full-Profile" et "Web-Profile" :

Les versions Java EE 6 , 7 et 8 existent en au moins 2 déclinaisons :

variantes (profiles)	Contenu	Déploiement	Serveurs
Full-Profile	Toutes les api de Java-EE	.ear	Serveurs complets avec conteneur d'EJB et conteneur WEB
Web-Profile	Api "web" + CDI + JPA + EJB en version "Lite" et JAX-RS depuis JEE7	.war	Serveurs complets et également serveurs simplifiés avec conteneurs "web" seulement (ex : TomEE / tomcat EE)

Principales restrictions de la version "Lite" des EJB 3.1 et 3.2 :

- pas d'accès remote (@Local seulement)
- pas de MDB / JMS (EJB asynchrone avec files d'attentes)

Evolutions des principaux besoins :

Epoque	Application	Profile nécessaire suffisant
2000-2013 (n-tiers / SOA)	Remote EJB (RMI) , JMS 1.1 , web-services "SOAP" et JSF	Full-Profile
2014-2018 (WOA)	JSF et/ou JAX-RS + javascript	Web-Profile
2019-202x (micro-services)	JAX-RS + javascript (JSF)	Web-Profile

Attention : le "micro-profile" de JakartaEE est encore très récent (pas encore de recul , pas encore considéré comme "mature") !!!!

1.2. Tableau des principales versions

API	Fonctionnalités	JEE5	JEE6	JEE7	JEE8 Jakarta	JEE9 Jakarta
Servlet	Cœur web/http	2.5	3.0	3.1	4.0	5.0
JSP	Pages (coté serveur)	2.1	2.2	2.2	2.3	4.0
JSF	Framework Web sophistiqué (MVC, ...)	1.2	2.0	2.2	2.3	3.0
EJB	Objet/service métier , transaction , ...	3.0	3.1 3.1 Lite	3.2 3.2 Lite	3.2 3.2 Lite	4.0 4.0 Lite
JPA	Persistance / ORM java	1.0	2.0	2.1	2.2	3.0
JMS	Envoi et réception de messages (Queue)	1.1	1.1	2.0 et 1.1	2.0 et 1.1	3.0
JTA	Transaction distribuée	1.1	1.1	1.2	1.2	2.0
JSTL	Standard TagLib	1.2	1.2	1.2	1.2	2.0
JavaMail	Envoi et réception de mail	1.4	1.4	1.4	1.4 ou 1.5	1.5 ou 1.6
JCA	Connecteur JEE (syst. propriétaire , ...)	1.5	1.6	1.6	>= 1.6	>= 1.6
JAX-WS	web-services Soap	2.0	2.2	2.2	2.3	>= 2.3
JAX-RS	web-services REST		1.1	2.0	2.1	3.0
CDI	Injection de dépendance (inter-container)		1.0	1.1	2.0	3.0
Validation	Validation via annotations		1.0	1.1	2.0	3.0

Nouvelles API (en version 1.0) disponibles depuis JEE7 : Java Api for WebSocket , Java Api for JSON , Concurrency Utilities for JEE , Batch Applications for Java Platform

Principales nouveautés de JavaEE 8 :

- **HTTP/2** supporté par **Servlet 4** , PushBuilder dans Servlet 4
- JSON-B (Json-Binding) , un peu comme jackson-databind .
- améliorations de JAX-RS (new reactive client)
- CDI en v2 (utilisable en javaSE)
- **Java EE Security Api** (pour cloud , Paas)
- Bean validation 2.0 (avec alignement java8 (Optional , ...))

1.3. Namespaces XML pour les fichiers de configuration

Attention : La marque "SUN" a été reprise/rachetée en **2009** par la marque "Oracle".

Les API les plus récentes ont maintenant des namespaces qui ont beaucoup évolués
 ("<http://java.sun.com>" devenu "<http://xmlns.jcp.org>")

Principales entêtes Xml des fichiers de configuration JEE :

web.xml (3.0 / JEE6)	<web-app xmlns:xsi=" http://www.w3.org/2001/XMLSchema-instance " xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd " version="3.0">
web.xml (3.1 / JEE7)	<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd " version="3.1">
web.xml (4.0 / JEE8)	<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd " version="4.0">

faces-config.xml (2.0 / JEE6)	<faces-config xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd " version="2.0">
faces-config.xml (2.2 / JEE7)	<faces-config xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd " version="2.2">
faces-config.xml (2.3 / JEE8)	<faces-config xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_3.xsd " version="2.3">

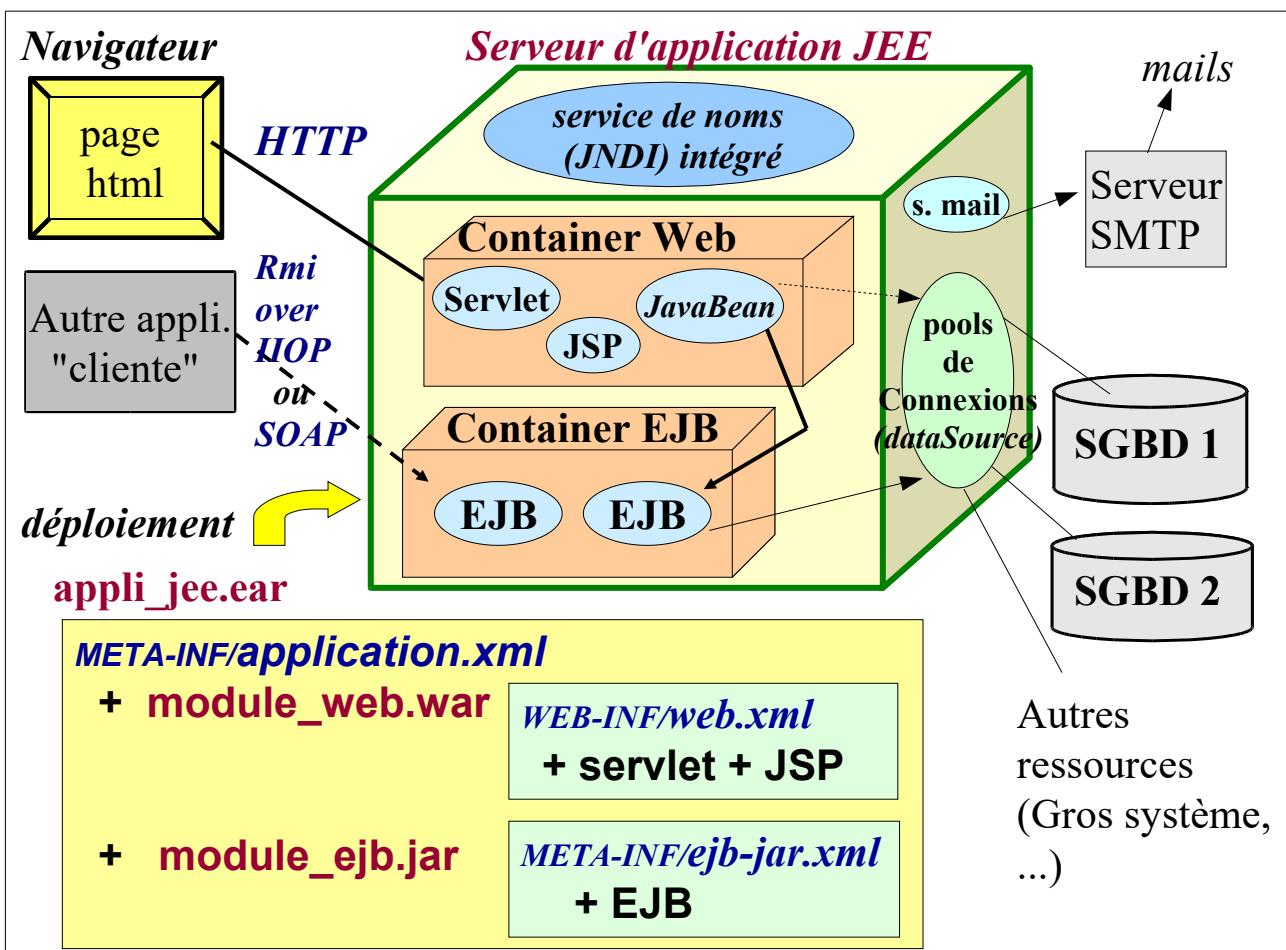
beans.xml (1.0 / JEE6)	<beans xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd " version="1.0">
beans.xml (1.1 / JEE7)	<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd " version="1.1">

beans.xml (12.0 / JEE8)	<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/beans_2_0.xsd " version="2.0">
-----------------------------------	--

persistence.xml (2.0 / JEE6)	<persistence xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd " version="2.0">
persistence.xml (2.1 / JEE7)	<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd " version="2.1">
persistence.xml (2.2 / JEE8)	<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd " version="2.2">

2. Serveurs d'applications Java EE

JEE peut également être vu comme un **modèle d'architecture** pour les **serveurs d'applications**. Les **spécifications JEE** indiquent clairement le rôle des "**container**" : Ceux-ci doivent offrir aux composants applicatifs qu'ils hébergent un accès normalisé aux API standards de JEE . Autrement dit , un **composant JEE** (*ex*: servlet , EJB, ...) fonctionne exactement de la même manière au sein des serveurs WebLogic , JBoss ou WebSphere car il peut appeler les mêmes fonctionnalités (mêmes API) et qu'il expose lui même les mêmes points d'entrées pour la gestion de son cycle de vie.



Depuis J2EE 1.2 , le déploiement d'une application JEE est standardisé:

- Un fichier ".war" (pour *Web ARchive*) contient tous les composants "web" et les fichiers de configurations associés (*WEB-INF/web.xml* , ...).
- Un fichier ".jar" (pour *Java ARchive*) contient tous les composants "EJB" et les fichiers de configurations associés (*META-INF/ejb-jar.xml* , ...).
- Un fichier ".ear" (pour *Enterprise ARchive*) regroupe différentes sous archives ("war", ".jar" , ...) et un fichier de configuration globale : *META-INF/application.xml* dont la balise **context-root** de l'**application WEB** indique l'URL relative de celle-ci.

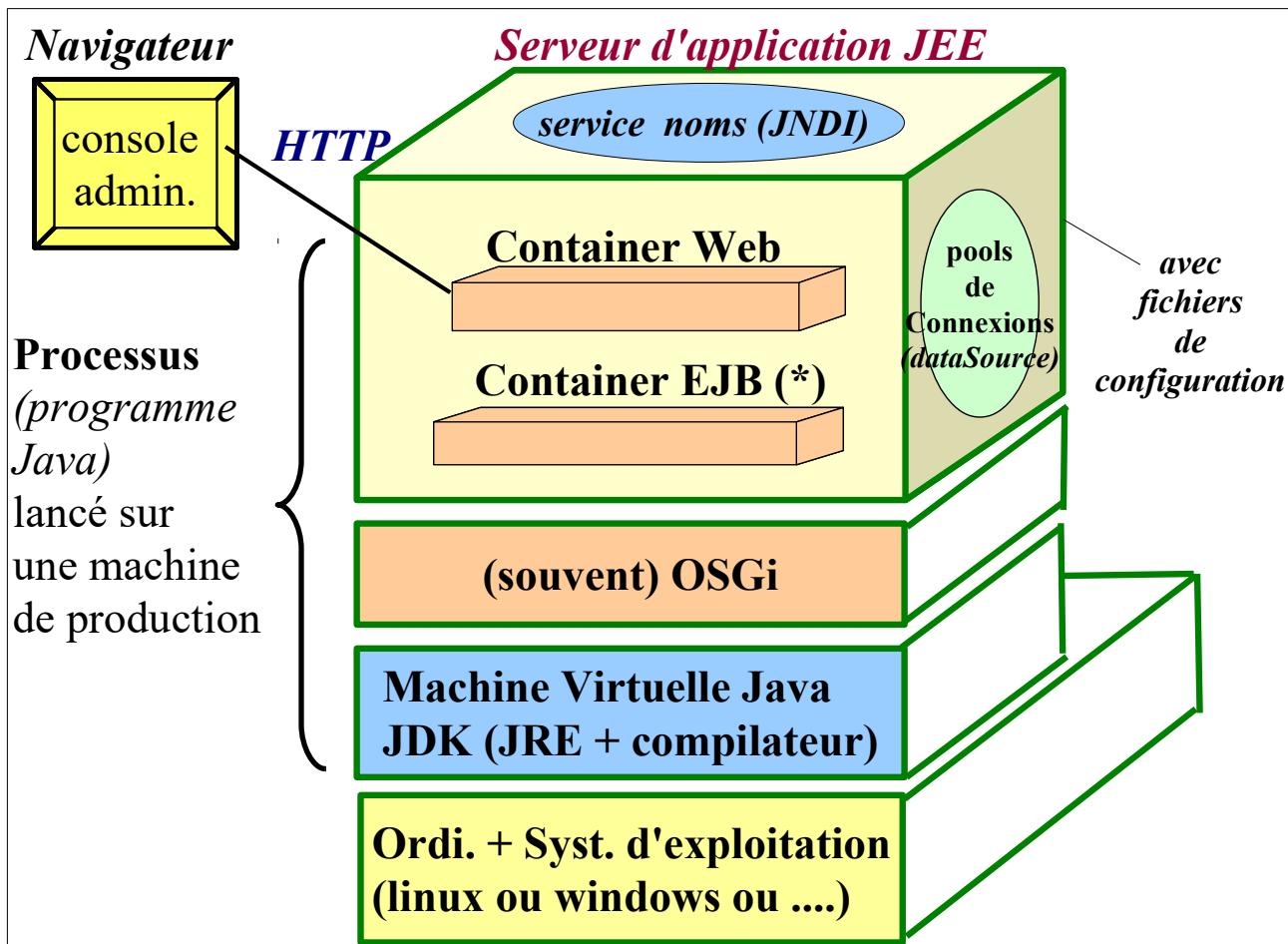
Au lieu de parler de J2EE 1.5 , Sun/JavaSoft a préféré baptiser **JEE5,6,7** les nouvelles versions des spécifications de sa plate-forme Java de niveau entreprise .

Les principaux apports de ces nouvelles versions sont les suivants:

- **EJB3 , 3.1 , 3.2** (avec api **JPA 1 , 2.0 , 2.1** pour la persistance des données).
- Nouveau support des **services WEB "Soap"** via l'api **JAX-WS** (mieux que JAX-RPC)
Support des **services WEB "REST"** avec l'api **JAX-RS** (v1.0 pour JEE6 v2.x pour JEE7)
- Intégration du framework **JSF** (1 puis 2) dans la partie WEB
- **Partie Web supportant l'injection de dépendances (@Ejb , @Resource , @Inject CDI)**
- ...

Un Serveur d'application JEE est avant tout un cas particulier de programme écrit en java et qui

s'exécute à l'aide d'une machine virtuelle Java (JVM) .



Certains mécanismes internes des serveurs d'applications JEE ont besoin de déclencher des compilations (ex: pages JSP transformées en Servlet à compiler) . Il faut donc s'appuyer sur le JDK complet (compilateur + JRE = Java Runtime Environment) .

La version du JDK (1.4 , 1.5 , 1.6 , 1.7 ou 1.8) a une très grande importance car elle conditionne les possibilités/fonctionnalités du serveur.

3. Evolution de JEE

Evolutions de J2EE, JEE5, JEE6

J2EE 1.0 à 1.2

Socle architecture = Servlet/JSP + JNDI + RMI + EJB
Formalisation des archives (.war, .jar, .ear)

J2EE 1.3 à 1.4 (WebSphere 5,6 , WebLogic 7,8,9, Jboss 3.2,4)

EJB 2.x (**MDB**, ..., Interfaces locales, ...), Connecteurs **JCA** (et .rar)

JEE5 (Jdk >= 1.5 , WebSphere7 , Jboss 4.2 ou 5 ,)

EJB3 (config. via annotations , Java Persistence Api)

Framework web **JSF 1** (Java Server Faces) , **IOC** ,

JAX-WS (Api simple et efficace pour Services Web)

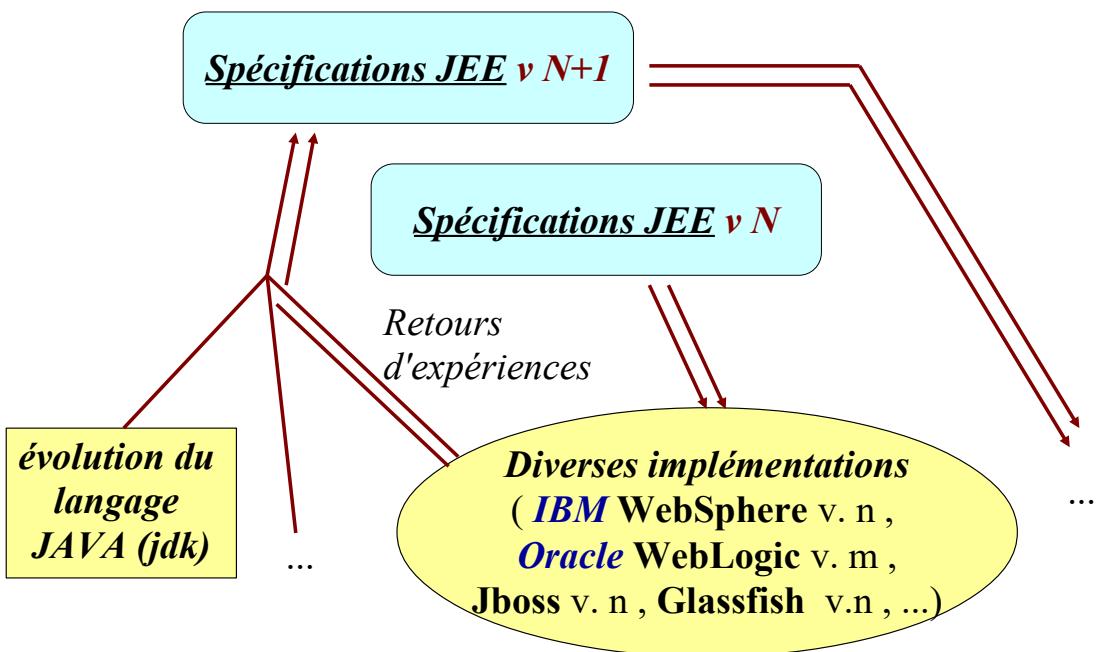
JEE6(jdk >=1.6 , Jboss 7 , Glassfish 3 , ..) :

EJB3.1 , JSF2 , annotations coté Web, JAX-RS 1 ,CDI (@Inject),...

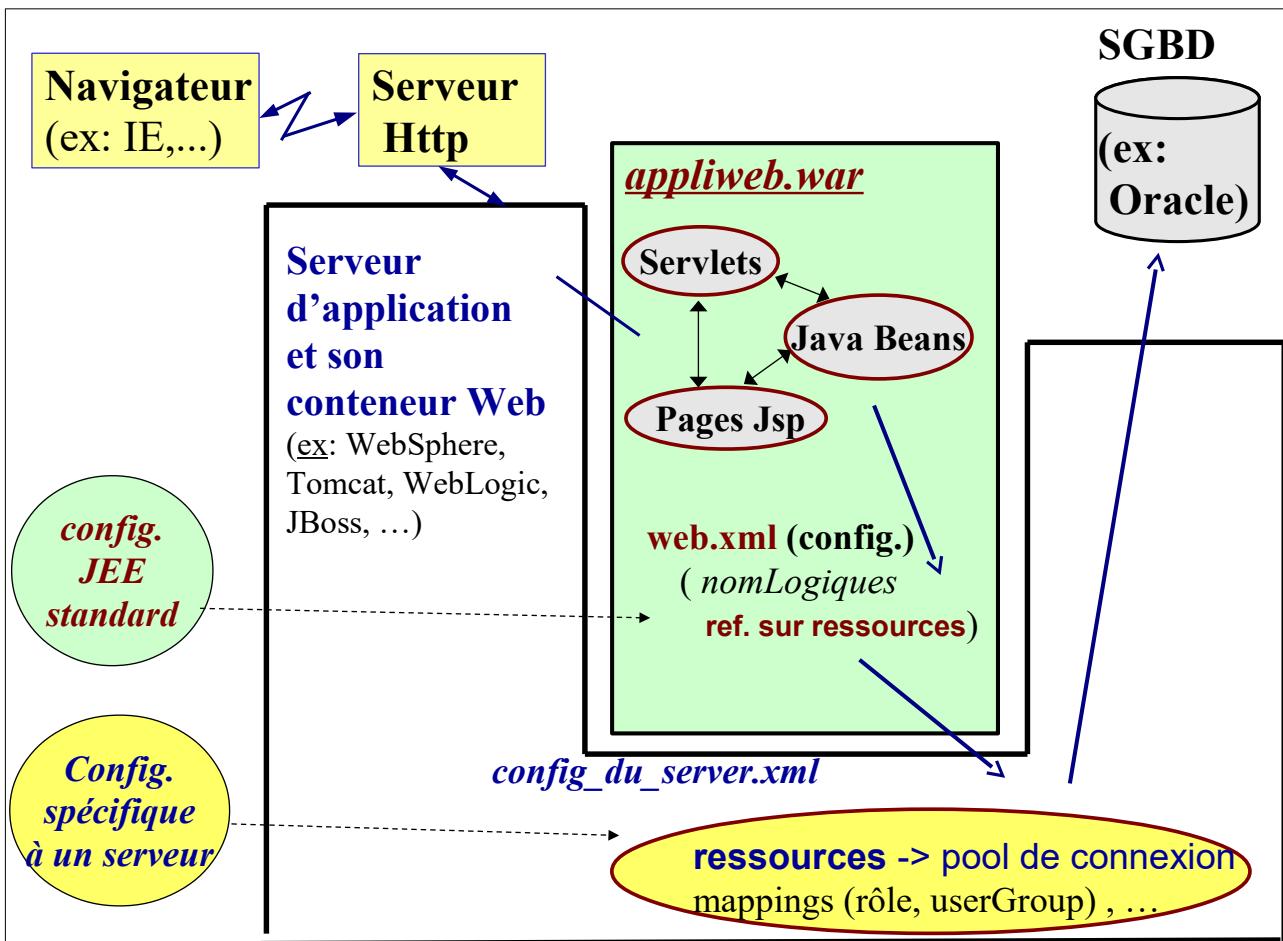
JEE7(Glassfish 4 , Jboss WildFly 9 , ...) :

EJB3.2 , JSF2.1, JAX-RS2, @Transactional sans EJB, ...

JEE : une *spécification* d'architecture qui évolue



4. Spécificités selon le serveur d'application JEE



Chaque serveur d'application se configure à sa façon (avec des fichiers de configuration différents) et quelquefois avec l'aide d'une console d'administration.

Exemples : ...\\glassfish4\\glassfish\\domains\\domain1\\config**domain.xml** pour glassfish
\\wildfly-9.0.2.Final\\standalone\\configuration**standalone-full.xml** pour Jboss

...

AU FINAL , on se retrouve "DEPENDANT DU SERVEUR d'APPLICATION" pour des CONFIGURATIONS SPECIFIQUES sur les points suivants :

- DataSource JDBC (accès au bases de données)
- Session de mail (config URL SMTP pour envoyer un mail)
- configurations JMS (Queue , ..) et JNDI (noms logiques des ressources locales ou distantes)
- sécurité JEE (realms , ...) ,

4.1. Principaux serveurs d'applications (JEE)

Serveurs d'applications	Marques/Editeurs	Caractéristiques
WebSphere	IBM	<ul style="list-style-type: none"> - Produit commercial avec le support d'une grande marque. - Serveur assez sophistiqué (très paramétrable et avec une bonne console d'administration). - surtout utilisé dans les grandes entreprises (banques, assurances, ...)
WebLogic	BEA --> Oracle	<ul style="list-style-type: none"> - Autre bon produit commercial (à peu près aussi sophistiqué que WebSphere)
Jboss (4.2 , 5.1 , 7.1) puis EAP (payant en prod) et wildFly (purement Open Source)	Jboss / Red Hat	<ul style="list-style-type: none"> - Open source , existe depuis longtemps - Souvent innovant sur les technologies java (jmx , ejb3, ...) - Utilisation très simple pour les tests durant la phase de développement - console d'administration rudimentaire.
...		
Tomcat (*)	Apache Group	<ul style="list-style-type: none"> - Open source faisant office de référence sur la partie "conteneur Web". - Serveur JEE simplifié (partie "conteneur web" seulement (sans EJB)).
TomcatEE	Apache Group	Assemblage de "Tomcat" + "OpenEJB/OpenJPA" + "OpenWebBean" (CDI) .
GlassFish	SUN --> Oracle	<ul style="list-style-type: none"> - Serveur JEE de référence (en partie open source) assez complet et assez innovant sur certaines technologies (BPEL, ESB/JBI ,) - Moins sophistiqué de "WebLogic" de la même marque

(*) **NB:** Tomcat peut être utilisé :

-soit de façon autonome en tant que mini serveur JEE (sans partie EJB)
 -soit en tant que partie "conteneur web" intégrée dans un autre serveur JEE plus complet .

- Certains anciens serveurs JEE (tels que "Jonas" de OW2) ne sont plus maintenus (faute de budget).

4.2. Serveur particulier "TomEE" (Tomcat EE)

(Tomcat + Jakarta EE = TomEE) est une version dérivée de tomcat qui est accompagnée des Api du "web-Profile" de Java EE (récemment renommé Jakarta EE) .

Autrement dit :

- le serveur traditionnel Tomcat ne comporte en lui que les api web fondamentales (servlet et jsp). c'est au développeur de récupérer des api complémentaires (JSF , JAX-RS, ...) via maven (en scope=compile)
- le serveur TomEE comporte déjà en lui les ".jar" pour les api standards de JEE (en version "WebProfile" (EJB-Lite , JSF , CDI , JAX-RS, ...)) . Beaucoup de dépendances maven peuvent être exprimées en mode "provided" et certains fichiers de configuration (META-INF/persistence.xml , ...) sont bien interprétés .

Attention : La partie JPA de TomEE est basée sur OpenEJB / OpenJPA plutôt que sur Hibernate .

Variantes de TomEE (versions récentes 7 et 8) :

- **TomEE webProfile** (servlet/jsp , jax-rs / **REST**, jsf , jpa , cdi , ejbLite , ...)
- **TomEE plus** (= TomEE webProfile + JAX-WS / **SOAP** + JMS)
- TomEE MicroProfile (= TomEE webProfile + api récentes "jakarta microProfile")

5. De EJB2 à EJB 3 à EJB 4

EJB2 (avant java 5 et annotations)	Configuration XML de dingue . Complètement "has been" et pas très au point .
EJB 3.x (à partir de java 5 et avec annotations)	Bien au point , fiable , standard efficace à partir des années 2005/2006 .
EJB 4	

6. EJB (Enterprise JavaBean)

Fonctionnalités des EJB (*valeurs ajoutées*)

- **Appels distants possibles** (via RMI-over-IIOP ou SOAP)
- **Bon support** pour **transactions distribuées**
- **Contrôle d'accès (authentification, ...)**
- **Module** réutilisable d'**objets "métier"**
- **Standard J2EE/JEE5** supporté par beaucoup de serveurs (WebSphere, WebLogic, JBoss, Jonas, ...)

Présentation des EJB

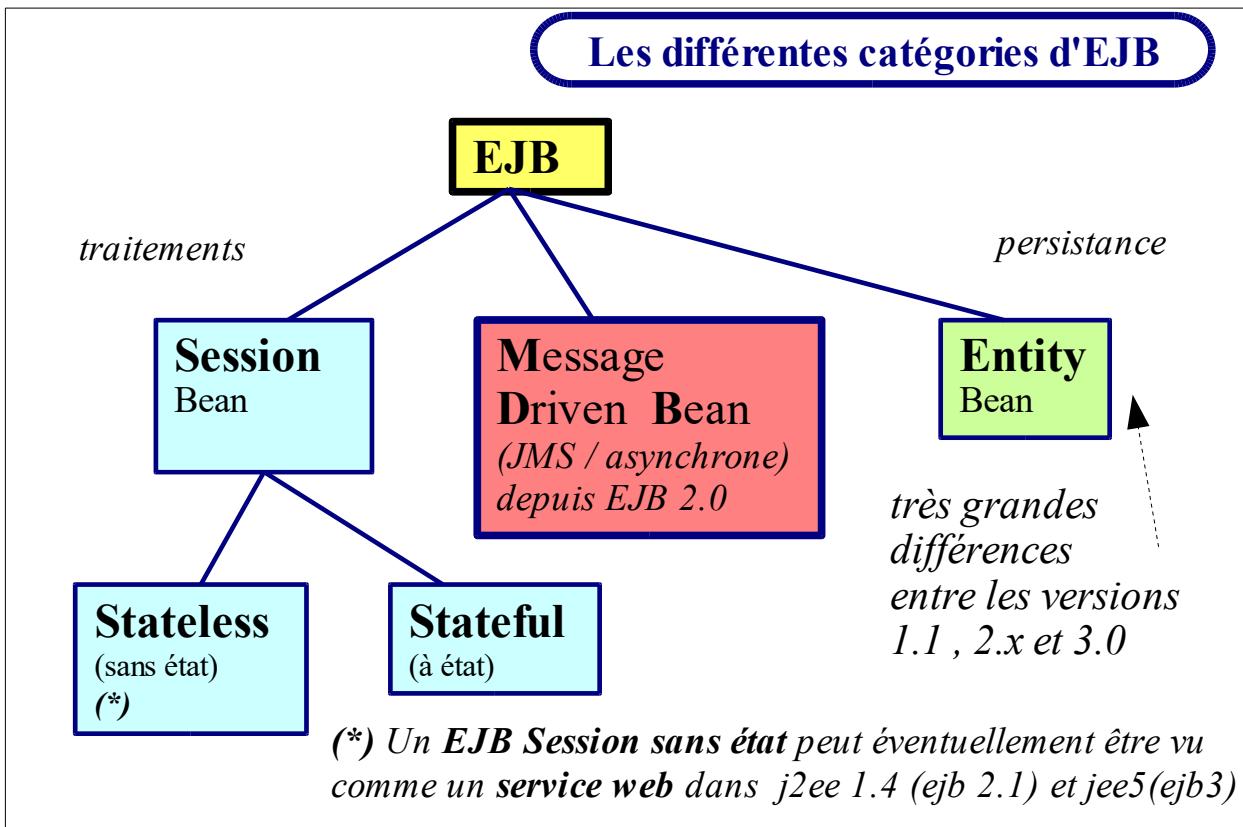
Un **EJB** correspond essentiellement à un **objet métier** (*traitement applicatif* ou *entité persistante*).

Le sigle **EJB** désigne avant tout une **spécification de composant métiers**: *comment ils doivent être écrit et le contrat qu'ils doivent respecter avec le conteneur EJB*

Principal objectif du framework "EJB":

-ne programmer que l'aspect "métier" des composants.
le serveur d'application (avec son container d'EJB) **prend en charge les aspects techniques** (sécurité, multi-tâches, transactions).

6.1. Entity et Sessions Beans



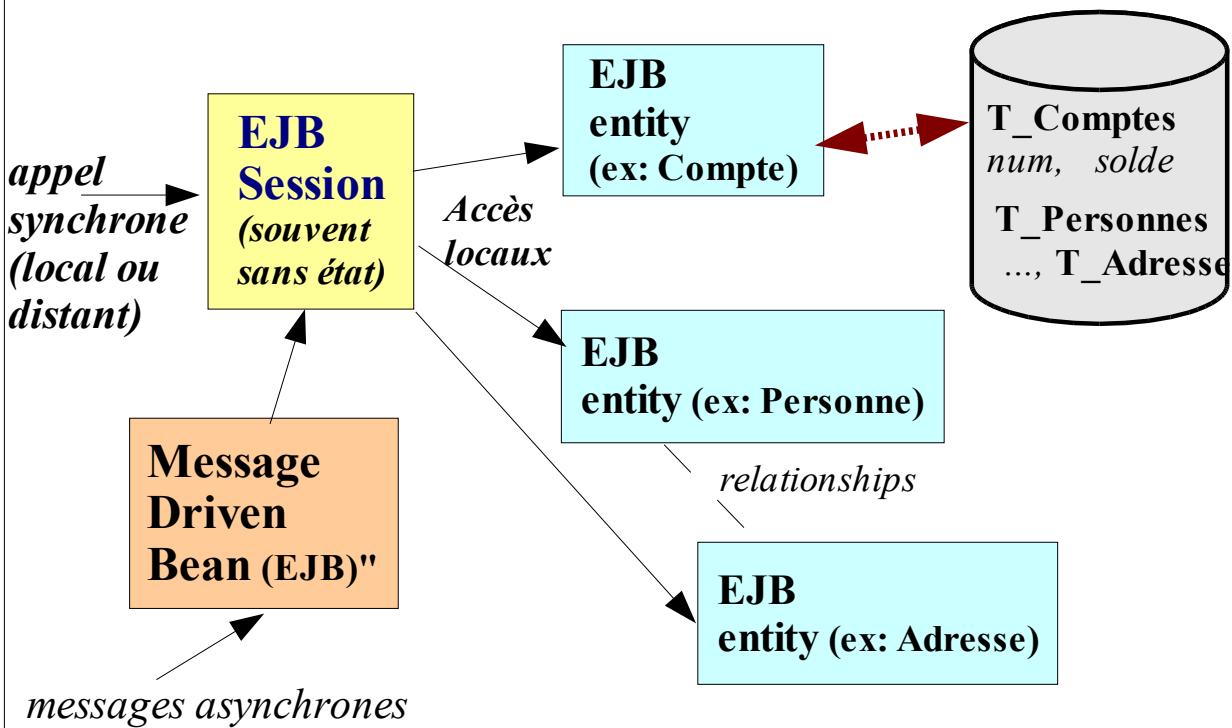
Différents types d'EJB

- **EJB Session** : Ces composants correspondent aux points d'entrée en mode synchrone des "*traitements métiers et applicatifs*". Ils sont étroitement lié à la session d'un utilisateur.
 ---> temporaires et non-partagés

- **EJB Entity** : Ces composants correspondent aux "*entités fondamentales du métier*". Ils représentent des données partagées par tous les utilisateurs du système
 ---> persistants et partagés

- **EJB Piloté par Messages** : Ces composants correspondent aux points d'entrée en mode **asynchrone** des "traitements applicatifs". Ils sont généralement déclenchés suite à la réception d'un message JMS ou SOAP. ---> temporaires & asynchrones

Relations classiques entre EJB d'un même module



Stateless (sans état) vs. Stateful (à état)

- (Ejb Session) **stateless** :
 - sans état, aucune donnée métier n'est conservée dans la mémoire de l'EJB entre deux appels (successifs) émanant d'un même client .
 - sert à factoriser des traitements atomiques entre les clients . Tous les paramètres nécessaires doivent être précisés d'un coup lors d'un appel d'une méthode autonome.
 - très bien optimisés par les serveurs d'application → bonnes performances.
- (Ejb Session) **stateful** :
 - avec état conversationnel, les données internes du bean sont conservées entre deux appels (ex: caddy électronique,).
 - sert à gérer une session utilisateur.

6.2. Qualités (A.C.I.D.) d'une transaction distribuée basique

Transactions distribuées

• **A.C.I.D.** ==> **Atomicity , Consistency , Isolation , Durability**

• L' **Atomicité** désigne le comportement "**tout ou rien**" (Le tout vu en tant qu'élément unique et atomique doit soit réussir , soit échouer). Il n'y a pas de demi-mesure.

• La **Consistance** d'une transaction désigne le fait que **les différentes opérations doivent laisser le système dans un état stable et cohérent**.

• Le concept d' **Isolation** signifie ici que **2 transactions concurrentes n'interfèrent pas entre elles** (Points critiques: résultats intermédiaires et opérations annulées).

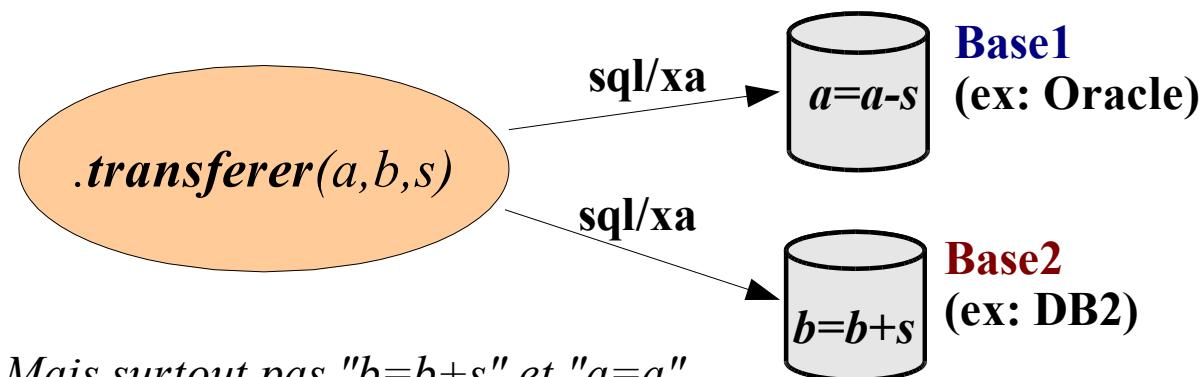
• La **Durabilité** indique que **les résultats d'une transaction doivent absolument être mémorisés de façon durable** (sur un support physique) de façon à survivre suite à une éventuelle défaillance(un fichier de Log peut également être très utile).

6.3. Protocole XA pour le commit à 2 phases

Protocole XA et commit à 2 phases (1)

Une **transaction distribuée** peut faire intervenir de **multiples ressources** telles que celles-ci par exemple:

- Base 1 (ex: Oracle via JDBC), Base 2 (ex: DB2 via JDBC)
- Moniteur transactionnel (ex: Tuxedo ou CICS via connecteurs).
- Système de message asynchrone (ex: MQSeries via JMS), ...



Protocole XA et commit à 2 phases (2)

De façon à ce que toutes les opérations à tous les niveaux (chacune des bases de données, ...) soient globalement annulées ou validées, on a recours à la technique suivante:

- 1 - Chaque ressource mise en jeu dans la transaction effectue des opérations dans une zone mémoire à part (ex: opérations SQL que l'on pourra éventuellement annuler) puis envoie un signal pour indiquer qu'à son niveau tout va bien.
 - 2 - Un élément "pilote de la transaction" centralise ces acquittements.
 - 3 - Si chaque protagoniste de la transaction distribuée a réussi sa tâche, le pilote envoie à chacun d'eux l'ordre d'entériner la mise à jour (commit final). Si un seul protagoniste de la transaction distribuée a échoué dans sa tâche, le pilote envoie à tout le monde l'ordre d'annuler la mise à jour (rollback final).
- Cette technique standard du **commit à deux phases** est formalisée au niveau d'un **protocole** normalisé dénommé **XA**.

6.4. Effets du contexte transactionnel sur les EJB

Transactions distribuées & EJB : Comportements

- ◆ Une transaction automatiquement gérée par le serveur (conteneur d'EJB) est implicitement validée lorsque toutes ses parties se sont bien passées (sans exception).
- ◆ Dès qu'une des opérations de la transaction génère une **exception système** héritant de *RuntimeException* (telle que **EJBException**) , l'ensemble de la transaction est alors automatiquement annulée.
- ◆ Si l'on souhaite **annuler explicitement une transaction** (suite à un test quelconque) il faut dans ce cas appeler la méthode **context.setRollbackOnly()** .

6.5. Attributs transactionnels sur EJB (approche déclarative)

Attribut de transaction tx	Transaction en cours	Transaction au niveau du Bean
Required (par défaut)	none	T2
	T1	T1
RequiresNew	none	T2
	T1	T2
Mandatory	none	error
	T1	T1
NotSupported	none	none
	T1	none
Supports	none	none
	T1	T1
Never	none	none
	T1	error

6.6. Annotations sur EJB3 concernant les transactions

@TransactionManagement (BEAN or CONTAINER) , **default=CONTAINER** à placer facultativement devant la classe d'un EJB session.

@TransactionAttribute (**MANDATORY** ou **REQUIRED** ou
REQUIRES_NEW ou **SUPPORTS** ou
NOT_SUPPORTED ou **NEVER**)

à placer facultativement devant les méthodes qui nécessitent un comportement transactionnel.

NB: Ces 2 annotations sont implicitement placées d'office avec les valeurs par défaut

(CONTAINER , REQUIRED) qui conviennent très bien dans 95% des cas.

---> Les EJB3 fonctionnent donc en mode transactionnel même si rien n'est indiqué .

6.7. Comparaison entre transactions Spring et EJB/JEE

Par défaut , Spring n'utilise pas JTA (Java Transaction Api) et les transactions sont donc gérées de façon simple (et souvent suffisante) sur une seule base de données.

Spring peut gérer des transactions distribuées sur plusieurs bases via xa et JTA en configurant des compléments (relativement complexes) tels que Atomikos ou bien Bitronix ou autres .

A l'inverse , un serveur d'application JEE et les EJBs gèrent par défaut les transactions de façon sophistiquée via JTA (et selon les spécifs JCA) .

Ceci dit , un micro-service a-t-il besoin de communiquer avec plusieurs bases de données ?

7. Spring vs CDI

Injection de dépendances (Spring vs CDI)

Spring	CDI
Standard de fait	Standard officiel JEE
@Component , ...	@Named , ...
@Autowired ,	@Inject , ...
@Resource	@Resource
@Qualifier en tant qu'annotation simple	@Qualifier en tant que métacomment complexe
Factory via @Configuration et @Bean	Factory via @Produces
Variantes de config via profiles Spring (@Profile , application-xyz.properties)	Variantes de config via alternatives (@Default, @Altyernative et META-INF/beans.xml)
@Scope("...")	@RequestScoped, @SessionScoped, ...

En gros , du coté des fonctionnalités , ça se vaut à peu près .

Spring est plutôt plus simple mais n'est pas un standard officiel . C'est cependant un standard de fait car énormément utilisé .

Spring peut éventuellement interpréter @Inject (de DI) comme @Autowired .

CDI implémenté par Weld (de Jboss) ou OpenWebBeans (de Apache) .

Le monde JEE des années 2000-2008 avait eu tendance à abuser de la division en conteneurs (Conteneur WEB et conteneur d'EJB , ça va encore mais trop de conteneurs c'est le fiasco de la norme JBI qui a échoué sur SOA vers 2008) .

Après avoir introduit une frontière entre les conteneurs WEB et EJB , JEE s'est amusé à effacer cette frontière via CDI qui permet d'injecter un EJB dans un composant WEB !!!!

VIII - Ecosystème Spring

1. Présentation du framework Spring

1.1. Historique et évolution de Spring

Versions de Spring	Possibilités au niveau de la configuration
Depuis Spring 1.x	Configuration entièrement XML (avec entête DTD) <bean>
Depuis Spring 2.0	Configuration XML (avec entête XSD) + .properties
Depuis Spring 2.5	Annotations spécifiques à Spring (@Component , @Autowired, ...)
Depuis Spring 3.0	Compatibilité avec annotations DI (@Inject , @Named)
Depuis Spring 4.0	Java Config (@Configuration , ...) et Spring boot 1.x (avec ou sans @EnableAutoConfiguration)
Depuis Spring 5.0	restructuration interne pour mieux intégrer java 8,9,10 et un début d'architecture asynchrone et réactive (Netty , WebFlux ,) Spring Boot 2.x bien au point

Spring (historique et évolution)

Complexe et lourd

J2EE 1.x et EJB 1 & 2

JEE 5 et EJB 3.0

@Entity (JPA1.0) , @EJB

JEE 6 et EJB 3.1

JPA 2.0 , @Named , @Inject

JEE 7 et EJB 3.2

JAX-RS 2 (WS-REST)

Simple et efficace (le printemps)

Spring 1.x

2003-2007

environ

Spring 2.5

@Component , @Autowired,
@Transactional

Spring 3.x

2006-2009

environ

Spring 4.x et 5.x

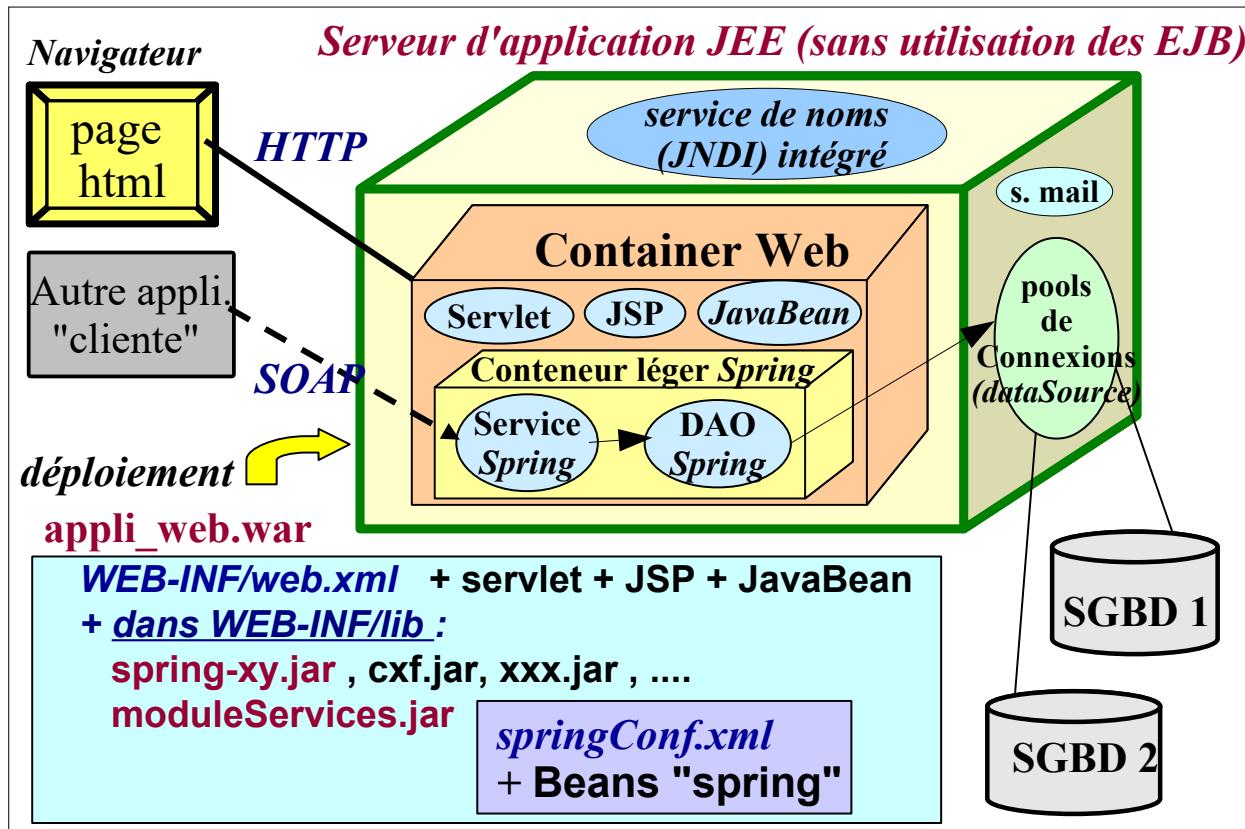
2009-2013

environ

Spring-boot , @Configuration ,
Spring-data , @RestController, ...

1.2. Architecture / Ecosystème Spring

Durant la première décennie du XXI siècle, Spring était à essentiellement considérer comme une alternative aux EJB et respectant les spécifications JEE :



Dès les premières versions, le framework open source "Spring" apportait les principales fonctionnalités suivantes :

- intégration de composants complémentaires inter-dépendants via le design-pattern "injection de dépendances / ioc". configuration souple et flexible
- prise en charge automatique et "déclarative" (via config xml ou annotations) des transactions (commit/rollback)
- intégration des principaux autres frameworks java/JEE (Hibernate/Jpa , Struts , JSF , JDBC , ...)
- intercepteurs (aop)
- tests unitaires simples (JUnit + spring-test)
- quelques éléments de sécurité (sécurité JEE simplifiée)
-

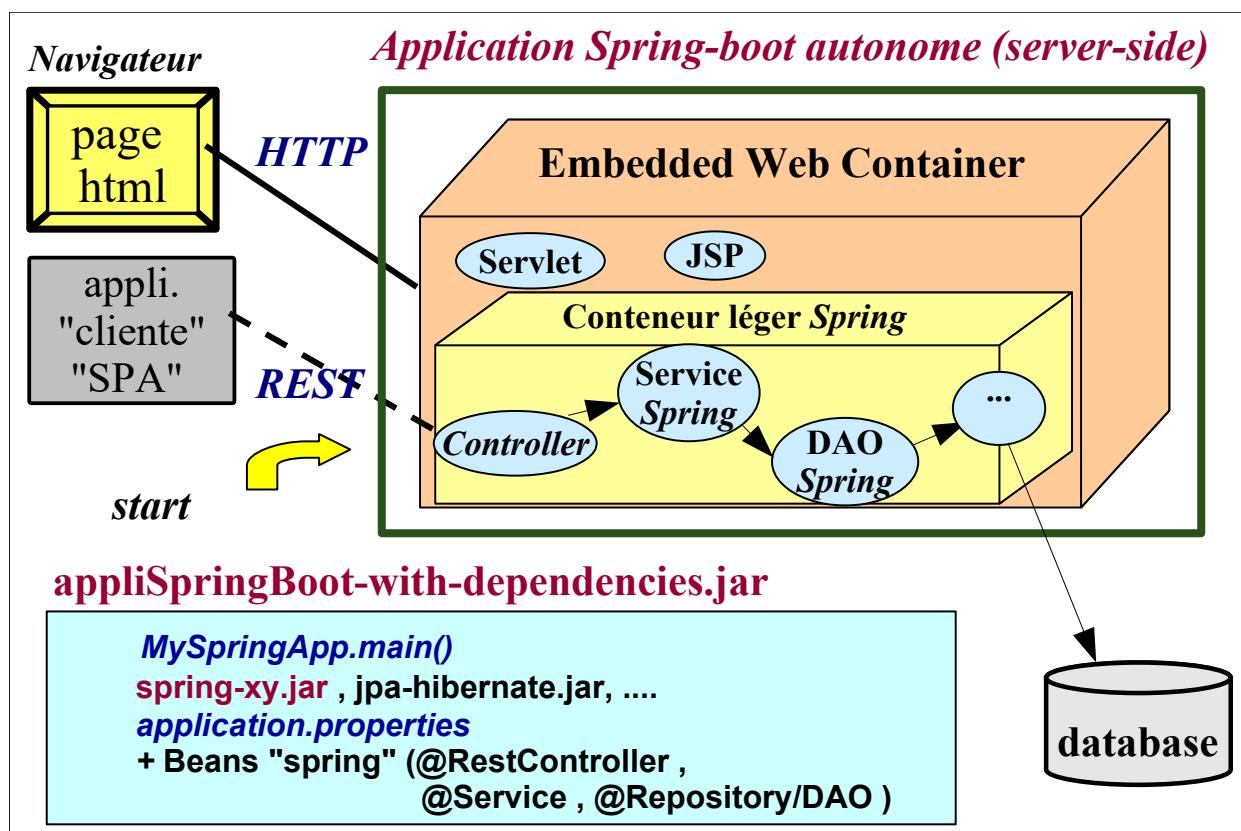
Le framework spring n'est pas associé à un grand éditeur de serveur JEE (tel que IBM , Oracle/BEA , Jboss) . Il a toujours laissé place à une très grande liberté dans le choix des technologies utilisées au sein d'une application java/JEE .

A partir de la version 4, Le framework spring a introduit tout un tas de spécificités très intéressantes qui se démarquent clairement des spécifications JEE officielles .

Principales fonctionnalités supplémentaires apportées par les versions 4 et 5 de Spring :

- **Spring boot** (démarrage complètement autonome . l'application incorpore son propre conteneur web (tomcat ou jetty ou netty ou ...)
- simplification de la configuration maven (ou gradle) via héritage de "POM/BOM/parent" .
- **Configuration java** (plus sophistiquée que l'ancienne configuration Xml , auto-complétion, rigueur , héritage , configuration conditionnelle intelligente)
- **AutoConfiguration** et simple fichier **application.properties** ou **.yml**
- **Spring Data** (composants "DAO" générés automatiquement à partir des signatures des méthodes d'une interface, implémentation possible via JPA et MongoDB , paramétrages possibles via @NamedQuery ou autres, ...)
- web services REST via **@RestController** de Spring-mvc
- sécurisation flexible via **Spring-security**
- autres fonctionnalités diverses (*actuators* : mesures de perf , ...) ,

Toutes ces fonctionnalités (bien pratiques) sont "hors spécifications JEE" et l'on peut aujourd'hui considérer que "**Spring**" forme un "**écosystème complet**" pour faire fonctionner des applications professionnelles "java/web".



1.3. Configuration Spring-boot simple

```
@SpringBootApplication
public class AppliSpringApplication {

    public static void main(String[] args) {
        SpringApplication.run(AppliSpringApplication.class, args);
        SpringApplication app = new SpringApplication(AppliSpringApplication.class);
        //EN PHASE DE DEV SEULEMENT
        app.setAdditionalProfiles("embeddedDB", "reInit", "dev");
        ConfigurableApplicationContext context = app.run(args);

        System.out.println("http://localhost:8080/appliSpringBiblio");
        //http://localhost:8080/appliSpringBiblio/index.html
        // et index.html sera trouvé dans src/main/resources/static
    }
}
```

scr/main/resources/application.properties

```
server.servlet.context-path=/appliSpringBiblio
server.port=8080

#pour demander à spring-data de générer automatiquement les classes d'implémentation des DAO/Repository
spring.data.jpa.repositories.enabled=true
```

application-dev.properties

```
logging.level.root=INFO
logging.level.tp.appliSpring=DEBUG

#ddl-auto=create pour demander (en phase de développement)
#à recréer les tables (à vide) à chaque redémarrage de l'appli (ou des tests)
spring.jpa.hibernate.ddl-auto=create
```

application-embeddedDB.properties

```
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.url=jdbc:h2:~/bibliodb
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.platform=h2
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

application-remoteDB.properties

```
spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/bibliodb?
```

```
createDatabaseIfNotExist=true&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
```

1.4. Test unitaire Spring simple

```
package tp.appliSpring.core.service;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import tp.appliSpring.core.entity.Devise;

@ExtendWith(SpringExtension.class) //si junit5/jupiter
@SpringBootTest
@ActiveProfiles({ "embeddedDB" , "dev" })
//@ActiveProfiles({ "remoteDB" , "dev" })
public class TestServiceDevise {

    private static Logger logger = LoggerFactory.getLogger(TestServiceDevise.class);

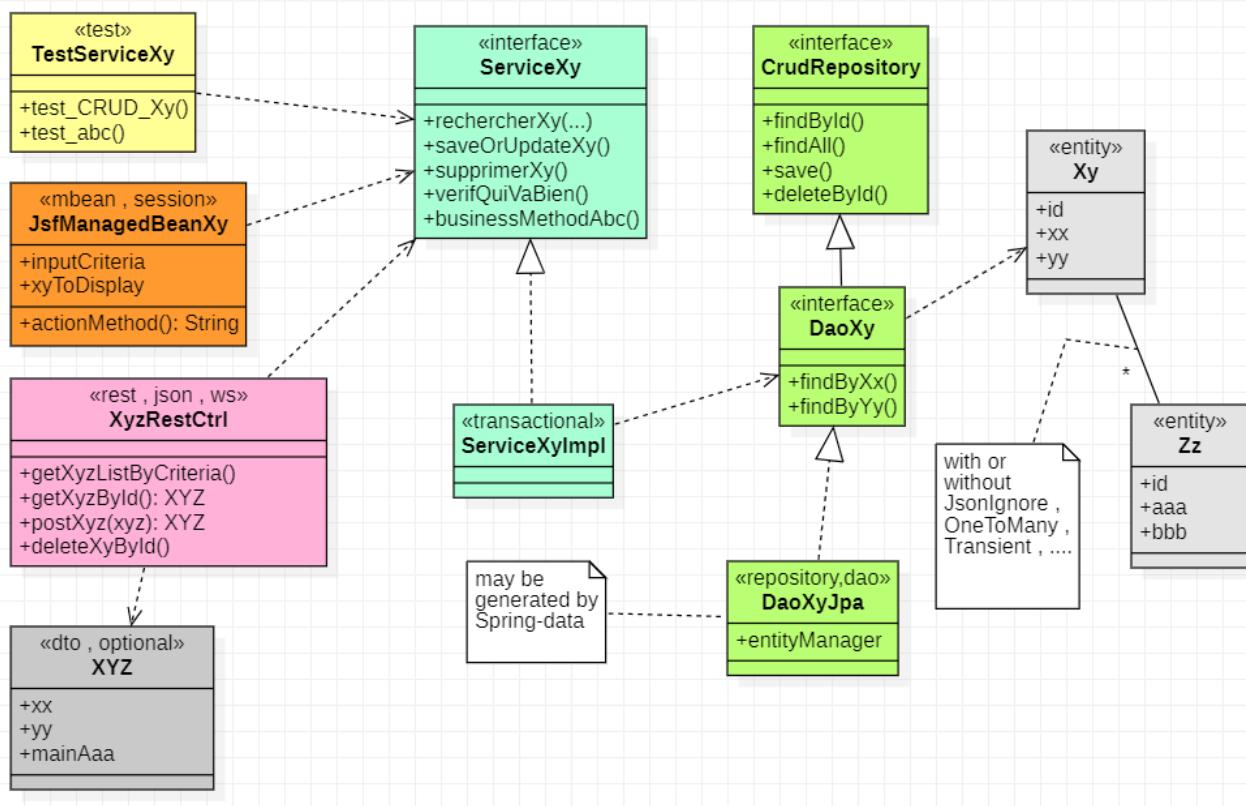
    @Autowired
    private DeviseService deviseService; //à tester

    @Test
    public void testRechercherDeviseParNumero() {
        Devise deviseEuro = new Devise("EUR","euro",1.0);
        deviseService.sauvegarderDevise(deviseEuro); //INSERT INTO

        Devise deviseRelu = this.deviseService.rechercherDeviseParCode("EUR");
        Assertions.assertEquals("euro",deviseRelu.getNom());
        Assertions.assertEquals(1.0,deviseRelu.getChange());
        logger.debug("deviseRelu=" + deviseRelu);

    }
}
```

1.5. DAO automatiques via Spring-Data



L'extension "Spring-Data" permet (entre autre) de :

- générer automatiquement des composants "DAO / Repository" modernes (utilisables avec des technologies SQL, NO-SQL ou orientées graphes telles que JPA, MongoDB, Cassandra, Neo4J, ...)
 - accélérer le temps de développement (l'interface suffit souvent, la classe d'implémentation sera générée dynamiquement par introspection et selon certaines conventions).
 - standardiser le format des composants "DAO/Repository" : mêmes méthodes fondamentales.
- On parle alors en termes de "composants DAO consistants" → des automatismes sont possibles (tests en partie automatique,).

1.5.a. Spring-data-commons

"Spring-data-commons" est la partie centrale de Spring-data sur laquelle pourra se greffer certaines extensions (pour jpa, pour mongo, ...).

"Spring-data-commons" est essentiellement constituée de 3 interfaces : **Repository**, **CrudRepository** et **PagingAndSortingRepository**.

- **Repository<T, ID>** n'est qu'une interface de marquage dont toutes les autres héritent.
- **CrudRepository<T, ID>** standardise les méthodes fondamentales (`findById`, `findAll`, `save`, `delete`, ...)
- **PagingAndSortingRepository<T, ID>** étend CrudRepository en ajoutant des méthodes

supportant le tri et la pagination.

Méthodes fondamentales de **CrudRepository**<T ,ID extends Serializable> :

<S extends T> S save (S entity);	Sauvegarde l'entité (au sens saveOrUpdate) et retourne l'entité (éventuellement ajustée/modifiée dans le cas d'une auto-incrémentation ou autre).
Optional<T> findById (ID primaryKey);	Recherche par clef primaire (avec jdk >= 1.8)
Iterable<T> findAll ();	Recherche toutes les entités (du type courant/considéré)
Long count ();	Retourne le nombre d'entités existantes
void delete (T entity);	Supprime une (ou plusieurs) entités
void deleteById (ID primaryKey);	
void deleteAll ();	
boolean exists (ID primaryKey);	Test l'existence d'une entité

Dépendance maven indirecte (avec spring-boot):

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Activation via application.properties (autoConfiguration) :

```
spring.data.jpa.repositories.enabled=true
```

Exemple d'interface de DAO/Jpa avec **JpaRepository** (héritant lui même de **CrudRepository**):

```
interface UserRepository extends CrudRepository<User, Long> {
    List<User> findByLastname(String lastname);
}
```

La classe d'implémentation sera générée automatiquement (si `@EnableJpaRepositories` ou si `<jpa:repositories base-package="..." />`)

il suffit d'une injection via `@Autowired` ou `@Inject` pour accéder au composant DAO généré .

Conventions de noms sur les méthodes de l'interface :

find...By, **read...By**, **query...By**, **get...By** and **count...By**,

Exemples :

```

List<User> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

// Enables the distinct flag for the query
List<User> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
List<User> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

// Enabling ignoring case for an individual property
List<User> findByLastnameIgnoreCase(String lastname);

// Enabling ignoring case for all suitable properties
List<User> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

// Enabling static ORDER BY for a query
List<User> findByLastnameOrderByFirstnameAsc(String lastname);
List<User> findByLastnameOrderByFirstnameDesc(String lastname);

```

methodNameWithKeyWords(?1,\$2,...)

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	... where x.lastname = ?1 or x.firstname = ?2
	FindByFirstname,	
Is, Equals	findByFirstnameIs, findByFirstnameEquals	... where x.firstname = ?1
Between	findByStartDateBetween	... where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	... where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	... where x.age <= ?1
Greater Than	findByAgeGreaterThan	... where x.age > ?1
Greater Than Equal	findByAgeGreaterThanOrEqual	... where x.age >= ?1
After	findByStartDateAfter	... where x.startDate > ?1
Before	findByStartDateBefore	... where x.startDate < ?1
IsNull	findByAgeIsNull	... where x.age is null

Keyword	Sample	JPQL snippet
IsNotNull,	<code>findByAge(Is)NotNull</code>	<code>... where x.age not null</code>
NotNull		
Like	<code>findByFirstnameLike</code>	<code>... where x.firstname like ?1</code>
NotLike	<code>findByFirstnameNotLike</code>	<code>... where x.firstname not like ?1</code>
StartingWith	<code>findByFirstnameStartingWith</code>	<code>... where x.firstname like ?1 (parameter bound with appended %)</code>
EndingWith	<code>findByFirstnameEndingWith</code>	<code>... where x.firstname like ?1 (parameter bound with prepended %)</code>
Containing	<code>findByFirstnameContaining</code>	<code>... where x.firstname like ?1 (parameter bound wrapped in %)</code>
OrderBy	<code>findByAgeOrderByLastnameDesc</code>	<code>... where x.age = ?1 order by x.lastname desc</code>
Not	<code>findByLastnameNot</code>	<code>... where x.lastname <> ?1</code>
In	<code>findByAgeIn(Collection<Age> ages)</code>	<code>... where x.age in ?1</code>
NotIn	<code>findByAgeNotIn(Collection<Age> age)</code>	<code>... where x.age not in ?1</code>
True	<code>findByActiveTrue()</code>	<code>... where x.active = true</code>
False	<code>findByActiveFalse()</code>	<code>... where x.active = false</code>
IgnoreCase	<code>findByFirstnameIgnoreCase</code>	<code>... where UPPER(x.firstname) = UPPER(?1)</code>

Paramétrage par défaut de JpaRepositories :

CREATE_IF_NOT_FOUND (default) combines CREATE and USE_DECLARED_QUERY

→ **on peut donc éventuellement personnaliser l'implémentation des méthodes.**

Utilisation de **@NamedQuery** à coté de **@Entity** (ou **<named-query ...>** dans orm.xml) :

Dans orm.xml (référencé par META-INF/persistence.xml ou ...) :

```
<named-query name="User.findByLastname">
    <query>select u from User u where u.lastname = ?1</query>
</named-query>
```

et/ou dans la classe d'entité persistante :

```
@Entity
@NamedQuery(name = "User.findByEmailAddress",
    query = "select u from User u where u.emailAddress = ?1")
public class User {

}
```

```
public interface UserRepository extends JpaRepository<User, Long>
{
    List<User> findByLastname(String lastname);

    User findByEmailAddress(String emailAddress);
}
```

Utilisation (un peu radicale) de @Query (de Spring Data) dans l'interface :

Sémantiquement peu être un trop peu radical pour une interface !!!

Exemple :

```
public interface UserRepository extends JpaRepository<User, Long> {

    @Query("select u from User u where u.emailAddress = ?1")
    User findByEmailAddress(String emailAddress);
}
```

==> et encore beaucoup d'autres possibilités / options dans la [doc de référence de spring-data](#) .

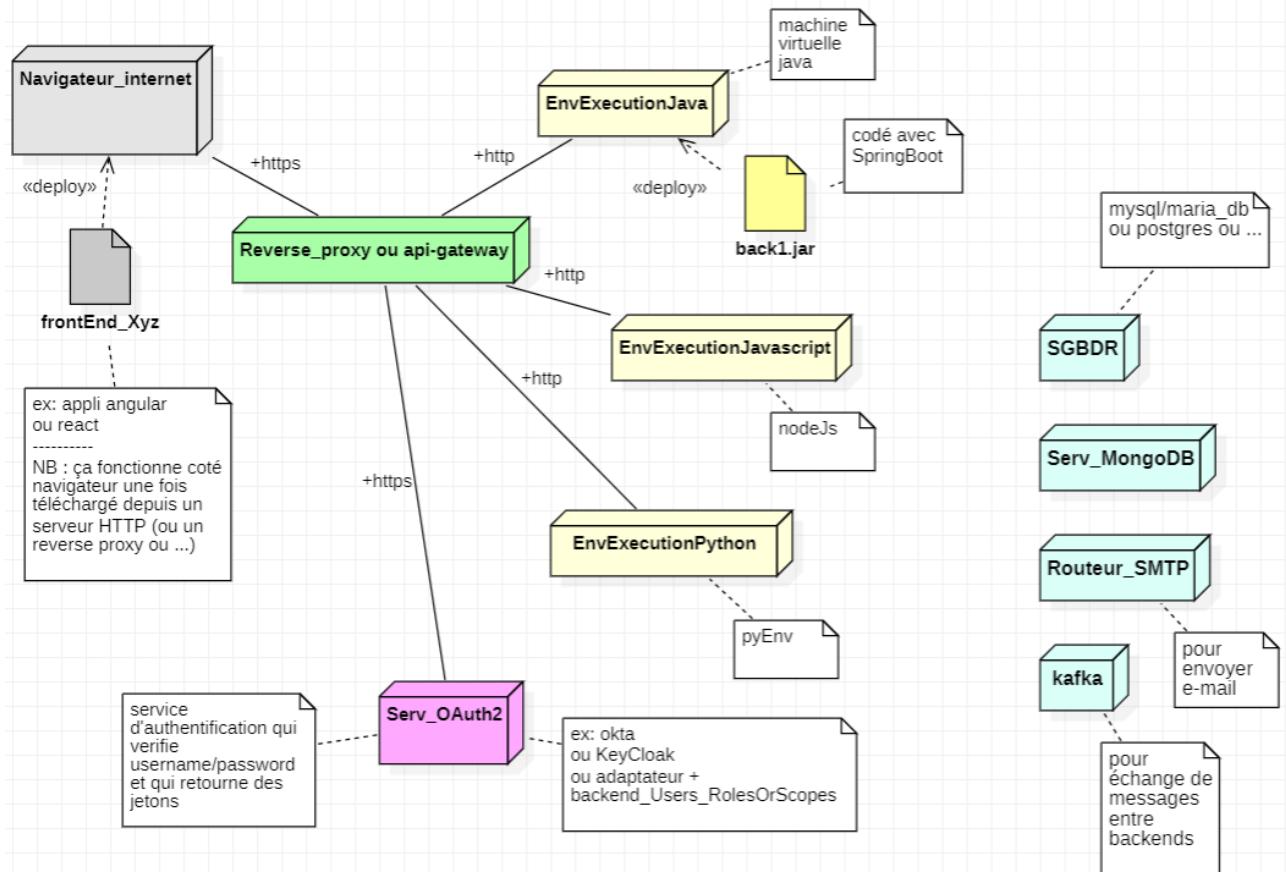
1.6. Quelques différences/correspondances entre Spring et JEE officiel

JEE Officiel	Equivalent Spring
WS-REST via JAX-RS	WS-REST via Spring-MVC et @RestController
Intercepteur sur EJB	SpringAop
Sécurité JEE	SpringSecurity

IX - Cloud java et microservices

1. Architecture micro-services

Architecture micro-services



1.1. Grands traits de l'architecture micro-services

Caractéristiques structurelles d'un micro-service

- **forte cohésion interne , périmètre fonctionnel très réduit**
(faire une seule chose et le faire bien/efficacement)
- **idéale indépendance** (conçus et développés indépendamment , testés et déployés *indépendamment* des autres (dans micro-conteneurs)
--> agilité / souplesse technologique)
- **beaucoup de délégation** (très grande sollicitation des réseaux informatiques)
- "**brique de code simple**" éventuellement packagée et déployée dans topologie "compatible cloud" sophistiquée
(ex : conteneur docker déployable dans un cluster)

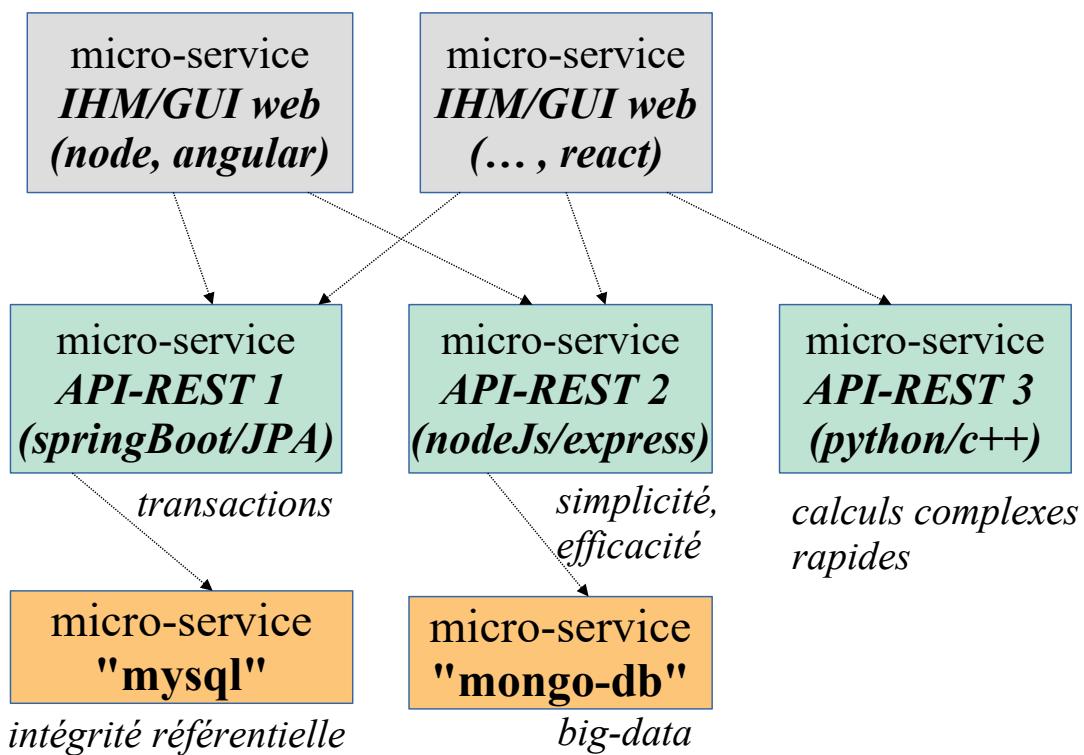
Agilité/souplesse technologique

- Mis à part quelques petits éléments protocolaires généralement imposés (tcp/ip , ...) , **un micro-service peut être programmé dans une technologie quelconque** (java , javascript, python, ...) car toutes les librairies et configurations spécifiques seront cachées/isolées au sein de micro-conteneurs opaques .
- *chaque équipe de développement peut donc choisir librement la technologie la plus appropriée pour développer son micro-service*

(ex : java/jee/spring-boot pour transactions courtes
nodeJs/express pour simplicité/efficacité,
base relationnelle (mysql, ...) pour données référentielles
mongoDB/... pour données en grand volume (big data)
elastic-search pour gestion historique et recherches
python et c/c++ pour service scientifique (calculs , ...)

...

Variétés technologiques des micro-services



Quelques impacts de l'architecture micro-service

- "périmètre réduit" + "liberté dans technologie de mise en oeuvre" font que :
 - les risques sont plus limités/circonscis .
(moins de problème d'incompatibilité entre technologies, si un sous projet échoue on change de technologie et on recode mieux le micro-service à problème . ce n'est pas l'ensemble qui est compromis)
 - beaucoup de liberté dans le choix des frameworks , langages et serveurs :
---> **marché des technos très ouvert** (ça bouillonne , ça part dans tous les sens , orientation "open source", les grands éditeurs "Microsoft, IBM, ..." nous imposent moins leurs "solutions coûteuses").
 - > **développeurs devant de plus en plus être polyglottes !!!** (*plein de langages/technos à apprendre/maîtriser !!!*).

Autres impacts de l'architecture micro-service

- **Tests d'intégration facilités :**
 - certains micro-services sont dépendants d'autres micro-servives en arrière plan (ex : api-rest nécessitant accès database)
 - une fois packagé sous forme de conteneur "docker" , un micro-service d'arrière plan pourra:
 - être complexe/sophistiqué en interne (ex : calculs , ...).
 - être vu comme une "boîte opaque" rendant un certain service utile (un minimum documenté)
 - être utilisé au sein de test d'intégration (alias "end-to-end") pour vérifier la bonne communication inter-service dès la phase de "pré-released" / "pré-production" du micro-service appelant .
 - prévoir peut être variantes/profils "avec ou sans cluster" durant les phases de tests (avec complexité progressive).

Factorisation et ré-utilisation des micro-services

Trouver le bon niveau de découpage / décomposition sachant que :

- un service simple/réduit peu plus facilement être réutilisé
- trop de délégations peuvent induire un très grand trafic réseau et un ralentissement dû aux sérialisations/dé-sérialisations des requêtes / réponses
- certaines données sont plus simples à relier/corréler localement qu'en mode "dispersé" : il faut idéalement modéliser (avec UML ou autre) pour étudier les aspects "cohésion non décomposable" et "indépendances possibles".
- ne pas hésiter à restructurer d'une version à l'autre (le "refactoring" n'est jamais un objectif mais c'est un "moyen souvent nécessaire à mettre en oeuvre").
- orchestration simple = "bonne idée de SOA à reprendre"

Résilience (tolérance panne partielle)

Au sein d'une architecture micro-service , l'inaccessibilité ponctuelle et les pannes partielles sont considérées comme des phénomènes normaux / courants .

Pour qu'un micro-service puisse continuer à fonctionner (éventuellement en mode "service légèrement dégradé") dans le cas où un élément dont il dépend ne fonctionne plus bien , il faudra anticiper des "plans B".

--> **Choisir des technologies "clusterisables"** (avec "load-balancing" , "rétablissement de connexions" , ...)

--> **Bien gérer les exceptions** (des 2 cotés : client et serveur)

--> **Bien paramétrier l'infrastructure hôte** (ex : "Kubernetes") qui dit coopérer avec la technologie "clusterisable" (ex : mysql, mongo)

Extensibilité/élasticité/scalabilité selon orientation "stateless"

D'emblée prévus pour l'infrastructure "cloud", les micro-services doivent pouvoir s'adapter à des redimensionnement de voilure (changement de la taille d'un cluster).

Ceci ne pose aucun problème pour les micro-services de type "API stateless" (ex : Web Service de calcul ,) mais pose de très nombreux problèmes sur l'aspect "cohérence des données réparties/dispersées" :

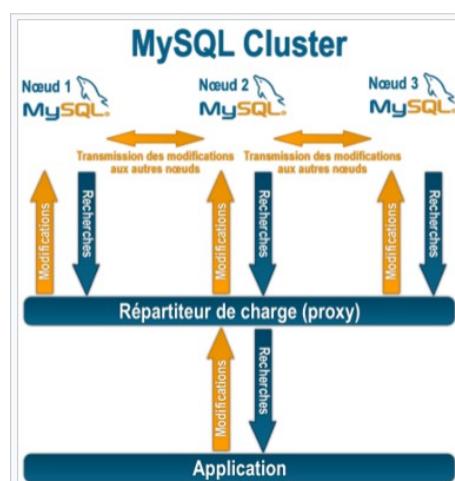
- théorème "CAP" spécifiant que l'on ne peut pas avoir en même temps les aspects "distribué / tolérance de panne" , "disponibilité / réponses quasi immédiates" et "transactions parfaites ACID" . il faudra privilégier un aspect prioritaire par rapport aux autres .
- intégrité référentielle en mode distribué/dispersé (cohérence entre données fonctionnelles/métiers réparties dans plein de petites bases?)

Cluster de données avec priorité aux écritures cohérentes

Lorsque l'on fait fonctionner "mysql" en cluster sur différentes machines (avec "load-balancing" et "fail-over") ,

- les *lectures simultanées* peuvent alors être *déclenchées en grand nombre* (relativement rapidement)

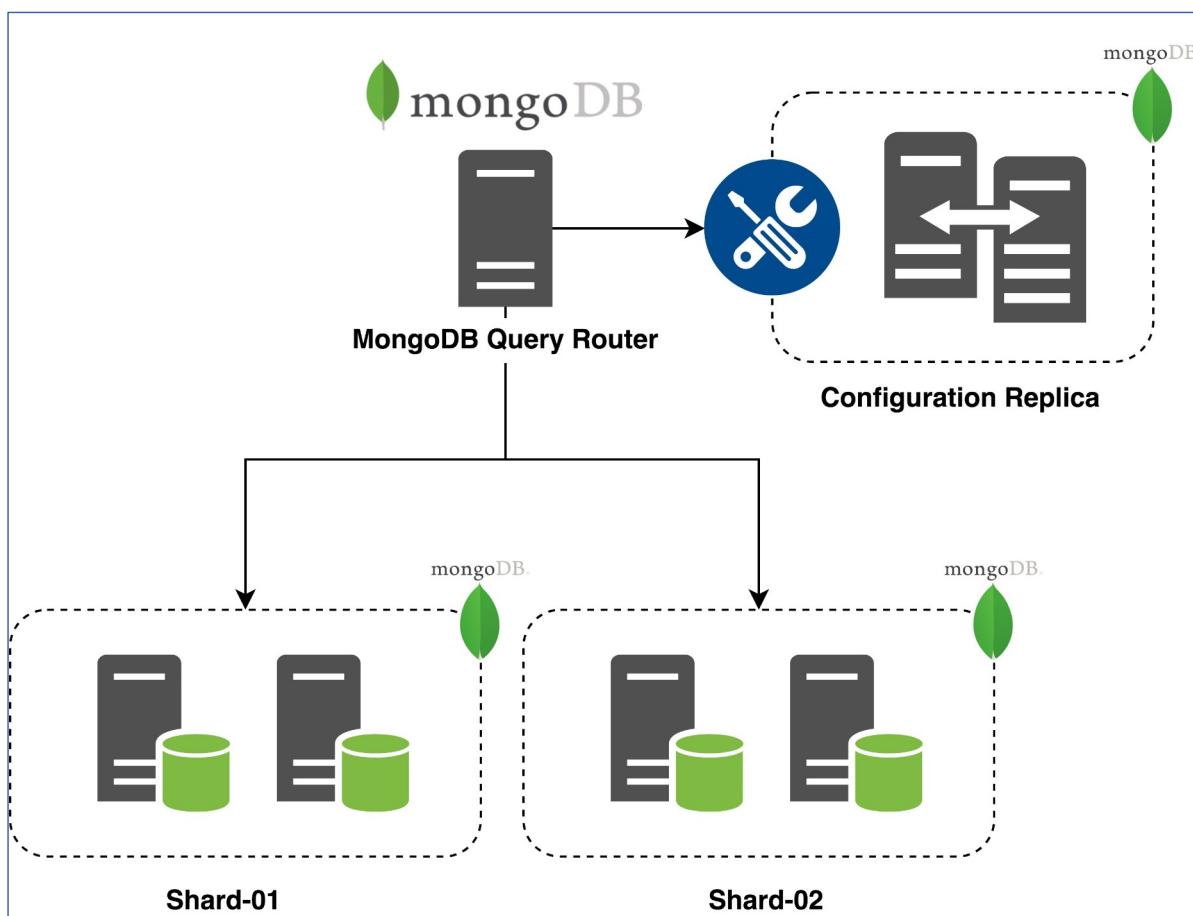
- les *écritures sont par contre ralenties* (car le système relationnel privilégie la cohérence des données écrites/modifiées via une réplication immédiate synchronisée vers les autres membres actifs du cluster)



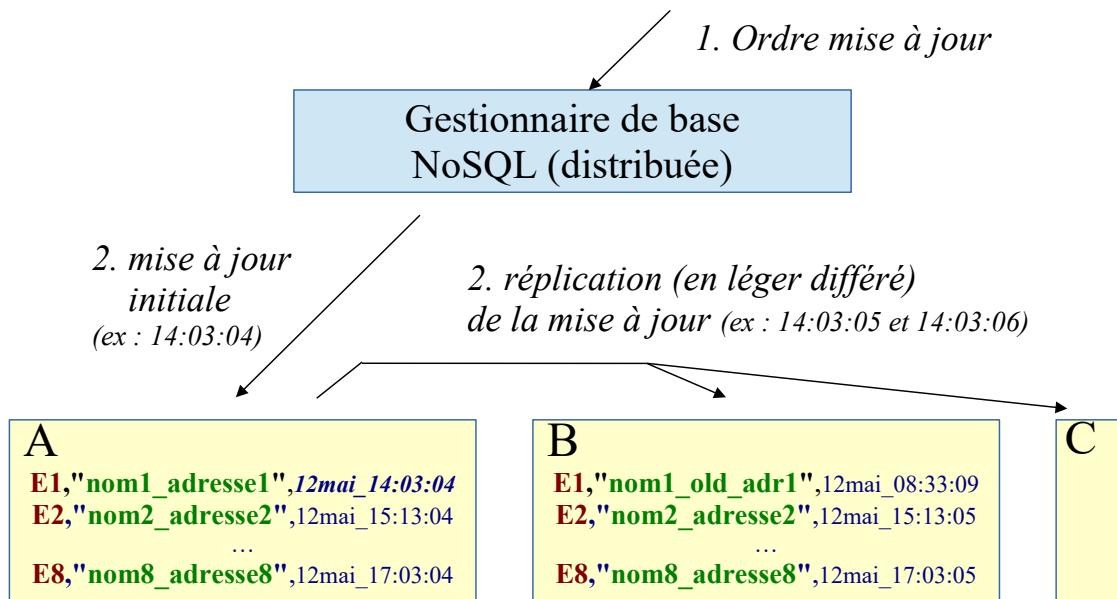
Cluster de données avec priorité aux écritures rapides et lectures immédiates (avec consolidation différée)

Lorsque l'on fait fonctionner certaines technologies "noSQL" (ex : "mongo db") en cluster sur différentes machines (avec "load-balancing" et "fail-over") ,

- les écritures sont quelquefois (selon le niveau transactionnel paramétré) effectuées relativement rapidement et répliquées de façon différée vers les autres membres actifs du cluster
- des lectures simultanées effectuées/redirigées vers différents membres du cluster peuvent quelquefois retourner des valeurs différentes (lorsque la réPLICATION différée n'est pas terminée)
NB : ce manque de précision/actualisation des valeurs lues est quelquefois "pas gênante" dans certains contextes (ex : données statistiques , big-data , ...)



Basic NoSQL (**key, value**, *timeStamp*)



Les synchronisations/réPLICATIONS peuvent s'effectuer dans tous les sens (A-->B,C ou B-->A,C , ...) . L'information technique/cachée "timeStamp" (ex : *12mai_14:03:04*) permet de connaître quelle est la version la plus à jour (la plus récente).

1.2. Docker , kubernetes , ...

Bonne exploitation des ressources matérielles

L'architecture "micro-service dans micro-conteneur" permet (avec de bons réglages) de généralement mieux exploiter les ressources matérielles (CPU, RAM, ...) qu'une architecture "application mono-bloc".

- une application mono-bloc (avec plusieurs couches logicielles "GUI" + "Tx" + "Accès données") comporte généralement quelques "goulets d'étranglement" difficiles à gérer/optimiser .
- il faut souvent surdimensionner la puissance de la machine hôte pour être certain d'anticiper certains pics de charge.

- un micro service correspond généralement à un seul "tiers" (soit "GUI" , soit "Api REST" , soit) et la consommation en ressource matérielle est plus régulière/homogène . En multipliant le déploiement de conteneurs identiques sur la même machine on peut ainsi assez bien exploiter la puissance d'une machine sans trop la sous-exploiter .

1.3. Docker et notion de conteneur

Micro-conteneurs "Docker"



Technologie légère et efficace de virtualisation
avec comme principaux apports :

- **Coûts optimisés** (car moins consommateur)
- **Déploiements rapides**
- **Portabilité** (linux/windows , isolation interne , compatibilité externe)

Principes de la "conteneurisation"



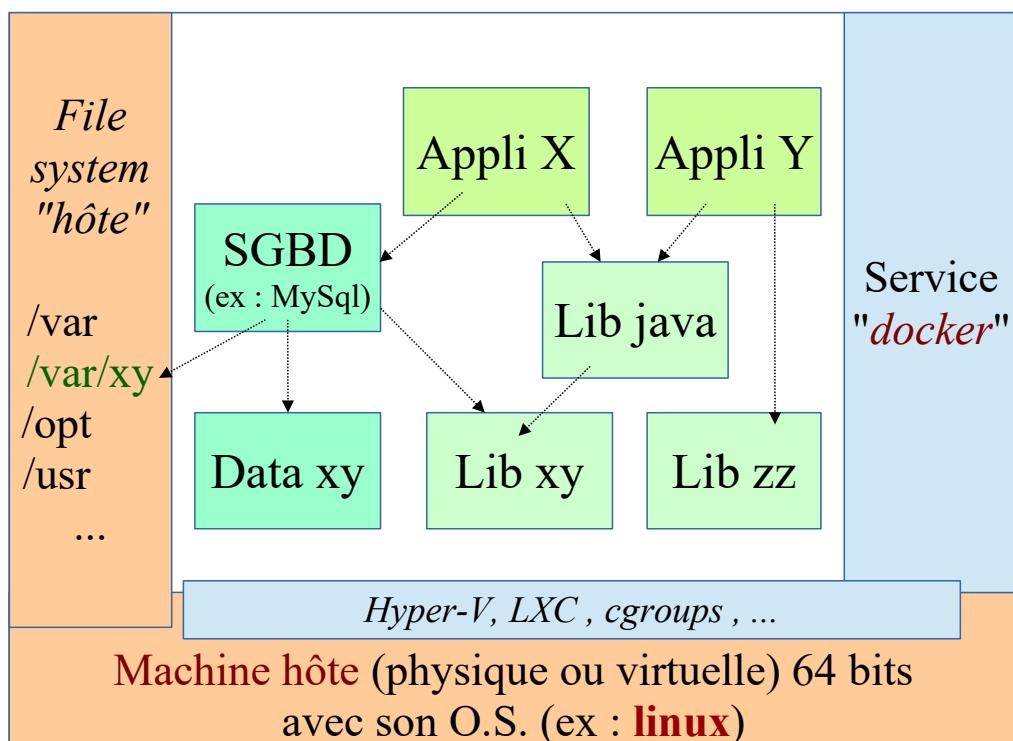
Vis à vis de plusieurs conteneurs co-localisés, permet un partage d'un système d'exploitation hôte unique, avec ses ressources (*fichiers binaires, librairies, pilotes de périphériques, mémoire vive, ...*)

Conteneur = environnement d'exécution

logicielle **complet** comportant :

- micro o.s. (debian, centos ou ...)
- librairies / dépendances
- logiciel de base (ex : mysql , node , ...)
- configuration logicielle
- le code d'une application ou d'un service

DOCKER

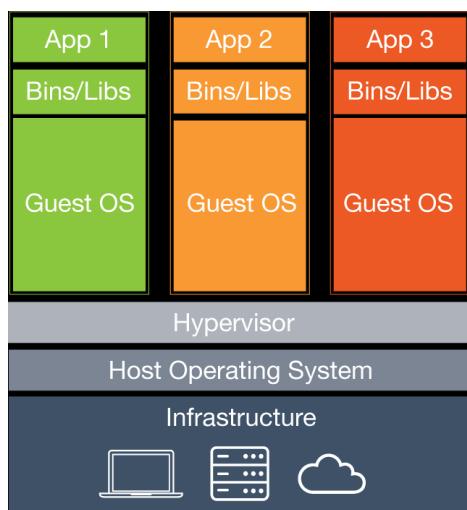


1.4. "Container" vs "VM"

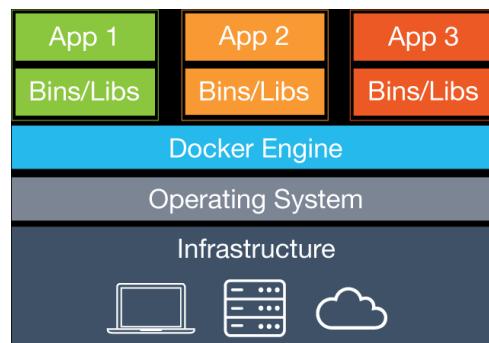
Bien que proche d'une VM (Machine Virtuelle) , Docker n'est pas une VM . Sa technologie de "Container" permet de bien mieux partager les ressources d'une machine physique (RAM , CPU , ...)

Evolution de la virtualisation ("VM" → "Conteneur")

V.M. (lourdes)
avec "*O.S. complets internes*"
gérées par hyperviseur
(ex : VmWare , VirtualBox)



Conteneurs (très légers)
avec "applications et
dépendances"
gérées moteur de conteneurs
(ex : Docker)



A bas niveau l'ordinateur (cpu/ ram / **bios**, ...) doit être prévu pour partager les ressources matérielles entre plusieurs conteneurs lorsqu'il sera en partie contrôlé par l' **hyperviseur "docker"** .

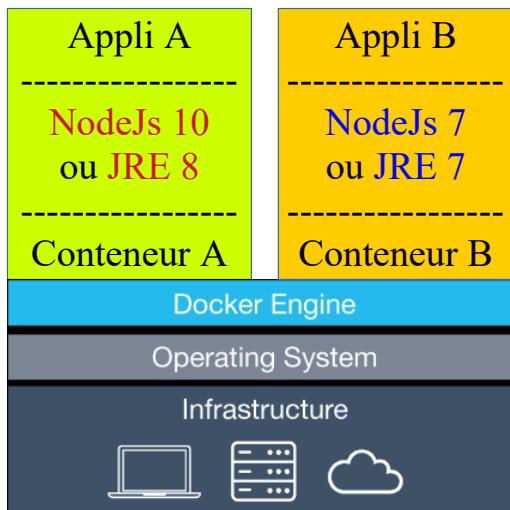
Chaque conteneur sera isolé des autres d'un point de vu "logique" et "fonctionnement interne". Par contre, les différents "conteneurs" qui vont s'exécuter sur un même ordinateur vont partager très efficacement les ressources de l'ordinateur (RAM, CPU, ...) :

- **très faible sur-consommation mémoire** (*comparée à celle d'une VM*)
- **démarrage très rapide** (*comparé à celui d'une VM*)

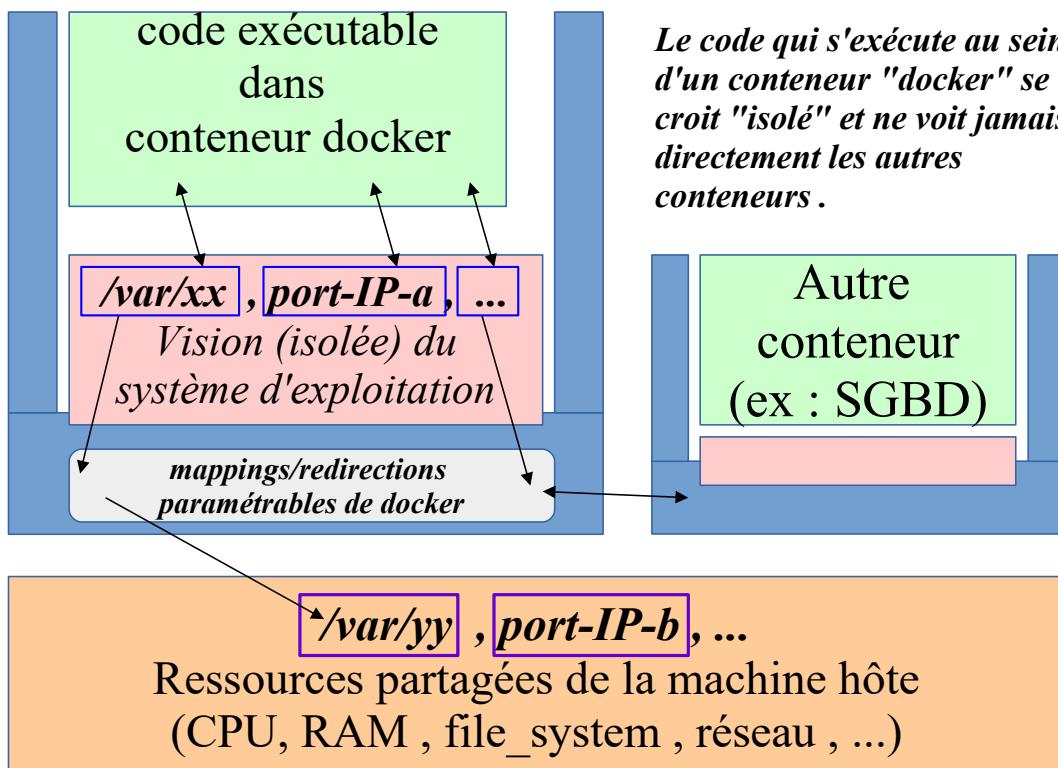
NB : Etant donné que toutes les distributions linux partagent un même type de noyau , un conteneur docker "debian" peut sans problème fonctionner sur un système "redHat" ou "centOs" et vice versa.

1.5. Isolation des conteneurs

Isolation des conteneurs



"Isolation virtuelle"

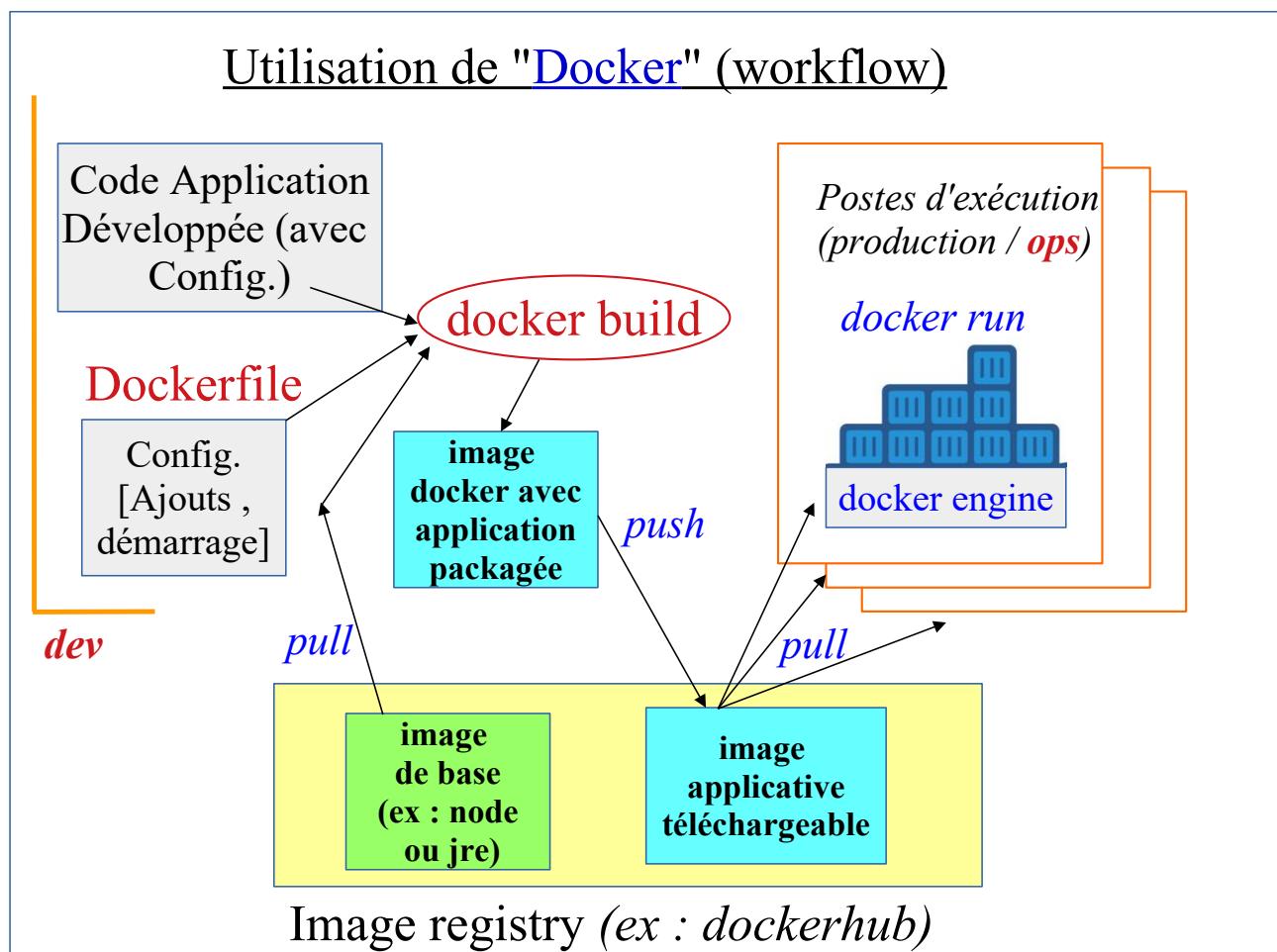


1.6. Référentiel d'images "docker" prêtes à l'emploi

Le site <https://hub.docker.com> permet de télécharger (gratuitement) toute une **série d'images "docker" prêtes à être installée sur n'importe quelle distribution linux** (fedora, redhat , debian , ubuntu , ...).

On y trouve une centaine d'images officielles ("debian" , "mysql" , "postgresql" , "mongo" , "node" , "tomcat" , "jenkins" , ...) et une multitudes d'images "non officielles".

1.7. "Docker workflow" (utilisation classique)



1.8. Spécificités de "docker"

Spécificités d'un conteneur "docker"

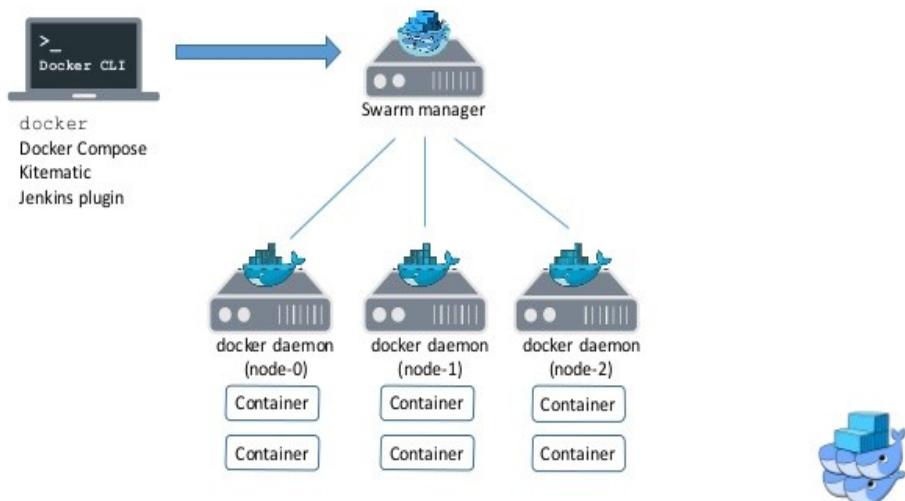
Points forts :

- démarrage assez rapide
- instanciations , démarrages , arrêts , suppressions facilement contrôlables à distance (via api REST) . Ce qui apporte beaucoup de souplesse dans la gestion des ressources (vite allouées, vites libérées)
- isolation , portabilité (tous linux et bientôt windows)

Points délicats (à avoir en tête)

- tailles de certaines images quelquefois importes (presque 1Go)
- nécessite réseau à relativement haut débit
- attention aux conflits potentiels entre les numéros de ports (heureusement reconfigurable) entre les éléments internes de différents conteneurs co-localisés.

Diversité des solutions "cloud" : "docker swarm"



1.9. Kubernetes

Kubernetes (k8s)

Kubernetes signifie "timonier" ou "pilote" en langue grecque .



kubernetes

Kubernetes (k8s) est une **plate-forme logicielle** permettant d'automatiser le déploiement, la montée en charge et la mise en œuvre de conteneurs d'application sur des clusters de serveurs .

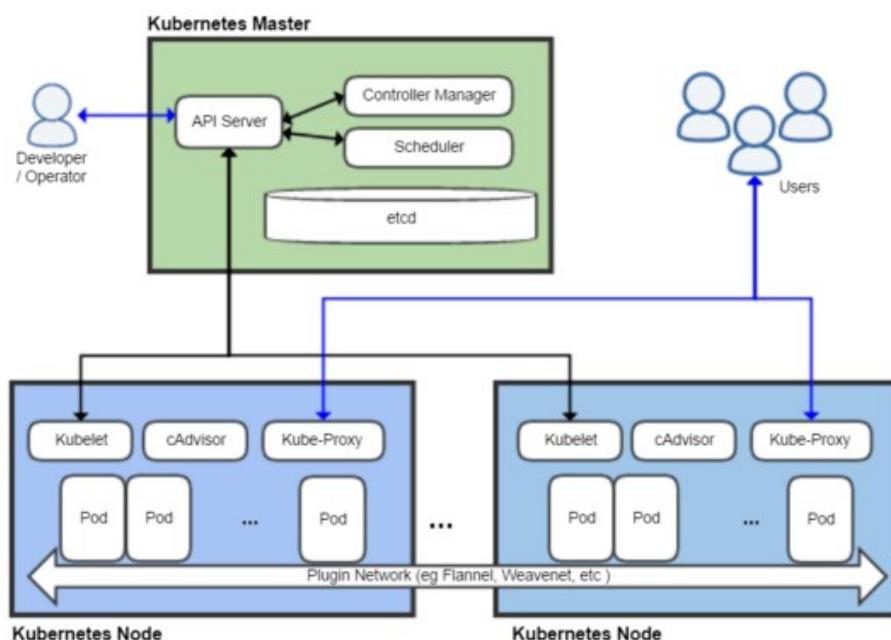
Kubernetes est très souvent utilisé avec des conteneurs "**docker**" mais il a été cependant conçu pour fonctionner avec d'autres sortes de conteneurs .

Kubernetes s'inscrit à fond dans la tendance architecturale "**DevOps , Cloud, micro-services**"

Historique , utilisation et évolution de Kubernetes

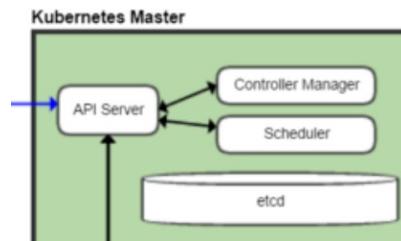
- Annoncée par **Google** en 2014 , la première version de kubernetes sort en **2015** . Kubernetes est le résultat de la réécriture en langage "**GO**" d'un ancien système "borg" permettant de gérer certaines infrastructures de "Google"
- D'origine "Google" , kubernetes a été offert à la "**Cloud Native Computing Foundation**" qui le gère actuellement en tant que projet "**open source**".
- **Kubernetes** *sert actuellement à arrêter et redémarrer des millions de containers chaque semaine.*
- Des services comme **Gmail** ou **Map** tournent dans des conteneurs gérés par Kubernetes
- En **2018**, **Kubernetes** représentait **51% du marché des orchestrateurs de conteneurs** contre **11% pour Docker Swarm** et **4% pour Mesos** .

Diversité des solutions "cloud" : Kubernetes



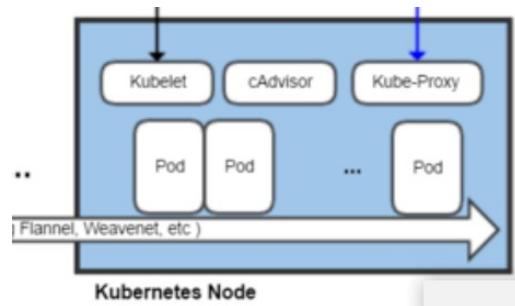
NB : Un "pod kubernetes" est un ensemble de "conteneurs co-localisés" (docker ou autres)

Composants internes de "kubernetes" sur une machine "maître"



- **etcd** = **base de données** distribuée et légère (clefs/valeurs) utilisée pour mémoriser de façon persistante la **configuration du cluster** .
- **API-Server** = Server d'api REST/JSON servant à contrôler le cluster (principal client = kubectl) .
- **Controller Manager** = Gestionnaire de "contrôleurs/superviseurs" (de replicas, de pods, de ...)
- **Scheduler** (ordonnanceur) : contrôle l'aspect "auto-adaptatif" d'un cluster kubernetes (ordres de "lancement / arrêt" selon états des noeuds : charge limite saturation, ...), S'occupe d'assigner un noeud disponible à un pod.
- **Service DNS** : intégré à Kubernetes, les conteneurs utilisent ce service dès leur création pour la résolution de nom à l'intérieur du cluster.

Composants internes de "kubernetes" sur une machine "worker"



- **Kubelet** (agent principal) : Kubelet est responsable de l'état d'exécution de chaque nœud (c'est-à-dire, d'assurer que tous les conteneurs sur un nœud sont en bonne santé). Il prend en charge le démarrage, l'arrêt, et la maintenance des conteneurs d'applications (organisés en pods)
- **cAdvisor** : agent qui surveille et récupère les données de consommation des ressources et des performances comme le processeur, la mémoire, ainsi que l'utilisation disque et réseau des conteneurs de chaque node.
- Le **kube-proxy** est l'implémentation d'un proxy réseau et d'un répartiteur de charge, il gère le service d'abstraction ainsi que d'autres opérations réseaux

1.9.a. Accès externe à un cluster kubernetes via LB7 / Ingress

Un "ingress" est un point d'entrée (virtuellement ou ...) unique (address Ip , num port) qui peut rediriger vers différents services internes via un aiguillage tenant compte d'un nom de domaine ou d'une fin d'url. Un peu comme "virtualHost" mais avec loadBalancing en plus .

Le terme anglais "*ingress*" peut se traduire en français par *pénétration, infiltration ou entrée* .

Exemple de fin d'url(analysé par ingress pour aiguillage) :

(en dev: http://hello.default.192.168.33.31.xip.io/ , http://s2.default.192.168.33.31.xip.io/)

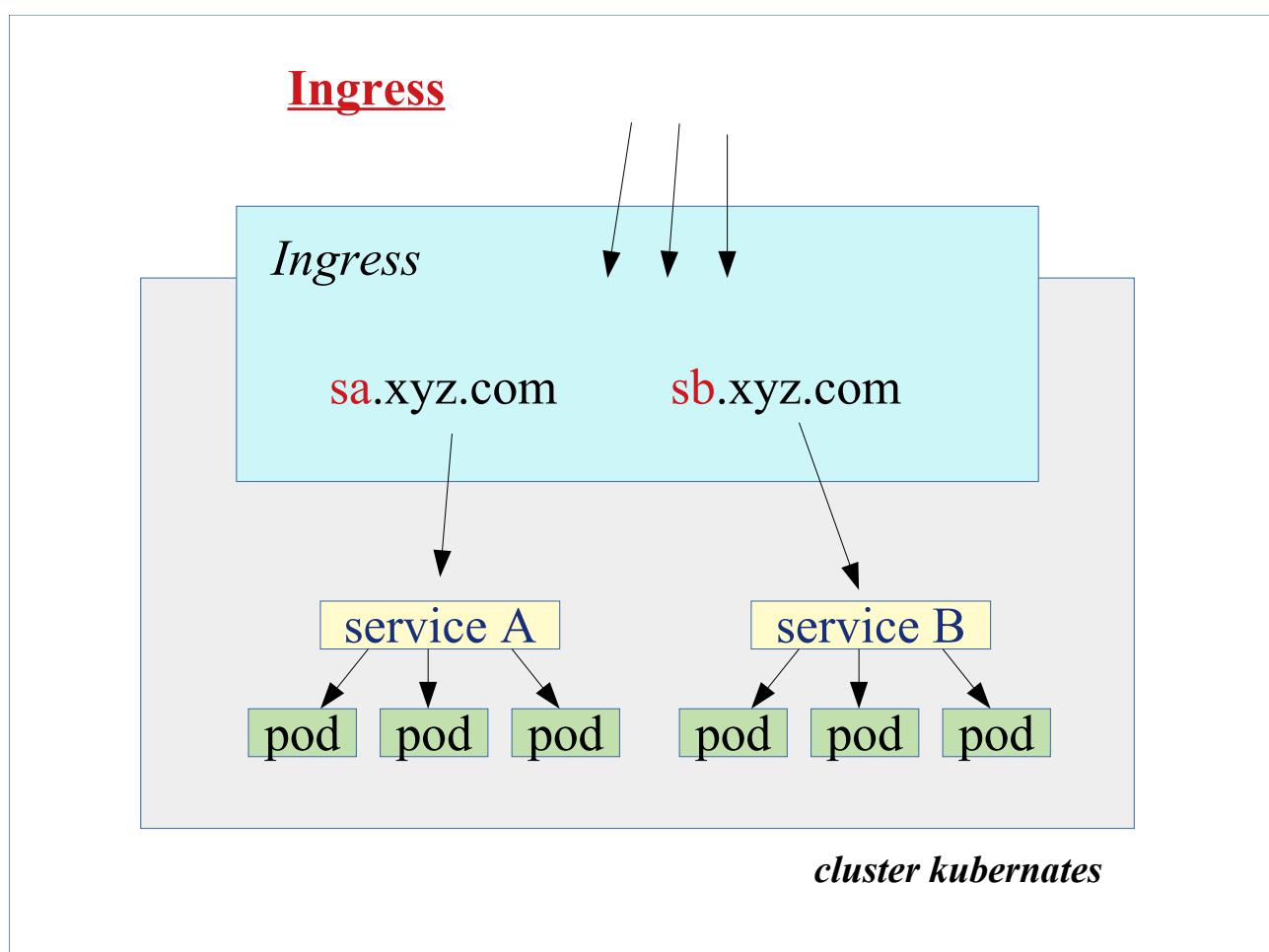
(en prod : http://www.xxx.com/s1 , http://www.xxx.com/s2 , ...)

...

NB :

Un "ingress" correspond à un "load-balancing de niveau 7 (LB7)"

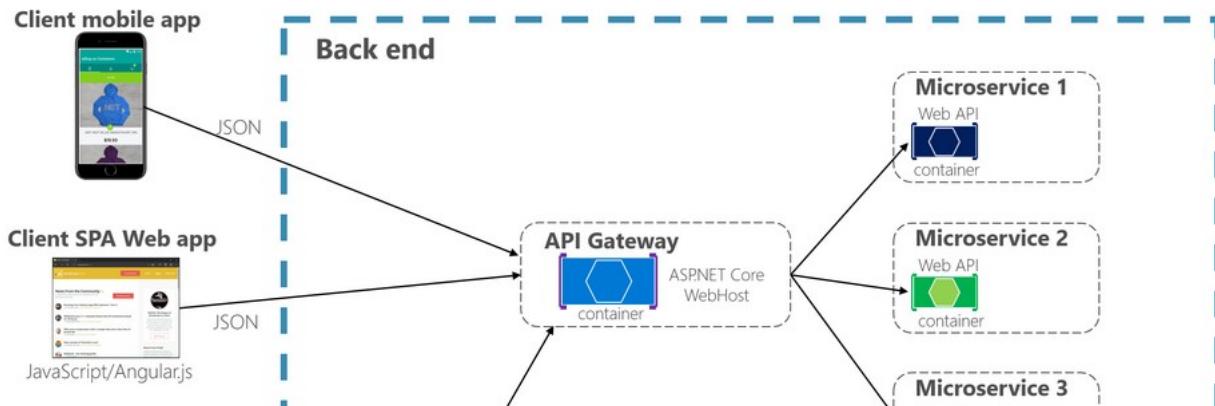
Un "ingress" est une vision abstraite d'un point d'entrée (avec aiguillage LB7) et beaucoup d'implémentations existent (purement logicielles ou pas , intégrée au fournisseur cloud ou pas, ...) Un "ingress" peut par exemple être géré par nginx-ingress .



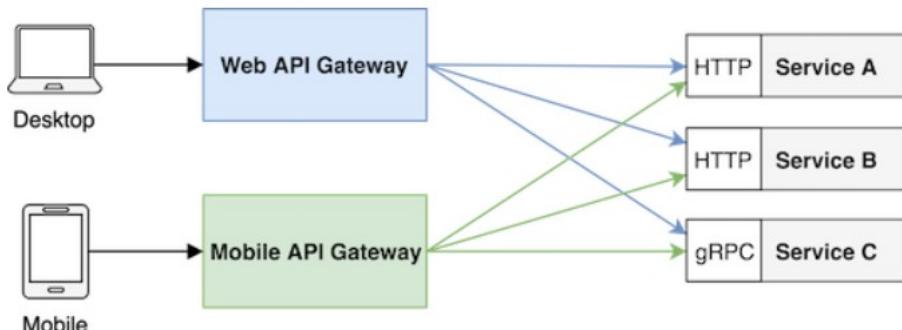
1.10. Api-gateway

Le réseau interne (intranet) d'une entreprise est déjà en soi un domaine à bien sécuriser. Lorsque certains services sont exposés à l'extérieur du "S.I.", il faut prendre encore plus de précautions (attaques de toutes sortes : force brute, déni de service, ...)

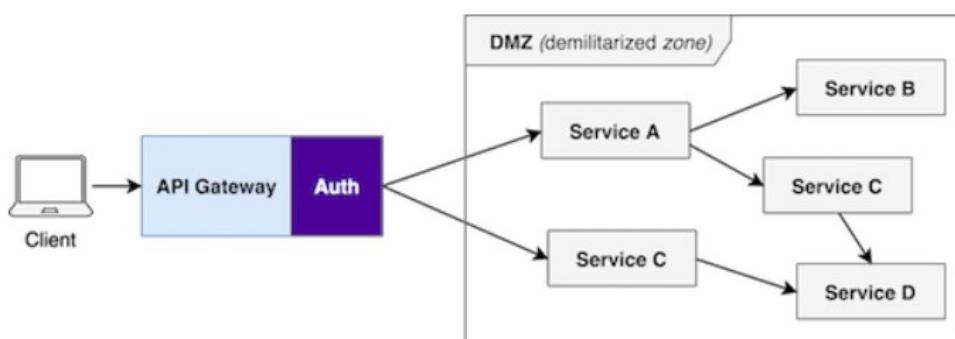
La solution la plus courante utilisée pour protéger l'accès aux services exposés à tout internet consiste à mettre en place des **serveurs d'intermédiation spécialisés appelés "Gateway"**.



On peut éventuellement envisager des "gateway" (ou point d'entrées de gateway) spécifiques aux types de clients externes ("desktop" ou "mobiles") :



On peut éventuellement concentrer la sécurisation des web-services sur un "gateway" :



Si le principal rôle d'un "Gateway" est la sécurisation , beaucoup de solutions "gateway" apportent "dans la façade exposée" tous un tas de valeurs ajoutées intéressantes :

- Authentification et autorisation
- Intégration de la découverte de service
- Mise en cache des réponses
- Stratégies de nouvelle tentative, disjoncteur et qualité de service (QoS)
- Limitation du débit
- Équilibrage de charge
- Journalisation, suivi, corrélation
- Transformation des en-têtes, chaînes de requête et revendications
- Liste verte d'adresses IP

NB : Dans le cadre d'un cloud public , la plateforme d'hébergement "cloud" (ex : Microsoft Azure ou ...) peut directement en charge le "Gateway API" (avec plus ou moins de fonctionnalités associées) .

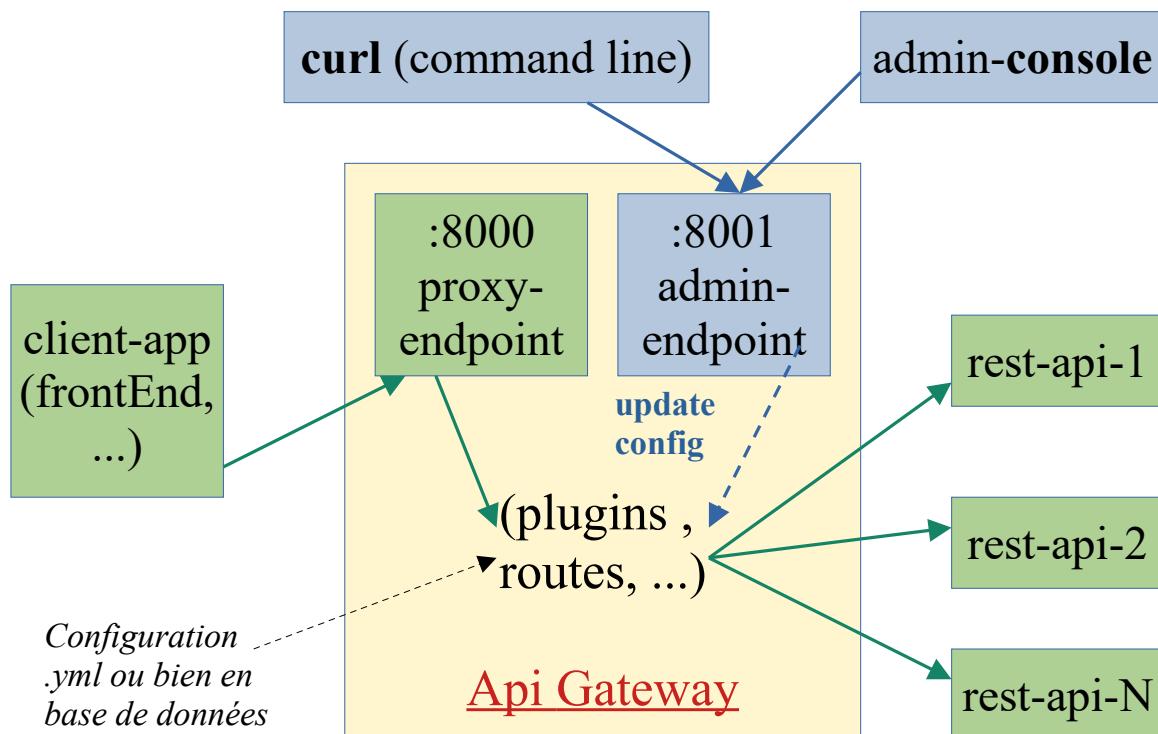
Fonctionnalités complémentaires ou incorporées :

Type de serveur	Fonctionnalités apportées
Serveur HTTP (ex : apache2.x , nginx , ...)	Permettre le téléchargement de site web (html + css + js)
Reverse-Proxy (ex : nginx , traefik, ...)	Rediriger certaines requêtes HTTP vers des serveurs en arrières plan (selon analyse des d'URL)
Api-Gateway (ex : Kong, ...)	Facade d'entrée (pour micro services / API-REST) gérant au minimum routing et sécurité/authentification . Gestion possible d'autres aspects (log , cache , ...) via plugins .
Gestionnaire de cluster (ex : swarm , kubernetes, ...)	Load balancing horizontal sur des services répartis sur plusieurs machines
Service-Mesh (ex : ISTIO)	Gestion avancée des communications inter-services (circuit-breaker , ...)

NB :

- Le serveur **nginx** est à la fois un **serveur HTTP** et un **reverse-proxy** .
- Certains "**api-gateway**" (tel que "**Kong**") sont basés sur **nginx** .
- Certaines fonctionnalités (ex : "load-balancing" , ...) sont à la fois gérées par les gestionnaires d'api , les gestionnaires de cluster et les "service-mesh" .
Plein de combinaisons sont possibles à ce niveau d'autant plus que les technologies évoluent et s'améliorent progressivement ...

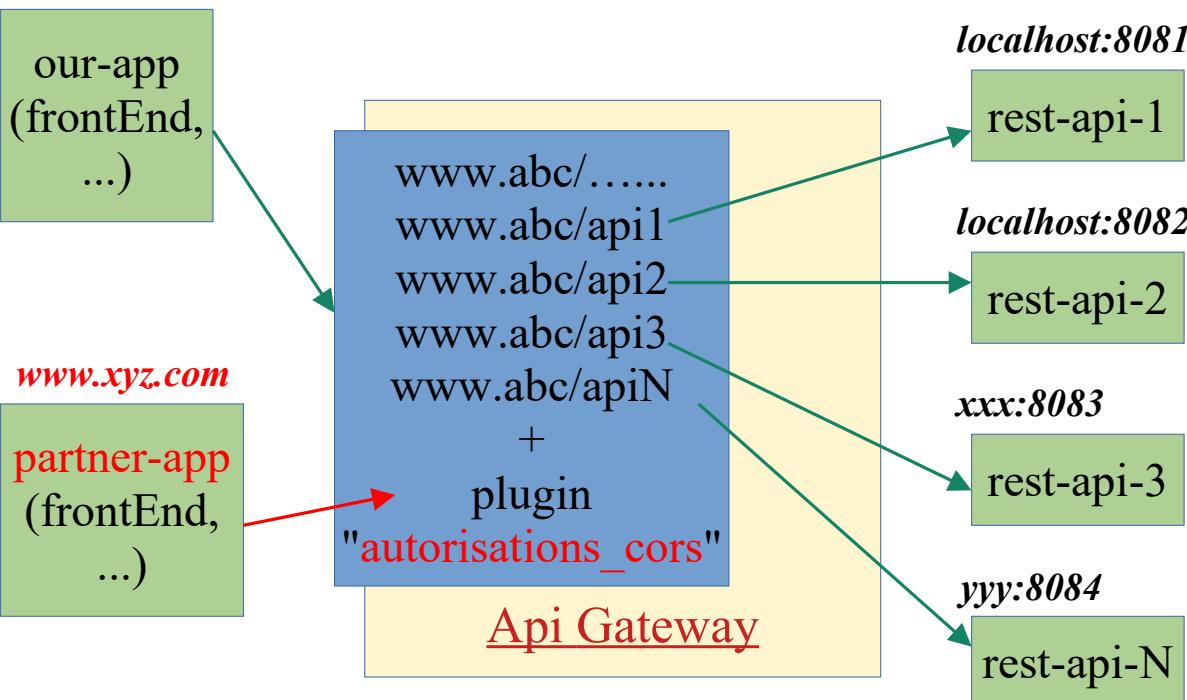
Api-Gateway (administration dynamique)



Grâce à cette configuration dynamique , on peut (en production notamment) reconfigurer certaines routes selon des changements en arrière plan ou bien réajuster certains paramètres (sécurité , log, ...).

Api-Gateway (virtual-hosts , routing , CORS)

www.abc:80_ou_8000



NB : en arrière plan du "api-gateway" , les différentes api REST sont potentiellement éparpillées sur des machines et des conteneurs différents.

Le gestionnaire d'api permet de configurer des **points d'accès** (alias "**routes**") qui :

- commencent tous par un même nom de domaine (ex : virtual-hosts **www.abc**)
- sont accessibles via le même numéro de port (:80 en direct ou :8000 en indirect)

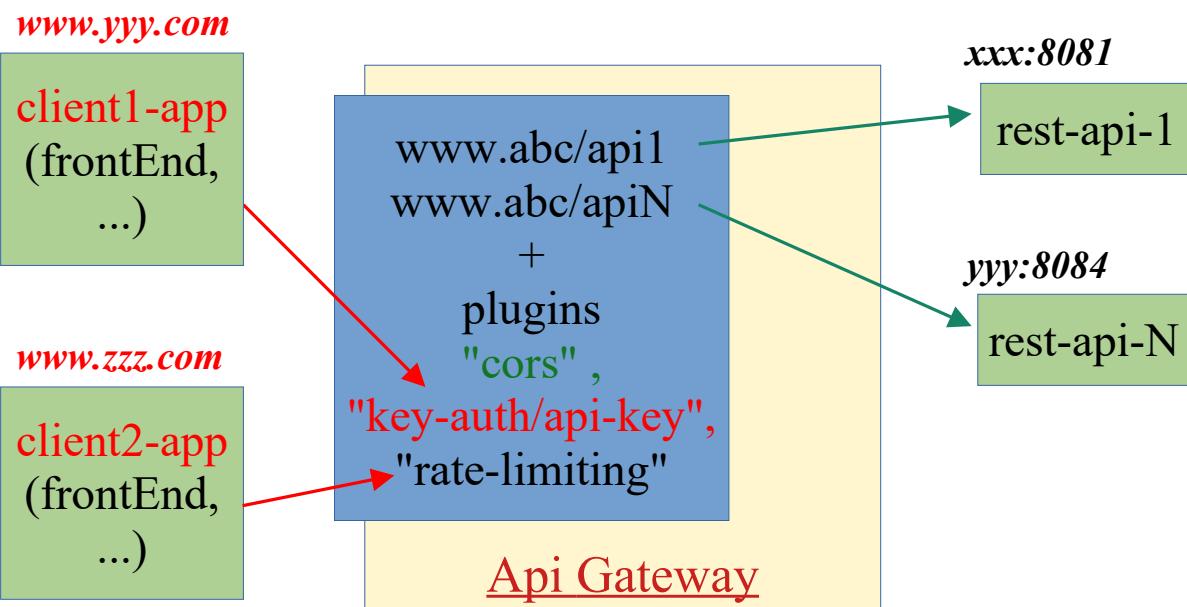
Ainsi , une application "frontEnd" de notre organisation/entreprise pourra sans problème effectuer des appels HTTP/ajax sans que cela nécessite des autorisations CORS .

Autrement dit , le "api-gateway" en tant que façade offre une vision unifiée des différents micro services gérés par l'entreprise abc .

Si une application "frontEnd" d'une autre organisation (entreprise cliente ou partenaire www.xyz.com) veut effectuer des appels ajax vers **www.abc/api...** il faudra alors configurer des autorisations CORS au niveau d'un des plugins du gestionnaire d'api .

Autrement dit, les micro-services en arrière plan (upstream) n'ont pas absolument besoin de configurer des autorisations CORS si c'est le gestionnaire d'api qui prend cela en charge .

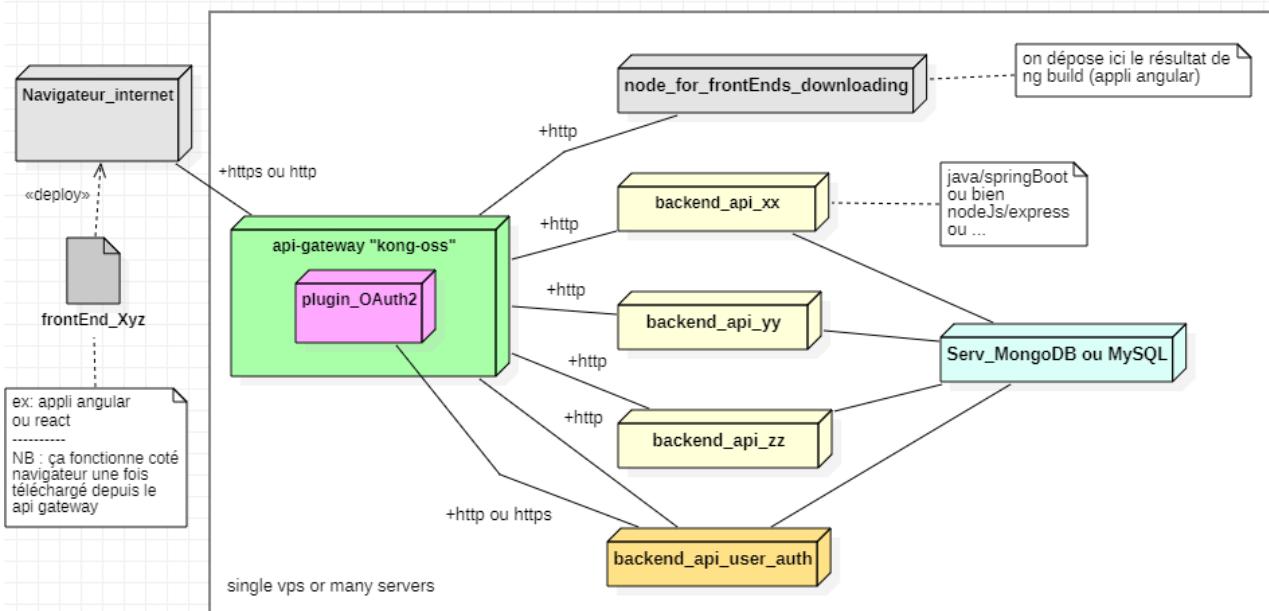
Api-Gateway (api-key , rate-limiting)



Le "*consumer*" www.yyy.com ne pourra accéder à l'api1 que via une URL de type **www.abc/api1?api-key=a2bc4g56** (ou via un champ "*api-key*" de l'*entête HTTP*) et *par exemple seuls 10000 appels par mois seront autorisés*

Ceci est très utile pour se protéger contre des éventuels abus .

Architecture micro-services légère (pour VPS)



Cette architecture micro-services est relativement légère (avec des backends node/express) et peut éventuellement fonctionner sur un seul VPS linux (virtual private server) pour environ 10 ou 12 euros par mois (OVH ou Gandi ou DigitalOcean ou autres).

Au sein de cette architecture , chaque partie (api-gateway , backend , ...) fonctionne idéalement au sein d'un conteneur docker

Etant donné que toutes les requêtes passent par l'intermédiaire api-gateway "kong-oss" pas besoin d'autorisation CORS
D'autre part , via un plugin (oauth2 ou oidc) du api-gateway :

- la génération des jetons d'authentification est simplifiée (collaboration avec backend_api_user_auth à configurer)
- la vérification des jetons est automatisée par kong-oss et son plugin .
- juste besoin côté "backend_api" de vérifier des champs de type "x-authenticated-userid" et "x-authenticated-scope" dans les entêtes des requêtes HTTP (interceptées et enrichies par le api-gateway).

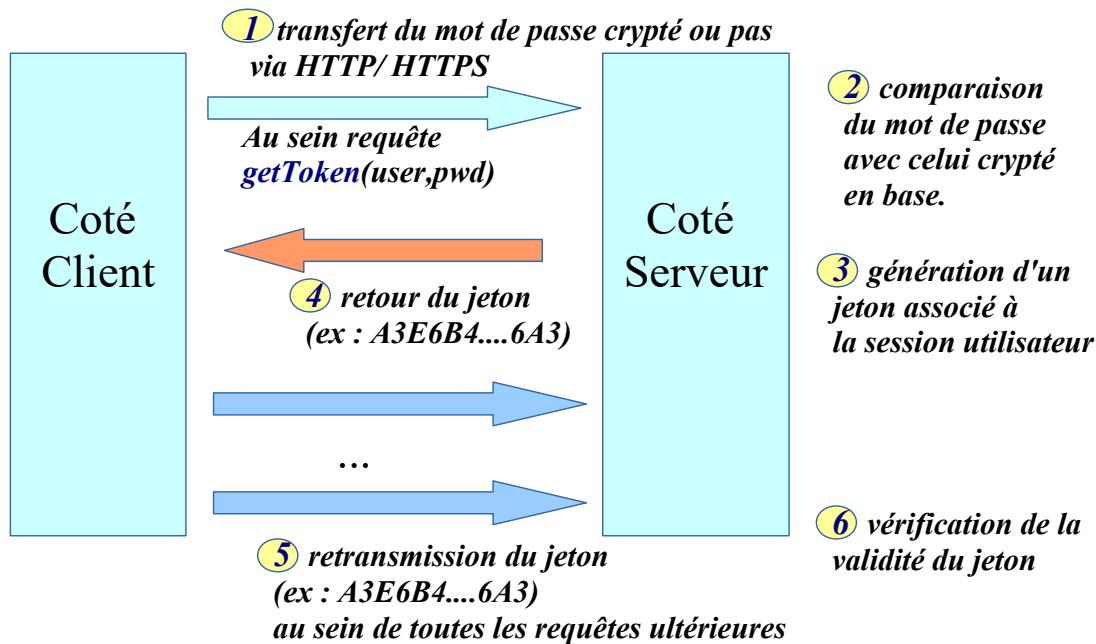
Bien qu'initialement légère , cette architecture est souple, modulaire et extensible.

On peut configurer tout un tas de plugins au sein de l'api-gateway "kong-oss" pour si besoin:

- placer des clusters en arrière plan (simples ou sophistiqués tels que kubernetes)
- gérer des caches
- gérer des contrôles d'api-key , etc .

1.11. Token JWT et OAuth2/OIDC

Jeton ("token") d'authentification valide le temps d'une session utilisateur



Bearer token (jeton au porteur) et transmission

Le champ **Authorization**: normalisé d'une entête d'une requête HTTP peut comporter une valeur de type **Basic ...** ou bien **Bearer ...**

Le terme anglais "**Bearer**" signifiant "**au porteur**" en français indique que la simple possession d'un jeton valide par une application cliente devrait normalement , après transmission HTTP, permettre au serveur d'autoriser le traitement d'une requête (après vérification de l'existence du jeton véhiculé parmi l'ensemble de ceux préalablement générés et pas encore expirés).

NB: Les "bearer token" sont utilisés par le protocole "O2Auth" mais peuvent également être utilisés de façon simple sans "O2Auth" dans le cadre d'une authentification "sans tierce partie" pour API REST.

NB2 : un "bearer token" peut éventuellement être au format "JWT" mais ne l'est pas toujours (voir rarement) en fonction du contexte.

Plusieurs sortes de jetons/tokens

Il existe plusieurs sortes de jetons (normalisés ou pas).

Dans le cas le plus simple, un **jeton est généré aléatoirement** (ex : **uuid** ou ...) et sa validation consiste essentiellement à vérifier son existence en tentant de le récupérer quelque part (*en mémoire ou en base*) et éventuellement à vérifier une date et heure d'expiration.

JWT (Json Web Token) est un **format particulier de jeton** qui **comporte 3 parties** (une entête technique , un paquet d'informations en clair (ex : username , email , expiration, ...) au format JSON et une signature qui ne peut être vérifiée qu'avec la clef secrète de l'émetteur du jeton.



Structure jeton "JWT / Json Web Token"

Header

```
base64enc({
  "alg": "HS256",
  "typ": "JWT"
})
```

Payload

```
base64enc({
  "iss": "toptal.com",
  "exp": 1426420800,
  "company": "Toptal",
  "awesome": true
})
```

Signature

```
HMACSHA256(
  base64enc(header)
  + '!' +
  base64enc(payload)
, secretKey)
```

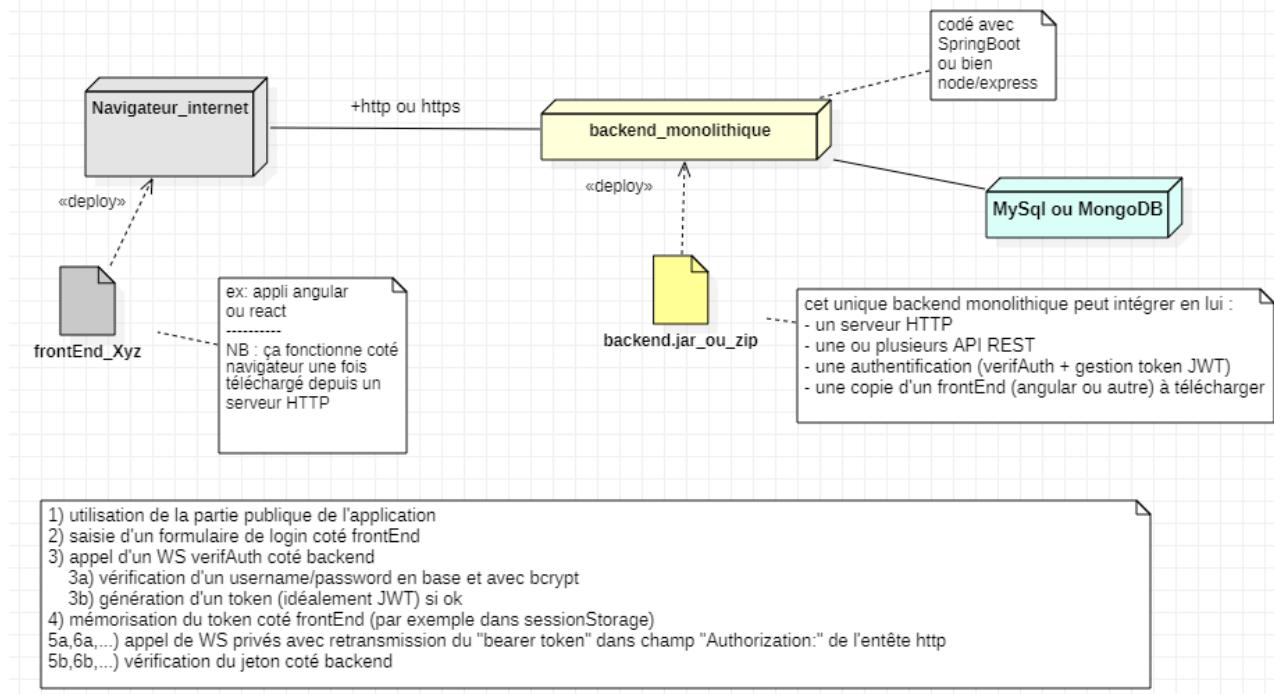
Exemple:

[eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJ0b3B0YWwuY29tIiwiZXhwIjoxNDI2NDIwODAwLCJodHRwOi8vdG9wdGFsLmNvbS9qd3RfY2xhaW1zL2lzX2FkbWluIjp0cnVILCjb21wYW55IjoiVG9wdGFsIiwiYXdlc29tZSI6dHJ1ZX0.yRQYnWzskCZUxPwaQupWkiUzKELZ49eM7oWxAQK_ZXw](https://www.toptal.com/api/auth/jwt)

NB: "iss" signifie "issuer" (émetteur) , "iat" : issue at time
 "exp" correspond à "date/heure expiration" . Le reste du "payload" est libre (au cas par cas) (ex : "company" et/ou "email" , ...)

Authentification monolithique (très limitée) sans OAuth2:

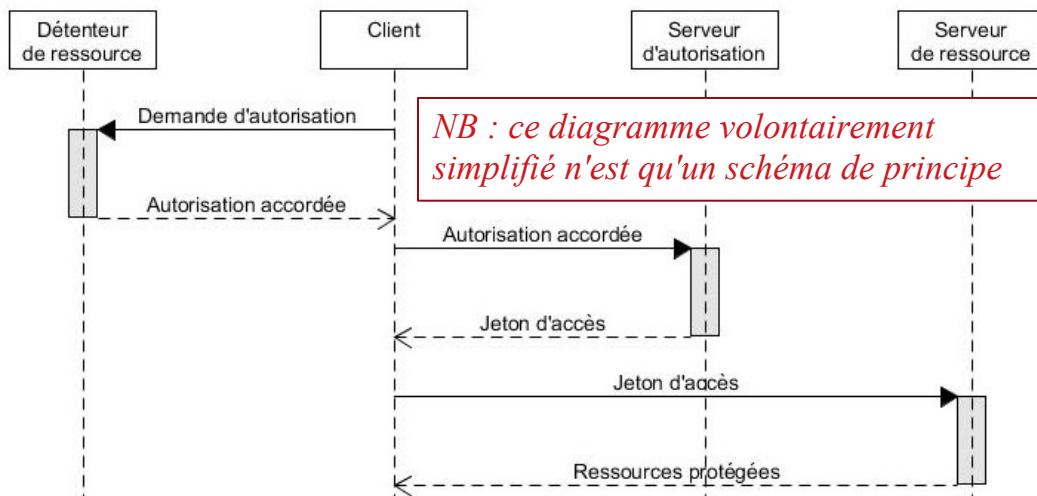
Authentification monolithique (1 seul backend)



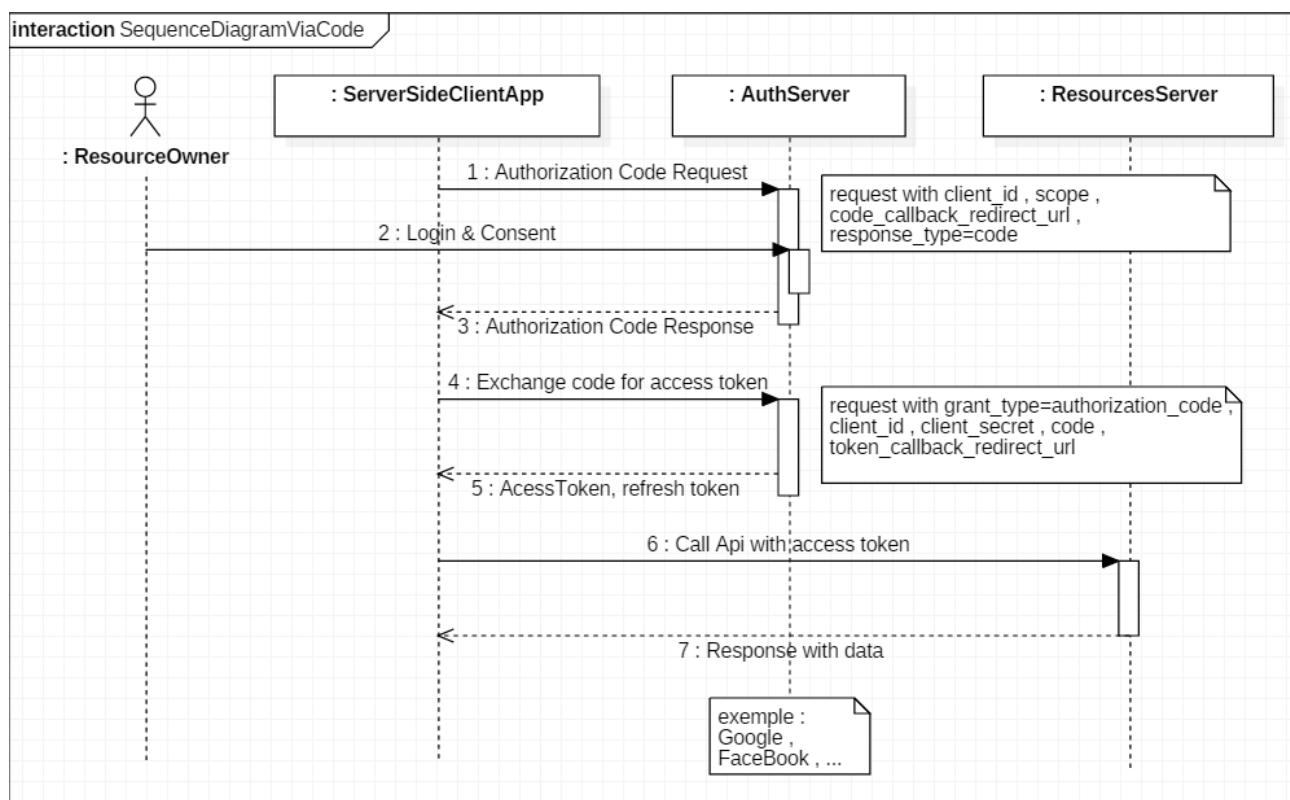
Authentification avec OAuth2 / OIDC:

Norme/Protocole "OAuth2"

OAuth (Open Authorization) existant en versions "1" et "2" , est une norme (RFC 6749 et 6750) qui correspond à un **protocole de "délégation d'autorisation"** . Ceci permet par exemple d'autoriser une application cliente à accéder à une API d'une autre application (ex : FaceBook , Twitter , ...) de façon à accéder à des données protégées.



Autorisation OAuth2 par code (inter-organisations)

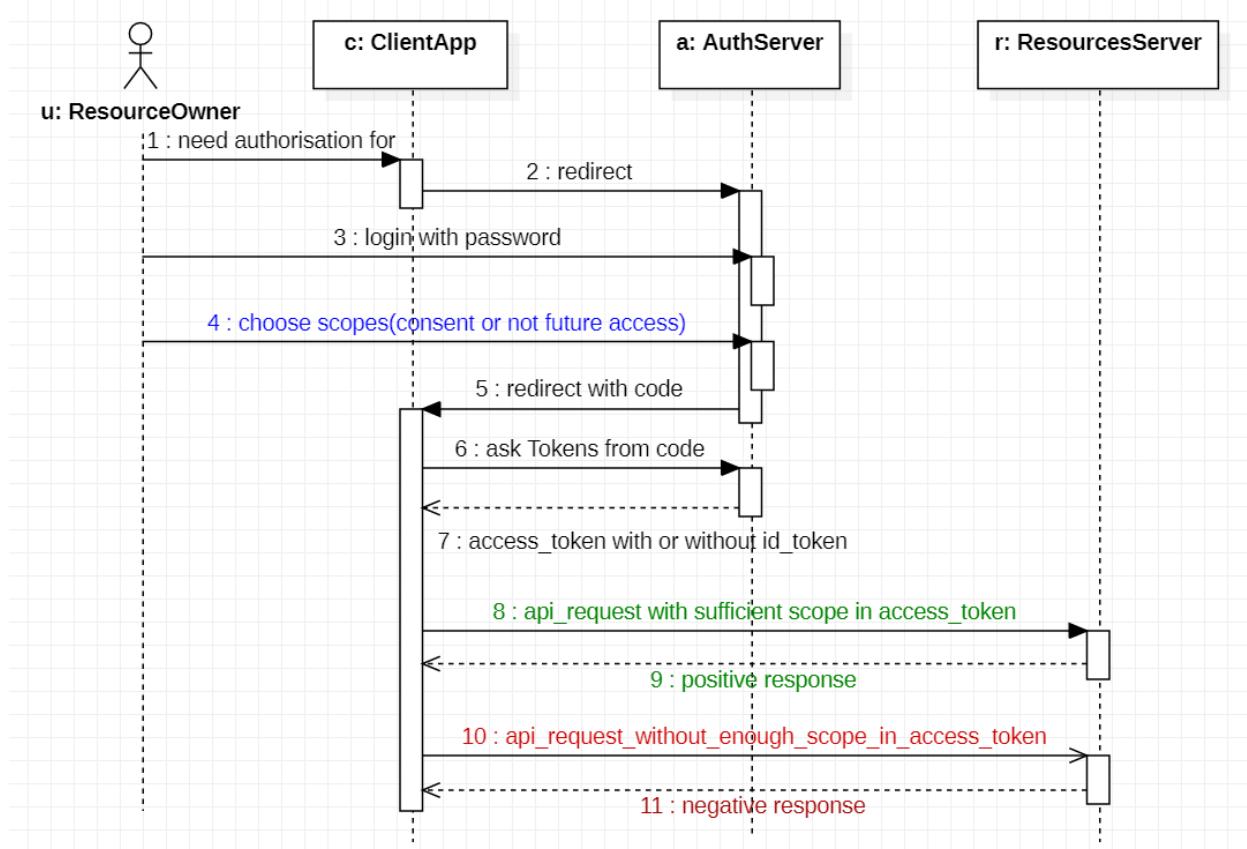


OpenId Connect

OpenID Connect (OIDC) est un protocole d'authentification forte basé sur une transmission standardisée ("*ID Token*" au format JWT) des informations sur l'utilisateur identifié .

OpenID Connect s'appuie en interne sur OAuth2 et permet d'**obtenir simplement des informations plus précises sur l'utilisateur à authentifier** .

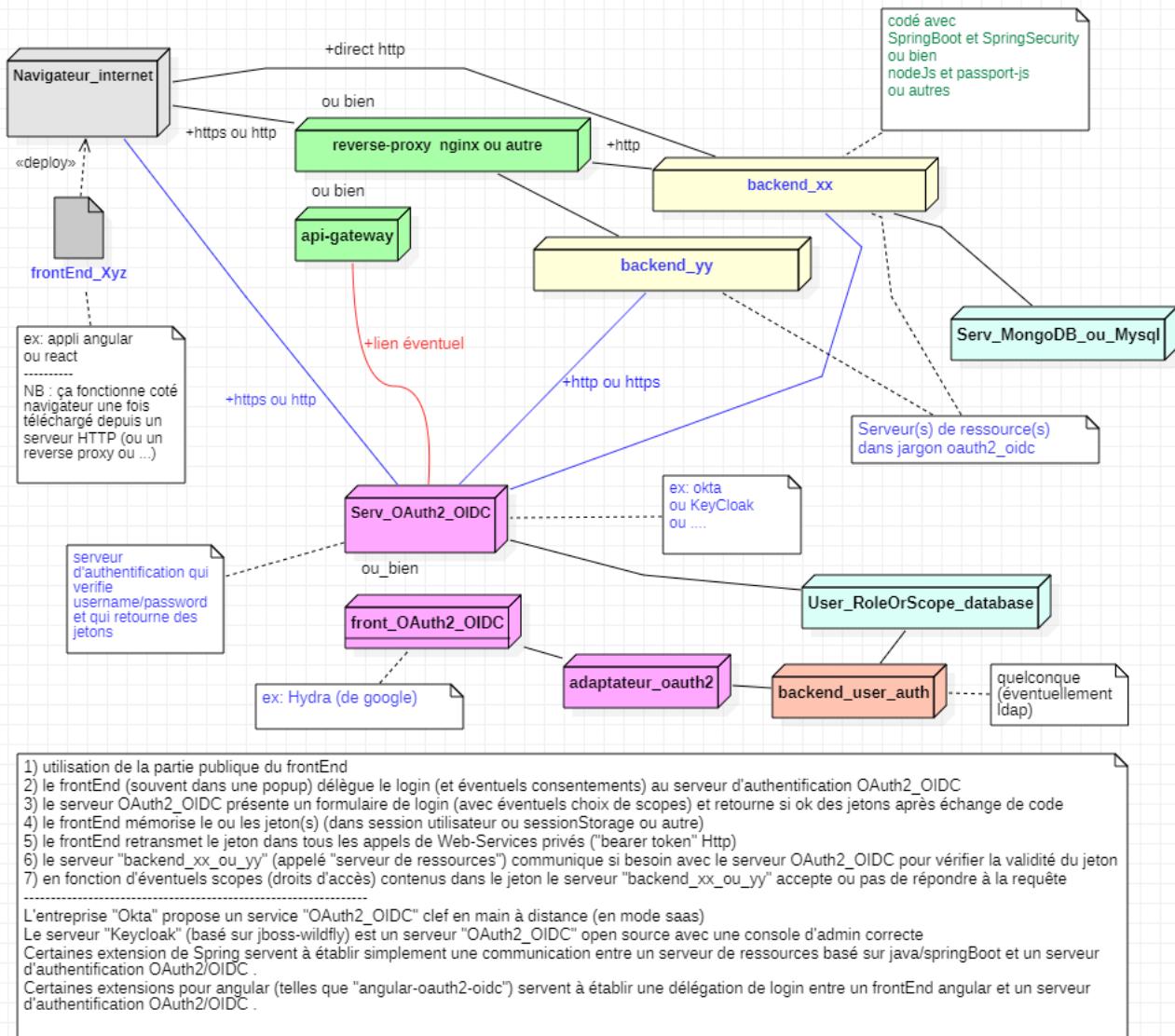
OAuth2 or OIDC with Scopes



Quelques serveurs d'authentification OAuth2/OIDC

- Google **Hydra** (très perfectionné et très performant mais complexe car besoin de compléments à personnaliser)
- **keycloak** est un serveur d'autorisation oauth2/oidc basé sur jboss wildfly et java >=8 (relativement simple à installer). Il est par défaut basé sur une base H2 et dispose d'une ihm intégrée pour configurer des utilisateurs
- **okta** est une entreprise spécialisée dans l'identité numérique et offre des services de type "*authorisation oauth2/oidc as a service*"
- Les grandes plate-formes "cloud" (**Azure** , **AWS**, ...) intègrent un service OAuth2/OIDC (ex : **Azure Active Directory**, **Amazon-Cognito** , ...)
- quelques autres

Authentification avec OAuth2 et/ou OIDC

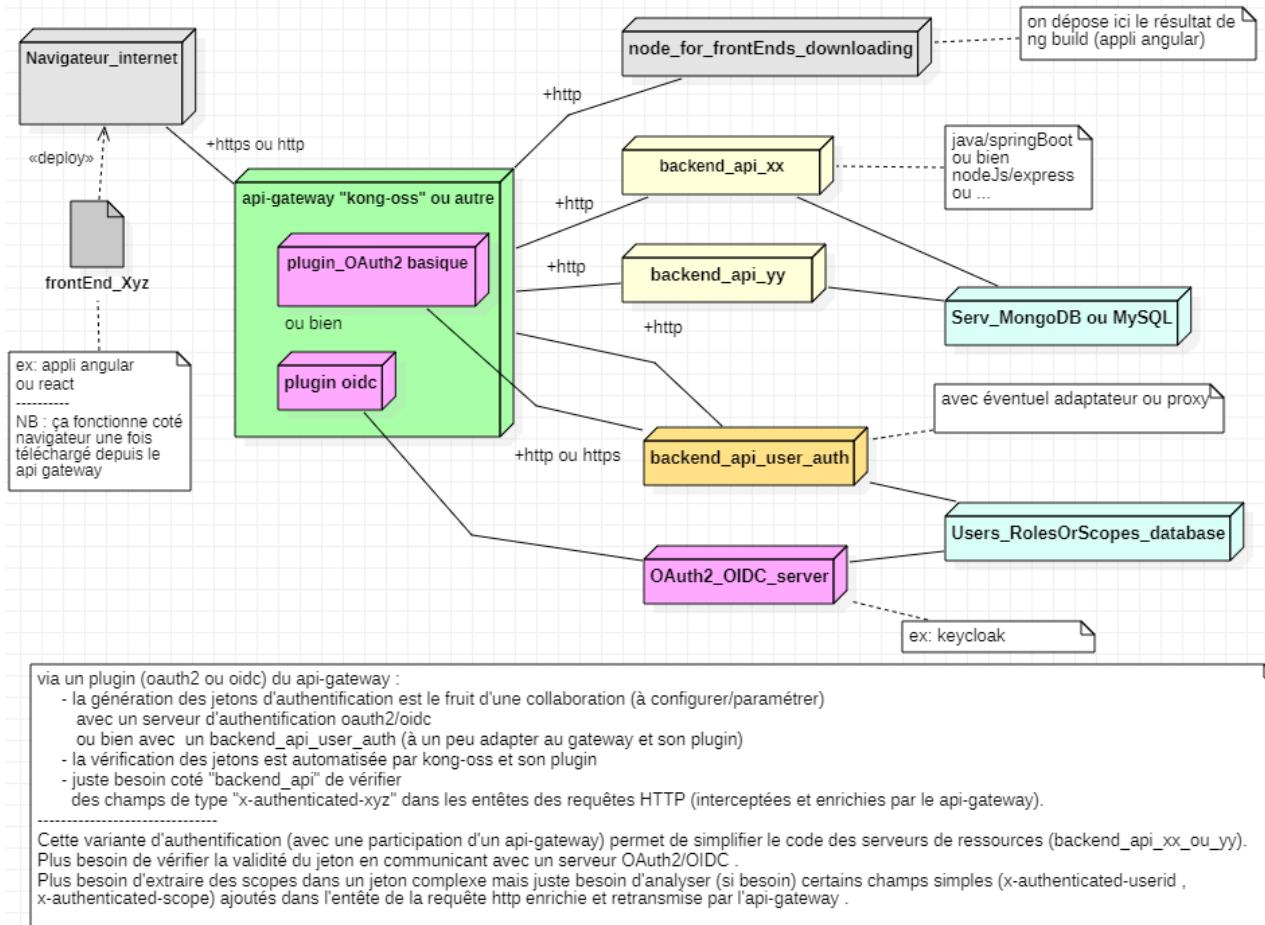


- 1) utilisation de la partie publique du frontEnd
- 2) le frontEnd (souvent dans une popup) délégué le login (et éventuels consentements) au serveur d'authentification OAuth2_OIDC
- 3) le serveur OAuth2_OIDC présente un formulaire de login (avec éventuels choix de scopes) et retourne si ok des jetons après échange de code
- 4) le frontEnd mémorise le ou les jeton(s) (dans session utilisateur ou sessionStorage ou autre)
- 5) le frontEnd retransmet le jeton dans tous les appels de Web-Services privés ("bearer token" Http)
- 6) le serveur "backend_xx_ou_yy" (appelé "serveur de ressources") communique si besoin avec le serveur OAuth2_OIDC pour vérifier la validité du jeton
- 7) en fonction d'éventuels scopes (droits d'accès) contenus dans le jeton le serveur "backend_xx_ou_yy" accepte ou pas de répondre à la requête

L'entreprise "Okta" propose un service "OAuth2_OIDC" clef en main à distance (en mode saas).
 Le serveur "Keycloak" (basé sur jboss-wildfly) est un serveur "OAuth2_OIDC" open source avec une console d'admin correcte.
 Certaines extensions de Spring servent à établir simplement une communication entre un serveur de ressources basé sur java/springBoot et un serveur d'authentification OAuth2/OIDC.
 Certaines extensions pour angular (telles que "angular-oauth2-oidc") servent à établir une délégation de login entre un frontEnd angular et un serveur d'authentification OAuth2/OIDC.

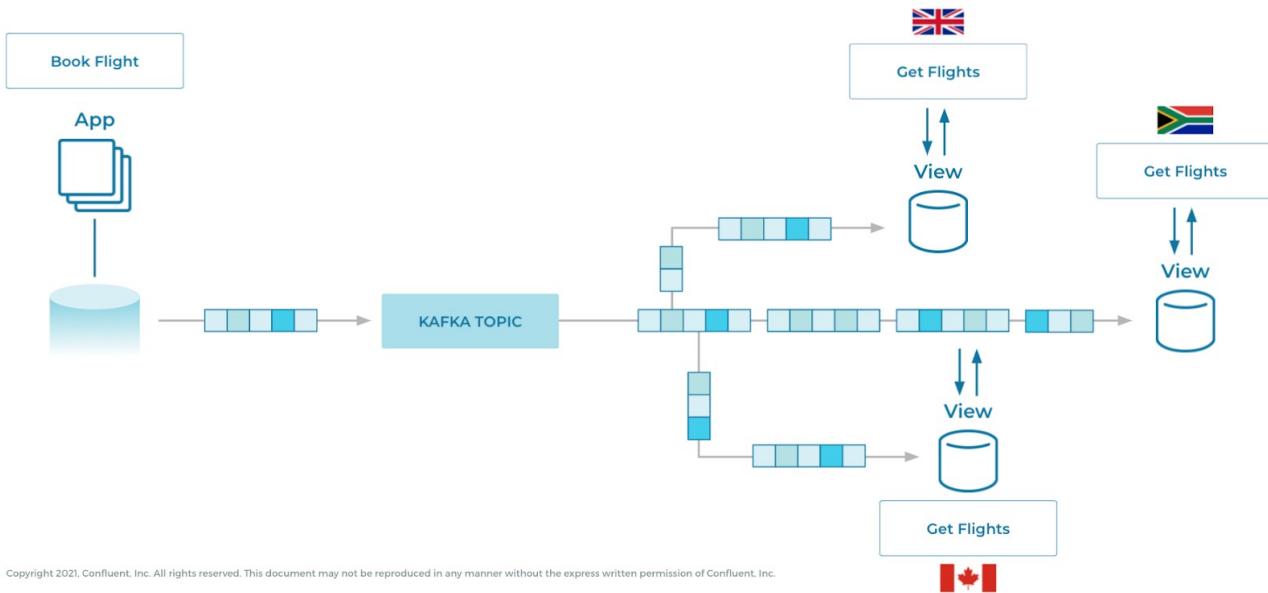
Authentification aidée via api-gateway :

Authentification avec api-gateway



1.12. Resynchronisations asynchrones (event sourcing)

Kafka (éventuellement via JMS en java)



source = <https://developer.confluent.io/learn-kafka/event-sourcing/event-sourcing-vs-event-streaming/>

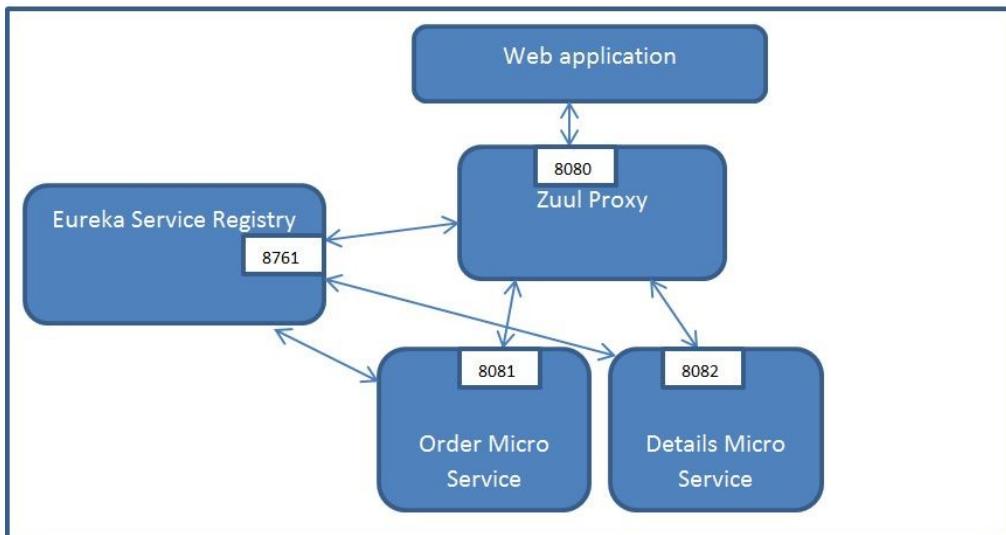
Intérêt principal de kafka (ou d'une techno de type "event sourcing") :

- Si chaque micro-service est indépendant des autres (avec sa propre base de données) , il manque un mécanisme de resynchronisation partiel (**une mise à jour au sein d'une base 1 doit être quelquefois être répercutée dans d'autres bases de données**) .
- **Kafka est un middleware orienté message très performant** (mieux que rabbitMq , activeMq , ...).

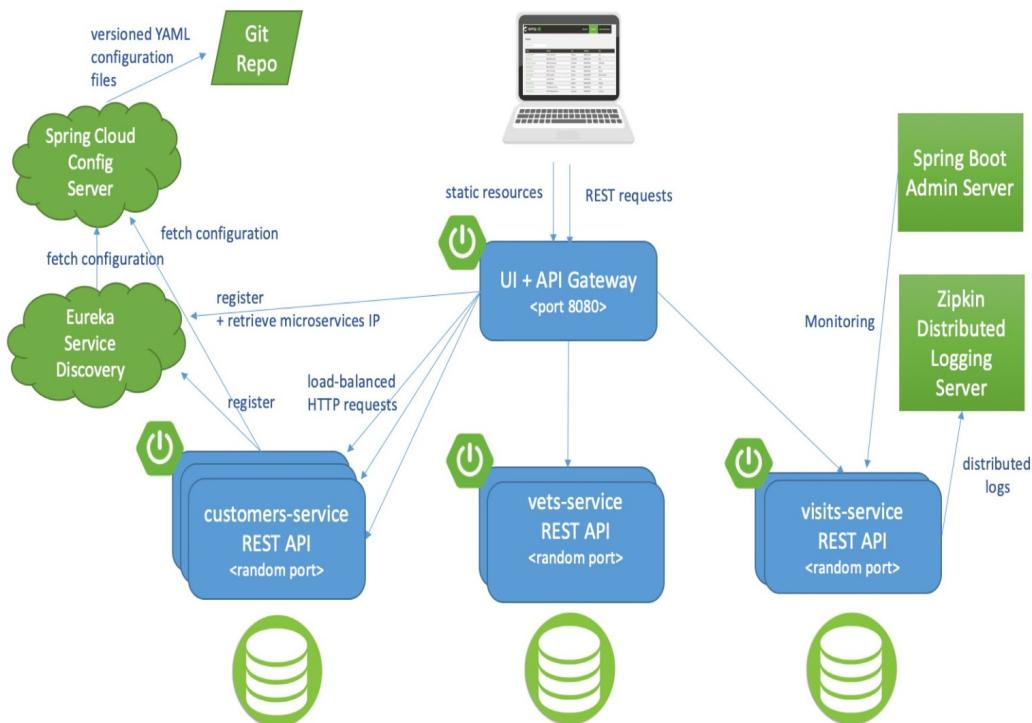
2. Java et le cloud

2.1. Spring-cloud

Diversité des solutions "cloud" : Netflix-OSS

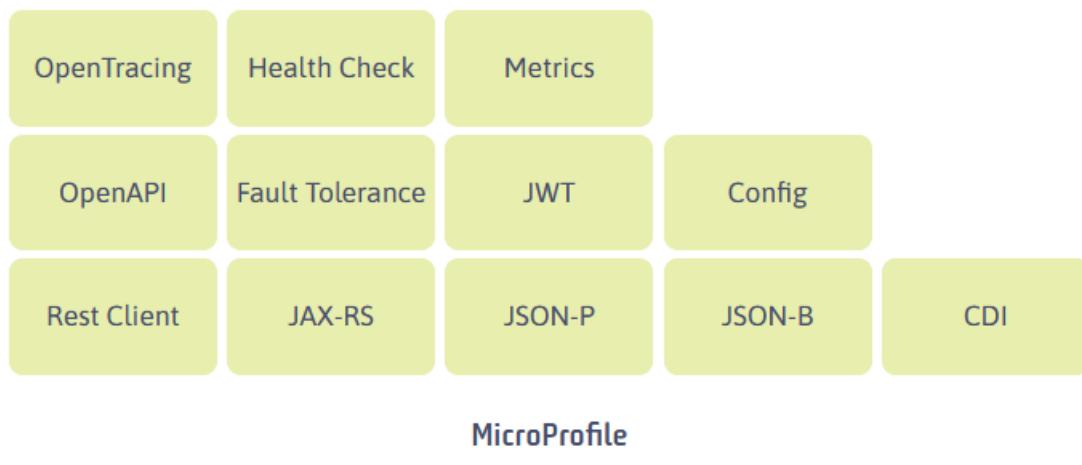


Diversité des solutions "cloud" : Spring-cloud



"spring-cloud" peut être vu comme une évolution de "netflix-oss"

2.2. JEE micro-profile



NB :

- L'api **JSON-P** (JSON-Processor) est une api de bas niveau pour sérialiser/desérialiser des données au format JSON
- L'api **JSON-B** (JSON-Binding) est une api de plus haut niveau qui permet de automatiquement convertir des objets java en JsonString et vice-versa .
JSON-B est paramétrable via des annotations et s'appuie en interne sur JSON-P
JSON-B est une api officielle qui fait à peu près la même chose que Jackson-databind .

- L'api **Config** sert simplement à récupérer des paramètres de configuration via `@ConfigProperty(name="myprj.some.param")` tout en offrant la liberté de préciser ces paramètres de différentes manières (`System.getProperties()` , `System.getenv()` , `META-INF/microprofile-config.properties` , ...), Les valeurs récupérées peuvent alors être directement injectées via `@Inject` de CDI et être utilisées par du code java.
- L'api **JWT** (*de JEE Microprofile*) sert (*au sein d'un serveur de ressources*) à récupérer/vérifier les informations d'un jeton JWT accompagnant traditionnellement certaines requêtes HTTP vers des WS REST (interface `org.eclipse.microprofile.jwt.JsonWebToken` , annotation `@Claim` pour initialiser automatiquement une variable d'instance java selon la valeur d'une partie du jeton et liens automatiques avec la sécurité standardisée de JEE : `getUserPrincipal()` , ... , `@RolesAllowed` , ...). *Mapping exact role-... selon provider de l'api et selon contexte .*
- L'api "OpenApi" (remplaçant maintenant swagger2) permet (via des annotations complémentaires `@Operation`, `@Parameter`, ...) de spécifier des informations sur la structure de certains WS REST de manière à pouvoir **générer automatiquement une documentation HTML ou autre sur une api REST** . En outre une description `.json` (générée automatiquement par openApi) permet d'aider à coder de futurs appels dans divers langages de programmation (`javascript` , `python`, ...)
- L'api **Fault-Tolerance** offre des annotations telles que `@Retry(maxRetries = 2)` et `@Fallback(fallbackMethod = "doWorkFallback")` qui servent à coder des délégations de services avec des mécanismes classiques pour gérer des problèmes d'inaccessibilité temporaire (ex : `circuit-breaker` ,) . Autrement dit , via du code de type "planB" , si notre WS-REST "S" devant appeler en interne un autre WS-REST "A" rencontre temporairement un problème (réseau ou ...) pour appeler convenablement "A" , il pourra renvoyer (faute de mieux) une réponse alternative explicite digne d'une application résiliente .
- L'api **RestClient** (*de JEE MicroProfile*) à voir comme un complément à la partie cliente de JAX-RS sert à construire simplement/rapidement des proxy_rpc pour appeler des WS-REST distants codés en java avec des implémentations JAX-RS (implémentant des *interfaces JAX-RS*) . Ceci permet un style de code par appels quasi direct de méthodes distantes plutôt qu'un style d'appel par URL/requête_http/analyse_statut_http et déserialisation_JSON .
- L'api **Health-check** (*de JEE MicroProfile*) sert à implémenter de manière souple et personnalisée des "endpoint" standardisés (/health/ready et /health/live) qui seront régulièrement invoqués par des contextes/environnements "cloud" de type "cloud fondry" ou "kubernetes" pour savoir une instance de notre application (fonctionnant généralement au sein d'un conteneur docker) est en bonne santé (fonctionne bien , prête à répondre , ... dans l'état "Up" ou bien "down").
- L'api **Metrics** (*de JEE MicroProfile*) sert à aider à implémenter (via des annotations prédefinies `@Count` , `@Gauge` , `@Metered` , `@Timed` , ...) des WS-REST en mode GET retournant des valeurs/mesures métriques dans un format un peu normalisé et quelquefois augmenté par des calculs statistiques (min, max, moyenne , ...).
- L'api "Open-tracing" sert à mettre en oeuvre des mécanisme de logs/traces distribués et paramétrables (à des fins de "debug" , "surveillance" , "traçages/optilisations" , ...).

Dans une application JEE basée sur un léger micro-profile , la **persistance** des données au sein

d'une base de données relationnelle est censée être gérée via l'api standard **JPA** .

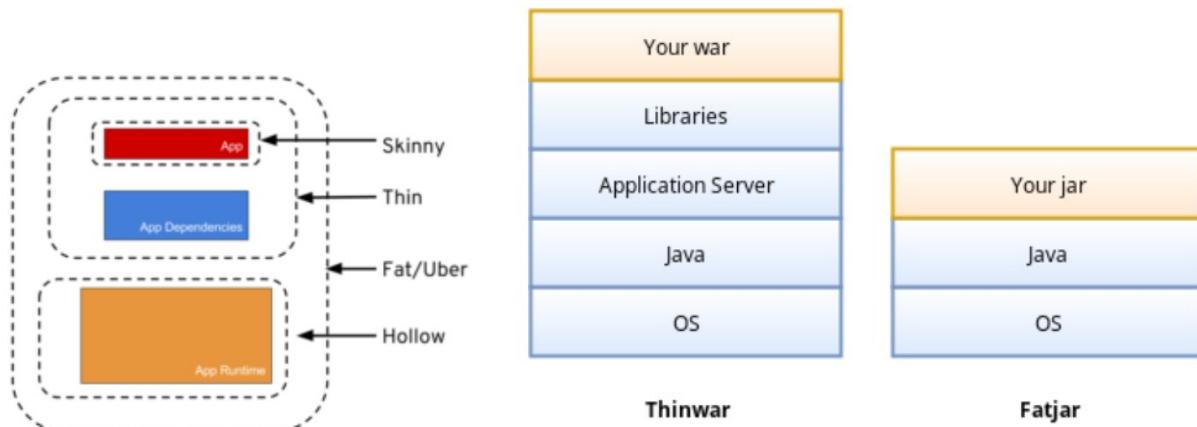
Pour ce qui est de la **gestion des transactions dans un contexte "micro-profile"**, on utilisera pas les **EJB** mais une **extension "JTA" pour "CDI"** permettant d'interpréter **@Transactional** via **des intercepteurs** . Cette extension est généralement fournie par le serveur (OpenLiberty ou autre).

Serveurs JEE récents et légers pour micro-profile :

Marque	Serveur	Caractéristiques
IBM	Open Liberty et WebSphere Liberty	
RedHat	Quarkus	Pour le cloud , avec GraalVM Native Image
Oracle	Helidon et Payara (<i>anciennement glassfish</i>)	???

Autres supports partiels de micro-profile : Wildfly et TomEE .

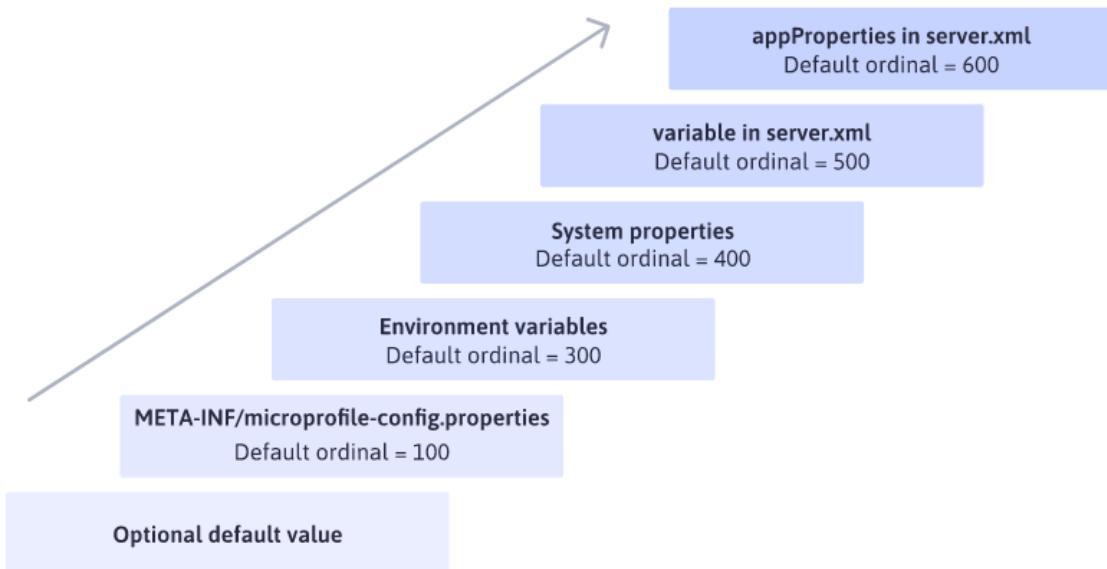
Déploiement moderne avec docker :



selon variantes/profiles d'un pom.xml de openLiberty (ou bien springBoot) .

2.3. Cas particulier: serveur "openLiberty"

Priorité par défaut des sources de configuration :



NB : plus l'ordinal est grand, plus c'est prioritaire .