

# **Sistemi Distribuiti M**

Luca Foschini

2021/2022

Enrico Valastro & Sofia Montebugnoli

## Sommario

<b>1. Applicazioni Enterprise e panoramica su J2EE .....</b>	<b>7</b>
<b>1.1 Modelli.....</b>	<b>7</b>
<b>1.2 Evoluzione delle architetture per Enterprise Application.....</b>	<b>8</b>
<b>1.3 Architettura J2EE per Application Server .....</b>	<b>12</b>
<b>1.4 Architettura J2EE per applicazioni N-tier.....</b>	<b>13</b>
1.4.1 Altre soluzioni.....	13
<b>1.5 Modelli a contenimento .....</b>	<b>14</b>
1.5.1 JEE.....	15
<b>1.6 Anteprima su EJB.....</b>	<b>15</b>
<b>1.7 J2EE per Appllicazione N-tier .....</b>	<b>16</b>
<b>1.8 Architettura EJB.....</b>	<b>17</b>
1.8.1 Componenti EJB.....	18
<b>1.9 Componenti e Container.....</b>	<b>19</b>
<b>2. Modelli a componenti ed Enterprise Java Beans .....</b>	<b>21</b>
<b>2.1 Enterprise Java Beans (EJB).....</b>	<b>21</b>
2.1.1 EJB e utilizzo da parte di client differenti .....	22
<b>2.2 EJB - 2.x.....</b>	<b>22</b>
2.2.1 Architettura EJB 2.x .....	22
2.2.2 Contratti in EJB 2.x.....	23
2.2.3 Terminologia.....	25
<b>2.3 Tipologie di componenti Bean .....</b>	<b>25</b>
2.3.1 Session Bean.....	26
2.3.2 Entity Bean .....	26
2.3.3 Message-Driven Bean (MDB).....	27
2.3.4 Scenari applicativi.....	28
<b>2.4 Scenari ed esempi pratici.....</b>	<b>29</b>
2.4.1 Esempi pratici .....	30
<b>2.5 RMI e interazioni .....</b>	<b>34</b>
<b>2.6 Deployment di un'applicazione .....</b>	<b>36</b>
<b>3. Java 5 &amp; Annotation .....</b>	<b>37</b>
<b>3.1 Sintassi .....</b>	<b>37</b>
<b>3.2 Categorie di Annotation .....</b>	<b>37</b>
<b>3.3 Custom Annotation .....</b>	<b>37</b>
3.3.1 Meta-Annotation.....	38
<b>4. Java Naming and Directory Interface (JNDI).....</b>	<b>40</b>
<b>4.1 JNDI architettura e concetti fondamentali .....</b>	<b>42</b>
<b>4.2 JNDI Provider .....</b>	<b>43</b>
<b>4.3 JNDI Context.....</b>	<b>44</b>
<b>4.4 JNDI DirContext.....</b>	<b>45</b>
<b>4.5 Uso di JNDI .....</b>	<b>46</b>

<b>4.6 Configurazione di JNDI.....</b>	<b>48</b>
<b>5. Enterprise Java Beans 3.x .....</b>	<b>50</b>
<b>5.1 Approccio EJB 3.0 .....</b>	<b>50</b>
<b>5.2 Tipologie di Bean in EJB 3.0 .....</b>	<b>51</b>
5.2.1 Session Bean.....	51
5.2.2 Message Driven Bean .....	53
<b>5.3 Dependency Injection.....</b>	<b>53</b>
<b>5.4 Interoperabilità e Migrazione fra EJB 3.0 e EJB 2.x .....</b>	<b>56</b>
<b>5.5 Servizi di sistema container-based.....</b>	<b>57</b>
5.5.1 Pooling e Concorrenza .....	57
5.5.2 Transazionalità .....	58
5.5.3 Gestione delle connessioni e risorse .....	63
5.5.4 Persistenza.....	63
5.5.5 Messaging.....	63
5.5.6 Sicurezza .....	65
<b>5.6 Gli intercettori.....</b>	<b>65</b>
<b>6. Persistenza, JPA e Hibernate.....</b>	<b>67</b>
<b>6.1 Java Persistence API (JPA).....</b>	<b>67</b>
<b>6.2 Entity .....</b>	<b>68</b>
6.2.1 ORM: Entity ed Ereditarietà .....	71
6.2.2 Entity Inheritance e Strategie di Mapping.....	72
6.2.3 ORM: Molteplicità nelle Relazioni .....	72
6.2.4 ORM: Direzionalità nelle Relazioni .....	72
6.2.5 Gestione a Runtime di Entity.....	73
6.2.6 Ciclo di Vita di Entity .....	75
6.2.7 Creazione di Query.....	76
6.2.8 Unità di Persistenza.....	77
6.2.9 Loading Lazy/Eager .....	78
<b>6.3 Hibernate .....</b>	<b>78</b>
6.3.1 Stati di oggetti Hibernate .....	80
6.3.2 Hibernate Caching.....	80
6.3.3 Hibernate Fetching.....	82
6.3.4 Query By Examples (QBE).....	82
<b>7. Java Messaging Service (JMS), ESB e JBI .....</b>	<b>83</b>
<b>7.1 Sistemi di Messaging .....</b>	<b>83</b>
7.1.1 Modello point-to-point .....	84
7.1.2 Modello publish-subscribe .....	84
7.1.3 Affidabilità dei messaggi .....	84
7.1.4 Transazionalità dei messaggi.....	85
7.1.5 Sicurezza nello scambio di messaggi .....	85
<b>7.2 Java Messaging System (JMS) .....</b>	<b>85</b>
7.2.1 Tipi di comunicazioni.....	86
7.2.2 Formato del messaggio .....	86
7.2.3 Interfacce.....	87
7.2.4 Affidabilità dei messaggi .....	90
7.2.5 Gestione delle transazioni di JMS .....	92
7.2.6 Selettori di messaggi .....	93
7.2.7 Java Messaging System JMS e Message Driven Bean MDB .....	93
<b>7.3 Enterprise Service Bus (ESB).....</b>	<b>93</b>

7.3.1	Service Oriented Architecture SOA .....	94
7.3.2	Web Services .....	94
7.3.3	Enterprise Application Integration EAI.....	96
<b>7.4</b>	<b>Java Business Integration .....</b>	<b>98</b>
<b>8.</b>	<b><i>Corba Component Model (CCM)</i>.....</b>	<b>101</b>
8.1	Corba 2.x .....	101
8.2	Corba Component Model (CCM) .....	102
8.3	Confronto con altre tecnologie .....	105
8.4	Esempio running di funzionamento .....	106
8.5	Componente .....	107
8.5.1	Facet .....	108
8.5.2	Receptacles.....	109
8.5.3	Scambio di eventi e Event Sources.....	109
8.5.4	Sink .....	109
8.5.4	Attributes.....	110
8.5.5	Navigation e Introspection in CCM.....	110
8.6	Implementazione dinamica dei componenti e supporto runtime .....	111
8.7	Container .....	112
8.8	Strategie Container-managed .....	113
8.8.1	Executor.....	113
<b>9.</b>	<b><i>Spring</i>.....</b>	<b>114</b>
9.1	Architettura di Spring .....	115
9.2	Aspect Oriented Programming (AOP) .....	116
9.3	Dependency Injection in Spring .....	118
9.3.1	BeanFactory.....	119
9.4	L' Hello World in Spring .....	120
9.5	Gli Interceptioni in Spring .....	127
9.6	Transazionalità .....	128
9.6.1	Recap su transazioni.....	128
9.6.2	Le transazioni in Spring.....	132
9.7	Considerazioni avanzate .....	133
9.8	Autowiring .....	133
9.9	Dependency Checking .....	134
9.10	Application Context.....	134
<b>10.</b>	<b><i>Java Management Extensions (JMX)</i>.....</b>	<b>136</b>
10.1	Introduzione ai monitoring systems.....	136
10.2	Java Management Extensions JMX .....	137
10.2.1	Architettura JMX.....	137
10.2.2	Livello Instrumentation .....	137
10.2.3	Livello Agente .....	138
10.2.4	Livello servizi distribuiti .....	139
10.3	Standard MBean.....	139
10.3.1	MBean Server registrazione .....	140

10.3.2 Invocazione servizi di gestione .....	140
10.3.3 Meccanismo di notifica .....	140
<b>10.4 Dynamic MBean .....</b>	<b>141</b>
<b>10.5 ModelMBean.....</b>	<b>142</b>
<b>10.6 Servizi Standard a Livello di Agente .....</b>	<b>143</b>
10.6.1 M-let service .....	143
10.6.2 Servizio di timer .....	143
10.6.3 Servizio di monitoring.....	144
10.6.4 Servizio Relation .....	145
<b>10.7 JMX remote API.....</b>	<b>145</b>
<b>10.8 Esempio di utilizzo di MBean .....</b>	<b>147</b>
<b>10.9 JMX in Application Server JBoss.....</b>	<b>149</b>
<b>11. <i>Clustering di applicazioni in WildFly/JBoss .....</i></b>	<b>152</b>
<b>11.1 Architettura JBoss .....</b>	<b>152</b>
<b>11.2 Comunicazione all'interno del cluster: JGroups.....</b>	<b>153</b>
11.2.1 Configurazione di JGroups.....	153
11.2.2 HA Partition .....	155
<b>11.3 Comunicazione Client-to-Cluster.....</b>	<b>156</b>
<b>11.4 Replicazione dello stato.....</b>	<b>157</b>
<b>11.5 Infinispan .....</b>	<b>158</b>
<b>11.6 Replicazione dello stato nel clustering di componenti EJB.....</b>	<b>160</b>
<b>11.7 Configurazione di JBoss/WildFly .....</b>	<b>160</b>
<b>12. <i>BigData .....</i></b>	<b>162</b>
<b>12.1 Standardizzazione industriale .....</b>	<b>163</b>
12.1.1 Open Data Center Alliance (ODCA).....	163
12.1.2 National Institute of Standards and Technology (NIST) .....	163
<b>12.2 InfoSphere Streams per Stream Processing.....</b>	<b>164</b>
<b>12.3 Batch Processing.....</b>	<b>166</b>
12.3.1 MapReduce .....	166
12.3.2 Esempio di MapReduce .....	168
12.3.3 Apache Hadoop .....	169
12.3.4 Google MapReduce vs Hadoop .....	170
12.3.5 Hadoop vs Spark .....	171
12.3.6 MapReduce Scheduling e limiti di Hadoop.....	171
<b>12.4 Fault-tolerance in DSPS (Distributed Stream Processing System) .....</b>	<b>172</b>
12.4.1 Active Replication .....	172
12.4.2 UpStream Backup .....	172
12.4.3 Checkpointing.....	173
<b>12.5 Gestione della variazione di carico.....</b>	<b>173</b>
12.5.1 LAAR .....	174
<b>12.6 Digital Twins.....</b>	<b>174</b>
12.6.1 IoTwins .....	175
<b>13. <i>Cenni di Node-JS .....</i></b>	<b>176</b>
<b>13.1 Event Loop .....</b>	<b>176</b>

<b>13.2 Thread vs Asynchronous Event Driven .....</b>	<b>176</b>
<b>13.3 Thread vs Eventi .....</b>	<b>177</b>
13.3.1 Esempio di uso di Node.js in server-side script PHP .....	181
13.3.2 Stili di programmazione: thread vs callback.....	181
<b>13.4 Moduli.....</b>	<b>182</b>
<b>14. Docker e Orchestrazione Container.....</b>	<b>186</b>
<b>14.1 Micro-servizi, DevOps, Container.....</b>	<b>186</b>
<b>14.2 Docker .....</b>	<b>188</b>
<b>14.3 Kubernetes .....</b>	<b>195</b>
<b>14.4 Applicazione Serverless e FaaS .....</b>	<b>200</b>

# 1. Applicazioni Enterprise e panoramica su J2EE

Un **componente software** è un pezzo di software che presenta particolari caratteristiche: è riutilizzabile e facilmente estendibile, è auto-contenuto mantiene al suo interno dettagli su come è implementato, quali sono i suoi comportamenti, quale è il suo stato ed espone verso l'esterno solamente la sua interfaccia, in particolare la caratteristica che lo distingue da altri è oggetti è che per essere eseguito necessita di un altro pezzo di software che ne sostiene run-time l'esecuzione, il **container**.

## 1.1 Modelli

Si ha sempre a che fare con problemi che si prestano a diverse soluzioni. Diviene necessario muoversi nello spazio delle soluzioni e scegliere la soluzione più adatta al caso specifico, è indispensabile scegliere la soluzione con il miglior trade-off tra costo e flessibilità. Lo scopo dei modelli è quello di comprendere le linee guida delle soluzioni andando oltre ai dettagli tecnologici. Alcuni modelli base sono:

- Modelli e paradigmi statici/dinamici
- Modelli e strategie preventivi/reattivi
- Modello per l'esecuzione nel sistema
- Modello delle entità per l'allocazione

### Modelli per l'esecuzione nel sistema:

Nell'ambito dei sistemi distribuiti si è interessati agli aspetti di **operatività, performance, e reale esecuzione distribuita**.

I possibili approcci sono i seguenti:

- **Modelli e paradigmi statici/dinamici:**

L'uso di modelli statici non permette di adeguare il sistema a fronte di variazioni, i modelli dinamici invece permettono di fare evolvere il sistema a fronte di variazioni ma hanno costi più elevati.

- **Modelli e strategie preventivi/reattivi:**

I modelli preventivi prevengono eventi o situazioni non desiderate e hanno un costo fisso sul sistema, i modelli reattivi introducono meno logica e limitano il costo sul sistema nel caso in cui gli eventi non si verifichino.

### Supporto al deployment:

Di fondamentale importanza è anche la fase di deployment, i possibili approcci sono:

- **Manuale:** in questo approccio l'utente determina ogni singolo oggetto/componente e lo trasferisce sui nodi appropriati con sequenza appropriata di comandi.
- **Approccio con File Script:** questo approccio prevede l'esecuzione di alcuni file di script per arrivare alla configurazione finale del sistema.

- **Approccio su modelli o linguaggi dichiarativi:** questo approccio prevede un supporto automatico alla configurazione attraverso linguaggi dichiarativi o modelli di funzionamento (file di deployment XML e annotazioni Java5).

### **Modello di allocazione:**

Le entità di un'applicazione possono essere statiche, dinamiche o miste. Nel caso di allocazione statica, data una specifica configurazione (o deployment) le risorse sono decise e allocate prima dell'esecuzione, nel caso di allocazione dinamica l'allocazione delle risorse è decisa durante l'esecuzione. I possibili approcci per l'allocazione sono:

- **Approccio implicito:**

l'utente prevede il mapping per ogni risorsa da creare prima dell'esecuzione. Questo approccio ha un costo elevato, in quanto l'utente prima dell'esecuzione deve prevedere un mappaggio per ogni risorsa, anche se questa non sarà utilizzata.

- **Approccio esplicito:**

il sistema si occupa del mapping delle risorse dell'applicazione (anche al deployment). Questo approccio ha un costo limitato in quanto il sistema si occupa del mappaggio delle risorse statiche e dinamiche su bisogno (by need).

- **Approccio misto:**

il sistema adotta una politica di default applicata sia inizialmente per le risorse statiche sia dinamicamente per l'allocazione delle nuove risorse e la migrazione di quelle già esistenti, eventuali indicazioni dell'utente sono tenute in considerazione per migliorare le prestazioni. Il costo di questo approccio è variabile, bilanciabile e adattabile: il sistema adotta una politica per ogni risorsa, o statica o dinamica, decisioni statiche possono essere ottimizzate prima del runtime, decisioni dinamiche possono essere a costo diverso, a seconda del carico del sistema.

## 1.2 Evoluzione delle architetture per Enterprise Application

Gli elementi costitutivi di un'applicazione enterprise sono:

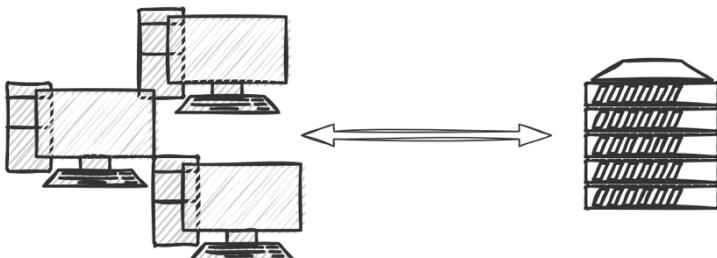
- Presentation logic
- Business logic
- Data access logic
- Servizi di sistema

L'evoluzione delle architetture porterà ad un sistema in cui tali elementi sono logicamente separati, permettendo così di ridurre la complessità degli strati.

### **Single tier:**

Questa primitiva architettura è basata su un modello centralizzato in cui logica di presentazione, di business e di accesso ai dati sono interdipendenti e fuse insieme in un'applicazione monolitica che risiede su un mainframe a cui si connettono in maniera diretta i terminali.

- **Pro:**



Nessuna necessità di gestione client-side.

Facilità di ottenere consistenza dei dati.

- **Contro:**

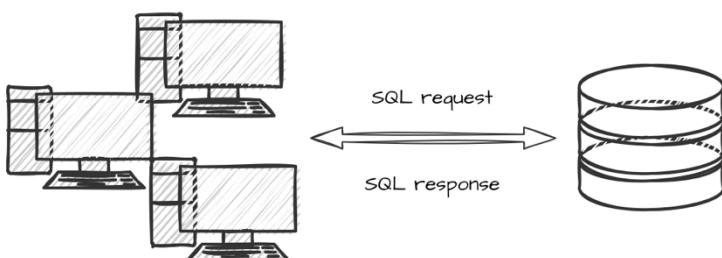
Funzionalità interdipendenti, difficoltà di aggiornamento, mantenimento e riutilizzo del codice.

### **Two tier:**

In questa seconda architettura si assiste ad una separazione tra la logica di accesso ai dati e la logica di business e di presentazione.

Dei “fat client” interagiscono con un database inviando query SQL e ricevendo dati raw.

La logica di presentazione, di business e di processamento del modello dei dati risiedono nell'applicazione client, il database contiene la logica di accesso ai dati.



- **Pro:**

Rispetto all'architettura single tier si ha indipendenza dallo specifico database.

- **Contro:**

Il livello di presentazione, il modello dei dati e logica di business sono ancora interdipendenti permangono pertanto le difficoltà di aggiornamento, di mantenimento e di riutilizzo.

Peraltro, il modello dei dati è tightly-coupled per ogni cliente, se cambia il database schema bisogna aggiornare tutti i client. Gli aggiornamenti devono quindi essere distribuiti a tutti i client, ciò comporta problematiche per quanto riguarda il mantenimento del sistema.

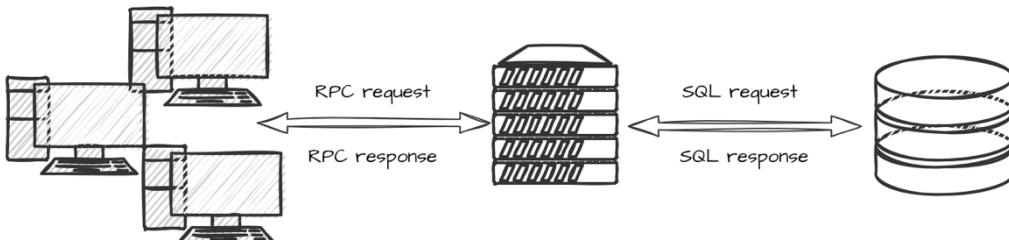
Si presentano inoltre problematiche legate alla scalabilità, in quanto ogni client apre una connessione con il database, se il numero di client cresce il DB non sarà più in grado di gestire le connessioni.

I dati vengono processati dal client, si ha quindi la necessità di trasferire “raw data” verso il client producendo overhead di rete.

### Three tier:

Architetture di questo tipo sono realizzate con dei “thin client” che ospitano solamente la logica di presentazione, per quanto riguarda le logiche di business e di processamento dei dati sono delegate ad un livello intermedio, la logica di accesso ai dati è contenuta nel terzo livello rappresentato dal database.

Il “middle tier”, si occupa di tutti i servizi di sistema (gestione della concorrenza, multithreading, transazioni, sicurezza, persistenza... ).



- **Pro:**

La logica di business è disaccoppiata dalla logica di presentazione ciò permette di modificarla in maniera più flessibile senza che impatti sulla logica di presentazione.

- **Contro:**

C’è un accoppiamento ancora abbastanza forte tra i client e il middle tier dovuto alle chiamate RPC (Remote Procedure Call). Complessità del middle tier. Scarsa riusabilità del codice.

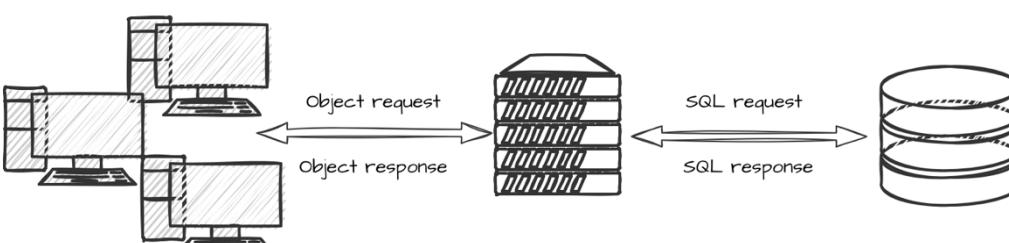
L’interazione tra client e middle tier è basata su tecnologie con un forte accoppiamento, che danno la possibilità di definire delle interfacce ma legata ad un modello funzionale. La naturale evoluzione di questa architettura è sempre un’architettura a tre livelli in cui però l’interazione tra client e middle tier è garantita tramite delle API basate su un modello a oggetti. Il middle tier è quindi un sistema Object oriented (CORBA o RMI).

### Three tiers (basato su Remote Object):

In questa architettura la logica di business e il modello dei dati sono encapsulati all’interno di oggetti ottenendo così una maggiore astrazione a livello di linguaggio di interfaccia.

I modelli ad oggetti più utilizzati sono CORBA (IDL) e RMI (Java).

(CORBA è un middleware per realizzare interazione tra oggetti distribuiti, come RMI, ma indipendente dal linguaggio utilizzato.)



- **Pro:**

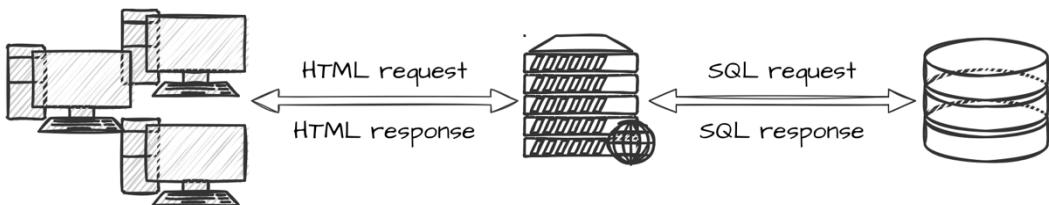
Meno strettamente accoppiato rispetto al modello basato su RPC.  
Maggior riusabilità del codice.

- **Contro:**

Middle tier ancora troppo complesso.

### Three tiers (Web Server):

In questa configurazione l'interazione tra il client e il middle tier è basata sul Web (HTTP, HTML). Il client deve disporre di un browser web che si fa carico della logica di presentazione, il middle tier gestisce la logica di business e il modello dei dati tramite tecnologie per “**dynamic content generation**”.



- **Pro:**

Supporto per tutti i client che possono interagire attraverso HTML e HTTP.  
Nessun problema di aggiornamento del software sul client.

- **Contro:**

Il middle tier resta ancora molto complesso, la logica di business deve far fronte alle problematiche specifiche dell'applicazione e di tutti i servizi di sistema (transazioni, concorrenza, sicurezza ecc...) in un'unica base di codice. Non c'è quindi una separazione netta tra la parte funzionale e quella non funzionale.

Attualmente la tendenza vede una transizione da un mondo monolitico e single tier ad un mondo multi tier che disaccoppia sempre di più, fa uso di tecnologie object oriented e facilita la standardizzazione del livello di presentazione utilizzando le tecnologie del web.

Tuttavia, restano ancora delle problematiche aperte, come visto in precedenza il middle tier rimane comunque molto complesso in quanto la parte di logica dell'applicazione non è ancora nettamente separata dalla parte di logica di tutti i servizi di sistema non funzionali, questo implica che essi vengano duplicati per ogni applicazione.

La soluzione a questa grande problematica consiste nell'introduzione di uno strato software ulteriore, il **container**, che si faccia carico di tutti servizi non funzionali cioè non legati alla logica applicativa. Tra le soluzioni proposte distinguiamo tra:

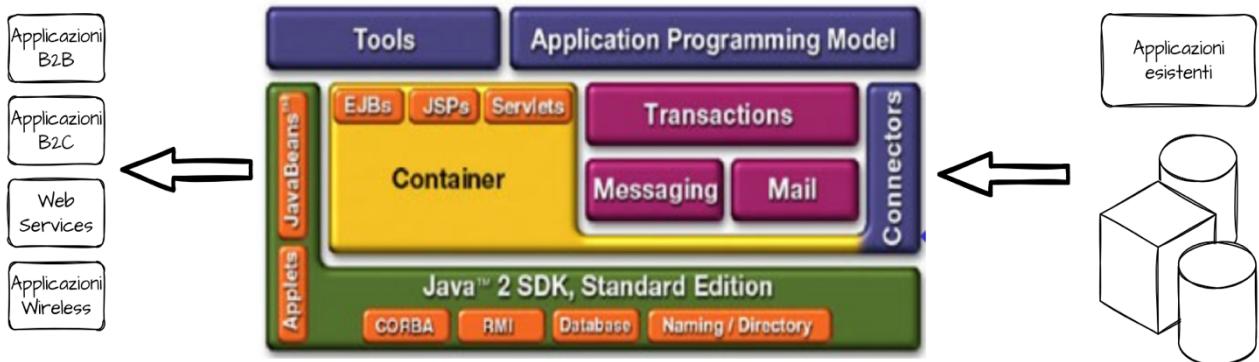
- **Soluzioni proprietarie:**

Usano il modello componente-container in cui i componenti gestiscono la parte di business logic e il container fornisce i servizi di sistema. Questo viene fatto in modo proprietario fornendo delle API (API proprietarie legate alla singola implementazione) per richiedere le funzionalità di sistema. Tra gli esempi più notevoli ci sono Tuxedo e .NET.

- **Soluzioni basate su standard aperti:**

Usano il modello componente-container come nel caso precedente ma forniscono i servizi di sistema in maniera ben definita in accordo a standard industriali tramite delle API standard. L'esempio più notevole è **JEE** (Java Enterprise Edition) che definisci questi standard, li implementa e abilita quindi questa interazione tra il componente (sviluppato in Java) e un container che realizza queste API standard.

## 1.3 Architettura J2EE per Application Server



### Client:

I client (a sinistra in figura) possono essere di diversi tipi:

- client che utilizzano le interfacce del Web interagendo in una logica **business to client**.
- client che sono essi stessi applicazioni o servizi e interagiscono quindi con una logica di tipo **business to business**, in questa logica tutte interazione sono abilitate in modo programmatico cioè automatizzate in modo che possano essere eseguite autonomamente da un software. Questo tipo di interazioni possono essere abilitate o tramite tecnologie del Web o tramite delle RMI.
- I client potrebbero essere dei **Web Services**, ovvero delle interazioni programmatiche che usano tecnologie come XML e standard per la rappresentazione dei dati e per l'effettuazione di chiamate. I WS sono una sorta di “super RPC” basate su delle interfacce che consentono la definizione dei servizi dei WS e una serie di tecnologie che consentono di scambiare richieste e risposte.
- I client potrebbero essere **applicazioni wireless** cioè delle applicazioni più leggere che interagiscono con l'applicazione Enterprise.

### Application Server:

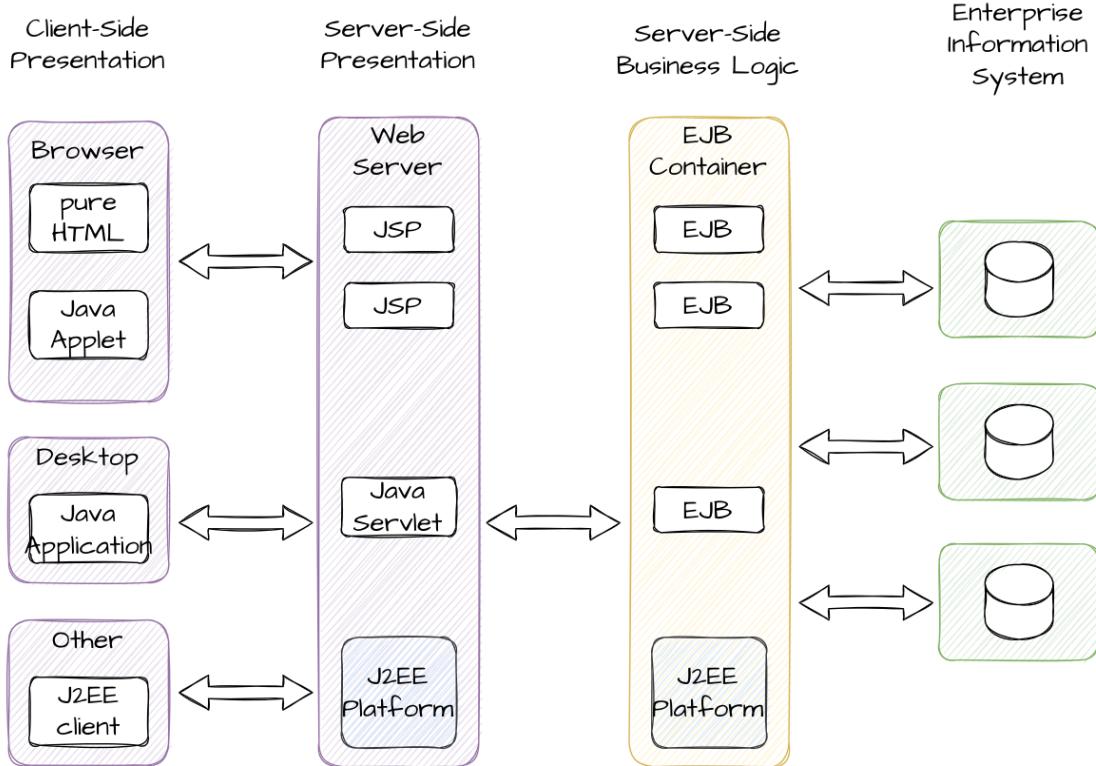
L'application server è costituito da:

- **Java standard edition** con tutte le funzionalità annesse, come RMI, Applets, la possibilità di interagire con un database tramite API standard, i servizi di naming e altro.
- **Container e supporto ai servizi di sistema** per abilitare l'integrazione con servizi esterni.
- **Supporto allo sviluppo** tutti i tool che permettono lo sviluppo di componenti (EJB) che vivono all'interno di questo ambiente.

### Enterprise Information System:

Rappresenta l'insieme di tutte le sorgenti dati esterne e l'interazione con servizi e infrastrutture preesistenti.

## 1.4 Architettura J2EE per applicazioni N-tier



Come già detto i client possono essere un browser, un applet (che comunica tramite RMI o tramite HTML), applicazioni desktop (che usano RMI) o altri “devices” che usano essi stessi il framework J2EE in una logica di interazione B2B.

Per quanto riguarda il middle-tier esso viene spacchettato in due parti:

- **Server-side presentation** si occupa della logica di presentazione, fa uso delle JSP (Java Server Pages) e delle Servlet.
- **Server-side business logic** realizza il container e la logica applicativa, cioè i componenti EJB.

Infine, a destra in figura l'insieme del mondo esterno con cui si intende interagire.

### 1.4.1 Altre soluzioni

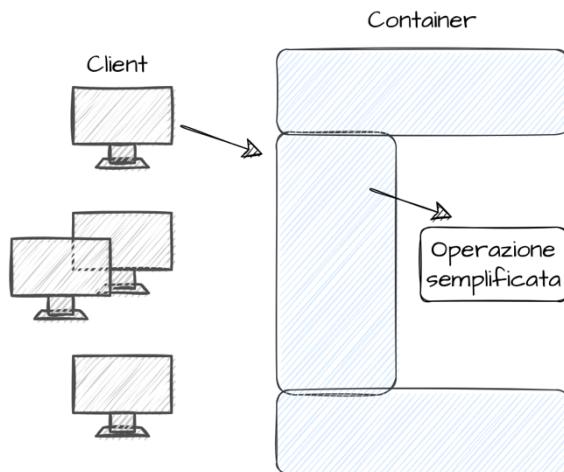
Oltre al J2EE esistono altre soluzioni. Il problema sta nel fatto che le tecnologie accennate finora garantiscono portabilità, facilità nella realizzazione delle componenti applicative con tutti i vantaggi di sistema, ma tali soluzioni sono ancora abbastanza pesanti dal punto di vista dell'esecuzione e sebbene facciano largo uso di standard tendono a far perdere elasticità alle interfacce di interazione tra i componenti rendendo più difficile l'evoluzione dei sistemi.

Nel 2003 nasce un progetto industriale che mira alla realizzazione dei cosiddetti **modelli a container leggeri**: Spring. Tale progetto mira a rendere più flessibili le applicazioni sviluppate all'interno del container leggero e a diminuire il carico durante l'esecuzione. Si parlerà di Spring in maniera più approfondita più avanti.

## 1.5 Modelli a contenimento

I container si fanno carico di tutte le funzionalità di sistema lasciando il programmatore libero di focalizzarsi sulle parti di logica applicativa. Il container gestisce tutte le funzionalità di sistema spesso introducendo politiche di default ed evitando che si verifichino errori.

L'idea principale del modello a contenimento prevede che i client non interagiscano direttamente con il componente di interesse, ma che passino attraverso il container che in qualche modo standardizza e facilita le operazioni di interazione. Il container al suo interno ospiterà il componente.



Questo concetto è applicabile a molte tecnologie diverse come CORBA, Engine per sistemi grafici, Container per servlet e così via.

Il Container può fornire automaticamente molte delle funzioni per supportare il servizio applicativo verso l'utente:

- **Supporto al ciclo di vita:** attivazione/disattivazione del servitore, la possibilità di mantenere lo stato interno del componente durante tutta la durata della sessione, la possibilità di rendere persistente lo stato del componente.
- **Supporto allo spazio di nomi:** questo è un aspetto fondamentale, infatti, in un sistema distribuito si ha la necessità di recuperare i servizi offerti da componenti sparsi, è necessaria quindi la presenza di un **discovery service**. Peraltro, in sistemi molto "larghi" possono presentarsi situazioni in cui non si ha a che fare con un singolo container ma un insieme di container **federati** cioè container distribuiti gestiti da diverse entità, fondamentale anche in questo caso è la presenza di un servizio di nomi per abilitare il coordinamento tra questi.
- **Supporto alla qualità del servizio:** nei sistemi enterprise è importante garantire un'eccellente qualità del servizio, questo implica che il sistema debba essere tollerante ai guasti, efficiente ciò significa avere un occhio di riguardo per tutti quell'indicators di **QoS** (Quality of Service).
- **Sicurezza:** rendere sicuro il sistema.

## 1.5.1 JEE

Particolare attenzione è rivolta a Java Enterprise Edition perché:

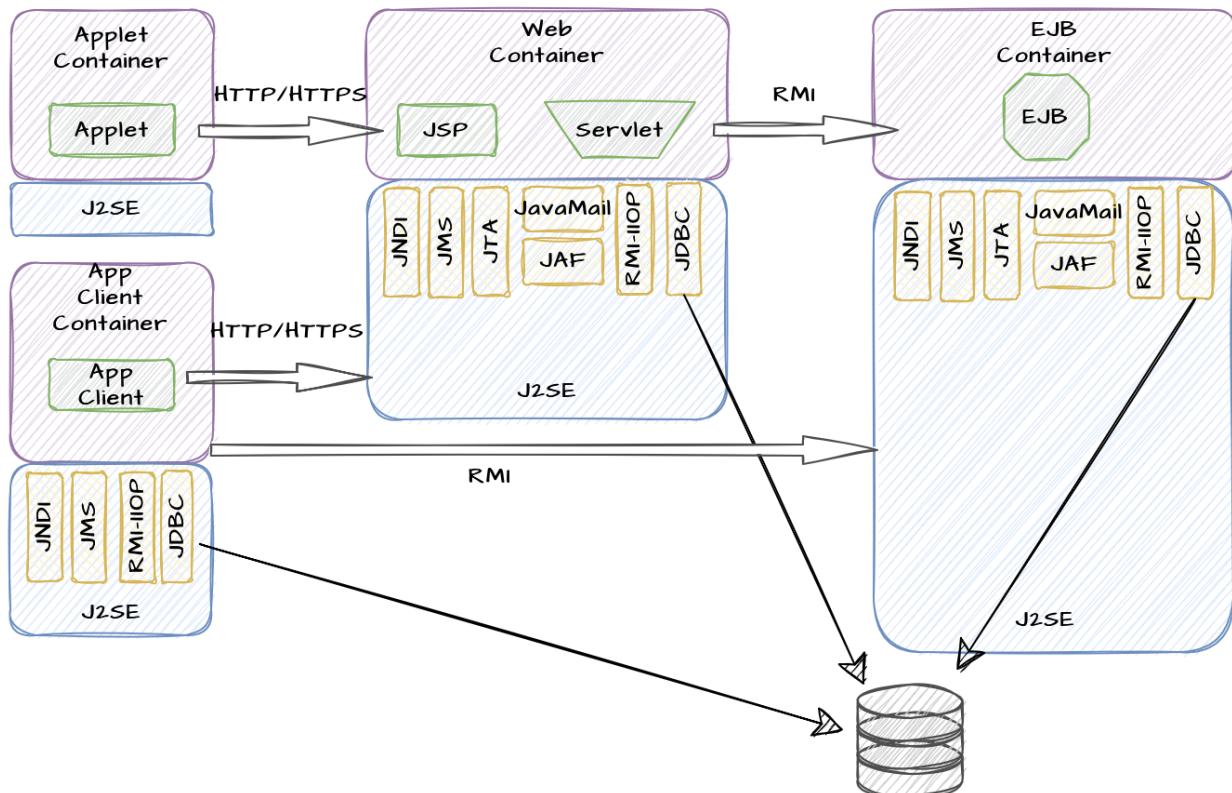
- è largamente diffusa a livello industriale
- definisce uno standard di riferimento basato su tecnologie Java e quindi molto portabile
- possiede una specifica ben fatta e condivisa da molti player industriali che la implementano
- possiede un insieme di test di compatibilità (CTS) che consentono di capire se una certa implementazione conforme allo standard oppure no.

L'utilizzo di JEE come application server evita il "lock-in" e quindi chi fa uso di queste tecnologie utilizzando solo API standard e non API proprietarie, può portare i componenti da un'implementazione del container Java ad un'altra. Inoltre, le applicazioni conformi allo standard JEE sono altamente portabili.

JEE offre un enorme ecosistema di implementazione e servizi di sistema conformi allo standard, lo standard ben definito permette a tale ecosistema di essere in continua crescita.

## 1.6 Anteprima su EJB

Gli **EJB** (Enterprise Java Bean) sono una tecnologia per sviluppare componenti server-side che facilitano lo sviluppo e il deployment di applicazioni multi-tier portabili con tutti i vantaggi precedentemente discussi del modello a componenti usato lato server. Inoltre, tutto questo grazie all'utilizzo di Java è portabile su architetture diverse.

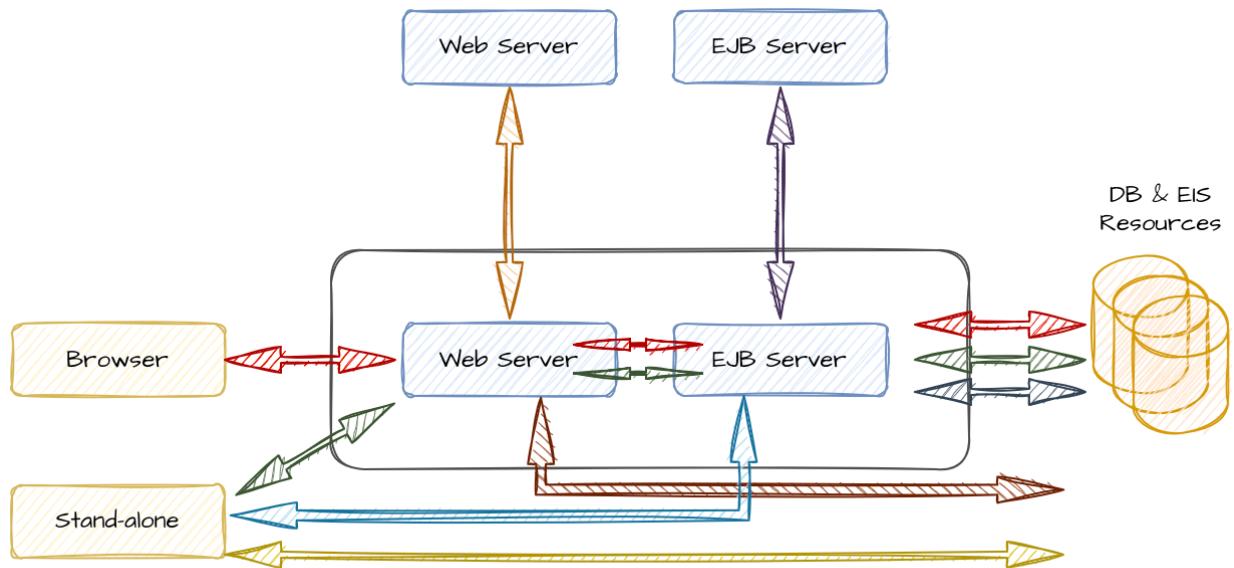


Il modello componente container può essere di diversi tipi:

- **Applet** il cui deployment viene fatto all'interno di una JVM SE
- **Web Container**
- **EJB Container**

Nella figura sovrastante in viola sono rappresentati i vari container, in verde i componenti che vivono all'interno di quel container, in azzurro vengono rappresentate le parti di supporto, cioè il “run-time environment”, in giallo tutte le API standardizzate per gestire per esempio la parte di “naming e discovery”, per gestire la parte di transazionalità, di messaggistica, di accesso ai database e così via e infine le frecce rappresentano i protocolli per gestire le interazioni (HTTP, RMI).

## 1.7 J2EE per Applicazione N-tier



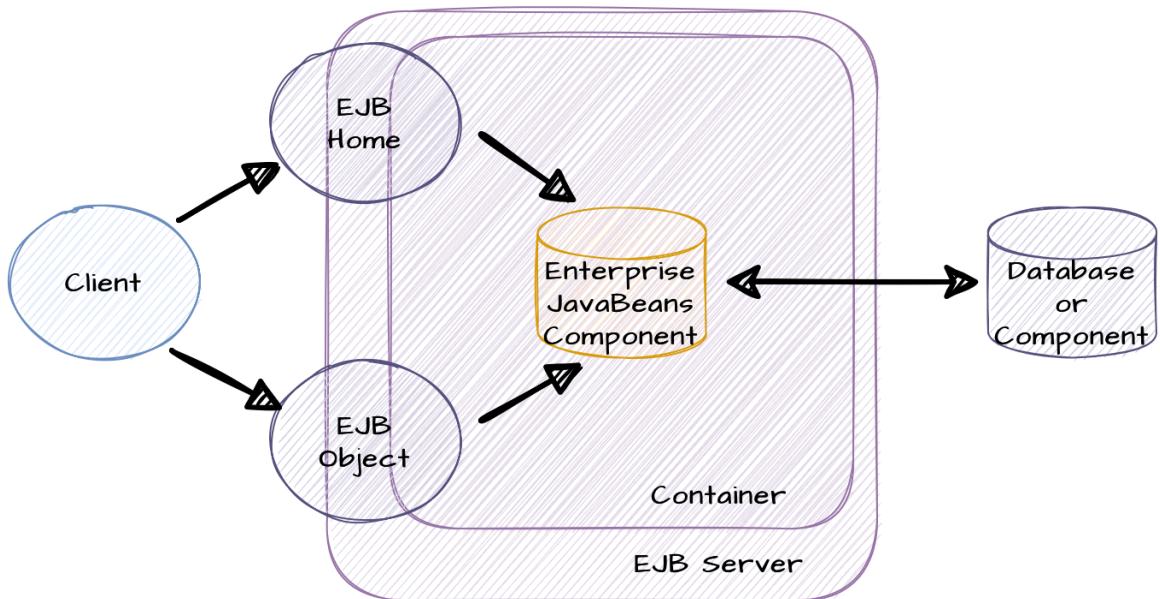
### Modello 4-tier e applicazioni J2EE:

In questo modello abbiamo un client che comunica tramite HTML con una parte di presentazione web basata su JSP e Servlet, EJB e il database con le connessioni al database.

### Modello 3-tier e applicazioni J2EE:

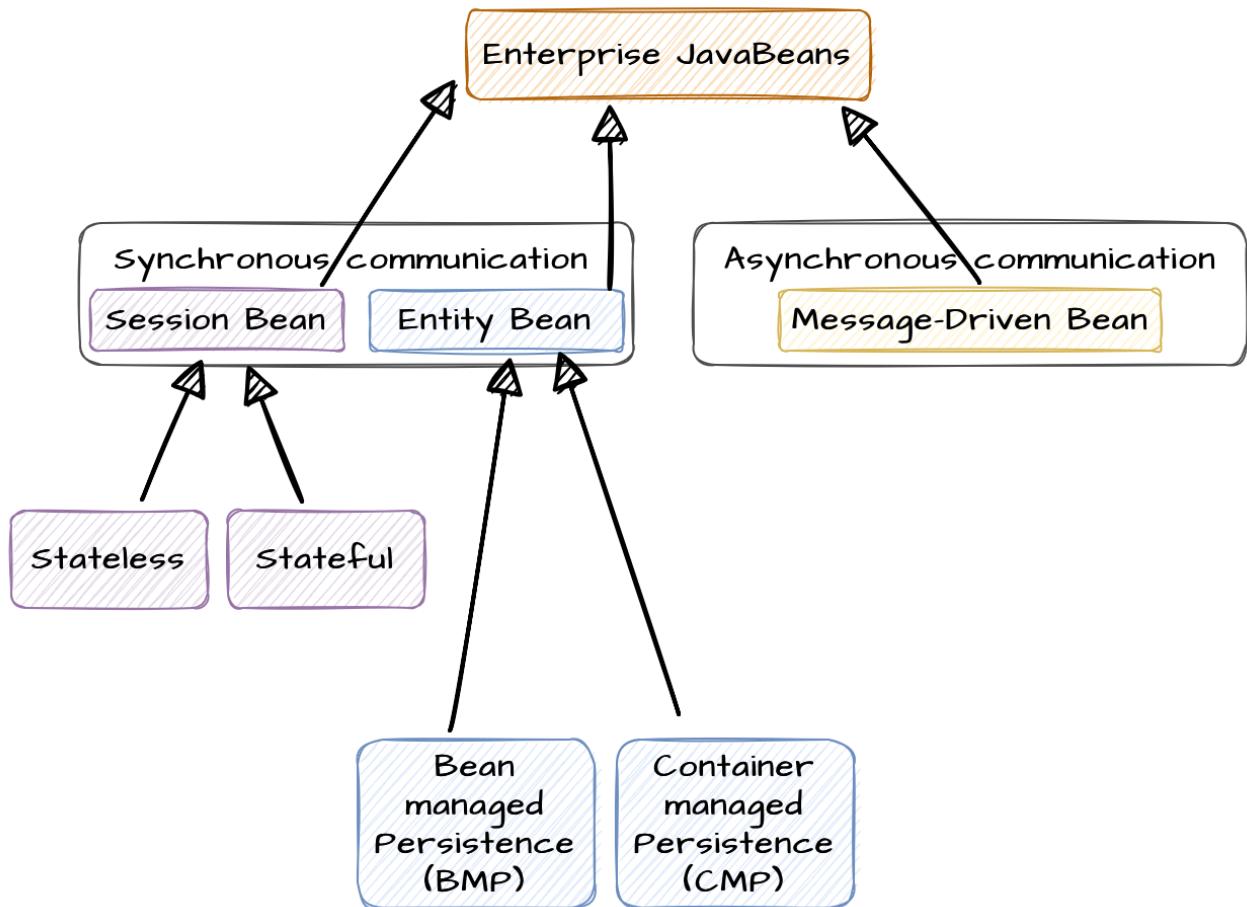
In questo modello il client parla tramite HTML con il server web che comunica direttamente con il database oppure, delle applicazioni stand-alone EJB che comunicano direttamente con l'EJB server e in ultima istanza con il database. Nelle applicazioni di tipo enterprise B2B le interazioni vanno tramite messaggi JMS o basate su XML.

## 1.8 Architettura EJB



Il client può interagire con il componente attraverso le interfacce **Home Interface** (EJB Home) che serve a recuperare il componente con il quale il client vuole interagire e **Remote Interface** (EJB Object) che serve ad esporre le interfacce di business. La logica di business è racchiusa all'interno dell'**Enterprise JavaBeans Component**, infine, lato backend troviamo il database. Il componente viene attivato all'interno di un container che risiede all'interno di un **EJB Server**.

### 1.8.1 Componenti EJB



I principali componenti presenti nella Java Enterprise Edition possono essere divisi in due categorie distinte, componenti che supportano comunicazioni di tipo sincrono (**Synchronous communication**) o comunicazioni di tipo asincrono (**Asynchronous communication**).

#### Synchronous communication:

Appartengono a questa categoria due diversi tipi di componenti:

- **Session Bean:** sono componenti con sessione, essi possono essere con stato, **Stateful**, o senza stato **Stateless**.
- **Entity Bean:** sono componenti che rappresentano le entità di business, la loro persistenza può essere gestita dal componente stesso, si parla in questo caso di **Bean Managed Persistence (BMP)**, oppure può essere gestita dal container, si parla in questo caso di **Container Managed Persistence (CMP)**.

#### Asynchronous communication:

Appartengono a questa famiglia i **Message-Driven Bean**, sono componenti che non vengono attivati su chiamata, ma da messaggi che arrivano in maniera asincrona.

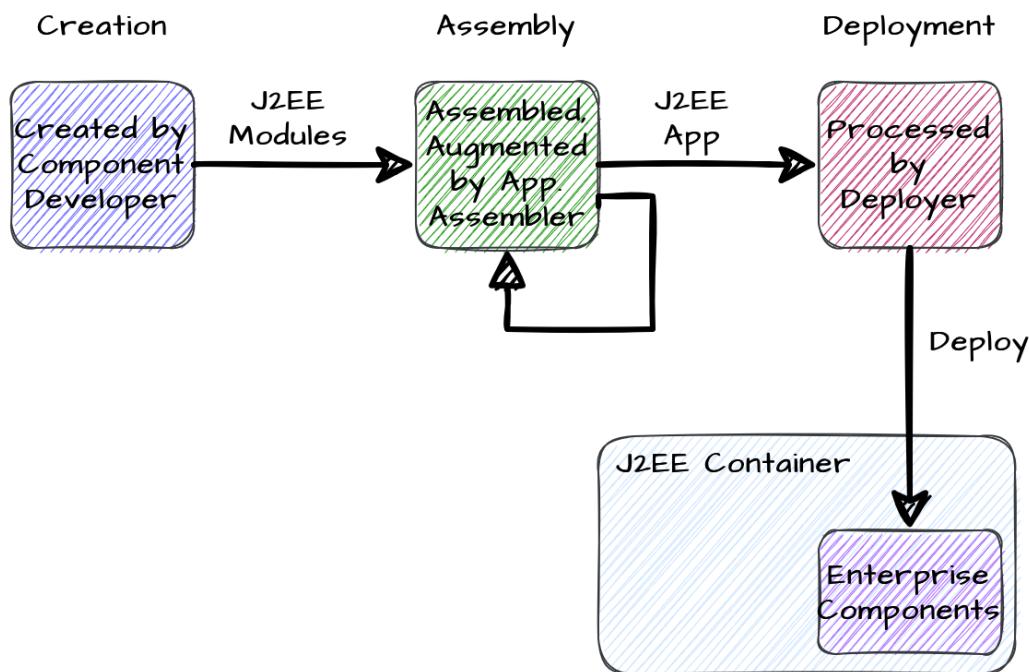
## 1.9 Componenti e Container

Il modello componente container permette, nello sviluppo di un'applicazione, di dividere i compiti, in modo che alcuni siano gestiti in autonomia dal container e altri siano gestiti dal componente. Il container si interpone nell'interazione con il client e attraverso delle semplici API aggiunge tutte le funzionalità che sono ortogonali alla logica di business che invece viene racchiusa all'interno del componente, è possibile che implementazioni diverse della stessa API gestiscano tali funzionalità in maniera diversa, ma in ogni caso saranno gestite. Tra le funzionalità demandate al container ci sono:

- **Concorrenza:** il container si occuperà di gestire tutti gli aspetti legati alla concorrenza (accessi concorrenti, threading).
- **Sicurezza:** il container si occupa di tutte le politiche di accesso.
- **Scalabilità:** il container decide se e come scalare il componente.
- **Replicazione:**
- **Transazionalità:**
- **Life-cycle management:** quando attivare/disattivare un componente.

Dall'altro lato al componente resta “solo” da gestire la parte di logica applicativa e di presentazione.

### Ciclo di vita delle Applicazione J2EE:



Il ciclo di sviluppo di un'applicazione enterprise parte dalla creazione dei bean da parte del **Component Developer** che si occupa anche di scrivere il **deployment descriptor** che in maniera dichiarativa istruirà il container sui comportamenti da assumere rispetto a tutte quelle funzionalità viste in precedenza (sicurezza, concorrenza, scalabilità...). In precedenza, il deployment descriptor era esterno e basato su XML, lo standard attuale prevede di inserire il deployment descriptor “embedded” nel codice tramite le Java Annotation. Il component developer rilascia, quindi, moduli EJB.

I moduli posso essere assemblati insieme in un'applicazione dall' **Application Assembler** che rilascia un'applicazione EJB a cui il **Deployer** aggiunge il deployment descriptor e poi dispiega in un server J2EE.

#### **Deployment Descriptor:**

Fornisce istruzioni al container su come gestire e controllare il comportamento di componenti J2EE, essendo scritto in un linguaggio dichiarativo si possono modificare le politiche rispetto alle funzionalità di sistema senza dover ricompilare il tutto. Il fatto che il deployment descriptor segua una specifica dichiarativa permette una forte portabilità del codice.

## 2. Modelli a componenti ed Enterprise Java Beans

### 2.1 Enterprise Java Beans (EJB)

Gli EJB presentano un'architettura a componenti che consente lo sviluppo e il deployment di applicazioni business distribuite basate su un modello a componenti. L'architettura EJB integra tutte le caratteristiche non funzionali discusse precedentemente, scalabilità, transazionalità e sicurezza, con la promessa di essere un'applicazione portabile, cioè una volta scritto il codice dell'applicazione esso può essere “deployato” su qualsiasi piattaforma server che supporti EJB. In sostanza: EJB rappresenta una tecnologia a componenti server-side, offre supporto allo sviluppo e al deployment di applicazioni distribuite Java Enterprise che siano multi-tier, transazionali, portabili, scalabili, sicure, ...

Per quanto riguarda i principi di design dei componenti EJB essi sono:

- **Auto-contenimento:** i componenti devono essere debolmente accoppiati (loosely coupled) per garantirne la riusabilità.
- **Interfacce:** il comportamento dei componenti deve essere definito tramite interfacce.
- **Gestione delle risorse:** i componenti non si occupano direttamente della gestione delle risorse, ma demandano al container.
- **Container:** le applicazioni sono affiancate da un container che semplifica la fase di sviluppo facendosi carico della gestione di tutti i servizi di sistema.
- **N-tier:** le applicazioni EJB sono N-tier
- **Session tier:** è un insieme di API pensate per l'applicazione e la gestione dell'interazione con i client.
- **Entity tier:** è un insieme di API che facilita l'interazione con il livello backend del database.

#### Auto-contenimento e Loosely coupling

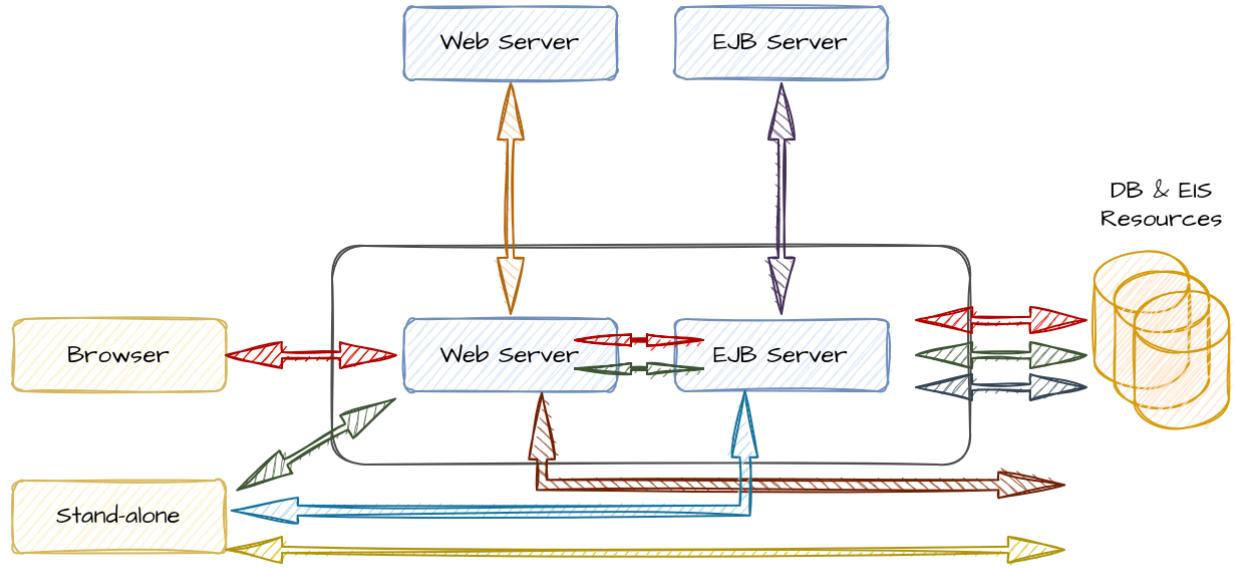
Questa caratteristica garantisce un'alta portabilità, lo scarso accoppiamento è ottenibile tramite la definizione di interfacce per l'interazione tra componenti e container. Lo sviluppo di componenti EJB non richiede conoscenza e visibilità approfondite dell'ambiente di esecuzione (container) e sebbene ciò da un lato semplifichi il lavoro in fase di sviluppo non è sempre auspicabile. In scenari in cui si ha a che fare con applicazioni che hanno una variabilità di carico molto ampia, e quindi necessitano di un'alta scalabilità, conoscere le politiche di gestione e avere una maggiore visibilità di come vengono gestite le risorse di sistema sarebbe più auspicabile. Per quanto riguarda l'interazione dei componenti EJB con i client esse sono specificate completamente in termini di interfacce Java. Le interfacce espongono i metodi che il client può invocare pur essendo all'oscuro dell'implementazione. Questo garantisce un elevato supporto alla portabilità e all'estendibilità.

#### Gestione delle risorse:

I componenti EJB possono accedere a risorse esterne (database, sistemi legacy) per mezzo del container che li ospita, non c'è quindi necessità all'interno della logica applicativa di esplicitare le operazioni di allocazione e de-allocazione.

La gestione delle risorse è quindi demandata al container con obiettivi di massima efficienza, la configurazione del container è compito, tipicamente, degli amministratori di sistema. Gli sviluppatori specificano i requisiti in modo dichiarativo, tramite annotazioni o tramite il deployment descriptor.

### 2.1.1 EJB e utilizzo da parte di client differenti



I componenti EJB, come visto in precedenza, possono essere utilizzati in diverse architetture N-tier. Le interazioni tra client e componenti EJB possono essere di diverso tipo, tipicamente si tratta di interazioni B2B (Business to Business) tra componenti EJB che interagiscono direttamente con il server EJB, oppure si possono avere componenti come JSP o Servlet che fanno parte del tier di presentazione e che interagiscono con gli EJB.

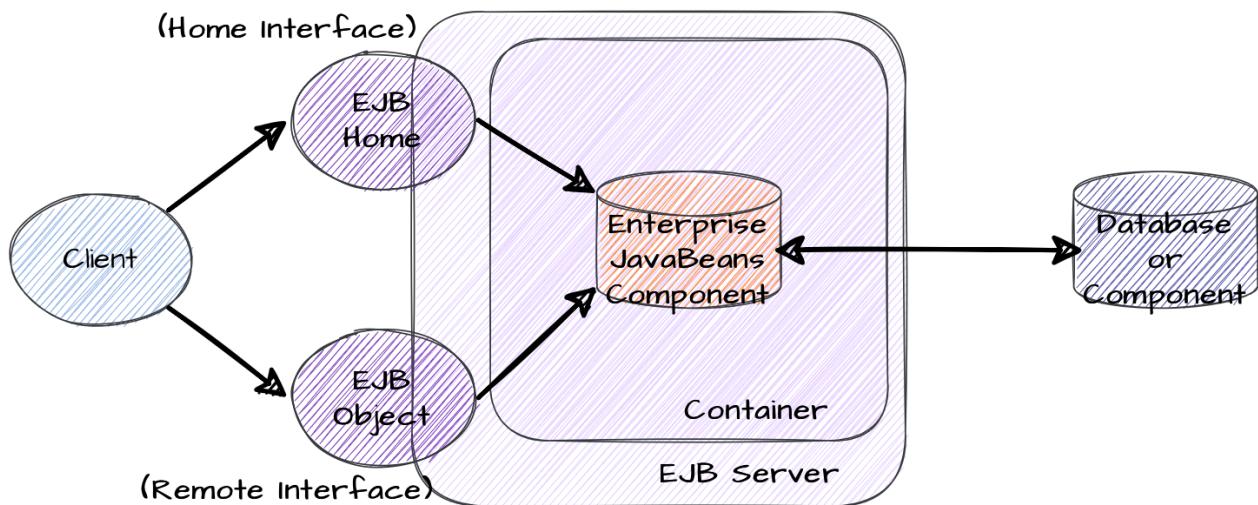
Sebbene gli EJB portino lato Server tutti i benefici del modello a componenti, separando la logica di business e il codice di sistema, offrendo un framework di supporto per componenti portabili, facilitando la configurazione a deployment-time tramite l'utilizzo di deployment descriptor e annotazioni, non sempre essi si prestano ad essere la soluzione migliore. Gli EJB sono interessanti dal punto di vista tecnologico quando il sistema che si progetta ha dei requisiti che sono di tipo enterprise, cioè è un sistema che deve necessariamente scalare in maniera automatica, deve semplificare l'integrazione con dei software preesistenti, consentire la modifica delle politiche in maniera veloce e flessibile. In altre situazioni, per esempio, quando si ha che fare con delle semplici applicazioni low-cost o delle pagine Web dinamiche l'utilizzo degli EJB è esagerato.

## 2.2 EJB - 2.x

Lo standard 2.x di EJB presenta dei problemi che hanno motivato il passaggio verso lo standard 3.x. Alcuni tra questi problemi riguardano:

- **La pesantezza**, che introduceva un elevato overhead in termini computazionali.
- **Non conformità a OO**, il framework aveva un modello di programmazione complicato e non conforme al classico modello OO.
- **Testing**, complicazioni in fase di testing.

### 2.2.1 Architettura EJB 2.x



L'architettura EJB nella versione 2.x è composta da tre parti fondamentali:

- **Home Interface:** definisce i metodi che saranno usati dai client per creare e ritrovare i componenti bean.
- **Remote Interface:** o Object Interface definisce i metodi di business per il componente.
- **Enterprise JavaBeans Component:**

A livello architetturale le due interfacce, Home e Object, forniscono un livello di indirezione tramite due oggetti, rispettivamente, EBJ Home e EBJ Object che vengono creati a tempo di deployment in automatico dal container.

L'oggetto EBJ Home lavorando come proxy locale intercetta la richiesta del client e interagisce con il container per portare a termine l'operazione di creazione dell'istanza del componente richiesto. Il grande vantaggio di questo approccio è che il proxy interponendosi fra client e container, per intercettare le chiamate, interagendo con il container può realizzare diverse politiche di deployment (esempio: se il proxy intercetta una richiesta per un certo componente "Y" ed esiste già, all'interno del container, un pull di istanze di Y non ne verrà creata una nuova ma sarà utilizzata una di quelle nel pull.).

L'oggetto EBJ Object intercetta le chiamate provenienti dal client, e permette di realizzare tutti i servizi di sistema (gestione della sicurezza, transazionalità, ecc...).

### 2.2.2 Contratti in EJB 2.x

I contratti in EJB servono a definire la vista del client sul mondo EJB (**Client view contract**) o a definire l'interazione tra il componente e il container (**Component contract**).

#### Client view contract

Il client view contract è un contratto tra il client, che richiede l'uso di un componente, e il container, con cui il client interagisce e che ospita il componente richiesto. Come già detto il client di un componente EJB può avere natura diversa, può essere un componente del Web (Servlet o JSP), un'applicazione Java standalone, un applet, un altro componente EJB o un client Web Services.

Il contratto include:

- **Home Interface:** fa le veci di una factory consentendo di creare e recuperare i componenti EJB.
- **Object Interface:** racchiude i metodi di business.
- **Identità dell'oggetto:** l'identificativo è fondamentale importanza per il servizio di nomi che si occupa del recupero della Home Interface.

### Component contract:

È un contratto tra il componente EJB e il container che lo ospita. Il contratto si occupa di:

- **Methods:** abilita le invocazioni dei metodi dai client.
- **Life cycle:** si occupa della gestione del ciclo di vita delle istanze dei componenti EJB.
- **Implements:** implementa le interfacce Home e Object, creando gli oggetti proxy discussi in precedenza.
- **Persistence:** se necessario fornisce supporto alla persistenza
- **System services:** gestirà tutti gli aspetti legati ai servizi di sistema come: transazionalità, sicurezza ecc...

Quindi la vista client di un componente EJB è definita strettamente dalle interfacce e dai contratti. Le operazioni di creazione e di ritrovamento dei componenti EJB sono standardizzate grazie alla Home Interface, che rappresenta il punto di partenza, e all'uso di identificativi a supporto del servizio di nomi, che si occuperà del recupero della Home Interface.

Il container EJB facendosi carico della gestione della concorrenza fornisce l'illusione di un ambiente single-thread, si occupa della gestione delle transazioni, delle problematiche di accesso e sicurezza e della gestione efficiente delle risorse con la possibilità di **pooling di istanze**.

Dal punto di vista degli EJB essi diventano dei componenti software auto-contenuti che sono presentati ai client tramite le due interfacce Home e Object che forniscono al client una factory per la creazione e il ritrovamento del componente EJB e i metodi di business del componente. Tutte le chiamate verso i metodi del componente sono intercettate dal container. Gli oggetti proxy che implementano le due interfacce sono **EJBHome** e **EJBOBJECT** sono generati automaticamente dal container e rappresentano il punto di contatto tra i client, il container e il componente EJB.

I client istanziano nuovi componenti EJB o trovano componenti esistenti con modalità ben definite.

I client utilizzano il servizio di nomi offerto da **JNDI**. JNDI è un servizio di nomi nel mondo Java, simile a RMI ma più generale che non assume, per esempio, di avere dall'altra parte un server RMI. Il client interroga JNDI per ottenere un oggetto proxy, in realtà JNDI restituisce al client il riferimento a uno stub dell'oggetto EJBHome (implementazione della Home Interface).

Il client possiede adesso lo stub di un oggetto EJBHome, può quindi utilizzare i metodi definiti dalla Home Interface e implementati da tale oggetto, **create()** e **find()**. Questi metodi

restituiscono al client un altro oggetto proxy, che in realtà è il riferimento a uno stub dell'oggetto EJBObject (implementazione della Object Interface), tale oggetto espone al client tutti i metodi del componente EJB.

Non c'è, quindi, un'interazione diretta tra il client e l'istanza del componente EJB, i client interagiscono sempre con oggetti proxy e tale interazione è mediata dal container.

Il container gestisce il pool di istanze di componenti, quando riceve una richiesta di creazione di un componente, se nel pool è presente un'istanza di quel componente inutilizzata, il container in modo trasparente restituisce al client quell'istanza.

Tutte le risorse esterne come database, Enterprise Information System, Sistemi di messaging, sono condivise fra i diversi componenti EJB e il container si fa carico di condividere l'accesso a queste risorse fra tutti i componenti che sono "deployati" su quel container.

Quindi dal punto di vista pratico il container si fa carico di:

- **Generazione:** generare automaticamente le classi concrete che realizzano l'interfaccia Home (remota o locale) e l'interfaccia Object (remota o locale).
- **Binding:** effettuare il binding dell'oggetto Home presso il servizio di naming (fondamentale dal momento che i client fanno lookup dei componenti attraverso JNDI).
- **Pooling:** creare e gestire un pool di istanze di componenti.
- **Caching:** effettuare il caching dei componenti acceduti di recente.
- **Connect:** gestire un pool di connessioni JDBC verso un database.

### 2.2.3 Terminologia

- **Classe Bean:** classe Java che implementa il componente.
- **Istanza di Bean:** La reale istanza dell'oggetto Bean all'interno di un EJB container.
- **Modulo EJB:** un file .jar che contiene tutte le classi dei componenti Bean.
- **Applicazione EJB:** è una pacchettizzazione, basata su file \*.ear di moduli EJB e quindi una collezione di componenti che costituiscono un'intera applicazione.
- **Interfaccia EJBHome:** interfaccia che definisce i metodi per la creazione/ritrovamento del componente.
- **Oggetto EJBHome:** chiamato anche oggetto factory è un'implementazione dell'interfaccia EJBHome.
- **Interfaccia EJBObject:** interfaccia Java che definisce i metodi di business del componente.
- **Oggetto EJBObject:** implementazione dell'interfaccia EJBObject.

## 2.3 Tipologie di componenti Bean

Come già anticipato precedentemente le principali tipologie di bean sono:

- **Session Bean**
  - Stateful session bean
  - Stateless session bean
- **Entity Bean**
  - Bean managed persistence (BMP)

- Container managed persistence (CMP)
- **Message Driven Bean**
  - Per Java Message Service (JMS)
  - Per Java API for XML Messaging (JAXM)

### 2.3.1 Session Bean

I session bean lavorano solitamente con un singolo client per volta, hanno un tempo di vita piuttosto breve, non sono quindi persistenti e vengono persi in caso di failure dell'EJB server. Il fatto di non essere persistenti implica che essi non rappresentino dati all'interno del database, permettono tuttavia di accedere e modificare i dati nel DB.

La classe corrispondente ad un session bean implementa l'interfaccia ***javax.ejb.SessionBean***.

I session bean possono essere utilizzati in diversi scenari:

- Per modellare oggetti di processo o di controllo specifici per un particolare client.
- Per modellare un workflow o attività di gestione all'interno dell'applicazione o per coordinare l'interazione fra bean.
- In un'architettura multi-tier per muovere la logica applicativa di business dal lato client al lato server

#### Stateless Session Bean

In questo caso il session bean non mantiene alcun tipo di stato, esegue una richiesta e restituisce il risultato senza salvare alcuna informazione di stato relativa al client. I session bean, quindi, possono essere definiti come elementi temporanei di business logic necessari per uno specifico client per un intervallo di tempo limitato.

Questo tipo di bean può essere usato in scenari come, per esempio, la consultazione di un catalogo merci, in situazioni del genere, infatti, non si ha la necessità di mantenere stato specifico per un certo client e la business logic può essere utilizzata senza necessità di accesso al database.

#### Stateful Session Bean

In questo caso il session bean può mantenere lo stato di uno specifico client. Durante tutta la durata dell'interazione tra il client e il session bean quest'ultimo mantiene una sorta di "soft-state" del client.

Questo tipo di bean può essere utilizzato in scenari come, per esempio, un carrello della spesa in cui si ha la necessità di mantenere lo stato per uno specifico client.

#### Pooling di Session Bean

Per quanto riguarda il pooling i due tipi di session bean vengono gestiti in maniera differente, gli stateless session bean non sono legati ad un particolare client e quindi, se un'istanza è libera può essere utilizzata da altri client, al contrario invece, per gli stateful session bean ogni istanza del bean è associata ad uno specifico client, per questo motivo è necessario preservare quell'istanza evitando che altri client la utilizzino.

### 2.3.2 Entity Bean

Gli entity bean forniscono una “vista ad oggetti” del database (tipicamente relazionale), il loro tempo di vita non è legato alla durata dell’interazione con il cliente; infatti, a differenza degli stateful session bean che mantengono un “soft-state” per uno specifico cliente, gli entity bean rappresentano uno stato persistente che viene mantenuto in un database e che viene condiviso tra tutti i clienti.

Componenti di questo tipo permangono nel sistema fino a che i dati esistono nel database, nella maggior parte dei casi sono sincronizzati con i relativi DB e lo stato che essi rappresentano è accessibile in maniera condivisa da clienti differenti.

La classe corrispondente ad un entity bean implementa l’interfaccia **javax.ejb.EntityBean**.

### Interazione tra client ed Entity Bean

I client possono interagire con gli entity bean per effettuare differenti operazioni:

- **Creazione:** creare un entity bean significa aggiungere una riga a una tabella di un database.
- **Ritrovamento:** trovare un entity bean significa determinare una riga in una tabella di un database esistente.
- **Rimozione:** rimuovere un entity bean significa eliminare una riga da una tabella di un DB.

C’è quindi un forte legame tra questi componenti e i database, al punto che ogni istanza di un entity bean ha un identificatore unico chiamato chiave primaria.

Gli entity bean possono essere usati per esempio per modellare e profilare un cliente, i dati del cliente devono essere persistenti, sono mantenuti in un database che è fault tolerant rispetto ai guasti del server, ogni cliente avrà un identificatore univoco e i dati del cliente potranno essere condivisi da diverse applicazioni.

### Container Managed Persistence (CMP)

La gestione della persistenza è totalmente demandata al container, i requisiti di persistenza sono specificati interamente nel deployment descriptor; quindi, gli sviluppatori di bean CMP non devono occuparsi della logica di persistenza all’interno del codice del componente. I pro di questo approccio sono: alta efficienza, performance e facilità di sviluppo e deployment.

### Bean Managed Persistence (BMP)

In questo caso la logica di persistenza deve essere gestita dallo sviluppatore del bean BMP, ciò permette un controllo totale dell’entity bean, ma ha come contro una complessità maggiore e una potenziale perdita di efficienza.

## 2.3.3 Message-Driven Bean (MDB)

I message-driven bean possono essere visti come consumatori di messaggi asincroni, in quanto tali non presentano un contratto chiaro verso i clienti, cioè, non posso essere invocati direttamente dal client.

I MDB non presentano le interfacce EJBHome e EJBObject, essi saranno attivati in seguito all’arrivo di un messaggio.

I client interagiscono in maniera indiretta con i MDB inviando messaggi verso le code o i topic su cui questi bean sono in ascolto, il message-driven bean riceve il messaggio e a questo punto può eseguire la sua logica applicativa.

I MDB sono privi di stato perché sono pensati per delle operazioni di vita breve. Come già detto sono dei consumatori di messaggi asincroni, il fatto di essere asincroni è importante perché evita di mettere in attesa il cliente che ha inviato il messaggio.

### Message-Driven Bean per JMS

In questo caso il bean MDB deve implementare l'interfaccia di ascolto `javax.jms.MessageListener` e il metodo `onMessage()` che verrà invocato a callback all'arrivo di un messaggio. Il bean viene configurato come listener per una queue o un topic JMS.

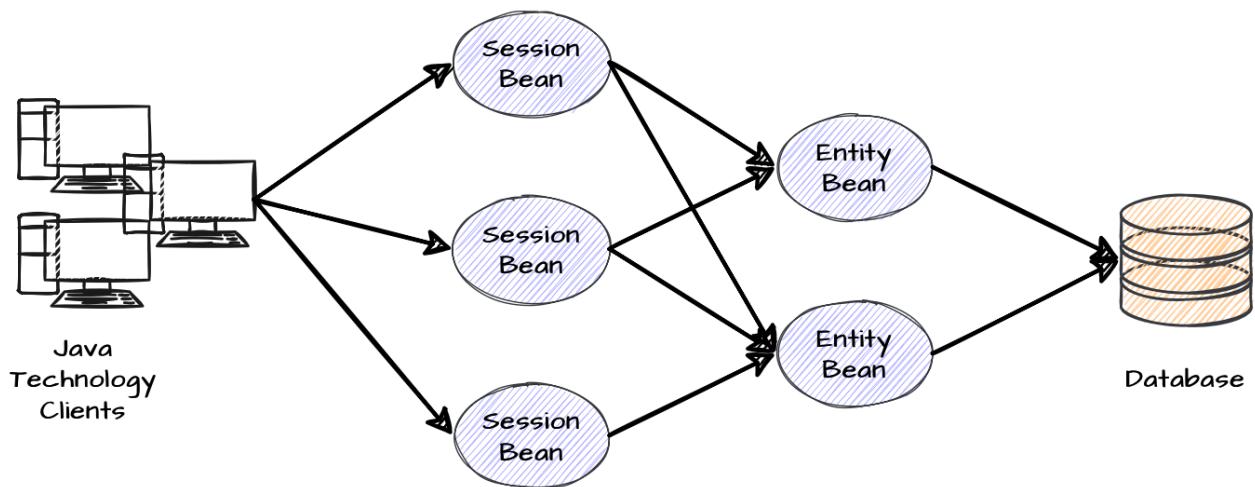
### API JMS

Le API JMS per l'invio di messaggi sono disponibili per qualunque tipo di componente EJB che quindi può avere una parte che lavora message-driven con interazioni che possono essere:

- **Molti a molti:** utilizzando una modalità pub/sub.
- **Uno a uno:** utilizzando una modalità unicast.

### 2.3.4 Scenari applicativi

L'immagine seguente mostra come interagiscono i client con i bean e come i bean stessi interagiscono tra loro. L'immagine mette solo in evidenza l'interazione con e tra componenti senza considerare gli aspetti legati al container, come, per esempio, la presenza della Home Interface e della Object Interface.



Agli estremi della catena di interazione vi sono, ovviamente, da un lato i client Java e dall'altro il database, in mezzo si pongono i componenti che danno vita all'applicazione. Il cliente interagisce con i session bean che realizzano la logica di controllo della sessione e la logica applicativa senza stato, se fosse necessario persistere uno stato i session bean possono interagire con gli entity bean.

Le design guide line dei componenti guidano lo sviluppatore nella scelta di uno o un altro bean:

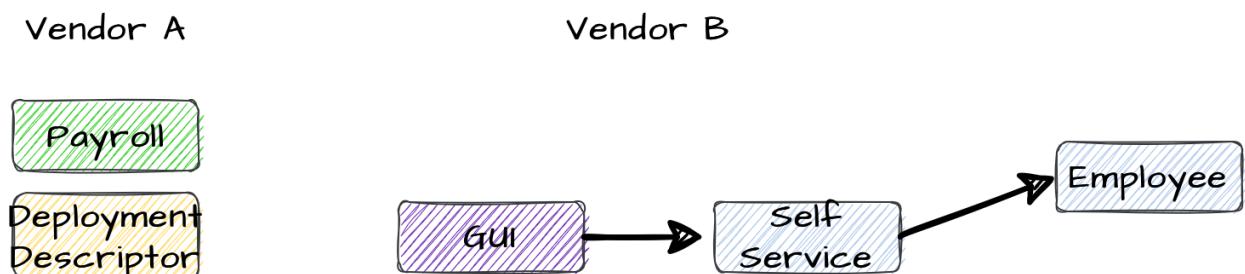
- **Session Bean**
  - Rappresenta un processo di business.
  - Ha, tipicamente, un'istanza per ogni cliente.
  - Tipicamente hanno un tempo di vita breve, legato alla vita del cliente.

- Sono transient.
  - Non persistono i dati in memoria persistente, quindi, non sopravvivono a crash del server.
  - Possono avere proprietà transazionali.
- **Entity Bean**
    - Rappresenta dati di business.
    - Le istanze sono condivise tra molteplici clienti.
    - Ha un tempo di vita lungo perché accoppiato al tempo di vita del dato che il bean rappresenta e che si trova all'interno del DB.
    - È persistente.
    - Sopravvive a crash del server.
    - È sempre transazionale.

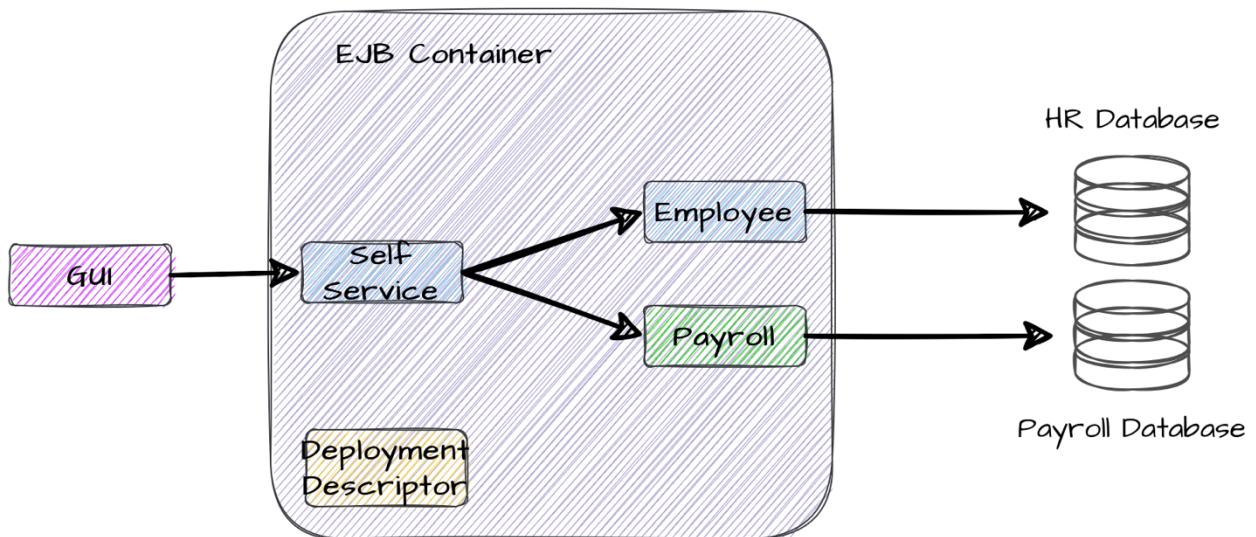
## 2.4 Scenari ed esempi pratici

J2EE è un'architettura pensata per velocizzare e industrializzare lo sviluppo di applicazioni. Gli standard e l'architettura di riferimento hanno il grosso vantaggio, oltre a semplificare il lavoro dello sviluppatore, di abilitare sempre di più la standardizzazione della produzione del software aprendo le porte alla possibilità di assemblare applicazioni utilizzando e facendo coesistere moduli sviluppati da "vendor" diversi, per esempio:

Si supponga di avere un produttore di software A (vendor A) specializzato nella modellazione e creazione di componenti busta paga che quindi sviluppa il componente "EJB Payroll", un secondo produttore di software B sviluppa altri componenti "Self Service" ed "Employee".



Il vendor B può utilizzare il modulo sviluppato dal vendor A per assemblare un'applicazione in cui il modulo "payroll" sviluppato da A coesista con i moduli sviluppati dal vendor B.



#### 2.4.1 Esempi pratici

Lo sviluppatore di EJB crea i moduli EJB (file EJB-JAR), ogni modulo contiene:

- **Classi di interfaccia**
  - Interfaccia EJBHome
  - Interfaccia EJBObject
- **Classi per il componente**
- **Deployment descriptor**
- Opzionalmente possono essere presenti delle classi d'appoggio dette **Classi Helper**

#### Interfaccia EJBHome

L'interfaccia EJBHome dichiara i metodi per la creazione, il ritrovamento e la distruzione dei bean, svolge, quindi, il ruolo di interfaccia factory. Il container implementa in maniera automatica l'interfaccia creando l'oggetto EJBHome che intercetta le chiamate del client per conto del container. Dall'altro lato il cliente ottiene un riferimento all'oggetto stub dell'oggetto EJBHome tramite JNDI. Come già detto l'interazione può essere remota e/o locale.

Un esempio di implementazione dell'interfaccia è il seguente:

```
// EJBHome
package com.ejb_book.interest;

import javax.ejb.*;
import java.rmi.*;

public interface InterestHome extends EJBHome{
    public Interest create() throws CreateException, RemoteException;
}
```

è un'interfaccia Java a tutti gli effetti che estende **EJBHome** e definisce il metodo **create()** che restituisce il componente d'interesse. Come si evince dal codice questa è l'interfaccia remota, infatti il metodo di creazione può potenzialmente sollevare delle **RemoteException**.

Per quanto riguarda l'interfaccia EJBHome locale essa è simile alla precedente ad eccezione del fatto che estende la classe **EJBLocalHome** e non solleva nessuna **RemoteException**:

```
// EJBHome
package com.ejb_book.interest;

import javax.ejb.*;
import java.rmi.*;

public interface InterestLocalHome extends EJBLocalHome {

    public InterestLocal create() throws CreateException;
}
```

### Interfaccia EJBObject

L'interfaccia EJBObject racchiude tutti i metodi della logica applicativa, il container implementa in maniera automatica l'interfaccia creando l'oggetto EJBObject.

Il client ottiene il riferimento all'oggetto stub di EJBObject come risultato dei metodi **create()** o **find()** invocati sullo stub dell'oggetto EJBHome.

Il cliente, quindi, non ottiene direttamente il componente ma uno stub dell'oggetto EJBObject che contiene la logica applicativa del componente, questo perché tale oggetto, come detto in precedenza funziona da proxy intercettando le richieste del client per conto del container.

Anche in questo caso la realizzazione di questa interfaccia può essere remota o locale.

Un esempio di implementazione dell'interfaccia è il seguente:

```
package com.ejb_book.interest;
import javax.ejb.*; import java.rmi.*;

// Interfaccia remota del componente Interest
public interface Interest extends EJBObject{

    public double getInterestOnPrincipal(double principal, double interestPerTerm, int terms)
                                         throws RemoteException;
}
```

L'interfaccia Object estende **EJBObject** e definisce tutti i metodi di logica applicativa. L'immagine riporta la definizione di un'interfaccia remota e quindi ogni metodo può sollevare delle **RemoteException**.

Per quanto riguarda la versione locale dell'interfaccia sarà identica a quella remota ad eccezione del fatto che i metodi non sollevano eccezioni di tipo Remote.

```
// EJBObject
package com.ejb_book.interest;

import javax.ejb.*;
import java.rmi.*;

public interface InterestLocal extends EJBLocalObject {

    // Calcola l'interesse da pagarsi ad un dato proprietario, ad uno specifico tasso di interesse
    public double getInterestOnPrincipal (double principal, double interestPerTerm, int terms);

}
```

## Client

Il cliente tramite il servizio di nomi di JNDI ottiene il contesto iniziale, effettua quindi delle operazioni di **Lookup** e **Narrowing** e ottiene un oggetto stub per l'oggetto EJBHome. Dall'oggetto EJBHome il cliente ottiene l'accesso all'oggetto EJB desiderato tramite il metodo *create/find* della Home interface che restituisce un oggetto remoto tipato. Adesso il cliente possiede uno stub tipato dell'oggetto remoto e tramite questo può invocare i metodi di business logic del componente, terminato l'uso del componente il cliente esegue le operazioni di **clean up** eliminando tutti gli stub e liberando le risorse.

Un esempio di implementazione di client è il seguente:

```
public class InterestClient {

    public static void main (String[] args) throws CreateException, RemoteException, NamingException {

        // passo 1: ottenere un'istanza di EJBHome (in realtà un oggetto
        // stub per l'oggetto EJBHome) via JNDI
        InitialContext initialContext = new InitialContext();
        Object o = initialContext.lookup("Interest");
        InterestHome interestHome = (InterestHome) PortableRemoteObject.narrow (o, InterestHome.cl

        // passo 2: creare un oggetto EJBObject remoto (in realtà
        // uno stub all'oggetto EJBObject remoto
        Interest interest = interestHome.create();

        double principal = 10000.0;
        double rate = 10.0;
        int terms = 10;

        System.out.println("Principal = $" + principal);
        System.out.println ("Rate(%) = " + rate);
        System.out.println ("Terms = " + terms);

        // passo 3: invocazione metodi di business
        System.out.println("Interest = $" + interest.getInterestOnPrincipal(principal, rate, terms));

        System.out.println("Total = $" + interest.getTotalRepayment(principal, rate, terms));

        // passo 4: clean up
        interest.remove();
    }
}
```

All'interno del metodo `main` tramite il metodo `getInterest()` invoca il servizio di nomi e crea lo stub per l'oggetto remoto. Il cliente ha creato un “legame” con l'oggetto remoto, possiede la Object interface del componente e può quindi invocare i metodi di business, infine, esegue il *clean up* tramite il metodo `remove()`.

Per quanto riguarda il metodo `getInterest()`, che si occupa di invocare il servizio di nomi e creare lo stub per l'oggetto remoto, esso è implementato come segue:

```
public Interest getInterest()
    throws CreateException,
           RemoteException,
           NamingException {

    InitialContext initialContext = new InitialContext();

    Object o = initialContext.lookup ("Interest");

    InterestHome home = (InterestHome) PortableRemoteObject.narrow (o, InterestHome.class);
    return home.create();
}
```

Il metodo `getInterest()` nasconde le parti di creazione dell'interfaccia Object, in modo da rendere plain Java il main visto precedentemente. Il metodo inizialmente recupera il contesto iniziale e su questo invoca il metodo `lookup()` passando l'identificatore unico “Interest” per trovare il riferimento alla Home Interface. Il metodo di `lookup()` restituisce un oggetto generico, a questo punto si esegue il metodo `narrow()` usando come attributi l'oggetto non tipato restituito dal metodo di `lookup()` e la classe con cui si vuole eseguire il “cast”, dopo questa operazione si ottiene lo stub all'interfaccia Home e su questo è possibile invocare il metodo `create()` per ottenere lo stub all'interfaccia Object.

N.B. Il narrowing, a differenza del casting che permette di tipare un dato purché ci sia omogeneità a livello di linguaggio, è un'operazione fondamentale nei sistemi distribuiti nei quali non è detto che ci sia omogeneità nei linguaggi dei vari componenti e né tantomeno che tutti i sistemi usino linguaggi OO, per esempio un client scritto in C vuole utilizzare le funzionalità di un componente scritto in Java.

## Descrittore di Deployment

Istruisce il container su come gestire il componente EJB circa gli aspetti di transazionalità, sicurezza, gestione dello stato, persistenza, ciclo di vita etc...

Il descrittore di deployment definisce un contratto tra produttore e consumatore del file ejb-jar, è un documento XML che supporta personalizzazione di tipo dichiarativo.

Un esempio di deployment descriptor è il seguente:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 2.0//EN' 'http://java.sun.com/dtd/ejbjar_2_0.dtd'>

<ejb-jar>
<display-name>Interest_ejb</display-name>
<enterprise-beans>
<session>
<display-name>InterestBean</display-name>
<ejb-name>InterestBean</ejb-name>
<home>com.ejb_book.interest.InterestHome</home>
<remote>com.ejb_book.interest.Interest</remote>
<local-home>com.ejb_book.interest.InterestLocalHome </local-home>
<local>com.ejb_book.interest.InterestLocal</local>
<ejb-class>com.ejb_book.interest.InterestBean</ejb-class>
<session-type>Stateless</session-type>
<transaction-type>Bean</transaction-type>
<security-identity>
<description></description>
<use-caller-identity></use-caller-identity>
</security-identity>
</session>
</enterprise-beans>

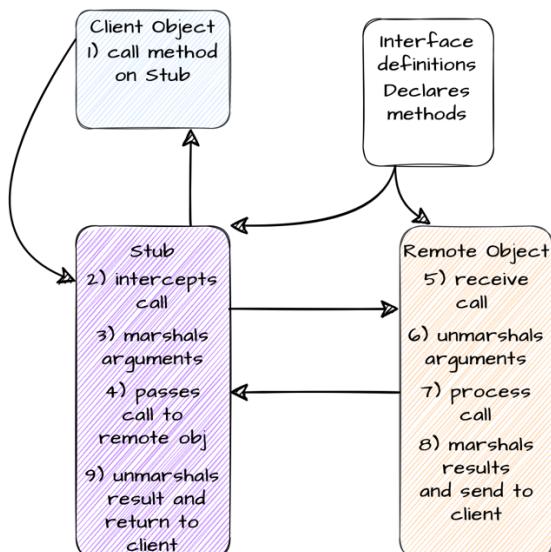
```

## 2.5 RMI e interazioni

In un'architettura distribuita gli oggetti che cooperano, spesso, risiedono su JVM diverse. Dal lato client vengono invocati i metodi di oggetti che si trovano lato server, necessariamente ci deve essere un meccanismo di comunicazione tra cliente e server.

Il **Remote Method Invocation (RMI)** è la tecnologia che consente la comunicazione fra cliente e server EJB. Le operazioni RMI sono costose perché bisogna effettuare la serializzazione/deserializzazione dei parametri, aprire, trasferire e chiudere una connessione RMI.

Il modello di interazione è il seguente:



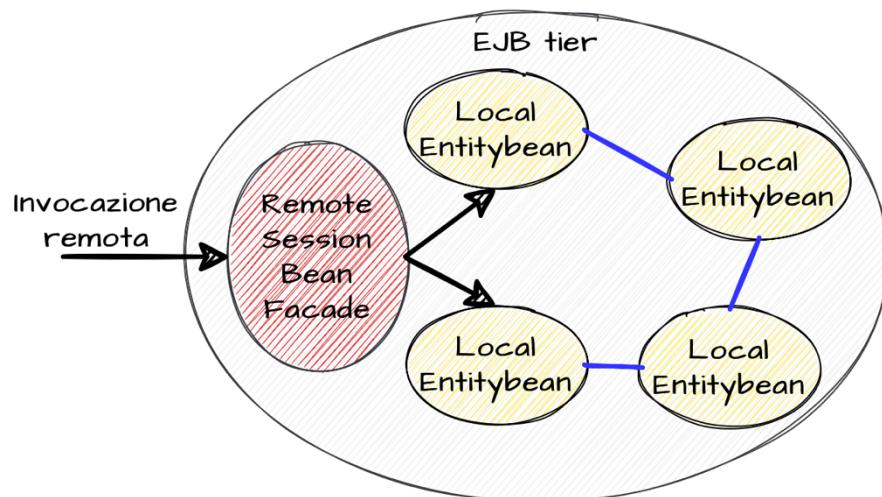
- **Client**
  1. Invoca un metodo sull'oggetto remoto
- **Stub**
  2. Intercetta l'invocazione del metodo
  3. Effettua il marshalling dei parametri
  4. Effettua la chiamata vera e propria all'oggetto remoto
- **Oggetto remoto**
  5. Riceve l'invocazione tramite il suo skeleton
  6. Effettua l'unmarshalling dei parametri
  7. Esegue l'invocazione localmente
  8. Effettua il marshalling dei risultati e li invia al client
- **Stub**
  9. Riceve i risultati, effettua unmarshalling e li restituisce al client

L'implementazione di RMI usata per la comunicazione è **RMI-IIOP**, IIOP (**Internet Inter-Orb Protocol**) è un protocollo di comunicazione del mondo CORBA, per questo la comunicazione ha un overhead ancora maggiore dovuto a CORBA.

Prima di EJB 2.0, RMI-IIOP doveva essere utilizzato anche se client e server risiedevano sulla stessa JVM, questo implicava il pagamento di un overhead non necessario. Per questo motivo a partire da EJB 2.0 è stata introdotta la “local interface”, le interfacce locali possono essere utilizzate quando il cliente esegue sulla stessa JVM del componente EJB d'interesse. L'utilizzo dell'interfaccia locale non comporta nessun overhead legato alla comunicazione RMI su IIOP, inoltre i metodi non sollevano RemoteException ed è possibile effettuare chiamate per riferimento. Tuttavia, l'utilizzo di local interface comporta un accoppiamento più stretto tra client e bean e una minore flessibilità di distribuzione e deployment.

Al fine di prendere il meglio dell'interfacciamento locale spesso si fa uso del **pattern facade** in cui un session bean (per operazioni sincrone) o un Message Driven Bean (per operazioni asincrone) con interfaccia remota invocano entity bean a loro locali. Questo permette di ridurre l'accoppiamento tra client e bean senza pagare un overhead eccessivo.

### Remote Session Bean Facade



## 2.6 Deployment di un'applicazione

Per effettuare il deployment di un'applicazione EJB sono necessari i seguenti file:

- **\*.EAR** (Enterprise ARchive): contiene l'intera applicazione EJB che si trova lato server. Per effettuare il deployment di una applicazione EJB, è sempre necessario creare un file \*.EAR anche se l'applicazione prevede un solo file EJB-JAR e nessun modulo Web. Al suo interno sono contenuti i file:
  - **\*.WAR** (Web ARchive): modulo Web (Servlet, JSP etc... ). È facoltativo.
  - **EJB-JAR (\*.JAR)**: modulo EJB al cui interno è possibile inserire uno o più componenti. In un file “.EAR” possono esserci uno o più moduli.
  - **application.xml**: descrittore di deployment dell'applicazione.
- **EJB-JAR**: è il formato standard per il packaging dei componenti EJB, utilizzato per raggruppare in un package componenti EJB sia assemblati che separati deve contenere necessariamente:
  - **Classe Bean**: la classe creata dallo sviluppatore.
  - **EJBHome**: l'interfaccia HOME che a run time sarà concretamente implementata dal container.
  - **EJBObject**: l'interfaccia OBJECT che a run time sarà concretamente implementata dal container.
  - **Primary key**: una classe che funge da chiave primaria nel caso di un entity bean.
  - **Deployment descriptor**: il descrittore di deployment in questo caso ha visibilità locale, limitata al modulo per cui è definito.
- **\*.JAR**: il file jar per il cliente EJB, questo file consiste di tutte le classi necessarie per il programma cliente. Chi si occupa del deployment dell'applicazione deve assicurare che questo file jar sia accessibile al class loader dell'applicazione cliente.

## 3. Java 5 & Annotation

Le annotazioni Java sono strumenti che permettono di aggiungere metadati all'interno del codice Java. Le annotation sono associabili a tutti gli elementi di un programma (package, classi e interfacce, costruttori, metodi, campi, parametri e variabili). Arricchiscono lo spazio concettuale del linguaggio aggiungendo espressività agli elementi.

Le annotazioni permettono di specificare informazioni relative a determinate entità senza dover ricorrere a descrittori esterni.

Le annotazioni vengono lette e gestite dal compilatore o da strumenti esterni. Non influenzano direttamente la semantica del codice, ma il modo in cui il codice può essere trattato da strumenti come VM e librerie e di conseguenza possono influenzare il comportamento runtime.

### 3.1 Sintassi

Le annotazioni sono strutturate nel seguente modo: possono essere formate da solo il nome dell'annotazione oppure possono avere dei membri (con membro si intende il parametro di ingresso dell'annotazione.) che vengono specificati come un insieme di coppie nome=valore. Se il membro è solo uno, il nome si può omettere.

### 3.2 Categorie di Annotation

- **Marker annotation:** non hanno membri, l'informazione è data dal nome stesso dell'annotazione. Esempio d'uso: `@MarkerAnnotationName`.
- **Single-value annotation:** hanno un solo membro. Esempio d'uso: `@SingleValueAnnotationName ("value")`.
- **Full annotation:** hanno più di un membro e si utilizzano come nel caso precedente.
- **Custom annotation:** sono annotazioni definite dal programmatore.

### 3.3 Custom Annotation

È possibile creare delle annotazioni personalizzate, di seguito viene riportato un esempio di annotazione personalizzata:

```
public @interface GroupTODO {  
    public enum Severity {CRITICAL, IMPORTANT, TRIVIAL} ;  
    Severity severity() default Severity.IMPORTANT;  
    String item();  
    String assignedTo();  
}
```

Si noti che l'annotazione è dichiarata di tipo `@interface`, ogni annotazione estende l'interfaccia `java.lang.annotation.Annotation`. Come si può notare gli elementi dell'annotazione vengono dichiarati come metodi. L'annotazione sopra definita può essere utilizzata come segue:

```
@GroupTODO (   
    severity = GroupTODO.Severity.CRITICAL;  
    item = "Figure out the amount of interest per month"  
    assignedTo = "Luca Foschini";  
)  
public void calculateInterest(float amount, float rate) { ... }
```

Le annotazioni hanno delle limitazioni:

- Non si possono avere delle relazioni di estensione fra tipi di annotation.
- I tipi di ritorno dei metodi di un'annotation devono essere primitivi.
- Una annotation non può lanciare eccezioni.
- Non sono permessi **self-reference** (AnnotationA non può contenere un membro di tipo AnnotationA) né **circular-reference** (AnnotationA non può contenere un membro di tipo AnnotationB se questo contiene un membro di tipo AnnotationA).

### 3.3.1 Meta-Annotation

Le meta-annotazioni sono annotazioni che possono essere applicate ai tipi di annotazione. Una speciale meta-annotazione predefinita definisce come possono essere usati i tipi di annotazione. Alcuni esempi di meta-annotation sono:

- **@Target**: specifica il tipo di elemento al quale si può allegare l'annotazione sui cui viene applicata.

```
@Target ( { ElementType.METHOD, ElementType.PACKAGE } )  
public @interface ExampleAnnotation { ... }
```

- **@Documented**: specifica che le annotation di tale tipo faranno parte della documentazione Javadoc generata.

```
@Documented  
public @interface ExampleAnnotation { ... }
```

- **@Inherited**: solo per annotazioni apposte a classi. Il tipo di annotazione verrà automaticamente ereditato dalle sottoclassi della classe alla quale viene allegata.
- **@Retention**: specifica la politica di mantenimento in memoria con cui compilatore e la JVM devono gestire le annotation:

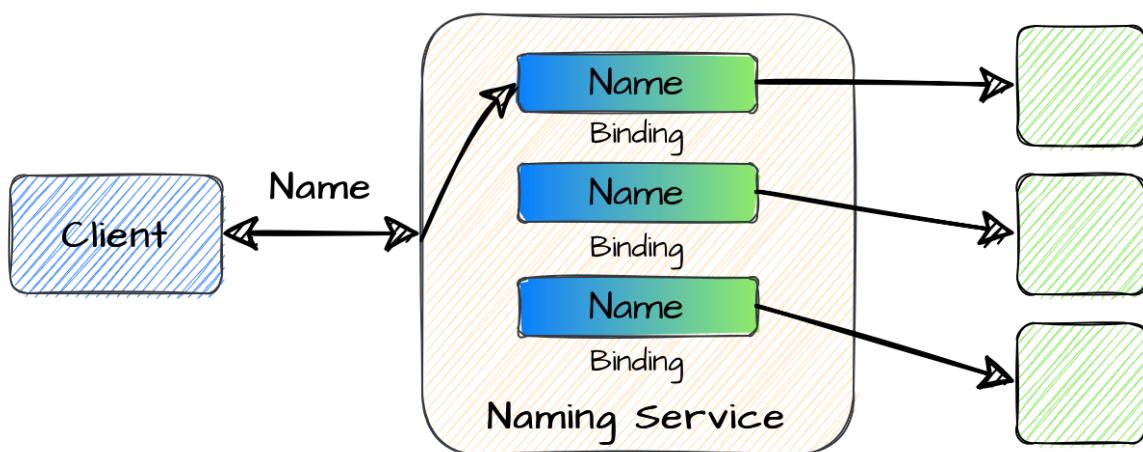
- **@Retention(RetentionPolicy.SOURCE)**: L'annotazione permane solo a livello di codice sorgente, non è memorizzata nel bytecode (.class file) e quindi viene ignorata dalla JVM. È utilizzata principalmente a tempo di sviluppo e compile-time.
- **@Retention(RetentionPolicy.CLASS)**: comportamento di default, l'annotazione verrà registrata nel bytecode dal compilatore, ma non verrà mantenuta dalla JVM a runtime. È usata tipicamente a tempo di caricamento.
- **@Retention(RetentionPolicy.RUNTIME)**: l'annotazione verrà registrata nel bytecode e potrà essere letta a runtime (mediante **reflection**) anche dopo il caricamento della classe da parte della JVM; utilizzabile anche all'interno del codice di supporto/applicativo a tempo di esecuzione, con proprietà eventualmente modificabili a runtime.

## 4. Java Naming and Directory Interface (JNDI)

### Servizi di naming

Un servizio di naming è un sistema che consente di associare ad un nome logico una risorsa (nome fisico, riferimento, oggetto). Esempi di sistemi di nomi sono:

- **DNS (Domain Name System)**: ad un nome logico associa l'indirizzo fisico del server.
- **RMI Registry**: ad un nome logico associa lo stub.
- **Portmapper (RPC)**: fornendo il numero di programma viene restituito versione, protocollo e porta.

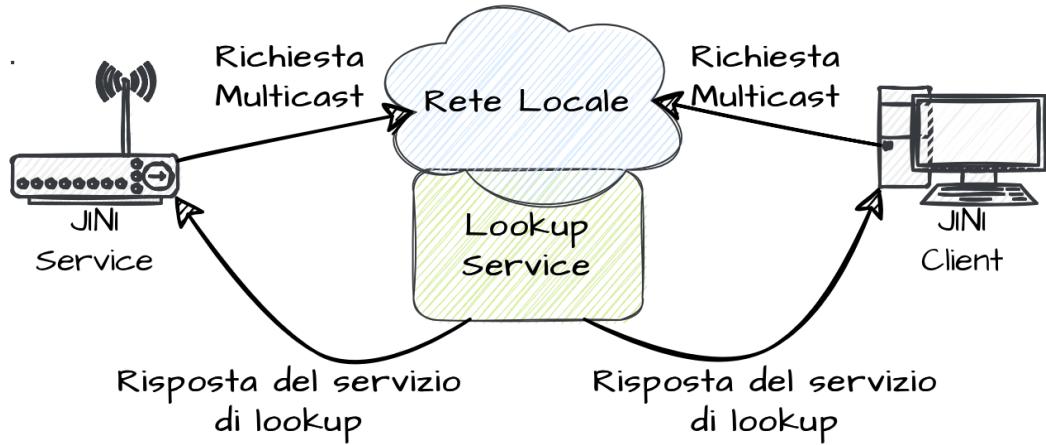


JNDI consente di interagire con diversi servizi di nomi, da servizi di nomi come DNS che offrono un semplice binding tra nome logico e nome fisico a servizi più avanzati come LDAP in cui i nomi sono coppie di elementi nome/valore separati da virgole (nome1 = valore1, nome2 = valore2... ). Per astrarre le diversità tra i formati dei nomi dei vari servizi di naming JNDI introduce una classe “**Name**” che può essere immaginata come un contenitore di nomi logici.

### Servizi di discovery

I servizi di discovery sono una famiglia di sistemi di nomi. Questo sistema viene usato, per esempio, quando un cliente invia una richiesta in broadcast in modo da trovare i dispositivi che sono presenti nella rete. Questo servizio gestisce una piccola quantità di nomi e il numero di scritture nella tabella è molto alto proprio perché la ricerca avviene in broadcast.

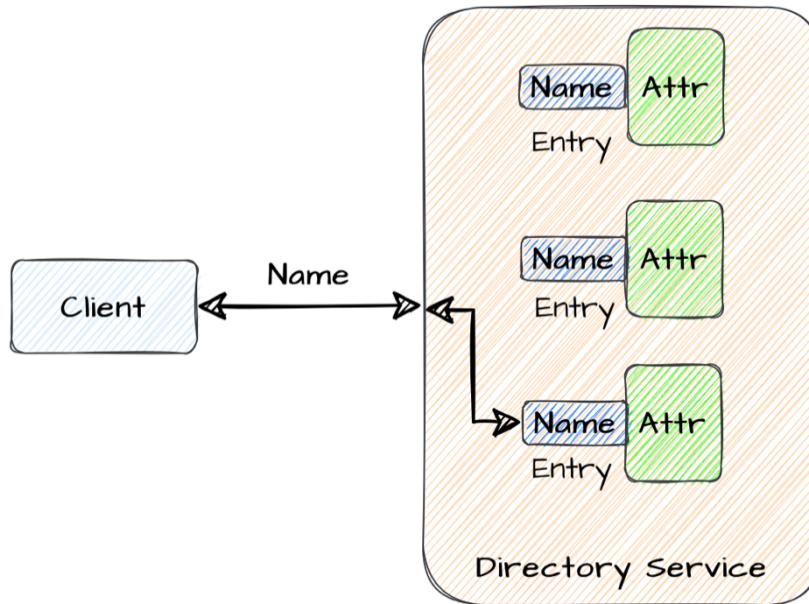
**JINI** è un protocollo Java per discovery che consente di associare ad un nome logico un nome fisico, che nel caso di JINI può essere, codice che verrà eseguito direttamente sul client JINI o un riferimento al server che verrà interrogato in remoto.



### Servizi di directory

I Servizi di directory sono una famiglia di sistemi di nomi in cui oltre al nome logico vengono memorizzate una serie di attributi (simili ai record di un DB). Gli attributi devono essere accessibili efficientemente in lettura e scalare molto bene su numeri grandi. A differenza dei servizi di discovery quelli di directory sono servizi di tipo più enterprise, quindi molto più assistiti, che garantiscono alta scalabilità e fault tolerance. Per esempio si potrebbe pensare ad un server di directory che mantiene tutti gli account di dipendenti e studenti di una università.

I servizi di directory servono alla gestione e alla distribuzione di informazioni condivise, per esempio, all'interno di un'azienda.



Ad ogni nome logico corrispondono una serie di attributi, che sono le informazioni condivise. Il servizio di directory può essere interrogato, non più con un nome unico, ma assegnando un valore agli attributi per recuperare le entry di interesse.

Il servizio directory può essere visto come un gestore di un insieme di entry conformi ad un certo schema.

Sebbene i servizi di directory presentano caratteristiche molto simili a quelle dei DB essi sono abbastanza differenti:

- A differenza dei database i servizi di directory hanno schemi prefissati.
- Nelle Directory, le operazioni sono molto più ottimizzate rispetto al DB perchè sono pensati come strumenti da usare in ambienti distribuiti.

## 4.1 JNDI architettura e concetti fondamentali

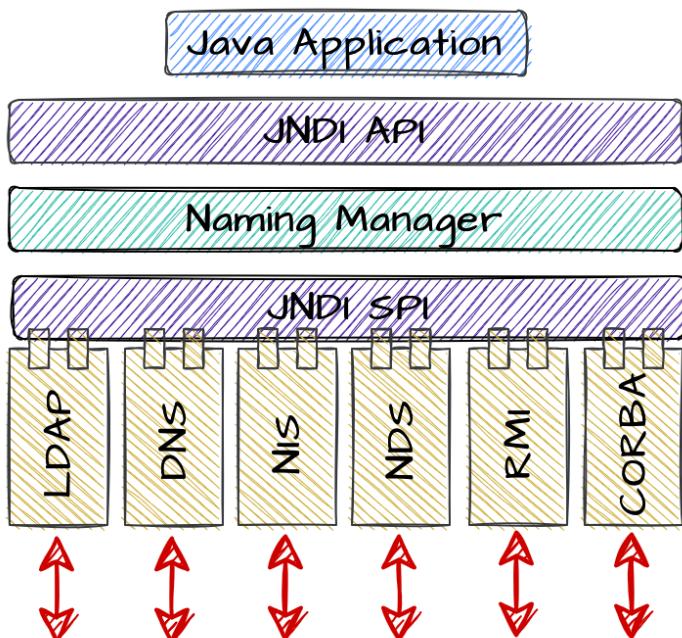
JNDI è un'interfaccia standard che consente di accedere in modo uniforme a servizi di naming già esistenti (Registry RMI, DNS, LDAP...). JNDI gioca un ruolo centrale in molte delle tecnologie Java-related come ad esempio JDBC, JMS, EJB.

JNDI consente a **JDBC** di memorizzare gli oggetti **DataSource** che specificano tutte le informazioni necessarie all'interazione con uno specifico database.

In **JMS** JNDI viene utilizzato per ritrovare le queue/topic con i quali un certo cliente JMS vuole interagire.

In **EJB 2.x** l'interfaccia EJBHome, che rappresenta l'entry point, viene pubblicata tramite JNDI. In **EJB 3.0** JNDI viene utilizzato in maniera più trasparente e pervasiva.

L'architettura di JNDI è la seguente:



Le **Java Application** interagiscono con il server JNDI attraverso delle API ben definite, le **JNDI API**. Al disotto di queste API, il **Naming Manager** gestisce le diverse implementazioni dei servizi di nomi supportati. Il **JNDI SPI** è un livello di astrazione che permette ai servizi di nomi di “agganciarsi” ed essere utilizzabili dall’applicazione.

L’obiettivo di JNDI è quello di standardizzare l’interazione con i servizi di nomi, utilizzando le JNDI API è possibile, infatti, cambiare servizio di nomi senza modificare il codice della Java Application.

I concetti fondamentali di JNDI sono:

- **Name:** il nome dell'oggetto registrato presso il servizio di nomi.
- **Binding:** associazione nome -> oggetto.
- **Reference:** il puntatore all'oggetto.
- **Context:** è un contenitore di coppie nome-oggetto.
- **Naming system:** è un insieme di context dello stesso tipo.
- **Naming space:** è un insieme di nomi all'interno di un naming system.

## 4.2 JNDI Provider

Per accedere ai servizi di nomi tramite JNDI si fa uso di plugin chiamati provider (blocchetti gialli della figura precedente), essi sono un'interfaccia di connessione verso uno specifico servizio di nomi esterno. Il provider JNDI offre supporto alla persistenza e alla distribuzione su rete dei binding nome-oggetto. Il provider JNDI dovrà essere opportunamente configuato attraverso le proprietà:

- ***java.naming.factory.initial***
- ***java.naming.provider.url***

Ad esempio, nel caso di registry RMI occorre configurare un JNDI provider per registry RMI:

- **Proprietà Initial:** com.sun.jndi.rmi.registry.RegistryContextFactory
- **Proprietà Url:** rmi://lia.deis.unibo.it:5599

Come detto in precedenza, il vantaggio di utilizzare JNDI, anziché direttamente il servizio di nomi, è quello di poter cambiare il naming service utilizzato modificando solo la configurazione del JNDI provider senza dover modificare il codice dell'applicazione.

Infatti per recuperare un oggetto remoto usando direttamente registry RMI l'applicazione dovrebbe contenere il seguente pezzo di codice:

```
import java.rmi.Naming;
...
IntRemota obj = (IntRemota)Naming.lookup("//lia.deis.unibo.it:5599/serverRemoto");
```

Questa porzione di codice rende la Java Application fortemente dipendente dal registry RMI, se si volesse cambiare servizio di nomi bisognere modificare il codice dell'applicazione.

Utilizzando JNDI invece, è possibile configurare un JNDI Provider settando le proprietà citate sopra, in questo modo si può ottenere il contesto e utilizzarlo per effettuare l'operazione di lookup offerta dalle JNDI API.

```
import javax.naming.*;
...
Properties prop = new Properties();
prop.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.rmi.registry.RegistryContextFactory");
prop.put(Context.PROVIDER_URL, "rmi://lia.deis.unibo.it:5599");

Context ctxt = new InitialContext();
IntRemota ctxt.lookup("lia.deis.unibo.it:5599/ServerRemoto");
```

Il vantaggio di questa diversa implementazione consiste nel fatto che se si intende utilizzare un servizio di nomi differente, basta modificare le porzioni di codice inerenti al settaggio delle “Properties”.

## 4.3 JNDI Context

**Context** è l'interfaccia che contiene metodi per aggiungere, cancellare, cercare, ridenominare oggetti di un sistema di nomi. Invece, **InitialContext** è l'implementazione di Context. I metodi che si trovano nell'interfaccia sono i seguenti:

- **Bind**

Consente di associare ad un nome logico un oggetto. È importante che il nome non sia già associato ad alcun oggetto.

```
void bind(String stringName, Object object)
```

- **Rebind**

Consente di riassegnare al nome logico, dato come primo parametro, un nuovo oggetto.

```
void rebind(String stringName, Object object)
```

- **Lookup**

Consente di cercare l'oggetto che corrisponde al nome logico dato come parametro di ingresso.

```
Object lookup(String stringName)
```

- **Unbind**

Dato un nome in ingresso ricerca la coppia nome-oggetto e la rimuove dal Context.

```
void unbind(String stringName)
```

- **Rename**

Consente di cambiare nome logico ad una coppia nome-oggetto già esistente.

```
void rename(String stringOldName, String stringNewName)
```

- **ListBindings**

Restituisce tutte le entry che si trovano nel sistema di nomi che fanno matching con il nome fornito in ingresso. In RMI, non è possibile avere due entry con lo stesso nome logico ma ci potrebbero essere dei sistemi di nomi che lo consentono.

```
NamingEnumeration listBindings(String stringName)
```

## 4.4 JNDI DirContext

L'interfaccia Context opera con naming services abbastanza semplici, che gestiscono binding di tipo nome-oggetto. Per quanto riguarda i servizi di nomi di tipo Directory, non è possibile usare l'interfaccia Context in quanto si ha la necessità di gestire dei binding più complessi.

**DirContext** è una sottoclasse di Context che ne estende le funzionalità standard aggiungendo i metodi che consentono di svolgere operazioni con attributi e ricerche su entry di directory. I metodi che si trovano nell'interfaccia sono i seguenti:

- **Bind**

Associa un nome a un oggetto e memorizza gli attributi specificati come parametro nella entry corrispondente. È importante che il nome non sia associato già ad alcun oggetto.

```
void bind(String stringName, Object object, Attributes attributes)
```

- **Rebind**

Consente di riassegnare ad un nome logico un nuovo oggetto.

```
void rebind(String stringName, Object object, Attributes attributes)
```

- **CreateSubcontext**

Crea un sottocontesto, eventualmente con attributi (come una cartella del file system al cui interno si possono creare altre cartelle).

```
DirContext createSubcontext(String stringName, Attributes attributes)
```

- **GetAttributes**

A partire da un nome logico ricerca la entry corrispondente e restituisce tutti i suoi attributi.

```
Attributes getAttributes(String stringName)
```

- **GetAttributes**

A partire da un nome logico ricerca la entry corrispondente e restituisce solo gli attributi specificati come secondo parametro.

```
Attributes getAttributes(String stringName, String [] rgstringAttributeNames)
```

- **ModifyAttributes**

A partire da un nome logico ricerca la entry corrispondente e permette di modificarne gli attributi. Le operazioni consentite sono: ADD\_ATTRIBUTE, REPLACE\_ATTRIBUTE e REMOVE\_ATTRIBUTE, l'operazione scelta può essere effettuata su più attributi.

```
void modifyAttributes(String stringName, int nOperation, Attributes attributes)
```

- **ModifyAttributes**

A partire da un nome logico ricerca la entry corrispondente e permette di modificarne gli attributi. Le operazioni consentite sono: ADD\_ATTRIBUTE, REPLACE\_ATTRIBUTE e REMOVE\_ATTRIBUTE. Possono essere effettuate operazioni diverse su uno o più attributi.

```
void modifyAttributes(String stringName, ModificationItem [] rgmodificationitem)
```

L'interfaccia DirContext offre anche delle funzionalità più avanzate come quelle di ricerca che permettono di eseguire delle query e recuperare tutte le entry che contengono un dato insieme di attributi. Dal momento che il numero di entry restituite da una singola query potrebbe essere molto grande (es. un sistema LDAP mantiene i profili di ogni persona in una grande città e una query richiede di ricercare tutti gli abitanti di sesso maschile) l'interfaccia permette di limitare le operazioni di ricerca in spazio (numero massimo di entry restituite) o in tempo (tempo massimo per la ricerca).

## 4.5 Uso di JNDI

Per prima cosa occorre scegliere un naming service provider, ad esempio, OpenLDAP o un'altra implementazione di LDAP. Dopo, bisogna aggiungere il nome del provider all'insieme di proprietà di ambiente tramite un oggetto **Hashtable**:

```
Hashtable hashtableEnvironment = new Hashtable();
hashtableEnvironment.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory")
```

A questo punto bisogna aggiungere ogni informazioni addizionale necessaria al naming provider. Ad esempio, per LDAP, l'URL che identifica il servizio, il context radice, il nome e la password per la connessione:

```
HashtableEnvironment.put(Context.PROVIDER_URL, "ldap://localhost:389/dc=etcee,dc=com");
HashtableEnvironment.put(Context.SECURITY_PRINCIPAL, "name");
HashtableEnvironment.put(Context.SECURITY_CREDENTIALS, "password");
```

A questo punto occorre creare l'oggetto **InitialContext**, o nel caso in cui il servizio di nomi sia di tipo directory, l'oggetto **InitialDirContext**

```
Context context = new InitialContext(HashtableEnvironment);
DirContext context = new InitialDirContext(HashtableEnvironment);
```

Adesso si può utilizzare il servizio di nomi per memorizzare informazioni come, ad esempio, un nome logico e un oggetto associato.

```
context.bind("name", object)
```

La semantica presenta delle analogie con RMI, ma è definita in maniera più flessibile. Ovviamente è possibile effettuare anche l'operazione duale che permette il ritrovamento di un oggetto a partire dal nome logico, l'oggetto restituito potrebbe essere il riferimento ad un oggetto remoto o un valore:

```
Object obj = context.lookup("name")
```

JNDI affida al JNDI service provider la definizione della semantica dell'operazione di memorizzazione di un oggetto. È possibile salvare:

- **Dati serializzati:** memorizza il valore del dato .
- **Riferimento:** memorizza il riferimento all'oggetto remoto.
- **Attributi:** nel caso di un directory service memorizza gli attributi associati a una data entry.

### Dati serializzati

La semantica serialized data (serializzazione) si usa per salvare tutto il contenuto dell'oggetto, è necessario che la classe dell'oggetto implementi l'interfaccia **Serializable**.

Quando si effettua l'operazione di *lookup* si recupera il contenuto dell'oggetto per copia. Tuttavia, non sempre una risorsa può essere serializzabile. Ad esempio, database, etc.

### Riferimento

Non sempre una risorsa è serializzabile, quindi per renderla raggiungibile dall'esterno si fa uso della semantica per riferimento. In questo caso, viene memorizzato solo il riferimento ad un oggetto, che deve implementare l'interfaccia **Referenceable**. Quando il cliente esegue

la lookup viene restituito il riferimento alla risorsa. Spesso, questo è l'unico comportamento supportato dal sistema di nomi.

### Attributi

Se si ha a che fare con un servizio di directory è possibile memorizzare un oggetto come una collezione di attributi. L'oggetto deve includere codice per scrivere il suo stato interno come un oggetto **Attributes** e deve fornire una factory per la ricostruzione dell'oggetto a partire dagli attributi.

Non tutti i linguaggi di programmazione conoscono il concetto di oggetto. Per questo motivo, utilizzare la semantica per attributi consente di eliminare il mismatch (disaccoppiamento) tra linguaggi differenti.

## 4.6 Configurazione di JNDI

JNDI è una interfaccia generica e generale che permette di accedere a uno specifico naming/directory service, occorre, per fare ciò, specificare quale service provider utilizzare, quale serve, etc...

La configurazione avviene attraverso proprietà di environment:

- **Standard:** sono proprietà indipendenti dal service provider che accomunano tutti i servizi di nomi ad esempio, LDAP, RMI etc. Si trovano nel package **java.naming**. Ad esempio, **java.naming.provider.url** o **java.naming.factory.initial**.
- **Service specific:** sono proprietà comuni per tutti i naming service provider che implementano uno specifico servizio o protocollo standard. Hanno prefisso **java.naming.service**. ad esempio: **java.naming.ldap**.
- **Feature specific:** comuni per tutti naming service provider che implementano una specifica feature. Ad esempio, SASL per autenticazione. Hanno prefisso **java.naming.feature**. ad esempio, **java.naming.security.sasl**.
- **Provider specific:** specifiche per un determinato naming service provider. Ad esempio, il servizio Sun LDAP ha una proprietà per abilitare il tracing: **com.sun.jndi.ldap.trace.ber**.

Le proprietà di ambiente possono essere configurate con metodi differenti:

- **Parametro environment:** vengono passati al costruttore di InitialContext. Ad esempio, si crea un oggetto HashTable.
- **File Application Resource:** si crea il file **jndi.properties** che contiene una lista di coppie attributo/valore, tale file viene utilizzato da JNDI al momento della creazione del contesto per configurare il JNDI provider. Questo tipo di configurazione è molto trasparente a livello programmatico.
- **Proprietà di sistema:** Sono coppie attributo/valore che Java definisce o usa a runtime per descrivere utenti, ambienti di sistema etc... Le proprietà di sistema possono essere aggiunte manualmente da riga di comando.
- **Parametri di applet:** ormai in disuso.

Quindi, riassumendo, le proprietà di ambiente possono essere specificate tramite file application resource, il parametro environment, le proprietà di sistema, e parametri di applet.

Ovviamente, è possibile utilizzare contemporaneamente più di uno di questi meccanismi.

Nel caso in cui una proprietà sia specificata da più di un meccanismo, generalmente i valori delle proprietà sono concatenati in una lista separata da virgole. Se sono presenti più valori per una proprietà ma questa richiede un solo valore, viene preso solo primo della lista.

## 5. Enterprise Java Beans 3.x

EJB 2.1 è stato considerato dagli sviluppatori una tecnologia molto potente, ma anche complessa da utilizzare. I problemi principali che EJB 2.1 presentava erano:

- Modello di programmazione non sempre naturale.
- Necessità di definire i descrittori di deployment.
- Elevato numero di classi e interfacce.
- Lookup dei componenti basato sempre su JNDI.
- Difficoltà di uso corretto.

L'introduzione di EJB 3.0 propone di risolvere tutte queste problematiche, introducendo un modello più semplice sia in fase di apprendimento che di utilizzo. A tal fine EJB 3.0 promuove un modello che presenta le seguenti caratteristiche:

- Un minor numero di classi e interfacce.
- L'introduzione del meccanismo di **Dependency injection** per semplificare la gestione di dipendenze tra componenti.
- Lookup dei componenti semplificato.
- Interfacce per il container e descrittori di deployment non necessari (grazie all'uso di annotation).
- Gestione della persistenza semplificata tramite l'introduzione delle **Java Persistence API** (JPA), un servizio di persistenza esterno alla specifica EJB, che si occupa delle operazioni di mapping tra oggetti e database relazionali, permettendo quindi di vedere gli entity bean come POJO (Plain Old Java Object, un oggetto Java puro).

Per quanto riguarda la retrocompatibilità, tutte le applicazioni EJB 2.x sono compatibili con il nuovo standard. Peraltro, EJB 3.0 facilita il passaggio dallo standard precedente a quello attuale.

### 5.1 Approccio EJB 3.0

Come già anticipato precedentemente, l'obiettivo della versione 3.0 è quello di alleggerire e semplificare l'utilizzo della versione 2.x, per questo motivo con la nuova versione non sono più necessarie le interfacce EJBHome ed EJBObject, non sono più necessari i descrittori di deployment ed inoltre, con l'introduzione del meccanismo di dependency injection, il cliente EJB non ha più la necessità di interfacciarsi, direttamente, con il servizio di nomi JNDI.

Al fine di rendere possibili tali cambiamenti EJB 3.0 fa largo uso delle Java annotation, e per semplificare ancora di più il lavoro dello sviluppatore, le annotazioni prevedono dei valori di default per i casi più usuali.

Questa semplificazione del lavoro svolto dallo sviluppatore impone che il container si faccia carico di più operazioni rendendo il tutto più trasparente a livello programmatico. Grazie all'uso delle annotazioni, il container è in grado di capire quali operazioni deve svolgere, come ad esempio, iniettare delle dipendenze o della logica specifica per realizzare i servizi di sistema, il tutto, come già detto, in modo dinamico e trasparente.

La versione EJB 3.0 elimina i requisiti sulle interfacce dei componenti, che quindi diventano delle **Plain Old Java Interface** (POJI). Non c'è più la necessità di definire l'interfaccia Home, basta infatti definire l'interfaccia con le operazioni di business.

L'uso estensivo delle annotazioni rende non necessari i descrittori di deployment, tuttavia, è ancora possibile utilizzarli, sia per motivi di retrocompatibilità con la versione 2.1, sia perché alcuni sviluppatori preferiscono usare questa metodologia. I descrittori di deployment hanno priorità maggiore rispetto alle annotazioni, permettendo quindi di effettuare l'override delle stesse. Inoltre, a partire da EJB 3.0 i descrittori possono essere anche parziali e incompleti, in tal caso i valori mancanti sono colmati dai valori di default delle annotazioni.

## 5.2 Tipologie di Bean in EJB 3.0

Come già detto in EJB 3.0 i session bean e i message-driven bean sono classi Java ordinarie, vengono quindi rimossi i requisiti di interfaccia e la tipologia del bean viene specificata tramite l'utilizzo delle annotazioni (**@Stateless**, **@Stateful**, **@MessageDriven**).

Gli entity bean non sono stati modificati e possono continuare ad essere utilizzati; tuttavia, per le entità che utilizzano la Java Persistence API (JPA) è possibile utilizzare l'annotazione **@Entity** per specificare che quella classe definisce un entity bean.

Per quanto riguarda le modalità di accesso ai bean, che possono essere locali o remoti, a default si assume che il componente sia di tipo locale, per definire un componente remoto si fa uso dell'annotazione **@Remote**, a differenza dello standard 2.1 i metodi remoti non sono più obbligati a lanciare una **RemoteException**. Quindi anche la gestione dell'accesso locale o remoto è effettuata tramite l'uso di annotazioni (**@Remote**, **@Local**, **@WebService**).

### 5.2.1 Session Bean

Con l'introduzione delle annotazioni si è reso possibile snellire il codice scritto dallo sviluppatore da tutte quelle parti legate alla definizione delle interfacce Home e Object, eliminare i requisiti sulle interfacce, ovvero l'implementazione dell'interfaccia **javax.ejb.SessionBean**, in particolare l'annotazione **@Stateless** viene utilizzata per definire un session bean senza stato e l'annotazione **@Stateful** per definire session bean con stato.

Il codice sottostante mostra la definizione di un session bean senza stato nello standard 2.1 e nello standard 3.0:

```
// EJB2.1 Stateless Session Bean: Bean Class
public class PayrollBean implements javax.ejb.SessionBean {

    SessionContext ctxt;

    public void setSessionContext(SessionContext ctxt) {
        this.ctxt = ctxt;
    }
    public void ejbCreate() {...}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}

    public void setTaxDeductions(int empId, int deductions) {
        ...
    }
}
```

```
@Stateless
public class PayrollBean implements Payroll {

    public void setTaxDeductions(int empId,int deductions) {
        ...
    }
}
```

Si noti come nella versione 3.0 dello standard la classe bean contiene solo il codice della logica applicativa (`setTaxDeductions`).

Inoltre, grazie all'uso delle annotazioni è possibile definire delle Plain Old Java Interface ed eliminare le interfacce Home e Object. L'esempio seguente mostra come il passaggio da uno standard all'altro semplifichi molto la scrittura delle interfacce:

```
// EJB 2.1 Stateless Session Bean: Interfacce

public interface PayrollHome extends javax.ejb.EJBLocalHome {
    public Payroll create() throws CreateException;
}

public interface Payroll extends javax.ejb.EJBLocalObject {
    public void setTaxDeductions(int empId, int deductions);
}
```

Lo standard 2.1 prevedeva la definizione delle due interfacce Home e Object che potevano essere locali o remote e nel caso di interfacce remote era necessario lanciare e gestire le `RemoteException`.

```
// EJB 3.0 Stateless Session Bean: Interfaccia di business
public interface Payroll {
    public void setTaxDeductions(int empId, int deductions);
}
```

In EJB 3.0 è possibile definire un'interfaccia POJI e con l'ausilio delle annotazioni specificare se si tratta di interfacce locali o remote, in quest'ultimo caso, peraltro, non sarà necessario né lanciare né gestire le `RemoteException` in quanto tale compito sarà demandato al container.

## 5.2.2 Message Driven Bean

La definizione di un message driven bean avviene tramite l'utilizzo dell'annotazione **@MessageDriven**, essi continuano ad implementare l'interfaccia del listener di messaggi **javax.jms.MessageListener**, per il resto basta implementare solo la logica applicativa senza dover implementare altre interfacce.

```
// EJB 3.0 Message-driven bean

@MessageDriven
public class PayrollMDB implements javax.jms.MessageListener {
    public void onMessage(Message msg) {
        ...
    }
}
```

## 5.3 Dependency Injection

Spesso accade che un bean ha necessità di usare risorse esterne, tali risorse sono “iniettate” quando l’istanza del bean viene creata. Le risorse di cui il bean ha bisogno sono, per esempio:

- **EJB Context**: le risorse messe a disposizione dal container.
- **Entity Manager**: per l’interazione con la parte di gestione degli entity bean.
- **Componenti EJB**: dipendenze da altri componenti EJB.
- **Data Source**: per le dipendenze da sorgenti dati esterne (database).

La dependency injection è realizzata principalmente tramite l’uso delle annotazioni:

- **@EJB**: per esprimere riferimenti ad interfacce di business EJB, oppure per riferimenti all’interfaccia EJBHome (per la compatibilità con i componenti EJB 2.1).
- **@Resource**: esprime qualsiasi altro tipo di riferimento: factory di connessioni, EJB Context e nel caso dei message-driven bean per indicare topic/queue di JMS.
- **@PersistenceContext, @PersistenceUnit**: utilizzare per gestire la persistenza (EntityManager)

L’uso delle annotazioni fa sì che il processo di injection possa avvenire a tempo di sviluppo, di compilazione o anche a runtime.

Grazie al meccanismo di Injection e alle annotazioni lo sviluppatore non ha più la necessità di interagire con le API JNDI, le dipendenze vengono espresse a livello del bean attraverso l’annotazione **@Resource**, a runtime si utilizza il metodo di lookup **EJBContext.lookup** come nell’esempio seguente:

```

@Resource(name="myDB", type=javax.sql.DataSource)
@Stateful
public class ShoppingCartBean implements ShoppingCart {

    @Resource
    SessionContext ctxt;

    public Collection startToShop (String productName) {
        ...
        DataSource productDB = (DataSource)ctxt.lookup("myDB");
        Connection conn = myDB.getConnection();
        ...
    }
    ...
}

```

L'annotazione **@Resource** permette di bypassare JNDI, recuperare il contesto e la risorsa d'interesse, a questo punto all'interno del metodo di business basta richiamare l'operazione di lookup. La classe bean risulta quindi un POJO.

Lato client, nello standard EJB 3.0, l'annotazione **@EJB** permette di recuperare il componente che verrà iniettato senza la necessità di effettuare operazioni di lookup, recuperare l'interfaccia Home etc...

```

// Vista cliente in EJB3.0

@EJB
ShoppingCart myCart;
...
Collection widgets = myCart.startToShop("widgets");
...

```

### Annotazione **@Resource**

L'annotazione **@Resource** serve a dichiarare dei riferimenti a delle risorse esterne, può essere applicata sia a livello di classe, di metodo e anche a livello di campo. Il container risolvere le dipendenze gestendo il meccanismo di dependency injection, a seconda delle annotazioni, e sfrutterà, dietro le quinte, JNDI per risolvere le dipendenze.

Il mapping con la risorsa JNDI è risolto tramite inferenza a partire dal nome e dal tipo.

L'annotazione ha i seguenti elementi:

- **Name:** nome logico con cui la risorsa viene cerata in JNDI. È opzionale a livello di campo o metodo e a default è “/nomeDellaClasse/nomeDelCampo”, nel primo caso, e nel secondo caso è basato sul metodo.
- **Type:** il tipo Java della risorsa. È determinato in base al tipo del campo o della proprietà su cui l'annotazione è posta.
- **Authentication Type:** tipo di autenticazione da usare. Usato solo con risorse di tipo connection factory.
- **Shareable:** possibilità di condividere la risorsa.
- **MappedName:** nome non portabile a cui associare la risorsa. Usato tipicamente per riferire la risorsa al di fuori dell'application server.
- **Description:** descrizione della risorsa.

Il container si occupa dell'injection della risorsa nel componente o a runtime o quando esso è inizializzato in base se l'annotazione viene specificata a livello di classe o di campo/metodo:

- **Campo:** se l'annotazione è posta a livello di campo l'injection avviene al momento dell'inizializzazione del componente.

```
public class SomeClass {  
    @Resource  
    private javax.sql.DataSource myDB;  
}
```

In questo caso poiché l'annotazione non specifica né name né type l'inferenza è fatta sulla base del nome della classe e del campo: **name=com.example.SomeClass/myDB** e **type=javax.sql.DataSource.class**.

- **Metodo:** se l'annotazione è posta a livello di metodo l'injection avviene al momento dell'inizializzazione del componente.

```
public class SomeClass {  
    private javax.sql.DataSource myDB;  
    ...  
    @Resource  
    private void setmyDB(javax.sql.DataSource ds) {  
        myDB = ds;  
    }  
    ...  
}
```

In questo caso il container inferisce il nome della risorsa sulla base dei nomi di classe e metodo: **name=com.example.SomeClass/myDB, type=javax.sql.DataSource**.

- **Classe:** se l'annotazione è posta a livello di classe l'injection avviene a runtime, peraltro in questo caso è obbligatorio utilizzare gli elementi name e type.

```
@Resource(name="myMessageQueue", type="javax.jms.ConnectionFactory")  
public class SomeMessageBean { ... }
```

Nel caso in cui sia necessario specificare delle risorse multiple sia usa l'annotazione **@Resources** che va posta a livello di classe:

```
@Resources({  
    @Resource(name="myMessageQueue", type="javax.jms.ConnectionFactory"),  
    @Resource(name="myMailSession", type="javax.mail.Session")  
})  
public class SomeMessageBean { ... }
```

## 5.4 Interoperabilità e Migrazione fra EJB 3.0 e EJB 2.x

Il passaggio dalla versione 2.1 alla 3.0 di EJB ha alla base l'idea che quanto realizzato fino a quel momento non andasse perduto. Per questo motivo EJB 3.0 mira ad essere completamente compatibile con lo standard precedente. Infatti, i componenti EJB 2.1 possono essere eseguiti senza nessun problema all'interno di un container EJB 3.0, inoltre i componenti EJB 3.0 possono essere clienti dei bean realizzati secondo la specifica 2.1 e viceversa.

Per esempio, lo snippet (frammento di codice) sottostante mostra come un cliente EJB 3.0 interagisce con un bean EJB 2.1:

```
// Vista cliente da EJB 3.X di un bean EJB 2.X

@EJB
ShoppingCartHome cartHome;

Cart cart = cartHome.create();
cart.addItem(...);
cart.remove();
```

Il cliente EJB 3.0 usa l'annotazione **@EJB** che permette di bypassare l'interazione con JNDI ma esplicita comunque il fatto di aver bisogno dell'interfaccia Home (`ShoppingCartHome`) perché lo Shopping Cart è un bean 2.1.

Dall'altro lato, i vecchi bean 2.1 possono utilizzare i componenti EJB 3.0, infatti le interfacce Home e Object vengono automaticamente mappate sulla classe del bean 3.0. Lo snippet sottostante riporta come un cliente EJB 2.1 interagisce con un componente EJB 3.0:

```
// Vista cliente da EJB 2.X di un bean conforme a EJB 3.X

Context initialContext = new InitialContext();
ShoppingCartHome myCartHome = (ShoppingCartHome)initialContext.lookup("java:comp/env/ejb/cart");
ShoppingCart cart = myCartHome.create();
cart.addItem(...);
cart.remove();
```

Nonostante Shopping Cart in questo caso sia un bean 3.0 il container crea per lui in automatico le interfacce Home e Object cosicché il cliente 2.1 possa interagire con lui come se fosse un bean 2.1.

## 5.5 Servizi di sistema container-based

Come già visto nel caso di EJB 2.1 i servizi di sistema messi a disposizione da parte del container sono molteplici:

- **Pooling e Concorrenza**
- **Transazionalità**
- **Gestione delle connessioni e risorse**
- **Persistenza**
- **Messaggistica**
- **Sicurezza**

### 5.5.1 Pooling e Concorrenza

Il pooling viene gestito in modi diversi in base se il componente ha stato oppure no. Il **Resource Pooling** viene utilizzato per gli Stateless Session Bean e per i Message Driven Bean. **Activation** viene utilizzato per gli Stateful Session Bean per risparmiare risorse.

#### Resource Pooling

L'idea base è quella di evitare di mantenere una istanza separata di ogni EJB per ogni cliente. Nel caso degli Stateless Session Bean il container mantiene un insieme di istanze del bean pronte per servire eventuali client. Non essendoci stato di sessione da mantenere fra richieste successive ogni invocazione di metodo è indipendente dalle precedenti.

Il ciclo di vita di uno Stateless Session Bean può essere riassunto dai seguenti stati:

- **No state:** il bean non è ancora stato istanziato, corrisponde allo stato iniziale e terminale del ciclo di vita.
- **Pooled state:** il bean è stato istanziato ma non è ancora stato associato a nessun cliente.
- **Ready state:** il bean è stato associato ad un cliente e pronto a rispondere a una sua richiesta.

Le istanze che si trovano nel pool di risorse ricevono un riferimento al contesto **javax.ejb.EJBContext**, questo contesto comune fornisce un interfaccia al bean per comunicare con l'ambiente EJB. Quando il bean passa allo stato ready vengo memorizzate nel Context le informazioni del client che sta utilizzando il bean.

Quanto detto finora è valido anche per i Message-driven bean. I MDB non mantengono stato della sessione e quindi il container può effettuare pooling in modo relativamente semplice. L'unica differenza sta nel fatto che ogni EJB container contiene pool diversi associati a destinazioni JMS diverse, ciascuno dei quali è popolato da istanze (anche di classi MDB diverse ma con la stessa destinazione).

#### Activation

Questo meccanismo è usato nel caso di Stateful session bean, cioè bean che sono legati con uno stato “soft” ad un determinato cliente. Si ha la necessità di mantenere lo stato della sessione tra bean e client, la gestione avviene in due fasi:

- **Passivation:** disassociazione fra l'istanza dello stateful session bean e il relativo oggetto EJB, con salvataggio dell'istanza in memoria (**serializzazione**).
- **Activation:** lo stato dell'istanza viene recuperato dalla memoria (**deserializzazione**) e riassociato all'oggetto EJB.

Tuttavia, la specifica EJB 3.0 impone che gli stateful session bean mantengano lo stato ma non richiede che la classe stateful session bean sia serializzabile, l'implementazione del meccanismo necessario al mantenimento dello stato dipende dallo specifico ‘vendor’.

Quando uno stateful session bean termina di eseguire la richiesta di un client non può rimanere legato a quel cliente senza far nulla solo per mantenere lo stato. Per superare questo problema si salva lo stato in memoria, in tal modo l'istanza dello stateful session bean sarà utilizzabile.

La procedura di activation può essere associata all'invocazione di metodi di callback, questo permette di eseguire metodi subito dopo l'attivazione, usando l'annotazione **@javax.ejb.PostActivate**, o prima della passivizzazione usando l'annotazione **@javax.ejb.PrePassivate**. Questo consente, per esempio, di chiudere/aprire connessioni a risorse esterne, serializzare riferimenti a risorse esterne (di default vengono mantenuti nello stato solo i riferimenti remoti ad altri bean, a SessionContext, al servizio EntityManager e all'oggetto UserTransaction).

Dal punto di vista della concorrenza il tutto è realizzato in modo che una singola istanza di un session bean sia associata ad un singolo cliente, per definizione infatti, i session bean non possono essere concorrenti in senso logico. È infatti vietato utilizzare thread a livello applicativo. Nel caso degli entity bean, invece, una singola istanza può essere acceduta da più clienti, in questo caso il tutto è gestito da una specifica per la persistenza (JPA).

Infine, nel caso dei message driven bean è necessario supportare la concorrenza, nel senso di processamento concorrente di più messaggi verso la stessa destinazione.

### 5.5.2 Transazionalità

Una transazione è un insieme di operazioni logiche (query) a cui corrispondono operazioni fisiche di lettura e scrittura sul DB. Le proprietà che una transazione deve rispettare sono quelle ACID (Atomicity, Consistency, Isolation e Durability) quindi la transazione è un'unità indivisibile di processamento che può: terminare correttamente (commit) oppure no (rollback).

Le transazioni possono essere gestite dal container (**Container-Managed Transaction**) o manualmente dal programmatore (**Bean-Managed Transaction**).

#### Container-Managed Transaction

La metodologia di default per la gestione delle transazioni è quella in cui le transazioni sono gestite dal container.

Si usa l'annotazione **@TransactionManagement** che può assumere come valori **CNTAINER** (valore di default) oppure **BEAN**.

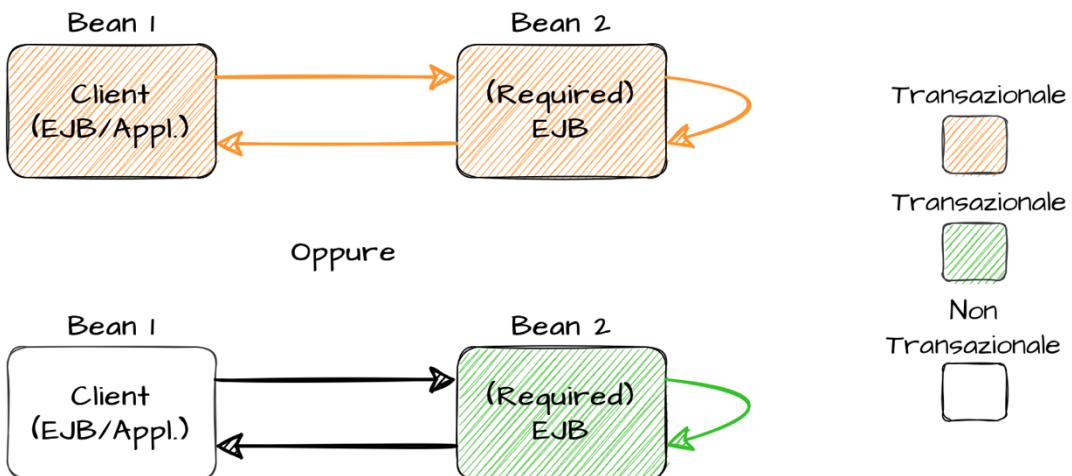
Una transazione è associata con l'intera esecuzione di un metodo: inizia immediatamente prima dell'inizio dell'esecuzione del metodo e viene “committata” immediatamente prima della terminazione del metodo. Con la modalità Container-Managed non si possono utilizzare metodi per la gestione delle transazioni che interferiscono con la gestione automatica del container. Ad

esempio, è proibito l'uso di commit o rollback di `java.sql.Connection`, di rollback di `javax.jms.Session` o dell'intera interfaccia `javax.Transaction.UserTransaction`.

Per rendere più flessibili le transazioni gestite dal container si possono modificare i valori degli attributi di transazione che permettono di controllare lo scope di una transazione. Ad esempio, si considerino BeanA e BeanB: se il BeanA invoca un metodo del BeanB, a default, viene creata un'unica grande transazione che inizia con il metodo di BeanA e termina solo quando è stato eseguito il metodo di BeanB. Se si vuole usare un approccio moderno è necessario rilassare la proprietà ACID della transazione. Per far ciò si fa uso dell'annotazione `@TransactionAttribute` i cui possibili valori sono: **REQUIRED** (default), **REQUIRES\_NEW**, **MANDATORY**, **NOT\_SUPPORTED**, **SUPPORTS**, **NEVER**. La tabella mostra schematicamente cosa succede a livello transazionale lato client e lato server per ogni attributo.

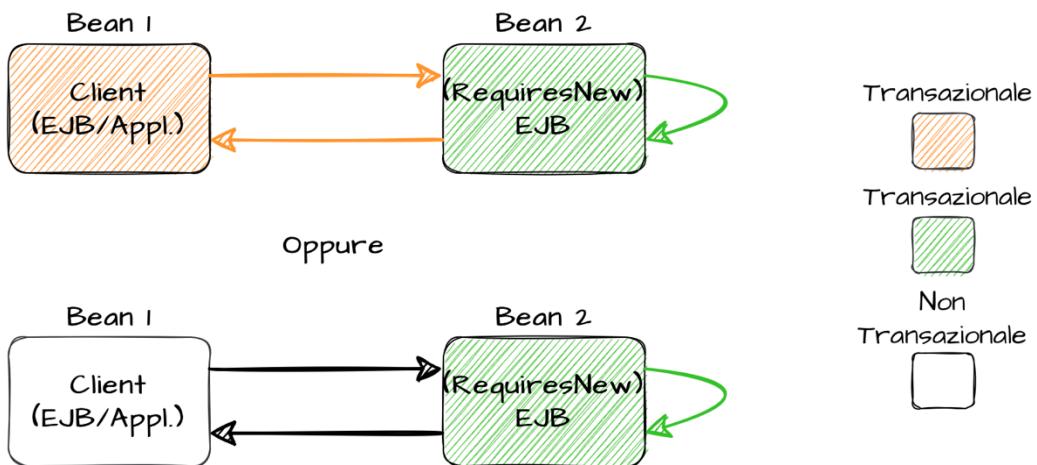
Attributo	Transazione lato cliente?	Transazione lato servitore?
<b>Required</b>	Nessuna	T2
	T1	T1
<b>RequiresNew</b>	Nessuna	T2
	T1	T2
<b>Mandatory</b>	Nessuna	Errore
	T1	T1
<b>NotSupported</b>	Nessuna	Nessuna
	T1	Nessuna
<b>Supports</b>	Nessuna	Nessuna
	T1	T1
<b>Never</b>	Nessuna	Nessuna
	T1	Errore

### Required



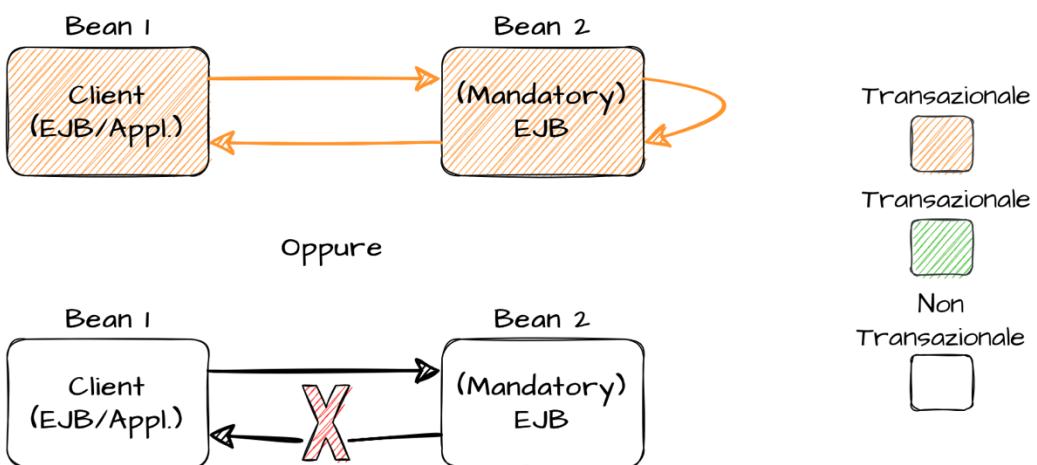
Nel caso **Required** se il Bean 1 lancia un metodo 'x' che richiede transazionalità e invoca un metodo 'y' del Bean 2, il metodo 'y' farà parte della stessa transazione del metodo 'x' del Bean 1, oppure, se il Bean 1 lancia un metodo 'x' che non richiede transazionalità e invoca un metodo 'y' del Bean 2, il metodo 'y' sarà transazionale (N.B il commit della transazione avviene nel metodo 'y' prima di ritornare al metodo 'x').

### RequiresNew



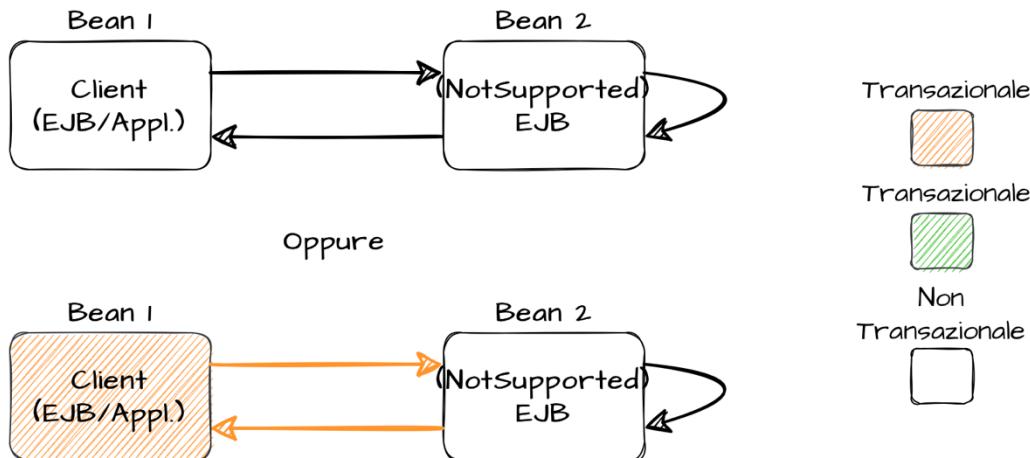
Nel caso **RequiresNew** se il Bean 1 lancia un metodo 'x' che apre una transazione T1 e invoca un metodo 'y' del Bean 2, il metodo 'y' aprirà una seconda transazione T2 il cui commit avverrà prima di ritornare al metodo 'x', oppure, se il Bean 1 lancia un metodo 'x' che non richiede transazionalità e invoca un metodo 'y' del Bean 2, allora il metodo 'y' apre una transazione T2 il cui commit avviene prima di ritornare al metodo 'x'.

### Mandatory



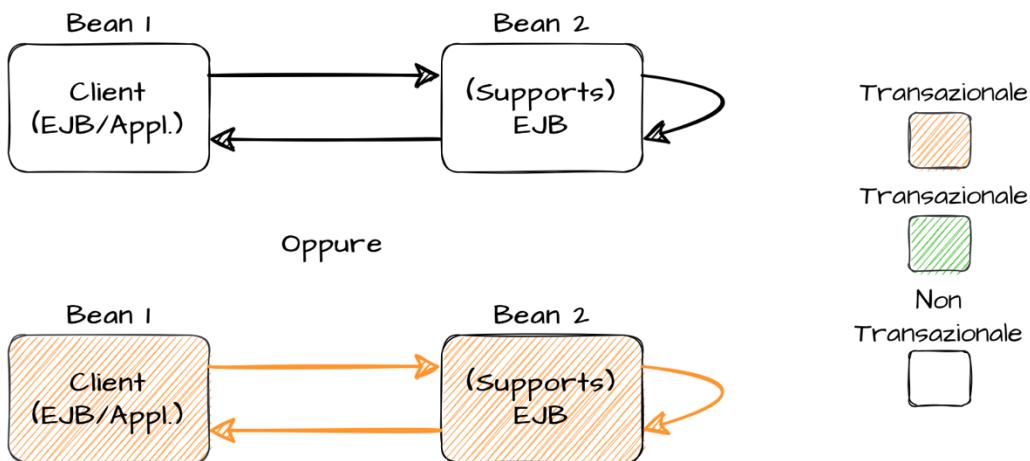
Nel caso **Mandatory** se il Bean 1 lancia un metodo 'x' che richiede transazionalità e invoca un metodo 'y' del Bean 2, il metodo 'y' farà parte della stessa transazione del metodo 'x', nella Mandatory non è possibile che il Bean 1 non abbia una transazione aperta, infatti, nel secondo scenario la X rossa ci fa capire che se il Bean 1 non apre una transazione il metodo 'y' del Bean 2 non può essere eseguito.

## NotSupported



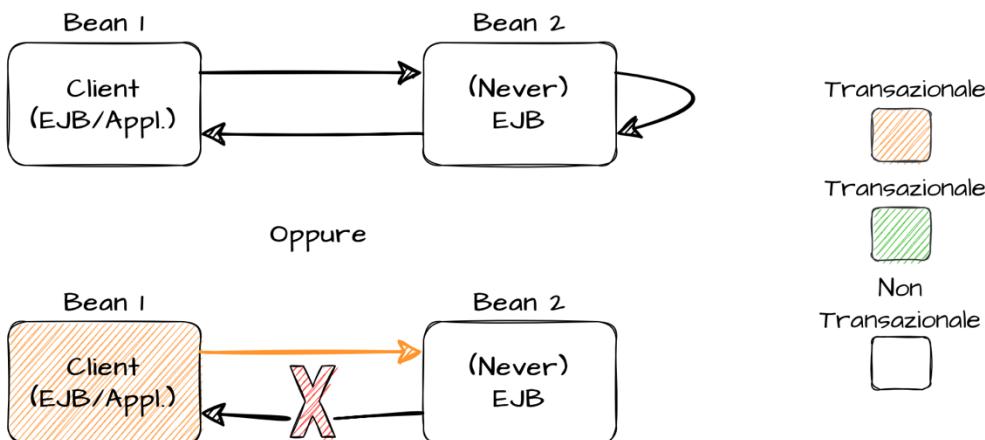
Nel caso **NotSupportedException** se il Bean 1 lancia un metodo 'x' che non richiede transazionalità e invoca un metodo 'y' del Bean 2, il metodo 'y' non sarà transazionale, oppure, se il Bean 1 lancia un metodo 'x' che richiede transazionalità e invoca un metodo 'y' del Bean 2, il metodo 'y' non sarà transazionale e quindi la transazione avviene solo sul metodo 'x' del Bean 1.

## Supports



Nel caso **Supports** se il Bean 1 lancia un metodo 'x' non transazionale e invoca un metodo 'y' del Bean 2 anche il metodo 'y' non sarà transazionale, oppure, se il Bean 1 lancia un metodo 'x' che richiede transazionalità e invoca un metodo 'y' del Bean 2, il metodo 'y' farà parte della stessa transazione del metodo 'x'.

## Never



Il caso **Never** è il duale di Mandatory, non si intende avere mai un comportamento transazione. Se il Bean 1 lancia un metodo 'x' che non richiede transazionalità e invoca un metodo 'y' del Bean 2, il metodo 'y' non sarà transazionale, oppure, se il Bean 1 lancia un metodo 'x' che richiede transazionalità e invoca un metodo 'y' del Bean 2 il metodo 'y' non sarà eseguito.

Se una transazione fallisce bisogna effettuare il rollback della transazione. L'operazione di rollback può essere scatenata da due cause:

- Eccezione del sistema, in questo caso il container automaticamente lancia il rollback.
- Invocando il metodo `setRollbackOnly` di `EJBContext`. `EJBContext` è un'interfaccia che consente di accedere a molte funzionalità del container.

Usando i Session Bean si ha la possibilità, utilizzando l'interfaccia `SessionSynchronization`, di sincronizzare l'esecuzione di un session bean con la transazione attraverso i metodi:

- **AfterBegin:** invocato dal container immediatamente prima dell'invocazione del metodo di business all'interno della transazione.
- **BeforeCompletion:** invocato dal container immediatamente prima del commit della transazione.
- **AfterCompletion:** invocato dal container immediatamente dopo il completamento della transazione (sia nel caso di commit che di rollback).

## Bean-Managed Transaction

In questa metodologia la transazione è gestita dal bean a livello programmatico, è quindi il programmatore che deve farsi carico delle operazioni che nel caso precedente erano effettuate automaticamente dal container. Questo comporta una maggiore complessità dal punto di vista del codice a fronte però di una maggiore flessibilità.

Si possono utilizzare le **Java Transaction API** per essere indipendenti dall'implementazione dello specifico transaction manager.

### 5.5.3 Gestione delle connessioni e risorse

Un componente può avere bisogno di utilizzare altri componenti e risorse, come ad esempio, database e sistemi di messaging. In JEE, il ritrovamento delle risorse desiderate è basato su un sistema di nomi ad alta portabilità come JNDI. In EJB 2.1 questa operazione era a carico del programmatore, a partire da EJB 3.0 con l'introduzione del meccanismo di injection è il container a farsi carico di interrogare JNDI e non il componente. Dal momento che il container si occupa del ritrovamento delle risorse per conto dei componenti, esso può effettuare un processo di ottimizzazione, per esempio, se molti bean richiedono l'accesso ad un determinato database il container anziché aprire e chiudere una connessione con il database per ogni bean, potrebbe riutilizzare le connessioni già aperte in precedenza per altri bean ottimizzando il processo, aumentando le prestazioni e abbattere la latenza.

### 5.5.4 Persistenza

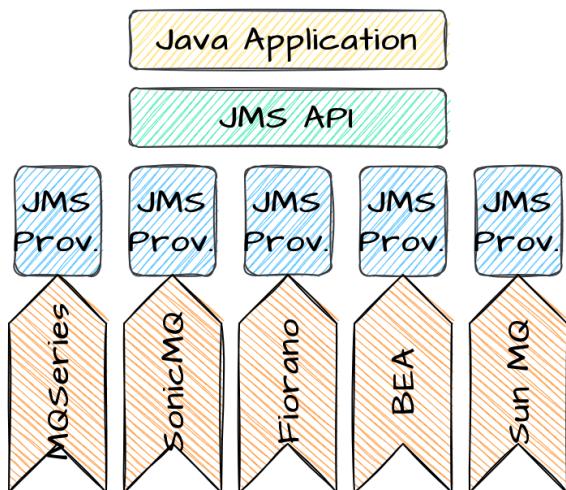
L'obiettivo è quello di semplificare il modello di programmazione degli entity bean offrendo supporto per il mapping tra il mondo ad oggetti e il mondo relazionale tramite l'uso di annotazioni.

Il nuovo modello di persistenza introdotto con EJB 3.0 vede le entità (entity bean) come delle semplici classi Java (POJO) che usano i metodi per effettuare il set/get di proprietà o variabili di istanza persistenti. L'entity manager svolge il ruolo di "Home" per le operazioni di entity (persist, remove, merge, etc...). Questi concetti saranno ripresi nel capitolo 6 parlando di JPA.

### 5.5.5 Messaging

I message driven bean e i servizi di messaggistica sono gestiti dal **Java Messaging System (JMS)**. La Java Enterprise Edition definisce come un cliente JMS acceda alle funzionalità supportate da un sistema di messaging di livello enterprise. JMS supporta le semantiche di tipo sincrono/asincrono, transacted, garantita, durevole.

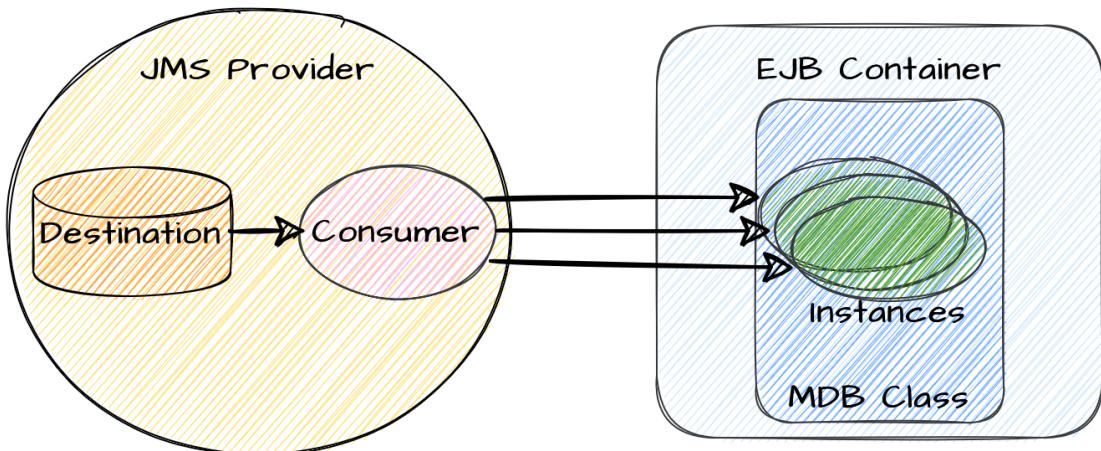
La specifica JMS presenta molte implementazioni offerte da 'vendor' diversi.



JMS supporta i modelli comunicazione:

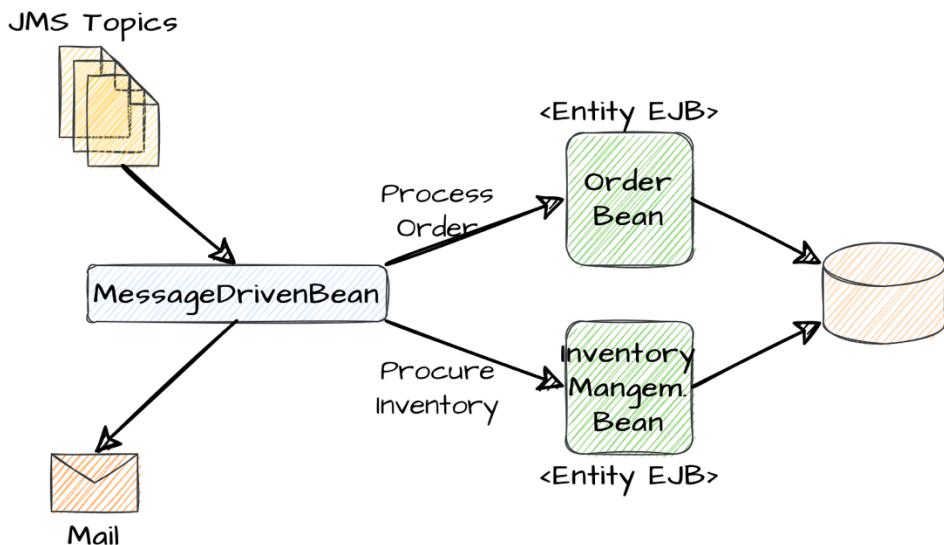
- **Point to Point**
- **Publish/Subscribe**

A livello concettuale la struttura di JMS può essere vista in questo modo:



Da un lato il JMS Provider che invia i messaggi, dall'altro all'interno del EJB Container, il Message-Driven Bean che riceve i messaggi.

Il message-driven bean sono visti come consumatori asincroni di messaggi JMS, ricevono il messaggio JMS ed eseguono la logica di business. Il fatto che gli MDB lavorino in maniera asincrona permette di utilizzarli in interazioni con altri componenti che realizzano una logica applicativa molto onerosa (in termini di tempo di esecuzione).



## 5.5.6 Sicurezza

La gestione della sicurezza è un altro servizio fornito dal container. Il container EJB svolge azioni di controllo dell'accesso sui metodi del bean:

- Consulta le policy di sicurezza (derivanti dal deployment descriptor e/o annotazioni) per determinare i differenti ruoli di accesso.
- Per ogni ruolo usa un contesto di sicurezza associato con l'invocazione per determinare i permessi di chiamata
- Se il chiamante è autorizzato, il container mappa le credenziali del chiamante sul ruolo autorizzato e passa il controllo all'EJB.
- Se il chiamante non è autorizzato, il container lancia un'eccezione.

Il container EJB basa le sue decisioni di sicurezza sui concetti di **Realm, Utenti, Gruppi e Ruoli**.

Il Realm è visto come una collezione di utenti di una singola applicazione, controllati dalla stessa policy di autenticazione. Gli utenti sono visti come singoli individui, i gruppi sono insiemi di utenti e ad ogni utente o gruppo viene assegnato un ruolo.

La configurazione della sicurezza avviene tipicamente a deployment time tramite annotazioni o deployment descriptor.

Le annotazioni possibili sono:

- **@RolesAllowed:** il cui valore è una lista di nomi o ruoli e permette l'utilizzo del metodo su cui è apposta solo ai nomi/ruoli specificati.
- **@PermitAll:** permette a chiunque di utilizzare il metodo su cui è apposta.
- **@DenyAll:** non permette a nessuno di eseguire quel metodo.
- **@RunAs:** per cambiare il ruolo di chi sta eseguendo un determinato metodo.

## 5.6 Gli intercettori

Gli intercettori sono oggetti capaci di interporsi sulle chiamate di metodo o su eventi riguardando il ciclo di vita di SB e MDB.

Il container si interpone sempre sulle invocazioni dei metodi della logica applicativa, gli intercettori, quando presenti, si interpongono dopo il container e prima dell'esecuzione della logica applicativa. Possono essere utilizzati, per esempio, per recuperare il tempo di esecuzione di un metodo.

L'intercettore è uno strumento potente che bisogna usare con prudenza, inserire al suo interno della logica applicativa (che per esempio cambia uno stato) sparge il codice che prima era interamente contenuto nel componente, da altre parti, diventa poi difficile controllare il codice. Gli intercettori seguono il modello di invocazione **AroundInvoke** perché vengono chiamati prima e dopo l'invocazione di un metodo. Gli intercettori sono invocati nello stesso stack di chiamate Java, con stesso contesto di sicurezza e di transazionalità.

Per gestire gli intercettori si fa uso delle annotazioni:

- **@Interceptors:** per associare una classe/metodo di un componente di business alla classe intercettore correlata.

- **@AroundInvoke:** per definire quale metodo della classe intercettore eseguire al momento dell'intercettazione.

Gli intercettori possono essere definiti a livello di:

- **File descriptor:** sono intercettori di default, si applicano a tutti i metodi di business di ogni componente nel file ejb-jar. Non ci sono annotazioni a livello di applicazione.
- **Classe:** si applicano ai metodi di business della classe bean.
- **Metodo:** per determinazioni più fine-grained e anche per fare overriding di associazioni precedenti.

## 6. Persistenza, JPA e Hibernate

### 6.1 Java Persistence API (JPA)

Una parte rilevante degli sforzi nello sviluppo di qualsiasi applicazione distribuita di livello enterprise si concentra sul layer di persistenza, dove avvengono accesso, manipolazione e gestione di dati persistenti, tipicamente mantenuti in un DB relazionale. Il **mapping Oggetto Relazione** (ORM) si occupa di risolvere il potenziale mismatch fra dati mantenuti in un data base relazionale (table-driven) e il loro processamento tramite oggetti in esecuzione. I database relazionali sono progettati per operazioni di query efficienti su dati di tipo tabellare, ma nelle applicazioni vi è la necessità di lavorare invece tramite interazione fra oggetti.

Prima di JPA, il tipico modo per accedere a dati era tramite metodi Java **Data Access Object** (DAO), per operare sui dati con operazioni di creazione, eliminazione e ricerca, questo implicava che persistenza e transazionalità fossero gestite in modo programmatico.

```
public SampleDAO samplelookup(String id) {  
  
    Connection c = null;  
    PreparedStatement ps = null;  
    ResultSet rs = null;  
    SampleDAO dao = null;  
    try {  
        c = getDataSource().getConnection();  
        ps = c.prepareStatement("SELECT ...");  
        ps.setString(1, id);  
        rs = ps.executeQuery();  
        if (rs.first()) {  
            dao = new SampleDAO(id, rs.getString(2), rs.getString(2));  
        }  
    }  
    catch (SQLException se) {  
        throw new SampleDAORuntimeException(se);  
    }  
    finally {  
        if (rs != null) try {rs.close();} catch (SQLException se) {}  
        if (ps != null) try {ps.close();} catch (SQLException se) {}  
        if (c != null) try {c.close();} catch (SQLException se) {}  
    }  
    return dao;  
}
```

L'adozione di questo metodo implicava la scrittura di una grande quantità di codice, con forte dipendenza dell'applicazione dal data-store. In questo senso **JPA** si pone come soluzione standard per riuscire a gestire velocemente la persistenza. Di seguito viene riportato un esempio di ricerca con chiave primaria tramite JPA:

```
Sample entity = entityManager.find(Sample.class, id);
```

## 6.2 Entity

Le Entità in JPA sono dei Plain Old Java Object (POJO) che vengono create attraverso l'invocazione di **new()** come per qualsiasi oggetto Java, non vi è nessuna necessità di implementare interfacce come richiesto per gli Entity Bean EJB 2.x. Ogni entità può avere stato sia persistente che non persistente, lo stato non persistente viene identificato tramite l'annotazione **@Transient**. Ciascuna entità può fare sub-classing di altre classi, sia Entity che non Entity, ed è serializzabile. Tutte le entità sono utilizzabili come detached object in altri tier per esempio in Hibernate. Inoltre, non sono necessari oggetti specifici addizionali per il trasferimento di dati, ovvero non vi è necessità di DTO esplicativi come in DAO.

Una Entity è un oggetto “leggero” appartenente a un dominio di persistenza, usualmente rappresenta dati di un DB relazionale: ogni istanza di Entity corrisponde a una riga in una tabella. Lo stato persistente di una Entity è rappresentato da campi persistenti o da proprietà persistenti. Questi campi/proprietà usano delle annotazioni per l'Object Relational Mapping (ORM): associazione con entità e relazioni per i dati mantenuti nel DB.

Le classi Entity devono essere definite dall'annotazione **javax.persistence.Entity (@Entity)**, avere un costruttore senza argomenti, che può essere **public** o **protected** (costruttori aggiuntivi sono ovviamente consentiti), inoltre nessun metodo o variabile d'istanza persistente deve essere dichiarata **final**.

Le Entity possono anche essere sottoclassi di classi entità oppure di classi “normali” come sottoclassi di Entity. Tutte le variabili di istanza persistenti devono essere dichiarate **private**, **protected**, o **package-private**, e possono essere accedute direttamente solo dai metodi della classe dell'Entity. Se una istanza di Entity è passata per valore (by value) come oggetto detached, ad esempio all'interfaccia remota di un Session Bean, allora la classe deve implementare l'interfaccia **Serializable**.

Le Entity possono avere campi persistenti cioè variabili di istanza a cui sono applicate annotazioni di mapping o proprietà persistenti, ovvero metodi getter/setter a cui sono applicate annotazioni di mapping. Tuttavia, non si possono utilizzare annotazioni di entrambi i tipi in una singola Entity. Ai campi persistenti è consentito l'accesso diretto alle variabili di istanza, sono identificati e gestiti come campi persistenti verso il DB tutti i campi non annotati con **javax.persistence.Transient (@Transient)**.

I campi persistenti presentano alcune proprietà: in primo luogo devono seguire le convenzioni sui metodi tipiche dei JavaBean, devono possedere metodi getter e setter (`getProperty`, `setProperty`, `isProperty`). Per campi o proprietà persistenti con valori non singoli (collection-valued) devono essere utilizzate Java Collection.

Ogni Entity deve avere un identificatore unico di oggetto. La chiave primaria di una Entity può essere semplice o composta, la chiave primaria semplice utilizza l'annotazione `javax.persistence.Id (@Id)` per indicare la proprietà o il campo chiave.

```
@Entity
public class Project {
    @Id
    private long id;

    ...
}
```

Mentre la chiave primaria composta può contenere un insieme di campi o proprietà persistenti, e in questo caso si utilizzano le annotazioni `javax.persistence.EmbeddedId (@EmbeddedId)` o `javax.persistence.IdClass (@IdClass)`:

```
@Entity @IdClass(ProjectId.class)
public class Project {
    @Id
    private int departmentId;
    @Id
    private long projectId;
    ...

}

public class ProjectId {
    private int departmentId;
    private long projectId;
}
```

Quando un'entità ha più campi chiave primaria, JPA richiede la definizione di una classe ID speciale collegata alla classe dell'entità utilizzando l'annotazione `@IdClass`. La classe ID riflette i campi della chiave primaria e i suoi oggetti rappresentano i valori della chiave primaria. Un modo alternativo per rappresentare una chiave primaria composta consiste nell'utilizzare una classe incorporabile.

```
@Entity
public class Project {
    @EmbeddedId
    private ProjectId id;
    ...
}

@Embeddable
public class ProjectId {
    private int departmentId;
    private long projectId;
}
```

Le classi di chiavi primarie prevedono alcuni requisiti che devono essere rispettati:

- deve essere dichiarato un costruttore di default.
- bisogna implementare i metodi **hashCode()** e **equals(Object other)**.
- bisogna rendere l'oggetto serializzabile.
- se la classe contiene campi/proprietà multiple della classe Entity, nomi e tipi dei campi/proprietà nella chiave devono fare match con quelli nella classe Entity.

```
@Entity
public final class LineItemKey implements Serializable {
    public Integer orderId;
    public int itemId;

    public LineItemKey() {
    }

    public LineItemKey(Integer orderId, int itemId) {
        this.orderId = orderId;
        this.itemId = itemId;
    }

    public boolean equals(Object otherOb) {
        if (this == otherOb) {
            return true;
        }
        if (!(otherOb instanceof LineItemKey)) {
            return false;
        }
        LineItemKey other = (LineItemKey) otherOb;
        return ((orderId==null ? other.orderId==null : orderId.equals(other.orderId))
                && (itemId == other.itemId));
    }

    public int hashCode() {
        return ((orderId==null?0:orderId.hashCode())^((int) itemId));
    }

    public String toString() {
        return "" + orderId + "-" + itemId;
    }
}
```

### 6.2.1 ORM: Entity ed Ereditarietà

In JPA è presente il supporto all'ereditarietà, all'associazione polimorfica e a query polimorfiche per le classi Entity. Le entity possono estendere classi non-Entity, e classi non-Entity possono estendere entità. Le classi Entity possono essere sia astratte che concrete. Se una query è effettuata su una Entity astratta, si opera su tutte le sue sottoclassi non astratte.

```
@Entity
public abstract class Employee {
    @Id
    protected Integer employeeId;
    ...
}

@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary;
    ...
}

@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;
}
```

Le Entity possono anche ereditare da superclassi che contengono stato persistente e informazioni su ORM, ma che non sono Entity (ovvero superclasse non decorata con `@Entity`), in questo caso si parla di **Mapped Superclass**.

```
@MappedSuperclass
public class Employee {
    @Id
    protected Integer employeeId;
    ...
}

@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary;
    ...
}

@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;
    ...
}
```

## 6.2.2 Entity Inheritance e Strategie di Mapping

In JPA è possibile decidere come il provider di persistenza debba fare mapping della gerarchia di classi Entity definita sulle tabelle del DB, si utilizza, per fare questo, l'annotazione `javax.persistence.Inheritance`, vi sono tre possibilità:

- **InheritanceType.SINGLE\_TABLE**: viene generata una tabella unica per l'intera gerarchia, tutte le proprietà sono memorizzate in questa tabella, insieme a un discriminator per stabilire il tipo di entity.
- **InheritanceType.JOINED**: viene creata una tabella differente per ogni classe Entity non astratta. Ogni tabella include solo lo stato dichiarato nella sua classe (e riferimenti alle tabelle in gerarchia).
- **InheritanceType.TABLE\_PER\_CLASS**: come JOINED, ma ogni tabella contiene tutto lo stato per un'istanza della classe corrispondente. Ogni tabella ha colonne con valore per ogni proprietà, comprese quelle ereditate dalle superclassi, non c'è bisogno di discriminator.

Il comportamento di default è quello del caso **SINGLE TABLE**, usato se l'annotation `@Inheritance` non è specificata alla classe radice della gerarchia di Entity.

La strategia **TABLE PER CLASS** offre una scarsa efficienza nel supporto a relazioni (e query) polimorfiche; di solito richiede query separate per ogni sottoclasse per coprire l'intera gerarchia. Invece, utilizzando **JOINED**, ogni sottoclasse ha una tabella separata che contiene i soli campi specifici per la sottoclasse, la tabella non contiene colonne per i campi e le proprietà ereditati. **JOINED** offre un buon supporto a relazioni polimorfiche, ma richiede operazioni di join, anche multiple, quando si istanziano sottoclassi di Entity, di conseguenza la performance è scarsa per gerarchie di classi estese. La scelta della strategia ottimale presuppone una buona conoscenza del tipo di query che si faranno sulle Entity.

## 6.2.3 ORM: Molteplicità nelle Relazioni

Vi sono quattro tipologie di molteplicità ORM che corrispondono a relazioni E/R:

- **One-to-one**: ogni istanza di un'Entity è associata a una singola istanza di un'altra Entity. Si usa l'annotazione `javax.persistence.OneToOne` (`@OneToOne`) sul corrispondente campo/proprietà persistente.
- **One-to-many**: ad ogni istanza di un'Entity sono associate più istanze di un'altra Entity. Si usa l'annotazione `javax.persistence.OneToMany` (`@OneToMany`).
- **Many-to-one**: il duale del precedente. Si usa l'annotazione `javax.persistence.ManyToOne` (`@ManyToOne`)
- **Many-to-many**: annotation `javax.persistence.ManyToMany` (`@ManyToMany`).

## 6.2.4 ORM: Direzionalità nelle Relazioni

Una relazione può essere monodirezionale o bidirezionale, questo determina anche come il gestore di persistenza aggiorni la relazione nel DB. Se la relazione è bidirezionale ogni entità

ha un campo o una proprietà che riferisce l'altra entità, ad esempio, ordine online avrà un campo che fa riferimento agli oggetti in esso contenuti e ogni oggetto avrà un campo che fa riferimento all'ordine che lo include.

È importante, in una relazione, stabilire quale delle due parti è proprietaria della relazione, per fare ciò bisogna assegnare un valore al campo **mappedBy** delle annotazioni di molteplicità (annotazioni del paragrafo 6.2.3).

La direzionalità nelle relazioni determina se una query può navigare o meno da una entità a un'altra. Questo tipo di relazioni sono utili nel caso di gestione di query da parte del container, inoltre la direzionalità si può utilizzare per navigare tra le entità e, per esempio, capire quali relazioni cancellare quando si utilizza la politica **Delete Cascade**, che significa ad esempio cancellare un oggetto parte di un ordine se viene cancellato l'ordine.

```
@OneToOne(cascade=REMOVE, mappedBy="customer")
public Set<Order> getOrders() {
    return orders;
}
```

## 6.2.5 Gestione a Runtime di Entity

Le Entity sono gestite da un **EntityManager**, cioè un'istanza di **javax.persistence.EntityManager**. Ogni istanza di EntityManager è associata ad un contesto di persistenza, che definisce lo scope all'interno del quale le istanze di Entity sono create, gestite e rimosse. All'interno di tale contesto è come se le Entity avessero un loro ciclo di vita. Il contesto di persistenza è, quindi, il luogo dove esistono tutte le istanze Entity. L'EntityManager può essere utilizzato demandando completamente la gestione al container o gestendolo a livello applicativo.

Il contesto di persistenza è l'insieme di istanze di Entity che esistono in un particolare datastore e che sono gestite da un singolo EntityManager.

L'interfaccia dell'EntityManager definisce i metodi utilizzabili per interagire con il contesto di persistenza (creazione e rimozione di istanze di Entity persistenti, ritrovamento di Entity tramite chiave primaria ed esecuzione di query).

L'EntityManager di JPA svolge funzionalità simili a **Hibernate Session**, **Java Data Objects PersistenceManager**.

L'EM controlla e gestisce il ciclo di vita di oggetti Entity eseguendo le operazioni:

- **persist()**: che inserisce una Entity nel DB.
- **remove()**: che rimuove una Entity dal DB.
- **merge()**: che sincronizza lo stato di entità detached.
- **refresh()**: che ricarica lo stato dal DB.

Inoltre, è possibile definire **listener** o metodi di **callback** che saranno invocati dal provider di persistenza quando una Entity transita da uno stato del ciclo di vita ad un altro. Per usare metodi di callback occorre annotare tali metodi nella classe della Entity o definirli all'interno di una classe listener separata. Le annotazioni correlate sono: **@PrePersist / @PostPersist**, **@PreRemove / @PostRemove**, **@PreUpdate / @PostUpdate**, **@PostLoad**.

## EntityManager a livello di Container (Container-Managed)

Nel caso di EntityManager a livello di container il contesto di persistenza è automaticamente propagato dal container a tutti i componenti applicativi che usano l'istanza di EntityManager all'interno di una singola transazione **Java Transaction Architecture** (JTA). Le transazioni JTA eseguono, generalmente, chiamate fra componenti applicativi che, per completare la transazione, devono avere accesso ad un contesto di persistenza. L'annotazione **@PersistenceContext** permette di eseguire l'injection del contesto di persistenza.

Propagazione automatica significa che i componenti applicativi non hanno bisogno di passare riferimenti a EntityManager per produrre modifiche all'interno della singola transazione, è il container che si occupa di questi aspetti.

```
@PersistenceContext
```

```
EntityManager em;
```

## Application-managed EntityManager

Negli application-managed EntityManager, invece, il contesto di persistenza non viene propagato ai componenti applicativi e il ciclo di vita delle istanze di EntityManager viene gestito direttamente dall'applicazione. Questo tipo di gestione dell'Entity Manager è da preferirsi quando l'applicazione necessita di diversi contesti di persistenza a cui bisogna associare diverse istanze di EntityManager. In questo caso, si usa il metodo **createEntityManager()** di **javax.persistence.EntityManagerFactory** per creare un nuovo Entity Manager.

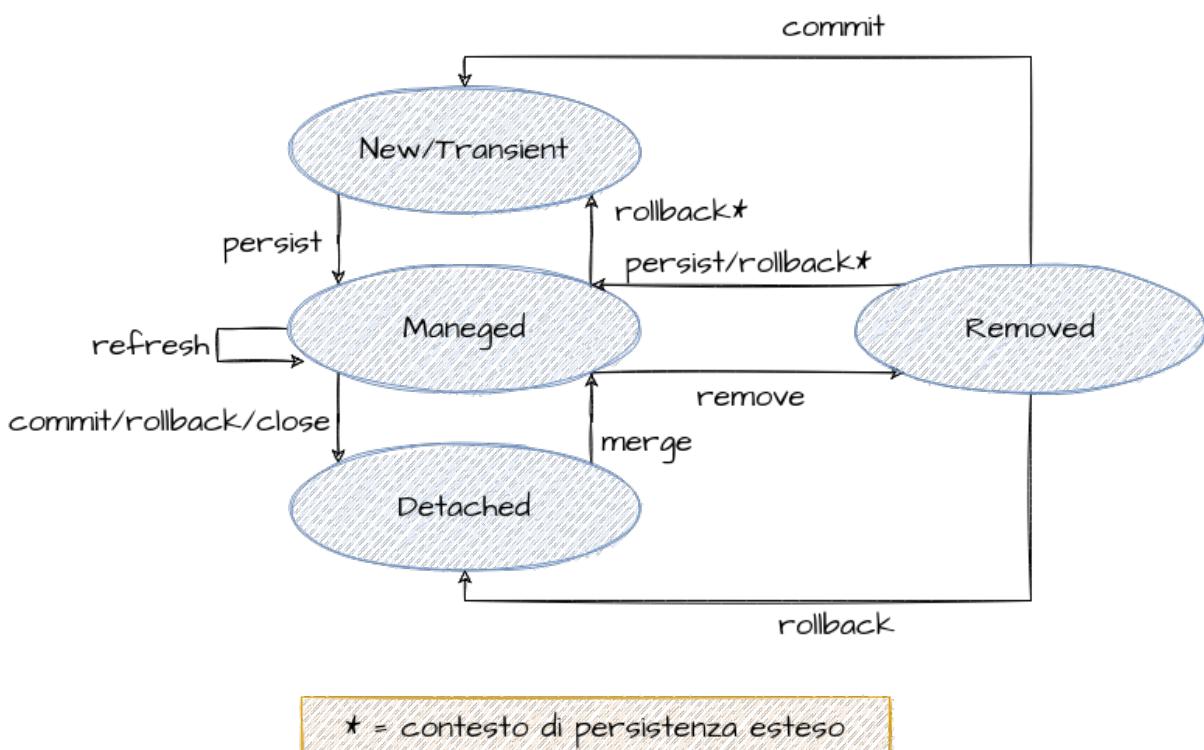
```
@PersistenceUnit  
EntityManagerFactory emf;  
EntityManager em = emf.createEntityManager();  
-----  
  
@PersistenceContext  
EntityManager em;  
public void enterOrder(int custID, Order newOrder) {  
    Customer cust = em.find(Customer.class, custID);  
    cust.getOrders().add(newOrder);  
    newOrder.setCustomer(cust);  
}
```

Di solito, quando l'applicazione è semplice ed è presente un solo DB si usa un solo Entity Manager. Invece, se l'applicazione è più complessa si usano molte Entity che fanno parte di DB diversi e di conseguenza servono più EntityManager. Tuttavia, è possibile che ci siano Entity Manager multipli verso lo stesso DB per gestire tabelle diverse e di conseguenza avere diversi contesti di persistenza.

## 6.2.6 Ciclo di Vita di Entity

La gestione delle istanze di Entity viene sempre effettuata tramite l'invocazione di operazioni sull'EntityManager. Le Entity vivono all'interno di un contesto di persistenza e possono trovarsi in quattro possibili stati differenti:

- **New/Transient**: le Entity sono nuove istanze, non hanno ancora identità di persistenza e non sono ancora associate ad uno specifico contesto di persistenza.
- **Managed**: le Entity sono istanze con identità di persistenza e sono associate ad un contesto di persistenza.
- **Detached**: le Entity sono istanze con identità di persistenza e correntemente (possibilmente temporaneamente) disassociate da contesti di persistenza.
- **Removed**: le Entity sono istanze con identità di persistenza, associate ad un contesto e la cui eliminazione dal datastore è stata già schedulata.



Nuove istanze di Entity diventano gestite e persistenti o invocando il metodo **persist()** di EntityManager o tramite un'operazione **persist()** invocata da Entity correlate con **cascade=PERSIST** o **cascade=ALL** nelle annotation di relazione. Questo comporta che i dati saranno memorizzati nel DB quando la transazione associata a **persist()** sarà completata.

- Se **persist()** è invocato per una istanza Entity nello stato Removed, questa ritorna nello stato di Managed,
- Se **persist()** viene eseguita su una Detached Entity viene sollevata una **IllegalArgumentException** o si assiste al fallimento della transazione.

```

@PersistenceContext
EntityManager em;
...
public LineItem createLineItem(Order order, Product product) {
    LineItem li = new LineItem(order, product, quantity);
    order.getLineItems().add(li);
    em.persist(li);
    return li;
}

@OneToMany(cascade=ALL, mappedBy="order")
public Collection<LineItem> getLineItems() {
    return lineItems;
}

```

Managed entity possono essere rimosse tramite l'operazione **remove()** o attraverso un'operazione di rimozione in cascata da Entity correlate con **cascade=REMOVE** o **cascade=ALL**.

- Se **remove()** è invocato su New Entity allora l'operazione viene ignorata.
- Se **remove()** è invocato su Detached Entity viene lanciata **IllegalArgumentException** o avviene il fallimento del commit della transazione.
- Se **remove()** è invocato su Entity già in stato di Removed l'operazione viene ignorata.

I dati relativi alla Entity sono effettivamente rimossi dal DB solo a transazione completata o come risultato di una operazione esplicita di **flush**. Nell'esempio tutte le Entity oggetto associate con l'ordine sono rimosse perché nell'annotazione di relazione il campo cascade ha valore **cascade=ALL**.

```

public void removeOrder(Integer orderId) {
    try {
        Order order = em.find(Order.class, orderId);
        em.remove(order);
    }
}

```

Lo stato di entità persistenti è sincronizzato con il DB quando la transazione associata compie il commit. Se il cascading è attivato e una Managed entity è in una relazione bidirezionale, i dati sono resi persistenti sulla base del lato di possesso (ownership) della relazione. Per forzare la sincronizzazione con il DB è possibile invocare il metodo **flush()**.

## 6.2.7 Creazione di Query

Si utilizzano i metodi **createQuery()** e **createNamedQuery()** di EntityManager, costruendo query conformi a **Java Persistence query language**. Il metodo **createQuery()** permette la costruzione di query dinamiche, ovvero definite all'interno della business logic:

```

public List findWithName(String name) {
    return em.createQuery(
        "SELECT c FROM Customer c WHERE c.name LIKE :custName")
        .setParameter("custName", name)
        .setMaxResults(10)
        .getResultList();
}

```

Il metodo `createNamedQuery()` si utilizza invece per creare query statiche, ovvero definite a livello di annotation, utilizzando l'annotazione `@NamedQuery`.

```
@NamedQuery(  
    name="findAllCustomersWithName",  
    query="SELECT c FROM Customer c WHERE c.name LIKE :custName")  
  
@PersistenceContext  
public EntityManager em;  
...  
customers = em.createNamedQuery("findAllCustomersWithName")  
    .setParameter("custName", "Smith")  
    .getResultList();
```

I parametri con nome (**named parameters**) sono parametri di query preceduti da `(:)`.

Sono legati ad un valore dal metodo `javax.persistence.Query.setParameter(String name, Object value)`.

JPA definisce un proprio linguaggio di querying per le Entity con una sintassi simile a SQL. L'obiettivo di questo linguaggio è offrire la possibilità di scrivere query portabili e indipendenti dallo specifico datastore. Il linguaggio di query di JPA usa gli schemi (**schema**) ottenuti dalla definizione delle Entity persistenti (relazioni incluse) per il modello dei dati, e definisce operatori/espressioni basati su questo modello. Lo scope di una query include tutte le entity correlate all'interno della stessa unità di persistenza (**persistence unit**).

## 6.2.8 Unità di Persistenza

Un'unità di persistenza, **Persistence unit**, definisce l'insieme di tutte le classi Entity potenzialmente gestite da un EntityManager in una applicazione. In altre parole, rappresenta l'insieme dei dati, di interesse per quella applicazione, contenuto in un datastore singolo. Le unità di persistenza sono definite all'interno di un file XML **persistence.xml**, distribuito insieme al file EJB-JAR o WAR, a seconda dell'applicazione sviluppata e secondo diverse specifiche di naming.

```
<persistence>  
    <persistence-unit name="OrderManagement">  
        <description> Questa unità gestisce ordini e clienti</description>  
        <jta-data-source>jdbc/MyOrderDB</jta-data-source>  
        <jar-file>MyOrderApp.jar</jar-file>  
        <class>com.widgets.Order</class>  
        <class>com.widgets.Customer</class>  
    </persistence-unit>  
</persistence>
```

Ad esempio, il file sopra definisce un'unità di persistenza chiamata `OrderManagement`, che usa una sorgente dati `jdbc/MyOrderDB` che è JTAware (consapevole di JTA). I tag `jar-file` e `class` specificano le classi relative all'unità di persistenza: classi Entity, embeddable, e superclassi mapped. L'elemento `jta-data-source` specifica il nome JNDI globale della sorgente dati che deve essere utilizzata dal container.

## 6.2.9 Loading Lazy/Eager

Con JPA vi è la possibilità di controllare il caricamento dei dati di entità correlate ad una applicazione. Le strategie di caricamento che possono essere adottate sono di due tipi:

- **EAGER**: in cui le entità correlate sono caricate quando viene caricata l'entità “padre”, ad es. perché coinvolta in una query.

```
@OneToMany(cascade=ALL, mappedBy="owner", fetch=EAGER)
```

- **LAZY**: le entità correlate sono caricate solo quando si “naviga” effettivamente su di esse.

```
@OneToMany(cascade=ALL, mappedBy="owner", fetch=LAZY)
```

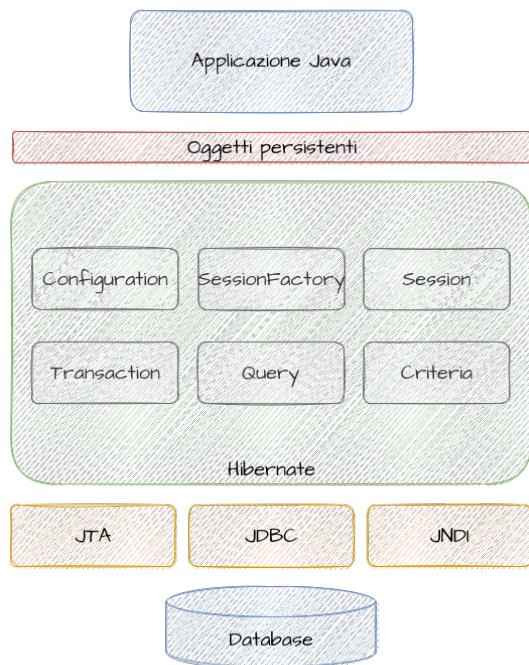
La linea guida per ottenere massime performance è quella di impostare **EAGER** su set di dati piuttosto ristretti, altrimenti utilizzare **LAZY**.

## 6.3 Hibernate

Hibernate è un framework **Object Relational Mapping** (ORM) per realizzare la persistenza di oggetti POJO. Hibernate permette la costruzione e l'utilizzo di oggetti persistenti seguendo gli usuali concetti di programmazione Object Oriented, mettendoli in relazione con i DB relazionali, senza vincoli derivanti dal modello orientato alle tabelle dei database. Infatti, è in grado di gestire il mismatch tra oggetti e relazioni.

Hibernate consente di effettuare operazioni di associazione, utilizzo e composizione, mantenendo l'ereditarietà e il polimorfismo del mondo a oggetti. Per rappresentare relazioni “multiple” utilizza le collection e garantisce elevate performance grazie all'**Object Caching**. Hibernate offre un supporto sofisticato al querying, attraverso **Criteria**, **Query By Example** (QBE), **Hibernate Query Language** (HQL) e SQL nativo.

L'architettura di Hibernate permette di astrarre dalle API JDBC/JTA sottostanti, il livello di applicazione può essere trasparente a questi dettagli.



La **SessionFactory** (`org.hibernate.SessionFactory`) è una classe factory per oggetti Session e clienti di **ConnectionProvider**, tipicamente ad ogni DB viene associata una factory dedicata. Questa classe si occupa di mantenere una cache di primo livello associata all'oggetto Session e utilizzata di default, mentre optionalmente può mantenere anche una cache di secondo livello, associata all'oggetto SessionFactory, con dati riusabili in diverse transazioni, anche a livello di cluster.

La **Session** (`org.hibernate.Session`) rappresenta un contesto di persistenza che è delimitato dall'inizio e dalla fine di una transazione logica associata ad essa. Inoltre, Session gestisce le operazioni del ciclo di vita degli oggetti persistenti come una cache, e opera come factory per oggetti **Transaction**.

Gli oggetti persistenti sono oggetti single-threaded che contengono stato persistente e logica di business, possono essere Java Bean o POJO, l'unico aspetto peculiare è che devono essere associati con uno e solo un oggetto Session. Tutte le modifiche effettuate sugli oggetti persistenti sono automaticamente riportate sulle tabelle DB (al loro commit). Appena si chiude la sessione, gli stessi oggetti diventano automaticamente “detached” e possono essere usati liberamente senza più vincoli dal DB. Gli oggetti transient o detached sono istanze non legate alla Session. Modificando gli oggetti non viene modificato il DB, eventuali modifiche al DB possono avvenire solo con operazioni **Persist** o **Merge** che li ricollegano al DB.

Le **Transaction** (`org.hibernate.Transaction`) sono oggetti single-thread che servono a specificare unità atomiche di lavoro astraendo dai dettagli delle librerie e dei framework sottostanti. Un oggetto Session può essere coinvolto in diverse Transaction, però la demarcazione delle transazioni deve comunque essere specificata esplicitamente.

### 6.3.1 Stati di oggetti Hibernate

Un'istanza di una classe persistente può assumere in ogni istante uno fra tre stati possibili, definiti all'interno di un contesto di persistenza:

- **Transient**: non appartenente a un contesto di persistenza.
- **Persistent**: appartenente a un contesto di persistenza.
- **Detached**: usualmente appartenente a un contesto di persistenza ma non attualmente.

Lo stato **Transient** non è mai associato a un contesto di persistenza, ovvero a una Session. Questo accade quando viene definita un'istanza POJO o Java Bean fuori da una Session, ciò implica che l'oggetto non ha identità di persistenza, cioè un valore per la primary key, e non ha righe corrispondenti nel DB.

Nello stato **Persistent** l'istanza è associata a una Session e il suo stato corrisponde a una riga del DB.

Un oggetto si trova nello stato **Detached** quando un'istanza che corrisponde a uno stato di persistenza viene staccata dalla Session stessa. L'istanza è stata associata a un contesto di persistenza in passato, ma quella sessione è stata chiusa oppure l'istanza è stata trasferita tramite serializzazione a un altro processo. L'oggetto ha identità di persistenza e forse una riga corrispondente in una tabella DB. Un oggetto si può trovare in uno stato Detached ad esempio quando deve essere inviato ad un altro processo, che lo utilizzerà senza necessità di avere un contesto di persistenza associato.

Le transizioni di stato sono sempre possibili:

- Oggetti Transient possono diventare Persistent tramite chiamate ai metodi `save()`, `persist()` o `saveOrUpdate()` dell'oggetto Session associato.
- Oggetti Persistent possono diventare transienti tramite l'invocazione di `delete()`. Ogni istanza di oggetto persistente restituita da `get()` o `load()` è Persistent.
- Oggetti Detached possono diventare persistenti tramite chiamate ai metodi `update()`, `saveOrUpdate()`, `lock()` o `replicate()`.
- Lo stato di una istanza Transient o Detached può transitare in Persistent anche tramite `merge()` e istanziazione di un nuovo oggetto persistente.

### 6.3.2 Hibernate Caching

In generale, il Caching in Hibernate migliora le prestazioni accedendo al database solo se lo stato necessario non è presente in cache. L'applicazione può avere bisogno di svuotare (invalidare) la cache se il DB viene aggiornato o se non è possibile sapere se la cache mantiene ancora copie aggiornate.

In Hibernate esistono due livelli di cache:

- **First level**: una cache di **primo livello** associata alla Session
- **Second level**: una cache di **secondo livello** associata alla SessionFactory, ovvero legata alla gestione del supporto al caching.

Le cache di primo livello hanno i confini della transazione e sono legate alla sessione, ottimizzano gli accessi, consentendo di effettuare meno query al DB, quindi, vi sono

solo statement iniziali e finali senza stati intermedi.

Le cache di secondo livello mantengono informazioni utilizzabili da diverse transazioni, principalmente mantengono oggetti persistenti disponibili per l'intera applicazione, non solo per l'utente che sta eseguendo le query e per il SessionBean associato.

Per l'invalidazione della cache viene prediletta una strategia ottimistica detta **Optimistic Concurrency Control** (OCC), si assume che la maggior parte delle transazioni verso il DB non siano in conflitto con le altre, questo permette di essere piuttosto “permissivi” nel rilasciare la possibilità di esecuzione, ottenendo un throughput molto alto.

Il versioning dei dati consente di effettuare dei check e controllare i dati mantenendo elevata scalabilità e concorrenza. Il **Version Checking** sfrutta numeri di versione o **timestamp** per fare la detection di aggiornamenti che possono determinare conflitti. Hibernate (e ora anche JPA) implementa il concetto di version-based OCC su ogni datastore, questo processo non è del tutto automatico. Una proprietà può essere marcata con l'annotazione **@Version**, questa operazione darà origine a una colonna speciale nel datastore.

```
@Entity  
@Table(name = "orders")  
public class Order {  
    @Id private long id;  
    @Version private int version;  
    private String description;  
    private String status;  
  
    ...  
}
```

Quando il gestore di persistenza “salva” questa entity nel datastore, la proprietà Version viene posta automaticamente al valore iniziale. Ogni volta che Hibernate aggiornerà l'entità verrà modificato in maniera automatica anche il valore della proprietà Version, la query effettivamente eseguita dal Persistence Manager sarà:

```
update orders set description=?, status=?, version=? where id=? and  
version=?
```

Ad esempio, se due utenti (Alice e Bob) caricano un ordine con versione 1, Alice decide di approvare l'ordine e lancia tale azione, di conseguenza lo stato è aggiornato sul datastore:

```
update orders set description=?, status=?, version=2 where id=? and  
version=1
```

Persistere l'aggiornamento del dato incrementa Version Counter a “2”. Nel frattempo, Bob, che possiede ancora la vecchia versione dei dati (version=1), lancia un update dell'ordine, la query eseguita risulta essere:

```
update orders set description=?, status=?, version=2 where id=? and  
version=1
```

Il risultato è che questo secondo update non fa match con alcuna riga (nessun match con clausola WHERE), Hibernate lancia un'eccezione `org.hibernate.StaleObjectStateException` (wrapped in `javax.persistence.OptimisticLockException`). Il risultato è che Bob non può eseguire l'update fino a che non ha effettuato il refresh della sua copia dei dati. Ovviamente serve una gestione opportuna delle eccezioni a livello applicativo.

### 6.3.3 Hibernate Fetching

Hibernate può adottare diverse strategie di **Fetching**, ovvero di caricamento dei dati dal DB alla memoria. Una strategia di Fetching determina come e quando i dati vengono effettivamente caricati dal DB per un'applicazione che usa gli oggetti di persistenza associati. La strategia di Fetching adottata ha ovviamente impatto sulle performance ottenibili dal sistema, e viene di solito dichiarata in un file di mapping o ne viene fatto l'override tramite specifiche query.

Le modalità di fetching possibili sono:

- **FetchMode.DEFAULT**: è la configurazione specificata nel mapping file.
- **FetchMode.JOIN**: Hibernate recupera i dati associati, anche collezioni, utilizzando un outer join all'interno della stessa select.
- **FetchMode.SELECT**: Hibernate effettua una seconda select separata per recuperare le entity o collection associate. Lazy fetching è il default per SELECT: la seconda select viene eseguita solo quando l'applicazione accede veramente ai dati associati.

### 6.3.4 Query By Examples (QBE)

Le **Query by Examples** sono ricerche di tipo associativo, lo stile è drasticamente differente da SQL per la ricerca dati nel DB. Per effettuare questo tipo di ricerche si va a popolare parzialmente un'istanza di un oggetto per permettere a Hibernate di costruire trasparentemente un criterio di scelta (criteria) usando l'istanza come esempio. Tale modalità solleva il programmatore dalla conoscenza del linguaggio SQL.

Ad esempio, se la classe `org.hibernate.criterion.Example` implementa l'interfaccia Criterion:

```
// cerca gli oggetti persona tramite un oggetto di esempio  
Criteria crit = sess.createCriteria(Person.class);  
Person person = new Person();  
person.setName("Shin");  
Example exampleRestriction = Example.create(person);  
crit.add(exampleRestriction);  
List results = crit.list();
```

## 7. Java Messaging Service (JMS), ESB e JBI

### 7.1 Sistemi di Messaging

L'importanza dei sistemi di messaging è dovuta alla possibilità di implementare una comunicazione disaccoppiata (loosely coupled) e asincrona, che, parafrasando nel rispetto delle definizioni della letteratura, si traduce in sincrono non bloccante. I messaggi sono lo strumento principale di comunicazione fra applicazioni, le quali utilizzano un modello a scambio di messaggi. Il software di supporto allo scambio di messaggi fornisce le funzionalità di base necessarie, per questo si parla di **Message Oriented Middleware** (MOM), **Messaging System**, **Messaging Server**, **Messaging Provider**, **JMS Provider**.

I vantaggi dei sistemi di messaging sono: l'indipendenza rispetto all'ambiente in cui si opera e alla locazione di rete, e la possibilità di operare in ambienti con un'elevata eterogeneità.

Nel modello client server, si assume che il cliente conosca la locazione del servitore ciò non è più necessario nei Messaging Systems.

Infatti, il sistema di messaggistica si occupa di smistare i messaggi verso il destinatario, ciò consente il completo disaccoppiamento spaziale e temporale. Dove per disaccoppiamento nello spazio si intende la mancata conoscenza della locazione del destinatario, mentre il disaccoppiamento nel tempo prevede la possibilità che mittente e destinatario non siano entrambi online al momento dell'invio o della ricezione di un messaggio.

Il modello a scambio di messaggi applicato all'architettura di un'applicazione distribuita di grandi dimensioni porta diversi vantaggi:

- **Scalabilità:** ovvero la capacità di gestire un numero elevato di clienti senza cambiamenti nella logica applicativa, nell'architettura e senza un elevato degrado nel throughput di sistema. Infatti, si tendono a incrementare le capacità hardware del sistema di messaging se si desidera una maggiore scalabilità complessiva.
- **Robustezza:** i consumatori, i produttori e la rete possono avere un fault senza causare un guasto all'intero sistema di messaging.

Il modello a scambio di messaggi permette di scalare le entità con precisione, ad esempio, nel caso dei **MOM** è possibile abbandonare l'architettura monolitica a favore dei micro-servizi, dividendo le applicazioni in varie funzionalità. Quindi il servizio di messaggistica supporta la scalabilità attraverso il disaccoppiamento. Inoltre, la robustezza dei MOM può garantire la transazionalità delle comunicazioni, e in caso di fault, i servizi non coinvolti possono continuare ad operare senza interruzioni.

Tra gli esempi notevoli di applicazioni che sfruttano i sistemi di messaging vi sono le transazioni commerciali utilizzate dalle carte di credito, i report con previsioni metereologiche, i workflow, la gestione di dispositivi di rete, la gestione di supply chain, il customer care, ma soprattutto sono utilizzati nelle architetture distribuite e cloud, a tutti i livelli, dai più bassi fino al livello applicativo. Le entità che effettuano lo scambio di messaggi sono dette **produttore** e **consumatore**, la comunicazione tra le due parti può seguire due diversi modelli di messaging: **point-to-point** e **publish-subscribe**.

Tutti i sistemi di messaging offrono i seguenti servizi:

- **Affidabilità:** operazioni con logica transazionale, ovvero trattano lo scambio di messaggi come transazione con la possibilità di persistere i messaggi.
- **Messaging distribuito**
- **Politiche di sicurezza**

I sistemi di messaging più sofisticati come i MOM possono supportare anche altre funzionalità, quali:

- **Qualità differenziata dei canali**
- **Transazioni sicure**
- **Auditing**
- **Load balancing**

### 7.1.1 Modello point-to-point

Nel modello **Point-to-Point** la comunicazione è un collegamento tra sole due entità. Questo modello viene utilizzato quando il produttore vuole contattare solo il proprio consumatore, ciò è particolarmente utile per i dispositivi mobili che hanno molte disconnessioni, appaiano e scompaiono online, e nei servizi, ovvero quando vi è la necessità di disaccoppiare molto le parti di un'applicazione, il MOM si comporta come un proxy mantenendo i messaggi.

Un messaggio è consumato da un singolo ricevente, ma possono esservi produttori multipli, la “destinazione” di un messaggio è una coda con nome detta **Named Queue**. Le code sono gestite con politica FIFO a parità di livello di priorità, tuttavia, i produttori possono inviare messaggi alla Named Queue specificando il livello di priorità desiderato. Questo ovviamente introduce tempi d’attesa, ma permette di gestire la priorità ed eventualmente il Quality of Service. Le code inoltre possono essere anche organizzate a tuple (argomenti) o in base al payload dei messaggi, con l’utilizzo di filtri per smistare i messaggi nelle varie Named Queue dei destinatari. Nell’ipotesi in cui il destinatario possa disconnettersi sarà necessario persistere i messaggi.

### 7.1.2 Modello publish-subscribe

Il modello **publish-subscribe** è un modello 1-N, in cui è presente un produttore e N consumatori. Il consumatore deve specificare al MOM l’interesse per una certa comunicazione. Il modello publish-subscribe è tipicamente utilizzato nelle bacheche dei social network in cui un messaggio è consumato da riceventi multipli.

La “destinazione” di un messaggio è un argomento con nome detto **Named Topic**, i produttori pubblicano su un topic, mentre i consumatori si “abbonano” a quel topic per ricevere i messaggi. In questa modalità non vi è persistenza, quindi i messaggi inviati quando un consumatore non era presente vengono persi.

### 7.1.3 Affidabilità dei messaggi

L'affidabilità nello scambio di messaggi da garanzie sulla consegna dei messaggi, sebbene, più la semantica di affidabilità è stringente più il throughput del sistema si abbassa. Inoltre, tutti i sistemi di messaging moderni supportano la persistenza dei messaggi, il che eleva ulteriormente il livello di affidabilità. JMS dà alcune garanzie nella consegna dei messaggi e permette al produttore di specificare il grado di affidabilità desiderato.

#### 7.1.4 Transazionalità dei messaggi

Nel modello a scambio di messaggi è possibile una produzione transazionale, il produttore può raggruppare una serie di messaggi in un'unica transazione, per cui o tutti i messaggi sono accodati con successo o non lo è nessuno. Nel consumo transazionale, invece, il consumatore riceve un gruppo di messaggi come serie di oggetti con proprietà transazionali, fino a quando tutti i messaggi non sono stati consegnati e ricevuti con successo i messaggi sono mantenuti permanentemente nella loro Queue o Topic. Per garantire transazionalità il MOM deve utilizzare un repository persistente.

Lo scope della transazionalità si può applicare sull'interazione tra consumatore del sistema MOM e il sistema MOM stesso, oppure tra produttore dei messaggi e MOM. Nei casi in cui si voglia una transazionalità forte si può applicare a tutto il percorso da produttore a consumatore. La terza opzione, tuttavia, è molto complessa e non viene garantita da molti MOM. Inoltre, il sistema di messaggistica può essere distribuito e ciò può rendere complicata la transazionalità. In pratica lo scope della transazionalità è di due tipi:

- **client-to-messaging system:** in cui le proprietà di transazionalità riguardano l'interazione fra ogni cliente e il sistema di messaging. Questo è lo scope supportato da JMS.
- **client-to-client:** in cui le proprietà di transazionalità riguardano l'insieme delle applicazioni produttore consumatore per quel gruppo di messaggi, questo non è supportato da JMS.

Come già detto anche il sistema di messaging può essere distribuito, infatti, sistemi di enterprise messaging possono realizzare una infrastruttura in cui i messaggi sono scambiati fra server nel distribuito, ciò complica ulteriormente la transazionalità.

#### 7.1.5 Sicurezza nello scambio di messaggi

Il MOM fornisce supporto alla sicurezza tramite i meccanismi di:

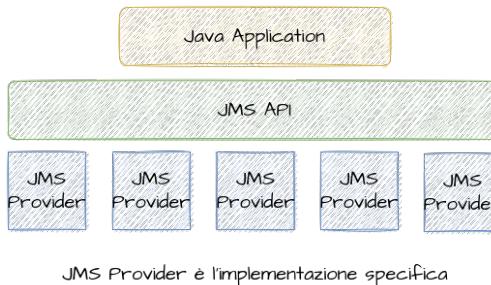
- **Autenticazione:** garantita dall'utilizzo di certificati.
- **Confidenzialità:** garantita dalla cifratura del payload dei messaggi.
- **Integrità:** garantita dall'utilizzo di digest (impronta) dei messaggi.

La sicurezza e la sua gestione sono dipendenti dal vendor del sistema di messaging. JMS non offre servizio diretto di sicurezza, ma esistono API che consentono di implementare varie politiche, JMS prevede unicamente la definizione del servizio.

### 7.2 Java Messaging System (JMS)

JMS è un insieme di interfacce Java, con associata definizione di semantica, che specificano

come un cliente JMS possa accedere alle funzionalità di un sistema di messaging generico.



JMS fornisce il supporto alla produzione, distribuzione e consegna di messaggi, inoltre, garantisce il supporto alle diverse semantiche per message delivery, sincrone o asincrone, bloccanti o non-bloccanti, con proprietà transazionali, ed assicura la realizzazione di comunicazioni con modello Point-to-Point (reliable queue) e Publish-Subscribe con selettori di messaggio lato ricevente, e cinque tipologie di messaggi possibili.

JMS è un supporto che fornisce interfacce generiche, non è una specifica. JMS è parte della piattaforma J2EE, ma non necessita di un EJB container per essere utilizzato, è solo fortemente integrato con esso. Gli obiettivi di JMS sono:

- Avere dei JMS provider generici che possano lavorare con sistemi di messaggistica preesistenti.
- Avere consistenza con le API dei sistemi di messaging esistenti.
- Indipendenza dal vendor del sistema di messaging.
- Copertura della maggior parte delle funzionalità comuni nei sistemi di messaging.
- Promozione della tecnologia Java per sistemi messaging.

Gli attori messi in gioco nell'architettura JMS sono:

- Clienti JMS e non-JMS.
- Messaggi.
- Provider JMS (sistema di messaging dipendenti dallo specifico vendor).
- Gli oggetti **Destination** e **ConnectionFactory** recuperabili tramite JNDI.

### 7.2.1 Tipi di comunicazioni

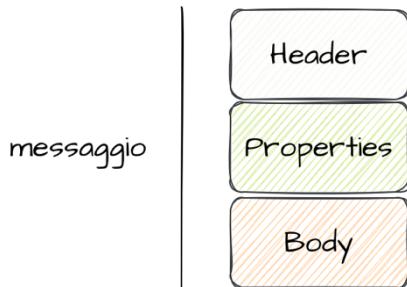
Nella comunicazione Point-to-Point i messaggi in una Queue possono essere persistenti o non persistenti. Nella comunicazione publish-subscribe, i messaggi non durevoli sono disponibili solo durante l'intervallo di tempo in cui il ricevente è attivo, se il ricevente non è connesso la semantica consente la perdita di ogni messaggio prodotto in sua assenza. I messaggi durevoli, invece, sono mantenuti dal sistema, che fa le veci dei riceventi non connessi al tempo della produzione dei messaggi, il ricevente non perde mai messaggi quando disconnesso.

### 7.2.2 Formato del messaggio

JMS definisce i formati dei messaggi e i payload possibili. I messaggi sono una modalità di comunicazione disaccoppiata fra le applicazioni. I veri formati che attualmente sono utilizzati

per l'encoding dei messaggi sono fortemente dipendenti dal vendor del sistema di messaging. Un sistema di messaging può interoperate completamente solo al suo interno, JMS fornisce quindi solo un modello astratto e unificato per la rappresentazione interoperabile dei messaggi attraverso le sue interfacce. I singoli vendor personalizzano i formati, che quindi sono fortemente dipendenti da essi, questo comporta una perdita di interoperabilità dovuta al fatto che Java lascia libertà nella definizione dei protocolli.

Il formato di un messaggio JMS è il seguente:



- **Header:** utilizzato per l'identificazione del messaggio e il suo routing, include la destination, la modalità di consegna (persistente, non persistente), il timestamp, la priorità, e il campo ReplyTo che serve al ricevente per rispondere al messaggio.
- **Properties:** JMS permette di aggiungere nuove features, dette proprietà dei messaggi. Queste sono coppie nome-valore personalizzate dai vendor e possono essere di vario tipo: campi application-specific, campi dipendenti da un sistema di messaging e specifici di un particolare sistema di messaging, oppure campi opzionali. Le proprietà di JMS sono: JMSDestination, JMSDeliveryMode (persistente o no), JMSMessageID, JMSTimeStamp, JMSRedelivered, JMSExpiration, JMSPriority, JMSCorrelationID, JMSReplyTo (destinazione fornita dal produttore, specifica dove inviare la risposta), JMSType (tipo del corpo del messaggio).

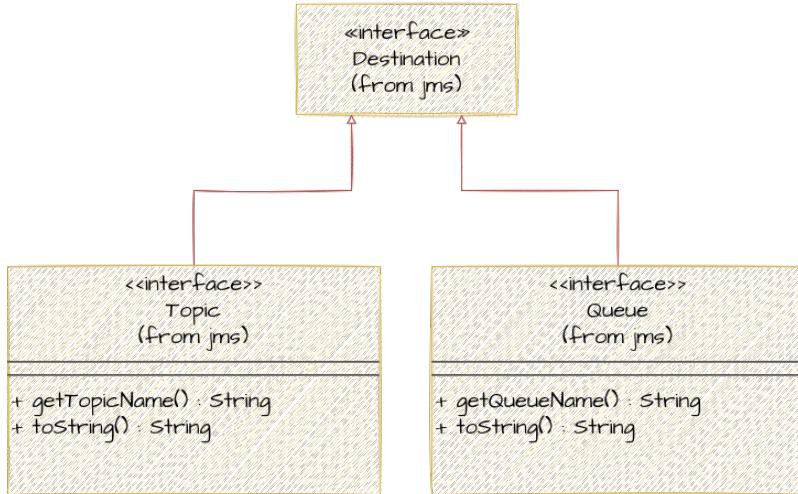
L'idea di dividere l'header dalla proprietà permette al MOM una visione più o meno granulare del messaggio, si possono guardare le proprietà, senza aprire il payload del messaggio.

- **Body:** rappresenta il payload, ovvero il contenuto, del messaggio, supporta diversi tipi di contenuto, ogni tipo è definito da una interfaccia locale utilizzata per interrogare il payload:
  - **StreamMessage:** StreamMessage contiene valori primitivi e supporta la lettura sequenziale.
  - **MapMessage:** contiene coppie nome-valore e supporta lettura sequenziale o by name.
  - **TextMessage**
  - **ObjectMessage**
  - **BytesMessage:** contiene byte “non interpretati” e viene utilizzato di solito per fare match con formati preesistenti.

### 7.2.3 Interfacce

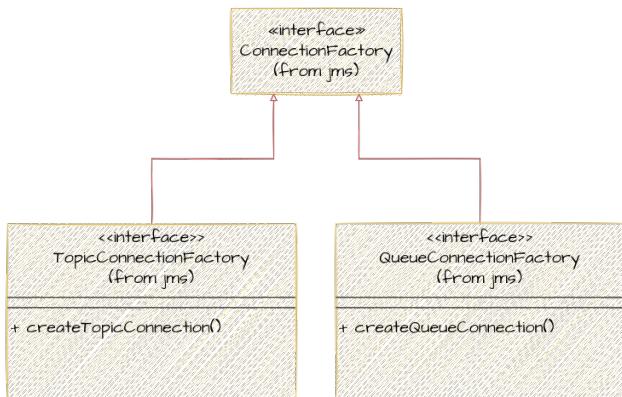
#### Destination

L'interfaccia Destination rappresenta l'astrazione di un Topic o di una Queue, non di un ricevitore di messaggi. Le interfacce Queue e Topic sono figlie dell'interfaccia Destination, e sono rispettivamente astrazioni di una destinazione point-to-point o pub-sub.



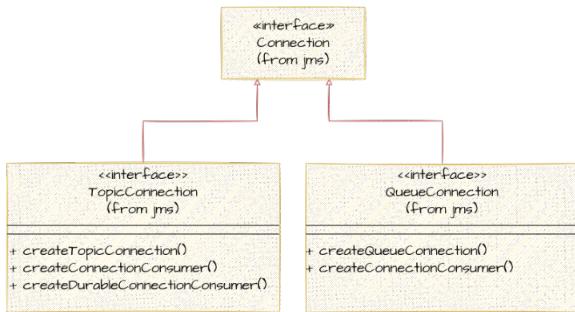
## ConnectionFactory

L'interfaccia ConnectionFactory viene implementata a partire dalla classe Factory per creare una connessione provider-specific verso il server JMS, è simile al gestore di driver (`java.sql.DriverManager`) in JDBC. Le interfacce figlie sono `QueueConnectionFactory` e `TopicConnectionFactory`.



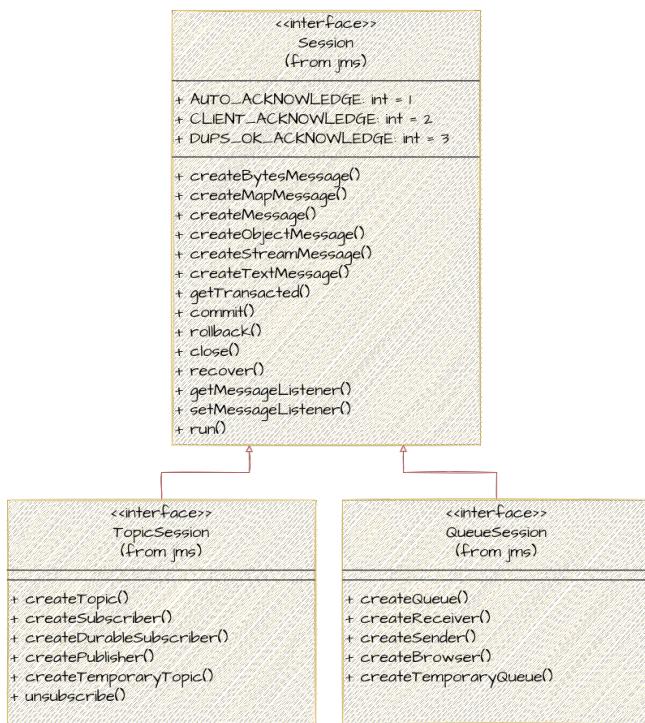
## Connection

Interfaccia Connection è un'astrazione che rappresenta un singolo canale di comunicazione verso il provider JMS. La connessione viene creata da un oggetto ConnectionFactory e quando si è terminato di utilizzare la risorsa la connessione viene chiusa.



## Session

L’interfaccia Session è creata da un oggetto Connection. Una volta connessi al JMS provider attraverso una Connection, tutte le operazioni si svolgono nel contesto di una Session attiva, ogni sessione è single-threaded, ovvero ogni operazione di invio e ricezione di messaggio avviene in modo serializzato. Le sessioni realizzano un contesto “limitato” con “proprietà transazionali”.



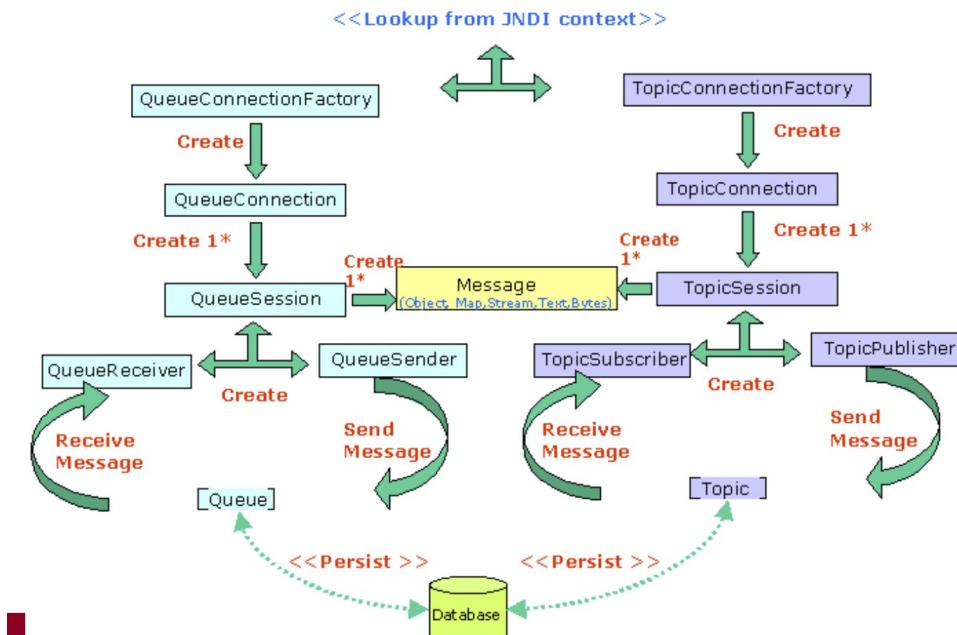
## Message Consumer e Message Producer

Le interfacce Message Consumer e Message Producer sono utilizzate dalla Session. Per inviare un messaggio verso una Destination, il cliente deve richiedere esplicitamente all’oggetto Session di creare un oggetto **MessageProducer**. Analogamente, i clienti che vogliono ricevere messaggi creano un oggetto **MessageConsumer** (collegato ad un oggetto Destination) attraverso la Session.

Sono possibili due modalità di ricezione dei messaggi:

- **Blocking:** La modalità blocking utilizza il metodo bloccante `receive()`.
- **Non-blocking:** Nella modalità non blocking il cliente registra un oggetto `MessageListener` presso una `Destination`, quando un messaggio è disponibile, il provider JMS richiama a callback il metodo `onMessage()` di `MessageListener`.

Il grafico sottostante mostra, dall'alto verso il basso, tutte le astrazioni per le due parti 1-1 e 1-N, sono presenti diverse tipologie di messaggi dovuti alle diverse persistenze:



#### 7.2.4 Affidabilità dei messaggi

JMS offre diversi livelli di affidabilità dei messaggi. Il livello più alto di affidabilità si ottiene quando si ha persistenza dei messaggi che è realizzabile con l'utilizzo di transazioni, oppure con la subscription durevole a un certo Topic o con la ricezione da Queue garantendo la persistenza del messaggio se il ricevente si è assentato per un certo tempo.

A livello base (Basic Reliability) l'affidabilità è garantita grazie all'uso di messaggi **ACK**, l'utilizzo di **messaggi persistenti**, la possibilità di definire dei **time to live**, la possibilità di configurare dei livelli di priorità e la possibilità di consentire l'**expiration dei messaggi**.

Il livello più avanzato (Advanced Reliability) garantisce l'affidabilità grazie all'uso di **abbonamenti durevoli** e l'utilizzo di **transazioni locali**, ovvero transazioni che non possono essere garantite in tutto il percorso end-to-end, ma solo tra consumatore e provider e/o tra provider e produttore.

#### ACK (ACKnowledgement)

La prima operazione eseguita dopo la ricezione di un messaggio è il processamento. Una volta eseguita la fase di processamento del messaggio, se necessario, avviene uno scambio di **ACK** tra il consumatore e il produttore. Lo scambio dell'ACK può avvenire con modalità diverse.

Nel caso di sessioni con transazionalità, l'ACK viene inviato in maniera automatica al commitment, nel caso in cui qualcosa sia andato storto si esegue l'operazione di rollback e quindi il rinvio di tutti i messaggi. In sessioni senza transazionalità il numero di ACK scambiati è variabile e dipende dal valore specificato come parametro nel metodo `createSession()`.

Sostanzialmente esistono tre tipi di ACK:

- **Auto acknowledgment:** ACK generato in maniera automatica dopo il ritorno con successo dei metodi `MessageConsumer.receive()` o `MessageListener.onMessage()`.
- **Client acknowledgment:** il cliente a livello applicativo si fa carico di inviare l'ACK con la chiamata al metodo `acknowledgement()`, questo tipo di ACK è cumulativo, cioè conferma tutti i messaggi inviati nell'intervallo che è passato dall'ultimo ACK a quello corrente.
- **Lazy acknowledgment:** JMS invia saltuariamente l'ACK in modo cumulativo. Questo approccio ha il minimo overhead per JMS, consente la presenza di messaggi duplicati.

Tutti i tipi di messaggi ACK hanno la possibilità di essere duplicati e quindi ritrasmessi.

Con la modalità **AUTO-ACK** vi sono differenze tra caso con Persistent e non persistent. Nel caso Persistent, supponendo il non fallimento dello storage dove sono salvati i messaggi, è possibile la duplicazione del messaggio, ciò consente che nel caso in cui un server fallisca, quando questo torna in modalità up and running, si renda conto del mancato invio di un ACK e possa rimandarlo al consumer; questo è possibile grazie allo storage persistente.

Nel caso **CLIENT-ACK** possono essere presenti duplicati, perché per situazioni simili alla precedente dove avvengono più ritrasmissioni, più messaggi possono essere stati persi a causa della politica cumulativa.

Con la modalità **LAZY-ACK** vi è duplicazione, ma i messaggi da rinviare potrebbero essere un numero maggiore rispetto alle modalità precedenti, poiché la decisione di mandare ACK è presa dal supporto in maniera arbitraria.

In generale è preferibile che le applicazioni siano idempotenti, cioè, che implementino la ritrasmissione.

### ACK in produzione

JMS prevede anche la possibilità di ACK in produzione, con modalità più semplici e gradi di libertà più limitati.

Gli ACK in produzione possono avere una semantica bloccante per la `send()`, localmente al produttore, che risulta in generale essere più semplice della `receive()`.

Il client manda il messaggio, il server lo persiste e manda l'ACK solo a seguito di questa operazione, a questo punto la **Publish** ritorna, il tutto avviene con molta più sincronicità.

Anche in questo caso vi è la ritrasmissione dei messaggi.

### Persistenza

JMS prevede due diversi modi di gestione della persistenza che si differenziano per la modalità di consegna.

- Modalità **Persistent:** richiede la persistenza, quindi il provider ha la responsabilità di non perdere il messaggio e questo è possibile grazie allo storage.

- Modalità **Non-persistent**: non dà garanzie rispetto al fault, ma non subisce il collo di bottiglia generato dallo storage, quindi, è possibile sostenere un high-rate nell'invio con migliori performance.

Vi è un trade-off tra il numero di ACK inviati, l'overhead che essi introducono e il livello generale di affidabilità che si vuole raggiungere. Quanto coinvolgere la parte applicativa nella gestione degli ACK è una scelta: coinvolgere molto il client nella gestione degli ACK può aiutare a diminuire l'overhead, a costo però di una maggiore complessità a livello applicativo.

### Priorità e TimeToLive

La priorità è un attributo che fa parte dell'**header** del messaggio (JMSPriority), visibile a livello di API, quindi rappresenta una parte funzionale che tutti i vendor devono trattare, questa è una decisione forte a livello di supporto presa al momento della progettazione.

La **priorità** è configurabile sia a livello di produttore del messaggio sia per il singolo messaggio, allo stesso modo funziona anche il **TimeToLive**, la priorità ha una scala di importanza da 0 a 9 e a default è impostata a 4, per il TTL bisogna invece impostare il valore in secondi. La priorità si imposta con il metodo `setPriority()` e il TTL si imposta con `setTimeToLive()` dell'interfaccia MessageProducer.

La configurazione dei livelli di affidabilità è spesso determinata da scelte di default o prese alla creazione della Destination.

Per la **Basic Reliability**:

- Persistenza scelta a livello di singolo messaggio (es. interfaccia MessageProducer).
- Controllo degli ACK scelta a livello di sessione, interfaccia Session.
- Livelli di priorità scelti a livello di singolo messaggio (es. interfaccia MessageProducer).
- Expiration time scelto a livello di singolo messaggio (es. interfaccia MessageProducer).

Per la **Advanced Reliability**:

- Sottoscrizione durevole scelto a livello di sessione, interfaccia Session;
- Transazionalità scelto a livello di sessione, interfaccia Session.

### Durable Subscription

Un **Durable Subscriber** si registra specificando un'identità univoca, perché un subscriber durevole potrebbe non essere sempre presente, quindi è necessario un naming durevole per ricondurre sempre i messaggi allo stesso subscriber. Se un Durable Subscriber non ha clienti attivi il provider JMS mantiene questi messaggi fino a quando non vengono effettivamente consegnati, oppure non avviene l'**expiration**. All'interno di una singola applicazione, una sola session può avere Durable Subscription a un determinato named-topic in un determinato istante.

## 7.2.5 Gestione delle transazioni di JMS

Lo scope delle transazioni in JMS è solo fra clienti e sistema di messaging, non fra produttori e consumatori. Quindi un gruppo di messaggi, all'interno di una singola transazione, è consegnato come un'unica unità lato produttore e in una singola transazione è ricevuto come un'unica unità lato consumatore.

Le Transazioni possono essere gestite localmente e sono controllate dall'oggetto Session. La transazione inizia implicitamente quando l'oggetto di sessione è creato e termina all'invocazione di **Session.commit()** o **Session.abort()**. La sessione è transazionale se si specifica il flag appropriato all'atto della creazione. Ad esempio:

**QueueConnection.createQueueSession(true, ...).**

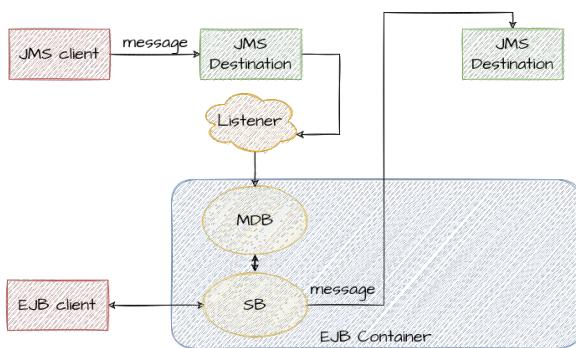
Eventualmente le transazioni possono essere anche gestite in modo distribuito, in tal caso devono essere coordinate da un transaction manager esterno, ovvero attraverso l'uso delle Java Transactions API (JTA). Le applicazioni possono controllare la transazione attraverso metodi JTA, tuttavia l'utilizzo di **Session.commit()** e **Session.rollback()** non è consentito. In questo modo, con l'utilizzo di JTA, le operazioni di messaging possono essere combinate con transazioni salvate su DB in una singola transazione complessiva.

## 7.2.6 Selettori di messaggi

I selettori sono filtri la cui logica di filtraggio è specificabile con stringhe SQL-like (SQL92) che possono lavorare sui messaggi in arrivo per estrarne solo alcuni. Lato receiver, le applicazioni JMS possono utilizzare selettori per scegliere i soli messaggi che sono potenzialmente di loro interesse, la selezione non può essere fatta sul contenuto dei messaggi, ma solo sulle proprietà e sull'header. I selettori consentono una gestione più semplice diminuendo l'overhead del provider e del supporto. Il fatto che non sia content base è dovuto alla retrocompatibilità con i precedenti MOM.

## 7.2.7 Java Messaging System JMS e Message Driven Bean MDB

I Message Driven Bean vengono istanziati e prelevati da un poll di istanze quando il messaggio viene ricevuto, il Java Messaging System provider invia i messaggi alle istanze MDB che si sono registrate per la ricezione. Il tutto avviene in maniera asincrona, con l'idea che ci sia un produttore che immette i messaggi in una Queue o in un Topic, con una semantica uno a molti, e un Listener che lo recapita al Message Driven Bean tramite un metodo di callback, a questo punto il MDB interagendo, eventualmente, con dei session bean può realizzare la logica applicativa.



## 7.3 Enterprise Service Bus (ESB)

Il tema principale su cui si soffermano gli **Enterprise Service Bus** è l'integrazione di sistemi di grandi dimensioni. In particolare, la possibilità di far coesistere parti di sistemi

legacy e preesistenti con parti sviluppate di recente. In questa direzione si sono sviluppati gli **Enterprise Service Bus**, infrastrutture molto larghe con principi di integrazione condivisi.

L'idea è quella di avere disaccoppiamento forte e quindi un Message Oriented Middleware a supporto della comunicazione, di utilizzare **Service Oriented Architecture (SOA)** con l'obiettivo di avere servizi che lavorano molto e comunicano solo quando vi sono riposte complete, quindi poco frequentemente.

Con ESB è possibile coniugare sistemi loosely coupled con servizi a grana grossa. La parola Bus significa facilitare la presentazione ovvero collegare dei servizi molto eterogenei attraverso una funzionalità di transformation e routing intelligence. Questo servizio garantisce anche la standardizzazione, è un middleware che disaccoppia molto la comunicazione. ESB utilizza un modello asincrono e pub-sub con intermediario.

Concretamente l'Enterprise Service Bus offre molte più funzionalità di un Message Oriented Middleware, garantisce lo scambio di messaggi che porta a un'elevata asincronicità con conseguente disaccoppiamento, oltre a collegare servizi molto eterogenei. Le principali caratteristiche sono:

- Disaccoppiamento
- Gestione dei “topic”
- Controllo degli accessi
- Struttura messaggi
- QoS configurabile

### 7.3.1 Service Oriented Architecture SOA

Una **Service Oriented Architecture** mette a disposizione:

- Servizi a grana grossa autonomi cioè non hanno bisogno di servizi esterni.
- Interfacce astratte ben fatte che definiscono contratti tra Consumer e Provider.
- Scambio di messaggi che compongono le operazioni invocabili sui servizi di input e output.
- Registri dei servizi con naming e trading.
- Possibilità di comporre i servizi in processi di business, ovvero molti servizi messi insieme seguendo le linee guide SOA.

Tutto questo ha l'obiettivo di ottenere accoppiamento debole, e quindi flessibilità di business, interoperabilità tra le applicazioni, indipendenza rispetto alle tecnologie di implementazione, grazie alle interfacce molto astratte. Un esempio è Azure che espone API con interfacce REST.

### 7.3.2 Web Services

I Web Services sono infrastrutture che soddisfano le caratteristiche SOA per l'interazione tra applicazioni, basati sul concetto di “servizio”, utilizzano interfacce web, e in particolare operano sulla porta “80”. I Web Services sfruttano essenzialmente tre tecnologie:

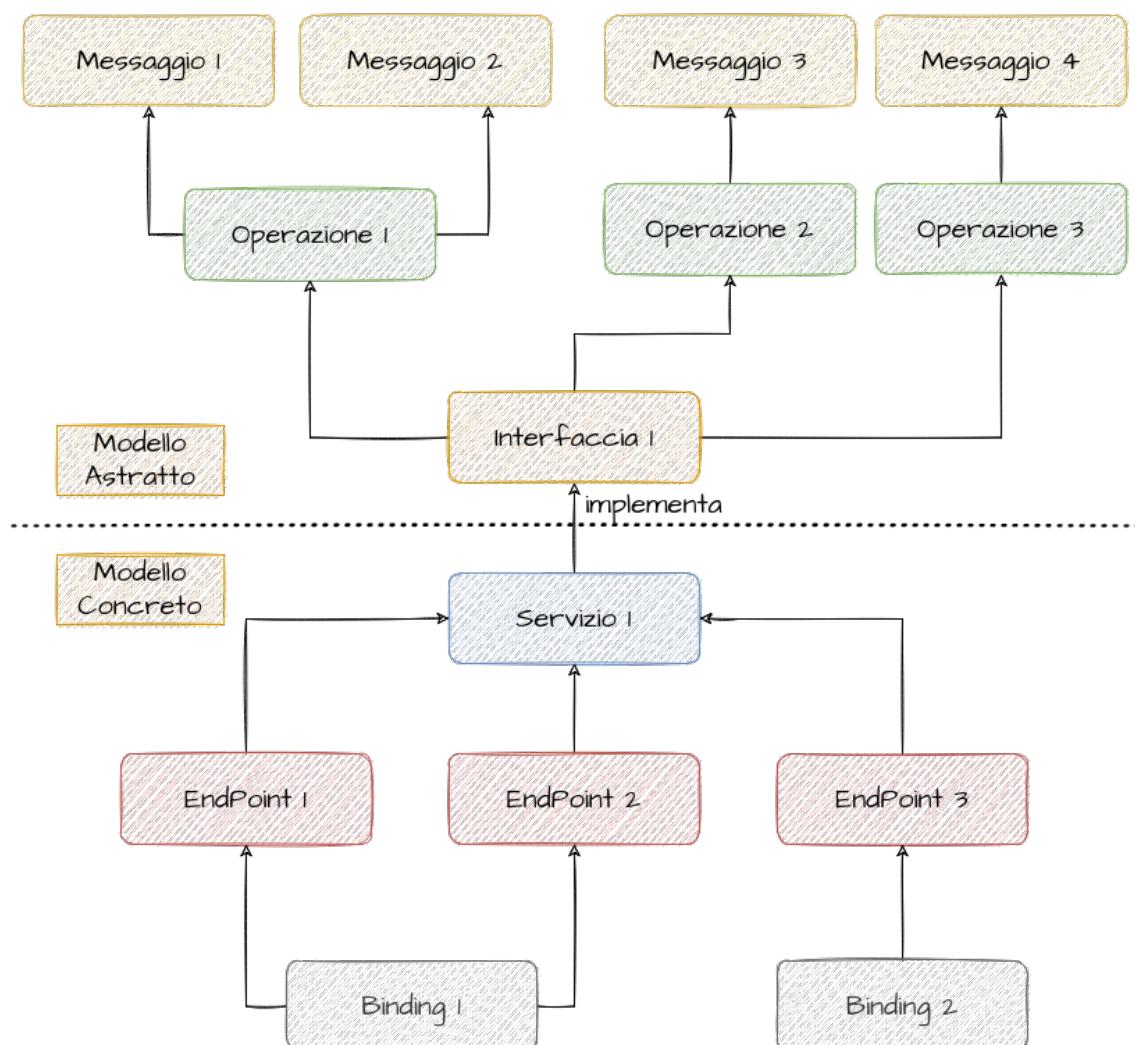
- **WSDL:** è un file contenente la descrizione del web service, scritto in XML contiene le funzionalità offerte da un servizio.

- **SOAP** è la descrizione di un protocollo per lo scambio di messaggi. I messaggi scambiati prendono il nome di “buste SOAP”, contengono file XML (indipendenti dal linguaggio) con diversi contenuti. Le buste SOAP possono essere trasmesse con http per garantire massima interoperabilità.
- **UDDI** è un servizio di trading che consente il discovery dei servizi, restituisce un file WSDL (file XML) che contiene la descrizione di quel servizio, la descrizione degli input e output del servizio e come collegarsi con esso.

La differenza tra un Web Service un servizio di Distributed Object Computing è la standardizzazione e la modalità di realizzazione dei servizi, nel caso dei Web Service non c'è dipendenza dall'implementazione.

WSDL (Web Service Description Language) è suddiviso in due parti, vi è una chiara separazione fra livello astratto (interfaccia) che contiene la definizione di operazioni di servizio come ingresso e uscita di documenti e struttura dei messaggi, e il livello concreto che implementa l'interfaccia, il quale presenta una serie di endpoint con indirizzo di rete e protocollo per definire i servizi (binding - per ogni interfaccia), tipico di tutte le soluzioni SOA.

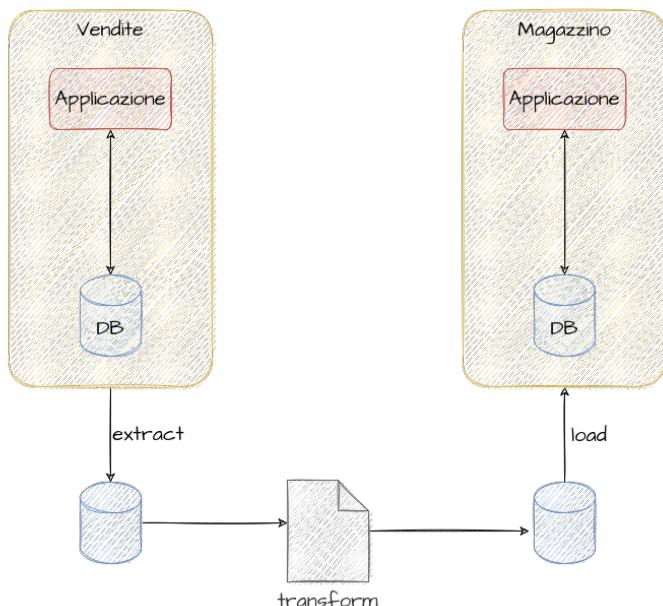
Nell'architettura SOA non è rilevante la tecnologia utilizzata si possono esporre più modi di comunicazione.



### 7.3.3 Enterprise Application Integration EAI

L'integrazione è un problema importante, solo 10% delle applicazioni è integrato (dati Gartner Inc.) e solo 15% di queste sfruttano middleware ad-hoc. Le tecnologie passate si sono rivelate inadeguate a causa di un'architettura "casuale", che è il risultato della composizione di diverse soluzioni adottate per i diversi sistemi nel corso degli anni. Tali circostanze comportano alti costi di mantenimento, rigidità (applicazioni tightly-coupled), prestazioni insoddisfacenti e scarsa scalabilità. Sono poche le applicazioni che nascono con la volontà di un'integrazione forte.

L'**Enterprise Application Integration** si occupa dell'integrazione di queste applicazioni, costruendovi intorno degli strumenti che le integrino.



Enterprise Application Integration tramite operazioni **Extract**, **Transform** e **Load** (ETL) permette di integrare applicazioni legacy. Si passa quindi per processi batch, tra un applicativo e l'altro vengono trasmessi i dati con FTP, i dati vengono trasformati in un formato utile per la seconda applicazione e vengono passate a questa con FTP e batch (questo è il funzionamento di molti servizi bancari). Tale soluzione ha alte latenze al momento delle batch, inoltre possono esservi fault al momento delle trasform. ESB vuole andare oltre rispetto a questa soluzione cercando di automatizzare il più possibile il processo di trasmissione e trasformazione, rendendolo il più possibile pulito.

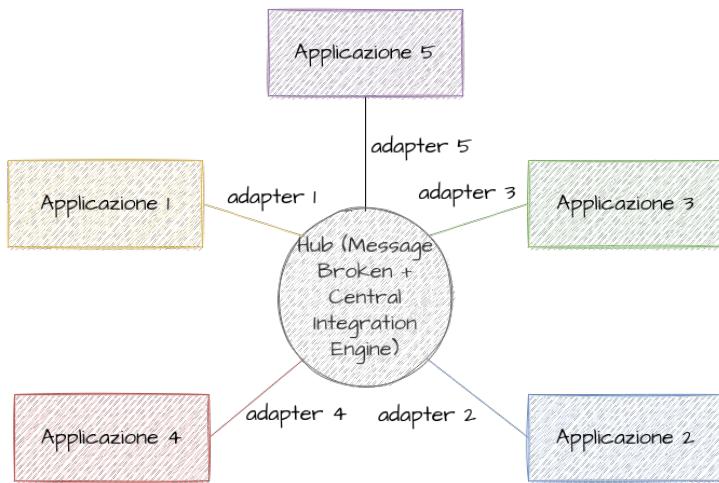
La soluzione proposta consiste nell'avere dei **Broker** e degli **Orchestration Engine** come facilitatori della comunicazione, non più provider ma entità intelligenti tra le due applicazioni. Le due architetture di riferimento sono **hub-and-spoke** e **bus**.

#### Hub-and-spoke

Hub-and-spoke è un'architettura a stella, in cui l'hub è il nodo centrale, che comunica con le applicazioni che hanno un **adapter** per un formato comune.

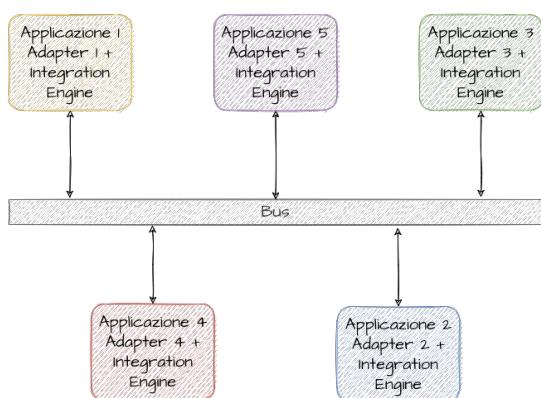
Un **Central Automation Engine**, un motore che consuma i messaggi, li analizza internamente

ed esegue il routing intelligente di essi tra le applicazioni, inoltrando i messaggi al destinatario corretto. Inoltre, vi sono tutti i servizi di sistema, tra cui persistenza e transazioni.



### Bus di interconnessione

L'altra architettura di riferimento è quella a bus, l'**Automation Engine** risiede nelle applicazioni, questo approccio permette di evitare il collo di bottiglia costituito dall'hub dell'architettura a stella. Il servizio a bus è un servizio punto a punto.



Tra vantaggi dell'architettura a stella vi è la facilità di gestione (centralizzata), ma ciò costituisce anche il principale svantaggio; infatti, l'hub è punto critico di centralizzazione che può rappresentare un collo di bottiglia, inoltre, ha anche ridotta scalabilità. L'architettura a bus ha la maggiore scalabilità, poiché è meno centralizzata, ma di contro è maggiore la difficoltà di gestione.

### Enterprise Service Bus

Enterprise Service Bus è la realizzazione della seconda architettura (Bus di interconnessione). L'idea principale è quella di:

- Avere una sorta di “lingua franca” con cui poter far comunicare i servizi.

- Orchestrare l'integrazione, rendendo ogni servizio punto centralizzato della gestione di tutti i servizi che si affacciano sull'enterprise service bus stesso.
- Registrare nuovi servizi.

Sul bus dell'ESB si vuole avere una lingua franca per comporre Web Services ovvero fare orchestrazione di web service. In questo senso l'adapter traduce la richiesta a un web service in una richiesta FTP per l'applicazione che non è un Web Service, mentre invece se l'applicazione è già un Web Service non c'è bisogno dell'adapter.

Orchestrazione ESB concetti chiave. L'orchestrazione avviene per le interazioni tra i servizi a grana grossa e per il routing intelligente. ESB tratta sia l'aspetto di descrizione astratta del servizio e sia la realizzazione concreta dei diversi binding. Consente di utilizzare tanti e diversi protocolli per scambiare l'informazione, senza imporre un protocollo unico. La descrizione astratta delle informazioni scambiate avviene nella parte astratta del WSDL, mentre con un biding concreto vengono descritte le interazioni con il mondo esterno con la parte concreta del WSDL, e ESB fa da orchestratore nel mezzo risparmiando al programmatore di implementare l'integrazione. L'orchestrazione è sia astratta che concreta specialmente nei routing intelligenti e nei vari servizi come auditing e logging.

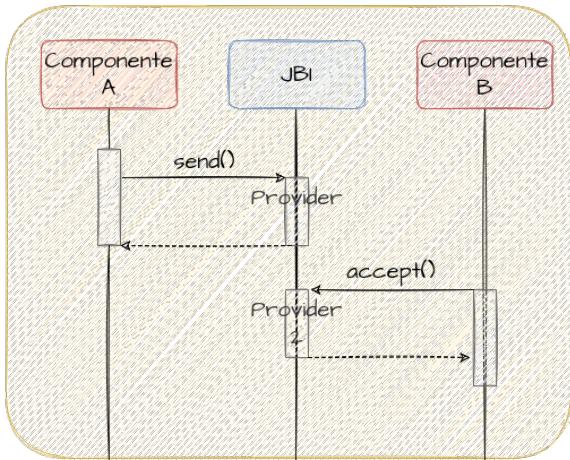
ESB è un'architettura altamente distribuita con integrazione basata su standard, mette a disposizione servizi di orchestrazione, mantiene l'autonomia delle singole applicazioni, consente un real-time throughput, mette in atto servizi di auditing e logging, consente l'adozione incrementale.

Nell'invocazione dei servizi, invece rende i servizi completamente disaccoppiati, utilizza pattern "find-bindInvoke" gestito automaticamente dall'infrastruttura, il progettista deve solo definire l'itinerario logico che i messaggi devono seguire, mentre i servizi si "limitano" a inviare e ricevere messaggi.

## 7.4 Java Business Integration

Nel mondo Java a supporto di tutto questo è presente JBI (**Java Business Integration**). In JBI vi sono servizi interni ed esterni, i servizi esterni non riescono a parlare direttamente con l'orchestratore per questi servizi ci sono dei biding component che fungono da proxy verso i servizi esterni, cioè sono degli adattatori per passare dal protocollo A al B.

I servizi interni offrono la possibilità di utilizzare il **Normalized Message Router** che si occupa dell'interazione tra componenti, nel routing delle informazioni. Sopra abbiamo la parte di servizi offerta dal framework tra cui l'orchestrazione dei servizi stessi offerta da BPEL che orchestra i servizi a grana grossa, con diagrammi di flusso che gestiscono le integrazioni. Sono offerti anche servizi grafici al programmatore per l'integrazione più facile dei servizi. La comunicazione tra componenti all'interno del bus non è diretta. Normalized Message Router, NMR, agisce da mediatore fra i vari componenti, ha il compito di fare routing dei messaggi tra due o più componenti, è distribuito e quindi consente di disaccoppiare Service Consumer da Service Provider garantendo un basso accoppiamento tra i componenti JBI. I messaggi sono scambiati in formato XML per cui la comunicazione è "technology-neutral" tra endpoint. Normalized Message scambiati sono definiti in formato indipendente e neutrale da qualsiasi specifica applicazione, tecnologia o protocollo di comunicazione, le trasformazioni di formato sono trasformazioni XSLT.

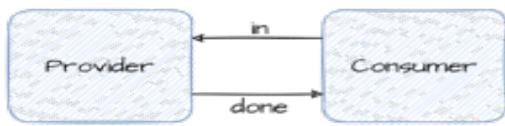


Per rispondere a ciascuna richiesta JBI è in grado di capire quale sia il provider migliore per il componente B, a quel punto B accetta il messaggio e continua l'interazione. Da notare come A non conosca B si è solo registrato a JBI.

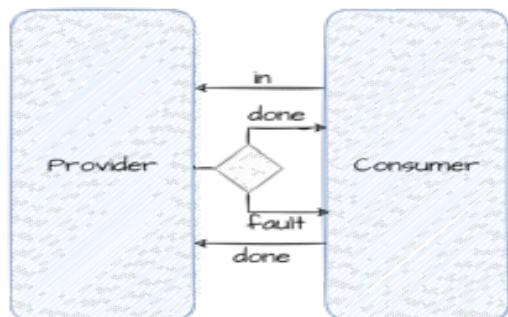
Componenti SOA e modello a scambio di messaggi basato su interposizione permettono un elevato grado di disaccoppiamento tra componenti con la possibilità di operare su messaggi (trasformazioni) in modo trasparente.

JBI supporta quattro possibili pattern di scambio messaggi:

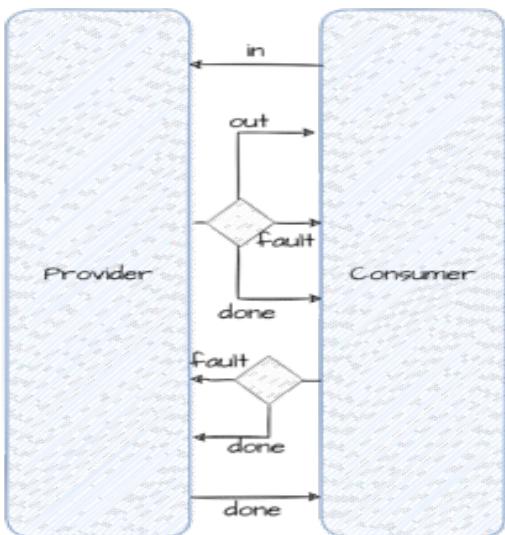
- **In-Only** per interazione/trasferimenti one-way, spesso molte integrazioni sono soddisfabili con questo pattern, prevede una conferma che tutto sia andato a buon fine.
- **Robust In-Only** per possibilità di segnalare fault a livello applicativo, simile a un handshake a tre vie con la conferma che le cose siano andate bene nelle due parti semantiche per gli errori o danni.
- **In-Out** per interazione request-response con possibilità fault lato provider ha una sola conferma da parte del consumatore.
- **In Optional-Out** per provider con risposta opzionale e possibilità di segnalare fault da provider/consumer (interazione completa).



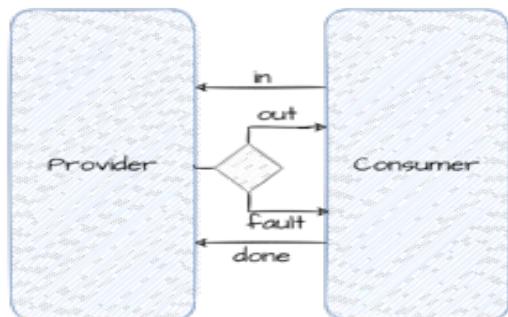
In-Only



Robust In-Only



In Optional-Out

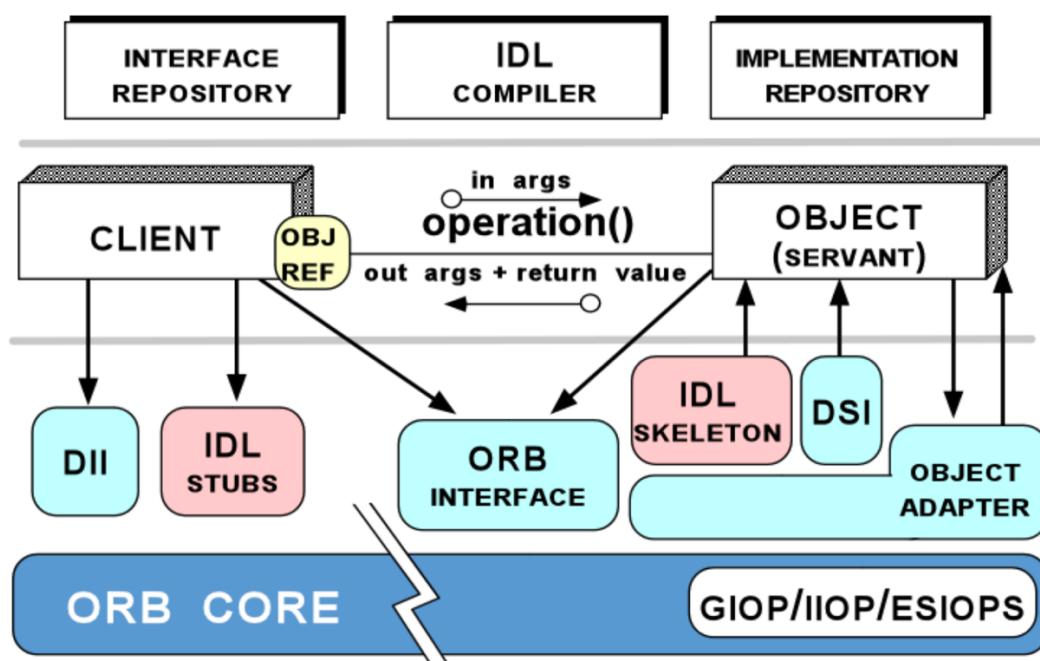


In-Out

## 8. Corba Component Model (CCM)

EJB 3.X non è l'unico modello a componenti distribuiti basato sull'approccio a container pesante ma esiste CORBA come potente container per oggetti distribuiti. Si immagini di voler realizzare un sistema distribuito che funzioni derivando dall'interazione sincrona bloccante fra parti di codice che non hanno il vincolo di essere scritte nello stesso linguaggio di programmazione. Con Java RMI si può creare un'applicazione che faccia invocazione di metodi remoti. Si vuole realizzare con CORBA la stessa cosa, ma si vuole anche che cliente e servitore possano essere scritti in qualsiasi linguaggio di programmazione.

### 8.1 Corba 2.x



Si ipotizzi di avere a disposizione del codice Pascal che si intende esporre all'esterno come codice ad oggetti. Per fare ciò utilizzando CORBA uno sviluppatore scrive un'interfaccia che deve essere esposta come oggetto **CORBA**. Per poter fare ciò utilizza il linguaggio **CORBA IDL** per creare un semplice file che dichiara qual è l'interfaccia del software originale. A partire da questo file d'interfaccia, l'infrastruttura CORBA deve essere in grado di generare l'**IDL Stub** e l'**IDL Skeleton**. Questi si occupano dell'integrazione del codice iniziale con CORBA e in particolare del **marshalling/unmarshalling** dei dati in ingresso e in uscita. Le operazioni di marshalling e unmarshalling sono molto importanti perché molto probabilmente il codice Pascal riceverà, dall'esterno, dei dati in un formato per lui incomprensibile.

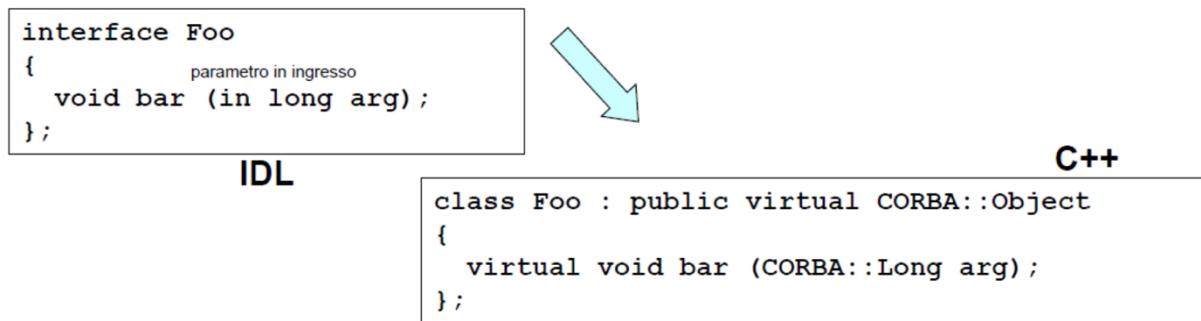
Lo **Stub** (che si trova lato cliente) si occupa di trasformare i dati che il cliente genera, nel formato a lui più congeniale, in un formato universale comprensibile da tutti.

Lo **Skeleton** (lato servitore) svolge l'operazione inversa, trasformando i dati dal formato universale al linguaggio specifico del servitore.

Quando il servitore risponde alla richiesta avvengono le operazioni inverse, cioè: lo **Skeleton** converte la risposta in formato universale e poi lo **Stub** la riconverte in linguaggio client. I due componenti Stub e Skeleton dipendono fortemente dalla conoscenza IDL di interfaccia.

L'**IDL compiler** fa parte dall'interfaccia IDL e si occupa della generazione di Stub e Skeleton. Gli **Object Adapter, Portable e Basic**, (POA e BOA) gestiscono l'attivazione del servitore.

Il funzionamento appena descritto è quello di CORBA con invocazione statica; infatti, la generazione di Stub e Skeleton avviene staticamente in fase di sviluppo. Se l'invocazione statica non è adatta o sufficiente allora si può utilizzare l'invocazione dinamica servendosi della **Dynamic Invocation Interface (DII)** e **Dynamic Skeleton Interface (DSI)**.



L'immagine sopra riportata mostra un esempio di file IDL di interfaccia, che viene passato ad un compilatore IDL per generare Stub e Skeleton.

Per recuperare l'oggetto CORBA il cliente interroga **l'Implementation Repository**. L'Implementation Repository è un registro che per ogni oggetto CORBA salva la locazione, per cui ogni riga del repository corrisponde ad una coppia nome dell'oggetto/riferimento. Ad ogni richiesta l'Implementation Repository restituisce un **Object Reference** CORBA che permette al cliente di effettuare la chiamata all'oggetto.

L'**Interface Repository** permette di registrare le interfacce CORBA. In uno scenario statico è ovviamente inutile, perché il cliente deve conoscere a tempo di sviluppo le locazioni e i riferimenti, per poter recuperare lo stub. Ma in uno scenario dinamico anche lo stub cambia, diventa dinamico amplia il set di operazioni possibili e prende il nome di **Dynamic Invocation Interface DII**.

Uno dei problemi di CORBA è la mancanza di un servizio a supporto dell'impacchettamento e del deployment del software che è, quindi, a carico dello sviluppatore. Una volta creata l'interfaccia, il programmatore si deve occupare di unire i vari pacchetti e farne il deployment. Le conseguenze legate a questo problema sono una scarsa manutenibilità e una scarsa penetrazione nel mercato.

## 8.2 Corba Component Model (CCM)

Corba offre una soluzione per il distributed object computing che definisce un modello a container per oggetti distribuiti, aggiungendo componenti e funzioni. Questo ha portato a una ridefinizione dello standard nel Corba Component Model (CCM).

Distributed Object Computing (DOC middleware) utilizza sempre un pattern **broker** che attraverso un proxy facilita la comunicazione tra il client e un servizio, inoltre non definisce solo API locali ma anche i protocolli da utilizzare “in mezzo” per facilitare la comunicazione delle due entità facendosi carico anche della parte di comunicazione, perché il **service access point** offerto

a livello applicativo prescinde dal linguaggio di programmazione. In CORBA la gestione della comunicazione è molto simile grazie all'utilizzo di un IDL che non dipende da nessun linguaggio o piattaforma in termini di interfacce applicative, così per ogni singolo linguaggio viene definito un pattern broker che delinearà il protocollo che segue le specifiche dello standard CORBA. Quindi CORBA facilita l'integrazione, non solo tra mondi object oriented ma anche tra mondi che non lo sono, rendendoli a oggetti nella loro interfaccia con il mondo esterno.

L'obiettivo di CORBA è lavorare con dispositivi molto leggeri e con poche risorse quindi il protocollo **GIOP** di CORBA è messo a disposizione per comunicare a byte. In questo senso, CORBA è talmente standard de facto che persino EJB utilizza RMI sopra al protocollo CORBA. CORBA automatizza una serie di funzionalità e servizi di sistema, tra cui:

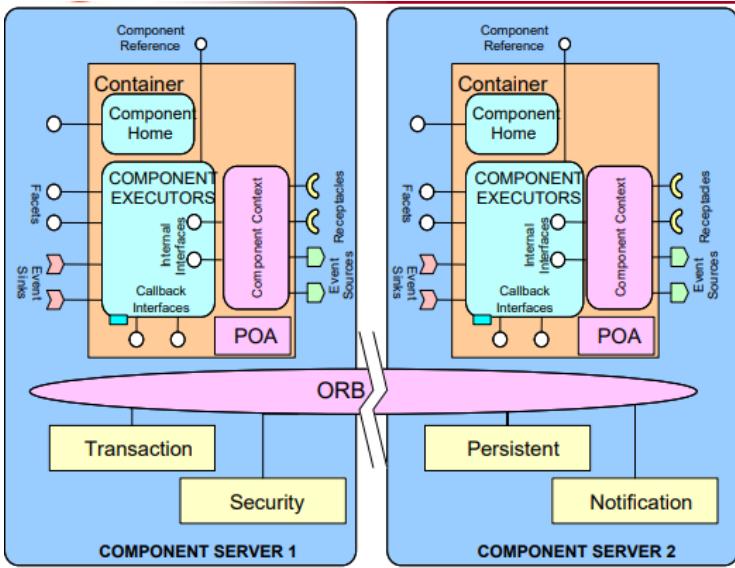
- Object location
- Connection e memory management
- Parameter marshaling e demarshalling
- Request demultiplexing
- Error handling e fault tolerance
- Object/server activation
- Concurrency e synchronization

I requisiti non-trivial di **Distributed Runtime Environment** (DRE) impongono collaborazione e coordinamento di oggetti e servizi multipli su diverse piattaforme. CORBA 2.x non riesce a rispettare tali requisiti poiché presenta alcuni problemi, come la mancanza di standard per server e node configuration, object e service configuration, application assembly, object/service deployment.

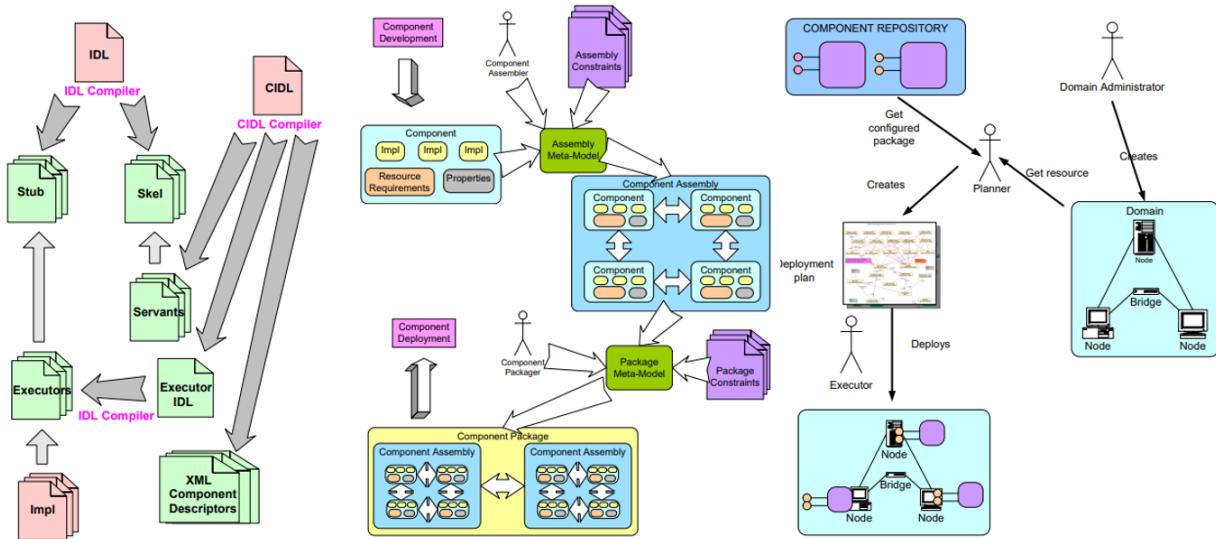
Le principali conseguenze di queste problematiche sono la scarsa adattabilità e manutenibilità, una crescita time-to-market e costi alti di integrazione. Per applicazioni anche poco complesse che richiedono l'utilizzo di alcuni oggetti e di servizi CORBA bisogna scrivere strumenti per il **Continuos Integration** e deployment.

Per andare oltre a questi limiti è stato proposto il **Corba Component Model** per far sì, che sia possibile creare dei componenti server che possano ospitare container e componenti, ovvero un'interfaccia HOME e una parte di componente che permette l'esecuzione.

I container definiscono operazioni che abilitano i **component executor** ad accedere a servizi comuni di middleware e politiche run-time associate, oltre ai servizi di supporto e la composizione di servizi a grana grossa.



Quindi agli strumenti classici si affiancano nuovi strumenti il **Component IDL Compiler** che aggiunge la parte di Executor e la realizzazione dei container, facilitando attraverso gli **XML component descriptor** il deployment del software, ovvero l'assembly e il packaging dei vari componenti. Il **Component Packaging** mette insieme metadati di implementation e configuration in assembly pronti per il deployment. Gli strumenti per **Component Deployment** automatizzano il deployment di component assembly verso component server



Prima si definiscono i componenti che possono richiedere l'utilizzo di vari oggetti, poi è possibile assemblarli e farli diventare componenti a grana grossa, a loro volta i componenti a grana grossa sono assemblabili. CORBA offre supporto alla fase di assembly e deployment mettendo a disposizione strumenti per facilitare l'assembly, e per facilitarne il deployment verso i component servant (anche nel distribuito). Con questi strumenti si può gestire il deployment attraverso un deployment plan, grazie al codice che si trova nel Component Repository su una rete di nodi distribuiti. La gestione è completa e ben pensata per ambienti distribuiti che coinvolgono molti nodi e ambienti molto larghi.

## 8.3 Confronto con altre tecnologie

Le tecnologie CORBA sono dorate, esse hanno avuto grande successo negli anni '80 e '90 soprattutto in ambito militare e bancario, purtroppo la sua popolarità ha subito un arresto a causa della mancata adozione ed implementazione da parte dei sistemi Microsoft che per motivazioni commerciali non hanno mai accettato tale standardizzazione, preferendo lo sviluppo di Web Services.

Le principali standardizzazioni sono le seguenti:

Name	Provider	Open Source	Language	URL
Component Integrated ACE ORB (CIAO)	Vanderbilt University & Washington University	Yes	C++	<a href="http://www.dre.vanderbilt.edu/CIAO/">www.dre.vanderbilt.edu/CIAO/</a>
Enterprise Java CORBA Component Model (EJCCM)	Computational Physics, Inc.	Yes	Java	<a href="http://www.cpi.com/ejccm/">www.cpi.com/ejccm/</a>
K2 	iCMG	No	C++	<a href="http://www.icmgworld.com/products.asp">www.icmgworld.com/products.asp</a>
MicoCCM	FPX	Yes	C++	<a href="http://www.fpx.de/MicoCCM/">www.fpx.de/MicoCCM/</a>
OpenCCM	ObjectWeb	Yes	Java	<a href="http://openccm.objectweb.org/">openccm.objectweb.org/</a>
QoS Enabled Distributed Object (Qedo)	Fokus	Yes	C++	<a href="http://www.qedo.org">www.qedo.org</a>
StarCCM	Source Forge	Yes	C++	<a href="http://sourceforge.net/projects/starccm/">sourceforge.net/projects/starccm/</a>

Di seguito sono elencate le principali caratteristiche di CORBA, EJB, COM e .NET mettendo in evidenza le differenze chiave:

### Enterprise Java Beans (EJB)

- Componenti CORBA creati e gestiti tramite l'interfaccia HOME.
- I componenti eseguono in container che gestiscono trasparentemente i servizi di sistema.
- I componenti sono ospitati da Application Component Server generici.
- I componenti possono essere realizzati in diversi linguaggi di implementazione.

### Microsoft Component Object Model (COM)

- Si possono avere diverse interfacce input e output per ogni componente.
- Sono presenti sia operazioni point-to-point sincrone e asincrone che eventi publish-subscribe.
- Capacità di component navigation e introspection.
- Supporto più efficace e flessibile, proprietà di distribuzione e QoS, meno multi-linguaggio e non lavora solo su Microsoft

### Microsoft .NET Framework

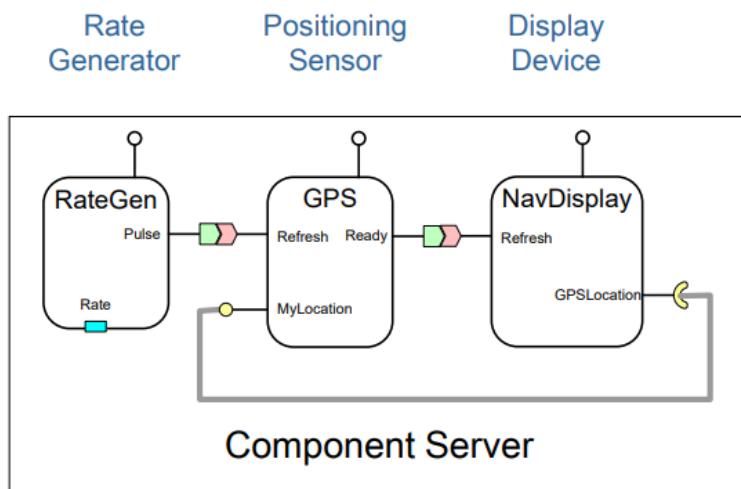
- Possono essere realizzati in diversi linguaggi programmazione
- Possono essere packaged per facilitare distribuzione
- MA possono eseguire su piattaforme multiple (non solo Microsoft Windows) e supportano il multi-linguaggio

### CORBA

- Si possono usare più linguaggi di programmazione.
- Corba ha delle HOME simili a EJB2
- Si può avere più di un container, se serve.
- CCM può avere diverse interfacce mostrabili ai clienti (non si può fare in EJB3, ma in COM sì) e lavorare ad eventi.
- Può navigare tra interfacce come il COM o .NET

## 8.4 Esempio running di funzionamento

Si supponga di avere un Component Server che ospita tre componenti. Il componente “Rate Generator”, il componente di calcolo della posizione, “GPS”, e un componente di display che mostra la mappa con la posizione corretta “NavDisplay”. Insieme formano un’applicazione con un determinato flusso di esecuzione.



`$CIAO_ROOT/examples/OEP/Display/`

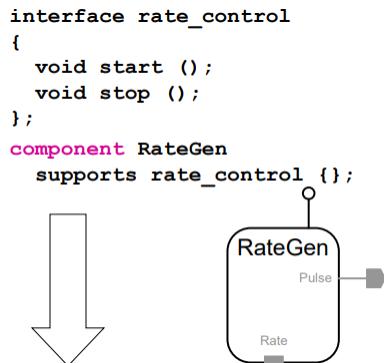
Il Rate Generator invia eventi Pulse (generati da un impulso), periodici, ai consumatori, per ottenere informazioni sul veicolo. Ha una porta **Rate** che è un parametro di configurazione definito a livello di deployment. Il Positioning Sensor (GPS) riceve eventi di refresh dai **Pubs**, ricalcola le coordinate cached disponibili via MyLocation facet, notifica sub via eventi Ready events. Il Display Device (NavDisplay) riceve eventi Refresh da **Pubs**, legge le coordinate correnti via GPSLocation receptacle, aggiorna il display.

Le frecce (verdi e rosa) servono per scambio di eventi e quindi per la gestione di scambio a eventi tipicamente sincrona non bloccante tra componenti a eventi. Il cerchio bianco invece indica la possibilità di configurare il componente.

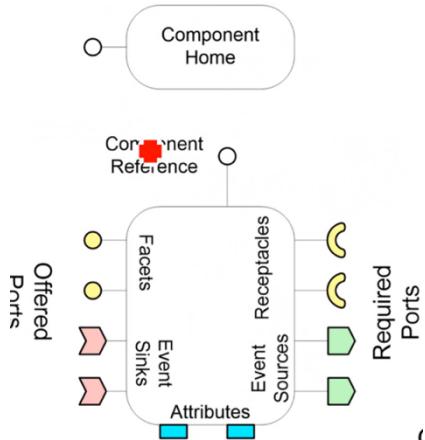
## 8.5 Componente

Il Corba Component Model è molto largo e completo, rispetto al mondo Corba 2 consente di aggiungere varie funzionalità, vi è una parte per l'integrazione di componenti, una per il packaging, fino ad arrivare all'assemblaggio dei vari componenti e infine facilitarne il deployment anche su reti di nodi distribuiti, vi è poi la parte di esecuzione in questi ambienti distribuiti.

Il componente è l'unità di composizione, di utilizzo e implementazione, è l'unità base di software utilizzabile in parti diverse. Per utilizzare il componente è possibile, grazie al mondo a oggetti, ereditare da un altro componente e supportare diverse interfacce preesistenti. Il componente offre una **Facet** che permette di attivare o disattivare il RateGen con i metodi **start()** e **stop()**, vi è poi l'interfaccia **rate\_control()** per controllare il Rate.



Le porte sono i punti di contatto tra i vari componenti. In particolare, vi sono le **Facet** che sono rappresentate dai cerchi gialli che espongono interfacce e quindi i metodi offerti da quelle interfacce, e i **Receptacle** che interagiscono con le Facet di altri componenti, soddisfacendo la necessità di combinare più Facet. Le **Sorgenti** e i **Sink** per eventi in uscita e in entrata e infine vi sono attributi per la configurazione dell'oggetto. La maggiore innovazione è rappresentata dalla Facet che consente di mettere insieme molti concetti differenti, permettendo sia comunicazioni sincrone bloccanti, che a scambio di messaggi e di eventi. Vi sono poi una Component Reference e una Component Home per creare nuovi componenti.



Il Corba Component Model si occupa di gestire il ciclo di vita dei componenti, tale gestione è integrata e standardizzata nel supporto. Si possono dichiarare diverse Home per definire diverse strategie di management. Grazie all'interfaccia Home è possibile definire varie strategie e applicarle al lifecycle management dei componenti, tale interfaccia Home è un metatype che ha riferimenti a interfacce e oggetti e gestisce una famiglia di componenti attraverso il tipo di lifecycle management definito. Ogni istanza di un oggetto è gestita da una sola istanza di Home, l'operazione base che si trova sempre nella Home è quella per la creazione dell'oggetto ovvero `create()`, il tutto in IDL 3 può essere riassunto con questa chiamata. Inoltre, questa interfaccia Home può ospitare operazioni addizionali user-defined.

I diversi componenti possono avere la necessità di collaborare fra loro, per questo ciascun componente offre viste diverse. Con Corba 2 era complesso definire una gestione avanzata del componente con eventuale parte relativa di gestione dinamica, con il Corba Component Model, il tutto è strutturato in un modello ben fatto.

### 8.5.1 Facet

Le Facet fanno parte di questo modello ben fatto, e sono chiamate “top of the lego”, che significa che le funzionalità che offrono al mondo esterno e la gestione di eventi stanno sopra e sotto nel “bottom of the lego” si trova ciò che ci si aspetta per legarsi al mondo sottostante, ovvero i Receptacle e i Sink. Le Facet sono le interfacce delle operazioni che i componenti offrono. Dal punto di vista logico, sono i servizi che il componente offre e dando modo di avere internamente componenti che realizzano diverse possibili interfacce. Le Facet definiscono interfacce con operazioni offerte, queste operazioni sono specificate tramite keyword `provides()` e rappresentano logicamente il componente stesso, la facet non è un'entità separata contenuta nel componente. Le Facet hanno riferimenti a oggetti indipendenti ottenuti tramite operazione `provide_*`() da una Factory e possono essere usati per implementare Extension Interface pattern. In IDL 2 il tutto deve essere fatto ragionando solo sui componenti di base. La Facet però non è una normale interfaccia: le facet possono essere multiple, un componente può esporre interfacce diverse a clienti diversi e possono essere modificate run time.

## 8.5.2 Receptacles

Il Receptacles specifica a livello di definizione le Facets di cui ha bisogno un determinato componente per lavorare correttamente. Attraverso l'uso della keyword `uses()` viene definito il Receptable che si aspetta una o più Facet. Le connessioni sono effettuate staticamente durante la fase di deployment o dinamicamente gestite dai container che supportano interazione con clienti o altri componenti via callback. Inoltre, CCM supporta connection establishment a runtime. Si può chiedere che durante il deployment vengano controllati i vincoli di Receptacles di cui un componente ha bisogno, oppure di controllare che tali vincoli vengano rispettati. Si può sempre chiedere in fase di deployment quale binding associare nel caso ci siano più vincoli di Receptacles. Quindi può servire:

- Staticamente per capire se un deployment è completo e sono presenti tutte le dipendenze.
- Dinamicamente per cercare un altro componente che faccia match con il Receptacles specificato. Si parla in CCM di possibilità di connessione a tempo di deployment e a tempo di runtime.

## 8.5.3 Scambio di eventi e Event Sources

Per quanto riguarda lo scambio di messaggi, CCM mette a disposizione lo scambio di eventi e le interazioni sincrone non bloccanti, con l'idea di definire dei collegamenti per consentire lo scambio di eventi. Lo scambio di eventi può essere basato su tipi statici oppure basato su `EventType`. Lo standard prevede una interazione di tipo push, poiché in Corba il servizio di scambio di eventi può usare sia una politica push che pull, ma per i componenti vi sono solo interazioni push. Per questo all'interno dei componenti sono presenti delle configurazioni che emettono Tick da due possibili tipi di Event Source: Publishes ed Emit con l'unica differenza che la parola chiave `publishes` per sorgenti Publisher prevede molti consumatori, mentre `emits` per sorgenti Emit prevede un solo consumatore. Chi riceve eventi può decidere di consumarli direttamente oppure può usare un broker, cioè un distributore di eventi, non dissimile da un distributore di messaggi come può essere JMS. In CORBA ce ne sono alcuni già notificati come Corba Object Service Notification.

## 8.5.4 Sink

Per connettersi al **Sink** di eventi può essere realizzata una comunicazione diretta o una comunicazione con intermediari. Si possono usare event-service del mondo Corba, oppure IDS standard per la distribuzione di eventi, con qualità e vincoli real-time anche in senso stretto. Dal punto di vista del componente questo emette sempre con interazioni push verso l'altro componente o verso la coda di eventi, questo facilita le cose soprattutto se l'altro componente non ha la pull.

Il Sink Refresh, definito con la parola `consumes`, riceve i tick. Si possono avere connessioni named a cui inviare eventi solo di specifiche tipologie. Event sink multipli dello stesso tipo possono essere subscriber della stessa sorgente, non vi è distinzione fra emitter & publisher, rende possibile connessi a sorgenti via object reference ottenuta tramite operazione `get_consumer_*`() su factory.

## 8.5.4 Attributes

Per rendere le implementazioni dei componenti più adattabili e flessibili, le proprietà dei componenti dovrebbero essere riconfigurabili. Questo presenta diversi problemi: le applicazioni non dovrebbero legarsi a una specifica configurazione troppo presto, non ci sono standard per specificare parametri configurabili per componenti in CORBA 2.x, vi è il bisogno di meccanismi standard per configurazione. La soluzione CCM è quella di configurare componenti con attributi di assembly/deployment, via home o nella fase di inizializzazione.

Gli attributes sono particolari proprietà che possono essere configurate dall'esterno, tipicamente dal container, in base alle informazioni che si passano sui file di deployment descriptor. La parola chiave è **attribute**. Questi attributi vengono definiti e viene dato loro un valore a tempo di deployment tramite file di configurazione esterni, ovvero dei descriptor di tipo XML.

## 8.5.5 Navigation e Introspection in CCM

I componenti hanno bisogno di essere “aggregati” per formare applicazioni complete, vi sono però alcuni problemi: i componenti possono avere porte multiple con diversi nomi e tipi, diventa dispendioso scrivere manualmente il codice necessario per collegare un insieme di componenti per una applicazione specifica. Per cui in CCM è stata implementata tale soluzione: vi sono interfacce per l'introspezione che consentono la scoperta dinamica delle capacità dei componenti, e generiche operazioni sulle porte per connettere componenti tramite strumenti esterni di deployment e configuration.

La capacità di navigation e introspection sono realizzate da **CCMObject**, che offre le seguenti interfacce: interfaccia **Navigation** per facet, interfaccia **Receptacles** per receptacle e interfaccia **Events** per porte per eventi. Navigation, permette, partendo dal riferimento base di un componente, di recuperare ogni sua facet via operazioni facet-specific automaticamente generate e supportate. Con il metodo **provide()** si può restringere la visibilità (narrowing) all'interfaccia d'interesse tale metodo restituirà il riferimento a cui si è interessati (es. **Components::CCMObject::get\_all\_facets()** & **Components::CCMObject::provide()** ).

La Navigation permette inoltre, partendo dalla facet, di ritrovare il riferimento base di un componente tramite **CORBA::Object::\_get\_component()**.

Tutto il codice per navigation e introspection è generato automaticamente dal compilatore CIDL come component servant.

Uso delle interfacce di navigazione di un componente.

```
int main (int argc, char *argv[]){
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

    // Get the NameService reference
    CORBA::Object_var o = ns->resolve_str ("myHelloHome");HelloHome_var hh = HelloHome::_narrow (o.in ());
    HelloWorld_var hw = hh->create ();

    // Get all facets & receptacles
    Components::FacetDescriptions_var fd = hw->get_all_facets ();
    Components::ReceptacleDescriptions_var rd = hw->get_all_receptacles ();

    // Get a named facet with a name "Farewell"
    CORBA::Object_var fobj = hw->provide ("Farewell");

    return 0;
}
```

In questo esempio si vuole recuperare una certa facet e invocare un metodo di tale interfaccia. Si parte dall'ORB che è un servizio di nomi che dà la possibilità di recuperare i servizi di base come il servizio di naming (e.g. RMI Registry). Una volta recuperato il servizio di nomi si possono invocare dei servizi di lookup, in particolare in questo caso si fa il lookup della “ComponentHome” registrata come MyHelloHome, attraverso il narrowing si può recuperare un oggetto tipato Home, a questo punto su quell’oggetto si applica il metodo **create** che crea il componente, che, adesso, può essere interrogato per ottenere una descrizione di tutte le facet offerte da quel componente stesso. Si può poi con il receptacle effettuare l’introspezione, con il metodo **provide** si può chiedere il recupero del riferimento all’implementazione di quella interfaccia, una volta ottenuto si può fare il narrowing e a quel punto effettuare la chiamata a un metodo. Le interfacce in CORBA non vengono mantenute insieme alle implementazioni ma in un repository che si chiama Interface Repository. Quindi le informazioni dell’interfaccia non sono insieme agli altri dati questo genera overhead quando bisogna recuperare molte interfacce diverse.

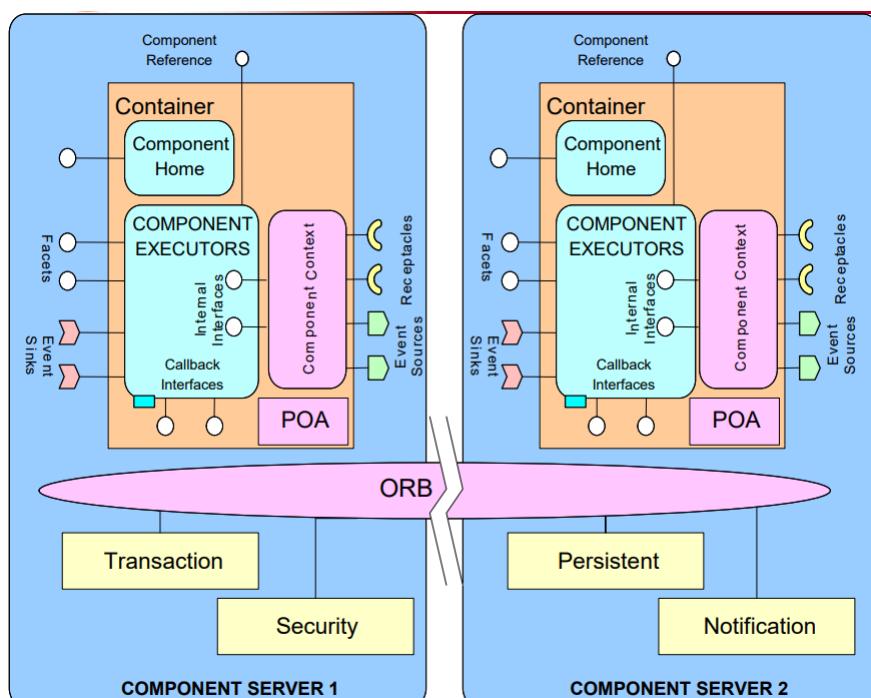
Riassumendo le principali caratteristiche di CCM per il supporto cliente si può dire che CCM definisce operazioni per life-cycle management (home), definisce che cosa un componente offre agli altri componenti, definisce che cosa un componente richiede da altri componenti, definisce quali modalità di collaborazione sono usate fra componenti. Le operazioni possono essere di tipo Point-to-point via operation invocation o Publish/subscribe via event notification, definisce quali attributi dei componenti sono configurabili.

## 8.6 Implementazione dinamica dei componenti e supporto runtime

Il supporto runtime dei componenti è possibile grazie al Component Server che facilita il deployment della configurazione delle applicazioni. Questo avviene perché Corba mette a punto astrazioni di alto livello per la gestione del Servant e una serie di tool che facilitano la configurazione attraverso tecniche di meta-programming con anche la possibilità di effettuare l’introspezione, ma anche grazie all’introduzione di una serie di parole chiave gestite dall’IDL compiler. Questo supporto a runtime facilita la gestione del ciclo di vita di questi componenti.

## 8.7 Container

Il CCM attraverso questa definizione del container va ad estendere il portable object adapter POA di base di Corba (che si occupa della gestione del ciclo di vita degli oggetti), il container va oltre l'object adapter nella gestione del loro ciclo di vita, indirizzandosi verso l'obiettivo della facilitazione dell'uso delle risorse. Ci sono poi i naming services la gestione della sicurezza, delle transazioni eccetera. Questi servizi esistevano ma dovevano essere utilizzati solo con oggetti a grana fine quindi oggetti che facevano molto poco; invece, con l'introduzione dei container si hanno servizi di più alto livello che con file di configurazione e deployment, automatizzano molti servizi che prima erano programmatici. Inoltre, vi è una gestione a call back su cui si può lavorare per capire come funzionano le cose e prendere misure dove necessario.



Anche nel modello CCM si possono definire categorie di componenti a seconda del tipo di implementazione di container. Vi sono i Service che sono componenti senza stato, per la rappresentazione di una sessione si può avere un componente Session con uno stato di tipo soft (stateful session bean), poi vi sono Process ed Entity entrambi durable con la possibilità di essere invocati dall'esterno con chiavi o per riferimento. Queste categorie possono essere specificate dichiarativamente tramite il CIDL file oppure programmate imperativamente.

Component category	Container Implementation type	Container type	External Type
Service	Stateless	Session	Keyless
Session	Conversational	Session	Keyless
Process	Durable	Entity	Keyless
Entity	Durable	Entity	Keyful

Il container offre interfacce verso l'interno e verso l'esterno, le interfacce verso l'esterno sono invocabili dai clienti, per interagire e configurare il container, quelle rivolte verso l'interno invece sono interfacce con il quale il container può offrire ai componenti possibilità di dialogo con il

contesto del container stesso, e per le quali attraverso l'implementazione delle callback è possibile l'interazione con il container.

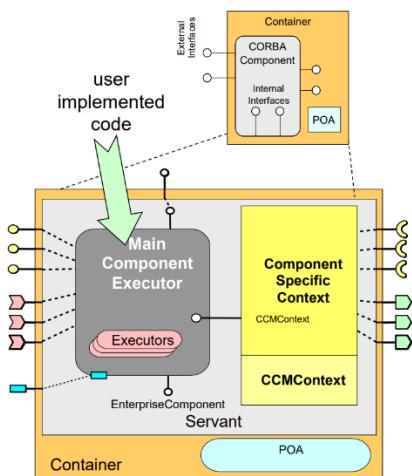
## 8.8 Strategie Container-managed

Ci sono diverse politiche di gestione, le strategie container managed cercano di disaccoppiare in modo forte la parte di politiche di configurazione dalla parte di implementazione del container, il container si fa carico della gestione dei servizi di sistema, ciò avviene preparando metadati XML o direttive da dare al Component IDL Compiler CIDL che dichiara la gestione dei servizi di sistema come transazioni, eventi, eccetera.

### 8.8.1 Executor

In particolare, lato server ci sono degli executor che possono andare a realizzare la business logic, vi è una suddivisione tra Component del business model e Home che realizza la gestione e la creazione dei componenti con la possibilità di avere lifecycle diversi. Le Home Executor sono monolitiche mentre i Component Executor possono essere sia monolitici con tutte le porte implementate dalla stessa classe, oppure le porte sono suddivise tra le varie classi che lo implementano. Lo sviluppatore partendo da file definiti dell'IDL nel Component IDL di CCM genera una serie di scatole vuote pronte ad accogliere la logica applicativa. Il programmatore deve solo aggiungere la propria logica applicativa.

Nel mondo Corba tutte questa funzionalità appartengono al mondo delle interfacce, Corba predisponde tutto per il linguaggio target per sollevare lo sviluppatore dalla gestione di tutti i problemi legati ai servizi di sistema, poi il programmatore si deve occupare di implementare le varie cose. Gli Executor vanno a eseguire nel container, il modello è un container pesante simile a EJB con alcune differenze. Gli Executor dei componenti devono implementare un'interfaccia locale per callback lifecycle ad uso del container, vi sono SessionComponent per componenti transienti, ed EntityComponent per componenti persistenti. Gli Executor dei componenti possono interagire con container e componenti connessi via context interface. L'interazione avviene attraverso un contesto che viene esposto con un container.



## 9. Spring

Spring è una soluzione a container leggero per la realizzazione di applicazioni Java SE e Java EE, si propone come alternativa/complemento a J2EE. A differenza di quest'ultimo, Spring propone un modello più semplice e leggero (soprattutto rispetto ad EJB) per lo sviluppo di entità di business. Tale semplicità è rafforzata dall'utilizzo di tecnologie come l'**Inversion of Control** e l'**Aspect Oriented Programming** che danno maggiore spessore al framework e favoriscono la focalizzazione dello sviluppatore sulla logica applicativa essenziale.

Spring è un framework “leggero” e grazie alla sua architettura estremamente modulare è possibile utilizzarlo nella sua interezza o solo in parte. L'adozione di Spring in un progetto è molto semplice, può avvenire in maniera incrementale e non ne sconvolge l'architettura esistente. Questa sua peculiarità ne permette anche una facile integrazione con altri framework esistenti, come ad esempio EJB per J2EE, Hibernate, iBates, JDBC per l'accesso a dati e O/RM, Java Persistence API per la persistenza, Struts e WebWork per Web tier.

Alcune delle funzionalità chiave di Spring sono:

- **Inversion of Control (IoC) e Dependency Injection**
- **Supporto alla persistenza**
- **Integrazione con web tier**
- **Aspect Oriented Programming (AOP)**

### Dependency Injection

La configurazione dei componenti è facilitata grazie all'uso dei principi di Inversion of control e dependency injection che mirano ad eliminare la necessità di binding ‘manuale’ fra i componenti. L'idea di base è quella di avere una factory globale (**BeanFactory**) che può essere utilizzata per ritrovare e gestire le relazioni fra componenti.

### Persistenza

Spring offre supporto alla persistenza fornendo un livello di astrazione generico per la gestione delle transazioni con database prescindendo dall'utilizzo di un EJB Container. Inoltre, Spring offre delle strategie built-in per JTA (Java Transaction API) e integra framework di persistenza come Hibernate, JDO, etc...

### Integrazione con Web Tier

Un intero modulo di Spring è dedicato alla realizzazione di un framework MVC (Model View Control) per applicazioni Web, con l'obiettivo di facilitare la realizzazione del web tier attraverso la generazione automatica di “viste” e la gestione del web flow per la navigazione a grana fine.

### Aspect Oriented Programming

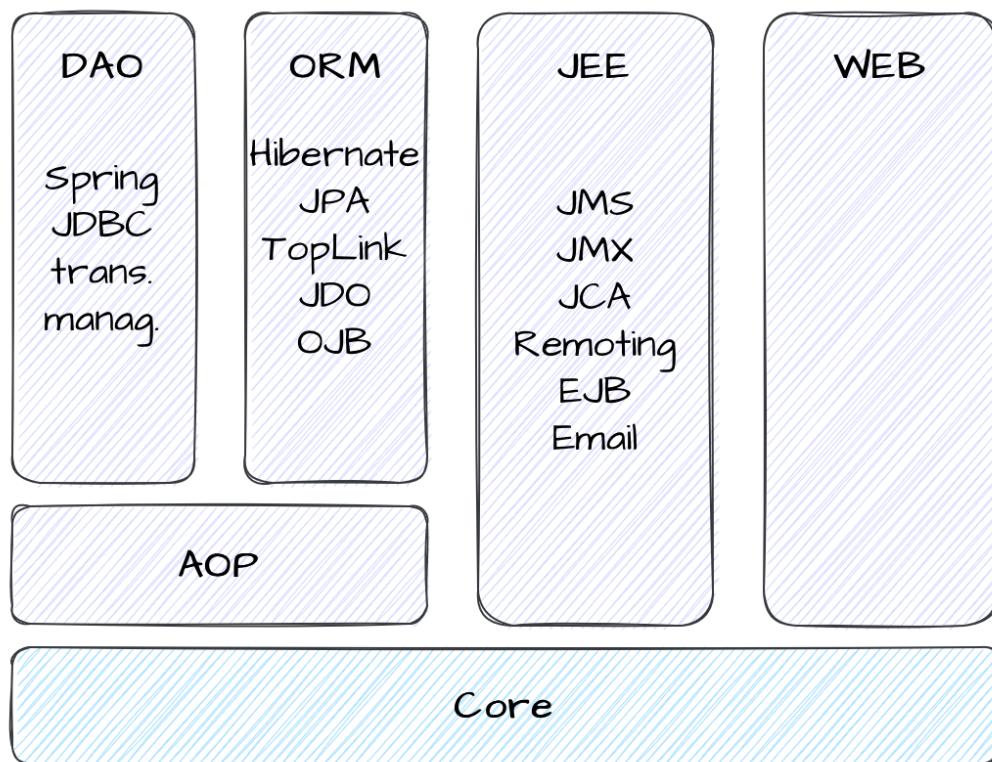
La programmazione orientata agli aspetti è un paradigma di programmazione basato sulla creazione di entità software, denominate aspetti, che sovrintendono alle interazioni fra oggetti finalizzate ad eseguire un compito comune. AOP offre supporto ai servizi di sistema, apportando miglioramenti, soprattutto, in termini di modularità (per il riutilizzo in contesti diversi) e facilità di testing delle applicazioni.

Spring rappresenta un approccio piuttosto unico che ha fortemente influenzato i container successivi verso tecnologie a microcontainer.

Le motivazioni per cui scegliere Spring sono diverse. Spring consente l'integrazione e la cooperazione fra componenti (secondo il semplice modello JavaBean) via Dependency Injection, pone molta importanza al disaccoppiamento, mette a disposizione **Test-Driven Development** (TDD), ovvero la possibilità di effettuare testing delle classi (POJO) senza essere legati al framework, semplifica l'uso di tecnologie diffuse e di successo utilizzando astrazioni che isolano il codice applicativo, eliminando il codice ridondante, gestendo le comuni condizioni di errore (caso delle **unchecked exception**), lasciando una parziale visibilità su queste tecnologie. La progettazione avviene per interfacce, con ottimo isolamento delle funzionalità dai dettagli implementativi, inoltre integra la programmazione dichiarativa via AOP che consente una facile configurazione degli aspetti.

## 9.1 Architettura di Spring

Uno dei principali vantaggi offerti da Spring è quello di poter escludere le parti del framework non necessarie all'applicazione che si sta sviluppando, includendo esclusivamente quelle utili. Questo è possibile grazie all'architettura modulare di Spring che è composta principalmente da cinque livelli.



Al livello più basso dell'architettura si trova il **Core** che svolge il ruolo cruciale di container leggero basato sul concetto di Inversion of Control e fa da collante consentendo l'interazione tra le varie parti dell'architettura Spring. Sul Core si trova il blocco **JEE** che gestisce l'integrazione con i servizi enterprise (JMS, JMX, EJB etc...), il blocco **Web** con tutti i framework che possono essere integrati (Spring Web MVC, Struts, JSP, etc...) e una parte più orientata al backend dati con il

blocco **ORM** per la gestione della persistenza e conversione dal mondo oggetti al mondo relazionale (Hibernate, JPA, etc...) e il blocco **DAO** per la gestione delle connessioni e soprattutto delle transazioni (JDBC), ORM e DAO sono realizzati grazie al blocco **AOP** che permette l'utilizzo di tecniche di Aspect Oriented Programming.

### **Core Package**

Rappresenta la parte principale di Spring e sopra di esso è costruito l'intero framework. Consiste in un container leggero che si occupa di Inversion of Control e della Dependency Injection. L'elemento fondamentale del core package è rappresentato dalla **BeanFactory**, un'implementazione estesa del pattern factory, che consente di disaccoppiare la parte di configurazione e dipendenza dalla logica applicativa. All'interno del Core Package il modulo **Context** estende i servizi basilari del Core aggiungendo le funzionalità tipiche di un moderno framework. Tra queste troviamo JNDI, EJB, JMX, e supporto agli eventi.

### **Dao Package**

Fornisce un livello di astrazione per l'accesso ai dati mediante tecnologie eterogenee tra loro come ad esempio JDBC, Hibernate o JDO. Questo modulo tende a nascondere la complessità delle API di accesso ai dati, semplificando ed uniformando quelle che sono le problematiche legate alla gestione delle connessioni, delle transazioni e delle eccezioni.

### **ORM Package**

Notevole attenzione è stata data all'integrazione del framework con i principali ORM in circolazione compresi JPA, JDO, Hibernate, e iBatis. Oltre a questo, il Data Access si occupa di fornire supporto per il Java Message Service e per svariate implementazioni di Object/XML Mapper come JAXB, Castor e XMLBean.

### **Web Package**

Come si intuisce dal nome, questo livello è responsabile delle caratteristiche Web del framework. Oltre alle funzionalità basilari, per la creazione di applicazioni Web, questo livello mette a disposizione un'implementazione del pattern MVC realizzando lo Spring MVC Framework (moduli Web-Servlet e Web-Portlet).

### **AOP Package**

Aggiunge al framework la funzionalità della programmazione Aspect Oriented. AOP offre un nuovo modo di programmare che porta notevoli vantaggi in tutte quelle operazioni che sono trasversali tra più oggetti. In Spring l'utilizzo di AOP offre il meglio di sé nella gestione delle transazioni, permettendo di evitare l'utilizzo degli EJB per tale scopo.

## 9.2 Aspect Oriented Programming (AOP)

L'aspect oriented programming ha come idea base quella di semplificare il più possibile le problematiche traversali alla logica applicativa intercettando le chiamate da remoto (cross-cutting concern) come ad esempio: Logging, Locking, Gestione degli eventi, Gestione delle transazioni, Sicurezza e auditing.

Tutti questi servizi sono presenti anche in EJB (servizi di sistema) e la loro gestione era affidata al container.

L'Aspect Oriented Programming si basa su alcuni concetti fondamentali:

- **Joinpoint**
- **Advice**
- **Pointcut e Aspect**
- **Weaving e Target**
- **Introduction**

L'idea di base è quella di intercettare le chiamate, inserendo della logica per la gestione dell'aspetto che si sta intercettando.

### **Joinpoint**

Il joinpoint è un punto all'interno del codice applicativo, anche determinato a runtime, in cui si può inserire della logica aggiuntiva rispetto alla logica applicativa. Esempi di joinpoint sono, per esempio, l'invocazione di metodi, l'inizializzazione di classi o l'inizializzazione di oggetti (creazione di istanze).

### **Advice**

L'advice è una parte di codice con logica addizionale (un metodo) che viene aggiunta alla logica di base, esso viene eseguito automaticamente quando viene raggiunto un determinato joinpoint. Esistono diverse tipologie di advice:

- **Before advice:** viene eseguito prima del joinpoint.
- **After advice:** viene eseguito dopo il joinpoint.
- **Around advice:** costituisce l'Advice più versatile in quanto consente di prendere il controllo dell'intero Join Point. Non solo è possibile specificare le stesse cose degli Advice precedenti ma è anche possibile decidere quando e se eseguire il Join Point o addirittura generare eccezioni.

### **Pointcut**

Il pointcut è un insieme di joinpoint che vengono verificati per definire quando eseguire un advice. Descrive le regole con cui associare l'esecuzione di un Advice ad un determinato Join Point. Sostanzialmente attraverso un predicato viene specificato che al verificarsi di un determinato Join Point (es: esecuzione di un certo metodo) sia applicato un determinato Advice (es: Before). I pointcut possono essere composti in relazione, anche abbastanza complesse, per vincolare il momento di esecuzione dell'advice corrispondente. La differenza tra pointcut e joinpoint è la seguente: il pointcut è l'insieme di tutte le invocazioni di metodo in una determinata classe mentre l'invocazione di un metodo è un tipico joinpoint.

### **Aspect**

L'aspect è un insieme di advice e pointcut, è l'equivalente di una classe in OOP ed è utilizzata nella programmazione ad aspetti per modularizzare i cross-cutting concern. Grazie agli aspect è possibile aggiungere dinamicamente comportamenti agli oggetti di dominio senza che questi ne siano a conoscenza.

### **Weaving**

Con weaving si fa riferimento all'effettivo processo di inserimento di aspect nel codice applicativo nel punto appropriato. Il weaving può avvenire a tempo di compilazione, per mezzo di un particolare compilatore, oppure a runtime. Ovviamente il weaving a runtime è più costoso in termini di overhead ma offre un maggiore potere espressivo.

## Target

Il target è un oggetto il cui flusso di esecuzione viene modificato attraverso un processo AOP attraverso il weaving, in alcuni casi viene anche indicato come oggetto con advice (advised object).

L' AOP può essere **statico** o **dinamico**. Nel primo caso il processo di weaving viene realizzato come un passo ulteriore del processo di sviluppo durante la build dell'applicazione, questo incide sul codice dell'applicazione che viene eseguito, il weaving può avvenire in diversi modi anche andando a modificare i BYTECODE. Ad esempio, in un programma Java, si può avere weaving attraverso la modifica del bytecode di una applicazione, senza la modifica del codice sorgente, i file bytecode possono essere modificati prima della messa in esecuzione questo richiede che durante la compilazione venga effettuato questo passaggio di modifica. Nel caso di AOP dinamico il processo di weaving viene realizzato dinamicamente a runtime ciò permette di cambiare il weaving senza la necessità di ricompilare l'applicazione; tuttavia, in questo caso bisogna intercettare il punto in cui cambiare il weaving e inserire gli advice, svolgendo quindi una continua attività di monitoraggio e iniezione durante l'esecuzione, questo rappresenta uno svantaggio perché causa un overhead durante l'esecuzione.

In Spring la realizzazione del Aspect Oriented Programming avviene tramite l'uso di un proxy. Perché sia possibile intercettare i metodi di un target object è necessario "wrapparlo" (si può pensare a una classe wrapper come ad una classe che ne avvolge un'altra) in una classe proxy, un'istanza della classe **ProxyFactoryBean**, e associargli tramite la proprietà **interceptorNames** la lista degli advice da utilizzare sul target object.

## 9.3 Dependency Injection in Spring

La dependency injection è l'applicazione più nota del principio di Inversion of Control. L'idea di base è che il container leggero si occupi di risolvere con l'iniezione, le dipendenze dei componenti attraverso l'opportuna configurazione dell'implementazione degli oggetti (push). Questa idea è opposta ai pattern più classici di istanziazione di componenti, in cui è il componente che deve determinare l'implementazione della risorsa desiderata (pull). Questo nuovo approccio consente di mantenere il codice scritto dagli sviluppatori il più possibile pulito in quanto le dipendenze sono risolte dal container leggero grazie al meccanismo di Inversion of Control.

Un esempio di dependency injection è presente anche in EJB 3.0, che attraverso le annotazioni aggiunge tutte le informazioni utili ad esprimere le dipendenze tra i vari componenti, con il vantaggio di una maggiore flessibilità (eliminazione del codice di lookup nella logica di business), nel caso in cui si verifichi un cambiamento delle risorse esterne non c'è la necessità di modificare il codice, la possibilità e facilità di testing senza alcun bisogno di dipendere da risorse esterne o dal container durante la fase di testing e con la possibilità, peraltro, di abilitare il testing automatico, ed infine un'elevata manutenibilità del codice che permette riutilizzo in diversi ambienti applicativi cambiando semplicemente i file di configurazione (o in generale le specifiche di Dependency Injection) e non il codice, dividendo nettamente la parte di programma da quella di configurazione.

In Spring la Dependency Injection può avvenire in due modi differenti:

## Dependency Injection a livello di costruttore

Le dipendenze vengono iniettate tramite il costruttore della classe, come nell'esempio seguente:

```
public class ConstructorInjection {  
    private Dependency dep;  
    public ConstructorInjection(Dependency dep) { this.dep = dep; }  
}
```

## Dependency Injection a livello di metodi “setter”

La modalità più frequentemente usata consiste nell'iniettare le dipendenze attraverso i metodi setter della classe, questa modalità offre il vantaggio di poter essere riutilizzata a runtime.

```
public class SetterInjection {  
    private Dependency dep;  
    public void setMyDependency(Dependency dep) { this.dep = dep; }  
}
```

### 9.3.1 BeanFactory

Lo **IoC Container** fornisce un contesto altamente configurabile per la creazione e risoluzione delle dipendenze di componenti chiamati bean (da non confondere con i JavaBean).

A differenza di quanto avviene nei cosiddetti “container pesanti” dove i componenti gestiti devono rispecchiare caratteristiche particolari o implementare opportune classi astratte fornite dal framework, in Spring un bean non deve aderire a nessun tipo di contratto e può essere rappresentato da una qualunque classe Java.

Lo IoC Container è realizzato da due interfacce:

- **BeanFactory**: che definisce le funzionalità di base per la gestione dei bean.
- **ApplicationContext**: che estende le funzionalità basilari aggiungendone altre tipicamente enterprise come, ad esempio, la gestione degli eventi o l'integrazione con AOP.

L'interfaccia BeanFactory rappresenta la forma più semplice di IoC Container in Spring e ha il compito di creare i bean necessari all'applicazione, risolvere le loro dipendenze attraverso l'utilizzo dell'injection, gestirne l'intero ciclo di vita dei bean.

Per svolgere questi compiti, il container si appoggia a configurazioni impostate dall'utente che, riflettendo lo scenario applicativo, specificano i bean che dovranno essere gestiti dal container, le dipendenze che intercorrono tra questi oltre alle varie configurazioni specifiche.

Esistono diverse implementazioni di **BeanFactory**, la più comune, è senza dubbio la **XmIBeanFactory** che permette di utilizzare uno o più file XML per descrivere la configurazione da utilizzare. Una volta creato l'oggetto **XMLBeanFactory**, questo legge un file di configurazione XML e si occupa di fare l'injection, questo processo è detto **wiring** della configurazione. **XmIBeanFactory** estende la classe **DefaultListableBeanFactory** per leggere le definizioni di Bean da un documento XML. Quindi la **XMLBeanFactory** legge la configurazione da un file XML e attraverso il metodo **getBean()** ottiene l'istanza logica del bean.

```
public class XmlConfigWithBeanFactory {  
    public static void main(String[] args) {
```

```

        XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource("beans.xml"));
        SomeBeanInterface b = (SomeBeanInterface) factory.getBean("nameOftheBean");
    }
}

```

Tutte le dipendenze di un bean da altri e tutte le configurazioni del bean sono informazione che vengono specificate nel file XML e vengono iniettate dalla factory. Un esempio di file XML è il seguente:

```

<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <bean id="..." class="...">
        <!-- eventuali risoluzioni di dipendenze e proprietà -->
    </bean>
</beans>

```

Spring supporta diversi tipi di parametri con cui fare Injection:

- **Valori semplici**
- **Bean all'interno della stessa factory:** si usa quando è necessario fare Injection di un bean all'interno di un altro bean (target bean). Si fa uso del tag **<ref>** all'interno del tag **<property>** o **<constructor-arg>** del target bean. Se il tipo definito nel target è un'interfaccia allora il bean iniettato dovrà essere un'implementazione di tale interfaccia, se il tipo definito nel target è una classe allora il bean iniettato deve essere della stessa classe o di una sottoclasse.
- **Bean in diverse factory**
- **Collezioni**
- **Proprietà definite esternamente**

Tutti questi parametri possono essere usati sia per Injection sui costruttori, sia per Injection sui metodi setter.

Il ritrovamento dei Bean avviene attraverso il naming dei componenti Spring che vengono specificati nei file XML. Ogni bean deve avere un nome unico all'interno della **BeanFactory** che lo contiene. La procedura di risoluzione dei nomi avviene seguendo le seguenti regole: se un tag **<bean>** ha un attributo di nome **id**, il valore di questo attributo viene usato come nome, se l'attributo **id** non è presente, Spring cerca un attributo di nome **name**, se né **id** né **name** sono definiti Spring usa il nome della classe del Bean come suo nome.

## 9.4 L' Hello World in Spring

Il blocco di codice seguente mostra un esempio di Hello World realizzato in Spring tramite il meccanismo della Dependency Injection, al fine di comprendere quali siano i vantaggi dell'uso della DI si parte mostrando una classe java base che esegue un “hello world”:

```

public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}

```

Il problema principale relativo a questa porzione di codice è il fatto che ogni volta che si vuole cambiare il messaggio si ha la necessità di modificare il codice e ricompilare il tutto, ciò rende questo codice poco riutilizzabile e poco estendibile.

Un primo passo verso la Dependency Injection è quello di disaccoppiare l'implementazione della logica del **message provider** (dice qual è il messaggio da stampare) dal resto del codice, tramite la creazione di una classe separata.

Oltre a questo, si disaccoppia anche l'implementazione della logica di message rendering (chi stampa il messaggio) dal resto del codice:

```
public class HelloWorldMessageProvider{
    public String getMessage() { return "Hello World!"; }
}

public class StandardOutMessageRenderer {
    private HelloWorldMessageProvider messageProvider = null;

    public void render() {
        if (messageProvider == null) {
            throw new RuntimeException("You must set the property messageProvider of class:"
                + StandardOutMessageRenderer.class.getName());
        }
        System.out.println(messageProvider.getMessage());
    }

    public void setMessageProvider(HelloWorldMessageProvider provider){
        this.messageProvider = provider;
    }

    public HelloWorldMessageProvider getMessageProvider() {
        return this.messageProvider;
    }
}
```

La classe **StandardOutMessageRender** ha una dipendenza dal message provider che gli permette di stampare il messaggio. La Dependency Injection è realizzata attraverso il metodo setter (`setMessageProvider`).

A questo punto il resto del codice (main) apparirà così:

```
public class HelloWorldDecoupled {

    public static void main(String[] args) {
        StandardOutMessageRenderer mr = new StandardOutMessageRenderer();
        HelloWorldMessageProvider mp = new HelloWorldMessageProvider();
        mr.setMessageProvider(mp);
        mr.render();
    }
}
```

A questo punto, la logica del message provider e quella del message renderer sono separate dal resto del codice. Tuttavia, il message renderer e il message provider sono ancora hard-coded (presentano dei valori costanti, e.g. “Hello World”, che non possono essere cambiati senza ricompilare il codice).

Il secondo passo, per rendere tutto ancora più disaccoppiato, consiste nell'uso delle interfacce:

```
public interface MessageProvider {
    public String getMessage();
}

public class HelloWorldMessageProvider implements MessageProvider {
    public String getMessage() {
        return "Hello World!";
    }
}

public interface MessageRenderer {
    public void render();
    public void setMessageProvider(MessageProvider provider);
    public MessageProvider getMessageProvider();
}

public class StandardOutMessageRenderer implements MessageRenderer {
    // MessageProvider è una interfaccia Java ora
    private MessageProvider messageProvider = null;

    public void render() {
        if (messageProvider == null) {
            throw new RuntimeException("You must set the property messageProvider of class:" +
                + StandardOutMessageRenderer.class.getName());
        }
        System.out.println(messageProvider.getMessage());
    }

    public void setMessageProvider(MessageProvider provider) {
        this.messageProvider = provider;
    }

    public MessageProvider getMessageProvider() {
        return this.messageProvider;
    }
}

public class HelloWorldDecoupled {

    public static void main(String[] args) {
        MessageRenderer mr = new StandardOutMessageRenderer();
        MessageProvider mp = new HelloWorldMessageProvider();
        mr.setMessageProvider(mp);
        mr.render();
    }
}
```

Ora è possibile modificare la logica del message renderer senza alcun impatto sulla logica del message provider e viceversa.

Tuttavia, vi sono ancora dei problemi irrisolti, l'uso di differenti implementazioni delle interfacce MessageRenderer o MessageProvider necessita comunque di una modifica (limitata) del codice della logica di business del main.

Per ovviare a tale problema si compie un terzo passo, che consiste nel creare una classe factory che legga, da un **property file**, i nomi delle classi desiderate per le implementazioni delle interfacce e le istanzi a runtime facendo le veci dell'applicazione:

```

public class MessageSupportFactory {
    private static MessageSupportFactory instance = null;
    private Properties props = null;
    private MessageRenderer renderer = null;
    private MessageProvider provider = null;

    private MessageSupportFactory() {
        props = new Properties();
        try {
            props.load(new FileInputStream("msf.properties"));
            String rendererClass = props.getProperty("renderer.class");
            String providerClass = props.getProperty("provider.class");
            renderer = (MessageRenderer) Class.forName(rendererClass).newInstance();
            provider = (MessageProvider) Class.forName(providerClass).newInstance();
        }
        catch (Exception ex) { ex.printStackTrace(); }
    }

    static { instance = new MessageSupportFactory(); }

    public static MessageSupportFactory getInstance() { return instance; }

    public MessageRenderer getMessageRenderer() { return renderer; }

    public MessageProvider getMessageProvider() { return provider; }
}

```

Il main a questo punto diventa:

```

public class HelloWorldDecoupledWithFactory {

    public static void main(String[] args) {
        MessageRenderer mr = MessageSupportFactory.getInstance().getMessageRenderer();
        MessageProvider mp = MessageSupportFactory.getInstance().getMessageProvider();
        mr.setMessageProvider(mp);
        mr.render();
    }
}

```

Per quanto riguarda il **property file** avrà la struttura seguente:

```

# msf.properties

renderer.class=StandardOutMessageRenderer
provider.class=HelloWorldMessageProvider

```

Ora le implementazioni di message provider e message renderer possono essere modificate tramite semplice modifica del file di proprietà. Tuttavia, questa soluzione necessita la scrittura di molto **glue code** (codice che collega parti di codice incompatibili) per mettere insieme l'applicazione, inoltre bisogna scrivere la classe MessageSupportFactory, e infine, l'istanza di MessageProvider deve essere ancora iniettata manualmente nel MessageRenderer.  
L'approccio di Spring a questo problema è mostrato nell'esempio seguente:

```

public class HelloWorldSpring {

    public static void main(String[] args) throws Exception {
        BeanFactory factory = getBeanFactory();
        MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
        MessageProvider mp = (MessageProvider) factory.getBean("provider");
        mr.setMessageProvider(mp);
        mr.render();
    }

    private static BeanFactory getBeanFactory() throws Exception {
        DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
        PropertiesBeanDefinitionReader rdr = new PropertiesBeanDefinitionReader(factory);
        Properties props = new Properties();
        props.load(new FileInputStream("beans.properties"));
        rdr.registerBeanDefinitions(props); return factory;
    }
}

```

I vantaggi raggiunti con questa soluzione sono l'eliminazione di glue code (MessageSupportFactory), una migliore gestione degli errori e meccanismo di configurazione completamente disaccoppiato, sparisce la dipendenza dalla classe, il messaggio HelloWorld non dipende più dalle classi. Rimangono, tuttavia, dei problemi, infatti, il programmatore deve avere conoscenza delle dipendenze del MessageRenderer, deve ottenerle e deve passarle a MessageRenderer. In questo caso Spring agisce come una classe factory sofisticata, nulla di più.

Per risolvere quest'ultimo problema (la dipendenza) si fa uso della Dependency Injection. L'esempio seguente mostra come, lavorando sul file di configurazione, si possa effettuare l'injection in modo dichiarativo evitando di sporcare il codice.

```

# File di configurazione

#Message renderer
renderer.class=StandardOutMessageRenderer
renderer.messageProvider(ref)=provider

#Message provider
provider.class=HelloWorldMessageProvider

```

Il metodo main() deve semplicemente ottenere il Bean MessageRenderer e richiamare render(), non deve ottenere prima il MessageProvider e iniettarlo nel MessageRenderer. Il **wiring** è realizzato automaticamente dalla Dependency Injection di Spring.

```

public class HelloWorldSpringWithDI {

    public static void main(String[] args) throws Exception {
        BeanFactory factory = getBeanFactory();
        MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
        // nota che non è più necessaria nessuna injection manuale
        // del message provider al message renderer
        mr.render();
    }

    private static BeanFactory getBeanFactory() throws Exception {
        DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
        PropertiesBeanDefinitionReader rdr = new PropertiesBeanDefinitionReader(factory);
        Properties props = new Properties();
        props.load(new FileInputStream("beans.properties"));
        rdr.registerBeanDefinitions(props);
        return factory;
    }
}

```

È possibile notare come le classi scritte in quest'ultimo esempio non facciano alcun riferimento a Spring, infatti, in Spring non si ha la necessità di implementare interfacce o estendere classi del framework. Spring, quindi, permette di scrivere delle classi POJO.

Il **property file** citato negli esempi precedenti è, di solito, sostituito da un file XML come il seguente:

```

<beans>
    <bean id="renderer" class="StandardOutMessageRenderer">
        <property name="messageProvider">
            <ref local="provider"/>
        </property>
    </bean>
    <bean id="provider" class="HelloWorldMessageProvider"/>
</beans>

```

Quindi è possibile rielaborare il codice del main come segue

```

public class HelloWorldSpringWithDIXMLFile {

    public static void main(String[] args) throws Exception {
        BeanFactory factory = getBeanFactory();
        MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
        mr.render();
    }

    private static BeanFactory getBeanFactory() throws Exception {
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource("beans.xml"));
        return factory;
    }
}

```

Inoltre, se si volesse cambiare il messaggio da stampare a video si potrebbe utilizzare la Dependency Injection sul costruttore, come nell'esempio seguente:

```

<beans>
    <bean id="renderer" class="StandardOutMessageRenderer">
        <property name="messageProvider">
            <ref local="provider"/>
        </property>
    </bean>
    <bean id="provider" class="ConfigurableMessageProvider">
        <constructor-arg>
            <value>Questo è il messaggio configurabile</value>
        </constructor-arg>
    </bean>
</beans>

```

A livello di codice il ConfigurableMessageProvider esporrà un campo nel costruttore che consentirà di modificare il messaggio, il renderer continuerà a usare il provider. Il ConfigurableMessageProvider sarà una delle possibili implementazioni del MessageProvider, a differenza della precedente implementazione ha la stringa configurabile e non embedded.

```

public class ConfigurableMessageProvider implements MessageProvider {

    private String message;

    public ConfigurableMessageProvider(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
}

```

Infine, è possibile stampare il messaggio “Hello World” a video utilizzando l’Aspect Oriented Programming. Si supponga, per esempio, di avere una classe con un metodo “writeMessage” che stampa a video la parola “World”, l’obiettivo è quello di inserire le parole “Hello” e “!” rispettivamente prima e dopo la parola stampata a video. Si ha la necessità di utilizzare un **around advice** utilizzando come **joinpoint** l’invocazione del metodo “writeMessage” (N.B spring supporta joinpoint solo a livello di metodo).

Gli advice sono scritti in Java (nessun linguaggio AOP-specific) e i pointcut tipicamente specificati in file XML di configurazione. I **pointcut** lavorano come intercettori e consentono se adeguatamente settati di cambiare la logica applicativa.

```

public class MessageWriter implements IMessageWriter {

    public void writeMessage() {
        System.out.print("World");
    }
}

```

```

public class MessageDecorator implements MethodInterceptor {

    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.print("Hello ");
        Object retVal = invocation.proceed();
        System.out.println("!");
        return retVal;
    }
}

public static void main(String[] args) {

    MessageWriter target = new MessageWriter();
    ProxyFactory pf = new ProxyFactory();
    // aggiunge advice alla coda della catena dell'advice
    pf.addAdvice(new MessageDecorator());
    // configura l'oggetto dato come target
    pf.setTarget(target);
    // crea un nuovo proxy in accordo con le configurazioni
    // della factory MessageWriter
    proxy = (MessageWriter) pf.getProxy();
    proxy.writeMessage();
}

```

Gli advice lavorano in maniera molto simile agli intercettori visti in EJB 3.0, infatti anche in Spring è possibile definire degli intercettori.

## 9.5 Gli Intercettori in Spring

L'intercettore Spring può eseguire immediatamente prima o dopo l'invocazione della richiesta corrispondente. La classe di un intercettore implementa l'interfaccia **MethodInterceptor** o estende **HandlerInterceptorAdaptor**.

Nell'esempio sottostante si mostra un tipico uso di intercettore:

```

public class MyService {

    public void doSomething() {
        for (int i = 1; i < 10000; i++) {
            System.out.println("i=" + i);
        }
    }
}

public class ServiceMethodInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation methodInvocation) throws Throwable {

        long startTime = System.currentTimeMillis();
        Object result = methodInvocation.proceed();
        long duration = System.currentTimeMillis() - startTime;
        Method method = methodInvocation.getMethod();
        String methodName = method.getDeclaringClass().getName() + "." + method.getName();
        System.out.println("Method '" + methodName + "' took " + duration + " milliseconds to run");

        return null;
    }
}

```

All'interno del file di configurazione vengono definiti: il componente su cui bisogna agire, l'intercettore, i target dell'operazione di intercettamento ed infine, bisogna specificare quali sono gli intercettori da usare sui ogni target.

```
<beans>
    <bean id="myService" class="com.test.MyService"></bean>
    <bean id="interceptor" class="com.test.ServiceMethodInterceptor"></bean>
    <bean id="interceptedService" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target">
            <ref bean="myService"/> </property>
        <property name="interceptorNames">
            <list>
                <value>interceptor</value>
            </list>
        </property>
    </bean>
</beans>
```

## 9.6 Transazionalità

### 9.6.1 Recap su transazioni

N.B. QUESTO PARAGRAFO NON FA PARTE DEL PROGRAMMA, È STATO TRATTATO A LEZIONE COME RIPASSO PER MEGLIO COMPRENDERE I PARAGRAFI SUCCESSIVI.

Una transazione è un'unità logica di elaborazione che, nel caso generale, si compone di molte operazioni fisiche elementari che agiscono sul DB. Le proprietà di cui deve godere una transazione si riassumono nell'acronimo **ACID** (Atomicity, Consistency, Isolation, Durability):

- **Atomicity** si basa sul principio del tutto o niente cioè o viene eseguita tutta correttamente oppure no.
- **Consistency** è garantita dal DBMS verificando che le transazioni rispettino i vincoli definiti a livello di schema del DB.
- **Isolation** richiede che venga correttamente gestita l'esecuzione concorrente delle transazioni.
- **Durability** le operazioni delle transazioni devono persistere nel tempo.

Date due transazioni queste possono essere: seriali e quindi la somma del tempo di esecuzione delle due transazioni è il tempo totale di esecuzione, oppure concorrenti e in questo caso si riduce il tempo di risposta.

time	T1	T2
1	R(x)	
2	W(x)	
...		
1000	R(x500)	
1001	commit	
1002		R(y)
1003		W(y)
1004		commit

$$\begin{aligned} \text{Tempo medio di risposta} &= \\ &(1001 + (1004-1))/2 \\ &= 1002 \end{aligned}$$

time	T1	T2
1	R(x)	
2		R(y)
3		W(y)
4		commit
5	W(x)	
...		
1003	R(x500)	
1004	commit	

$$\begin{aligned} \text{Tempo medio di risposta} &= \\ &(1004 + 3)/2 \\ &= 503.5 \end{aligned}$$

A sinistra un esempio di due transazioni seriali e il calcolo del tempo medio di risposta, a destra un esempio di transazioni concorrenti e il calcolo del tempo medio di risposta.

Eseguire più transazioni in maniera concorrente è necessario per garantire buone prestazioni: si sfrutta il fatto che, mentre una transazione è in attesa del completamento di una operazione di I/O, un'altra può utilizzare la CPU. La concorrenza però va gestita, se le varie transazioni interferiscono tra di loro si possono avere problemi in lettura e in scrittura. Gli scenari critici possibili dovuti all'esecuzione di transazioni concorrenti sono:

- **Lost update:** Si verifica quando due transazioni accedono allo stesso dato, con conseguente perdita di stato, per esempio due persone comprano entrambe l'ultimo biglietto di un concerto.
- **Dirty read:** Si verifica quando una transazione accede ad un dato, e lo modifica, un'altra transazione legge il dato modificato, ma nel frattempo la prima transazione esegue, per qualche motivo, il rollback riportando il dato al suo valore originale. Il dato letto dalla seconda transazione sarà un dato “sporco” non più consistente con il database.
- **Unrepeatable read:** Si verifica quando il valore di un dato cambia molto frequentemente, questo comporta che due letture consecutive dello stesso dato potrebbero avere valori diversi.
- **Phantom row:** Si verifica quando vengono eseguite due query identiche ma il risultato di una è diverso dal risultato dell'altra.

### Lost Update

Il problema nasce perché T2 legge il valore di X prima che T1 (che lo ha già letto) lo modifichi (entrambe vedono l'ultimo biglietto disponibile).

T1	X	T2
R(x)	I	
X=X-1	I	
	I	R(x)
	I	X=X-1
W(x)	O	
commit	O	
	O	w(x)
	O	commit

### Dirty Read

In questo caso il problema è che una transazione legge un dato che non è consistente con il database. Le operazioni svolte dalla transazione T2 si basano su un valore di X intermedio, e quindi non stabile.

T1	X	T2
R(x)	O	
X=X+1	O	
W(x)	I	
	I	R(x)
rollback	O	
	O	...
	O	...
	O	commit

### Unrepeatable read

Il problema in questo caso è che una transazione legge un dato che ha un certo valore, immediatamente dopo un'altra transazione modifica il valore di quel dato, così quando la prima transazione tornerà a leggere il dato troverà un valore diverso dal primo.

T1	X	T2
R(x)	O	
	O	R(x)
	I	X=X+1
	I	w(x)
	I	commit
R(x)	I	
...	I	
commit	I	

### Phantom row

Questo caso si può presentare quando vengono inserite o cancellate tuple che un'altra transazione dovrebbe logicamente considerare.

**T1:**

```
UPDATE Prog
SET Sede = 'Firenze'
WHERE Sede = 'Bologna'
```

**T2:**

```
INSERT INTO Prog
VALUES ('P03', 'Bologna')
```

Prog	
CodProg	Città
P01	Milano
P01	Bologna
P02	Bologna
P03	Bologna

T1	T2
R(t2)	
R(t3)	
...	
W(t2)	
W(t3)	
	W(t4)
...	
Commit	
	Commit

T1 "non vede" questa tupla!
   


In SQL si può scegliere di operare a diversi livelli di isolamento. SQL definisce quattro livelli di isolamento, che sono **uncommitted read**, **read committed**, **repeatable read**, **serializable**:

- **ISOLATION\_READ\_UNCOMMITTED**: possono accadere dirty read, unrepeatable read e phantom read.

131

- **ISOLATION\_READ\_COMMITTED**: le dirty read sono rese impossibili, possono accadere solo unrepeatable read e phantom read.
- **ISOLATION\_REPEATABLE\_READ**: possibilità delle sole phantom read, dirty e unrepeatable rese non possibili.
- **ISOLATION\_SERIALIZABLE**: tutte le possibilità spiacevoli sono rese impossibili. Viene garantita la proprietà ACID.

## 9.6.2 Le transazioni in Spring

Con scarsa sorpresa in Spring una transazione può essere:

- **Locale**: una transazione specifica per una singola risorsa transazionale, ad esempio un'unica risorsa di un database
- **Globale**: una transazione gestita dal container, questa può includere risorse multiple e distribuite

Il programmatore può specificare in modo dichiarativo che un metodo di Bean deve avere proprietà transazionali. L'implementazione delle transazioni in Spring è basata su AOP, le chiamate ai metodi transazionali vengono intercettate per gestire la transazione. Non c'è nessuna necessità di modificare la logica di business, né al cambio della proprietà di transazionalità desiderata né al cambio del provider di transazionalità.

Spring offre diversi livelli di isolamento delle transazioni, come nell'elenco sopra riportato:

- **ISOLATION\_DEFAULT**
- **ISOLATION\_READ\_UNCOMMITTED**
- **ISOLATION\_READ\_COMMITTED**
- **ISOLATION\_REPEATABLE\_READ**
- **ISOLATION\_SERIALIZABLE**

Spring offre diversi livelli di propagazione delle transazioni (come in EJB 3.0), questi vincoli sono modificabili in modo atomico senza cambiamenti in altre parti, le alternative disponibili sono:

- **PROPAGATION\_REQUIRED**: supporto alla propagazione della transazione di partenza. Crea una nuova transazione se non era transazionale il contesto di partenza.
- **PROPAGATION\_SUPPORTS**: supporto alla propagazione della transazione di partenza, esegue non transazionalmente se la partenza non era transazionale.
- **PROPAGATION\_MANDATORY**: supporto alla propagazione della transazione di partenza; lancia un'eccezione se la partenza non era transazionale.
- **PROPAGATIONQUIRES\_NEW**: crea una nuova transazione, sospendendo quella di partenza, se esistente.
- **PROPAGATION\_NOT\_SUPPORTED**:
- **PROPAGATION\_NEVER**:
- **PROPAGATION\_NESTED**:

## 9.7 Considerazioni avanzate

Alla base dell'architettura Spring, c'è l'idea di Inversion of Control (prima che in EJB 3.0) e di container leggeri realizzati attraverso le factory per l'istanziazione, il ritrovamento e la gestione delle relazioni fra oggetti. Le factory supportano due modalità di oggetto:

- **Singleton (default):** in questa modalità viene creata un'unica istanza, condivisa, dell'oggetto. Un cliente C1 richiede alla BeanFactory un'istanza logica di un componente X, la BeanFactory crea un'istanza, la inizializza, risolve le dipendenze e la restituisce al cliente C1. Se un secondo cliente C2 richiede alla BeanFactory un'istanza del componente X, allora la gestione a Singleton porterà la BeanFactory a restituire al cliente C2 il riferimento all'istanza generata in precedenza. La modalità a Singleton è ideale per oggetti stateless (che non richiedono stato e possono essere condivisi), in modo da ridurre la proliferazione di istanze nel codice applicativo.
- **Prototype:** in questa modalità ogni operazione di ritrovamento di un oggetto, produrrà la creazione di una nuova istanza. Questa modalità è utile per gestire oggetti stateful in modo da far avere ad ogni cliente un'istanza distinta.

Un bean di Spring può essere lui stesso una factory, questo permette di aggiungere un ulteriore livello di indirezione. Viene usato di solito per creare oggetti con proxy.

La possibilità di semplice Dependency Injection tramite costruttori o metodi semplifica il testing delle applicazioni Spring, per esempio, è semplice scrivere un test JUnit che crea l'oggetto Spring e configura le sue proprietà al fine di eseguire il testing.

Il container **IoC** non è invasivo: molti oggetti di business non dipendono dalle API di invocazione del container, ed inoltre è facile “introdurre” vecchi POJO in ambiente Spring.

## 9.8 Autowiring

Spring può occuparsi automaticamente di risolvere le dipendenze tramite introspezione delle classi Bean. In questo modo, il programmatore può evitare di specificare esplicitamente le proprietà del Bean o gli argomenti del costruttore (tag `<ref>`) nel file XML. Le proprietà del Bean sono risolte attraverso il matching basato su nome o su tipo:

- **autowire = “name”:** l'autowiring viene fatto sui nomi delle proprietà, cioè sui metodi setter del bean (`setNomeProprieta()`).
- **autowire = “type”:** l'autowiring viene fatto sui tipi di proprietà del bean, cioè sui tipi degli argomenti dei metodi setter (`setNomeProprieta( Type arg)`).
- **autowire = “constructor”:** l'autowiring è fatto sui tipi degli argomenti del costruttore.

Questo meccanismo ricorda molto la gestione di default delle dipendenze in EJB 3.0 in cui le dipendenze erano risolte attraverso JNDI.

## 9.9 Dependency Checking

Il dependency checking è utilizzabile per controllare l'esistenza di dipendenze non risolte senza ricevere spiacevoli sorprese quando è già fatto il deployment di un Bean all'interno di un container Spring. È utile per tutte le proprietà che non hanno valori configurati all'interno della definizione del Bean, per i quali anche l'autowiring non ha prodotto alcun setting. Questa è una caratteristica utile quando ci si vuole assicurare che tutte le proprietà (o tutte le proprietà di un determinato tipo) siano state configurate correttamente su un Bean. Vi sono diverse modalità possibili e configurabili:

- **None**: nessun tipo di check.
- **Simple**: dependency checking effettuato solo per tipi primitivi e collection.
- **Object**: dependency checking effettuato solo per le dipendenze da altri bean all'interno della stessa factory.
- **All**: dependency checking effettuato per tutto.

## 9.10 Application Context

Per accedere ad alcune funzionalità avanzate di Spring, non è sufficiente l'uso della semplice interfaccia BeanFactory, l'**ApplicationContext** è un'estensione dell'interfaccia **BeanFactory**. L'**ApplicationContext** oltre a fornire tutte le funzionalità base, integra, ad esempio, la gestione delle transazioni e di AOP. **ApplicationContext** si utilizza in modalità più tradizionale cioè invocando direttamente dal codice la chiamata di cui si ha bisogno. Le funzionalità aggiuntive dell'**ApplicationContext** non sono supportate di base dal BeanFactory perché nell'ottica di container leggero questo appesantirebbe il container. Tra le funzionalità aggiuntive di **ApplicationContext** vi sono ad esempio:

- Interfacce per la gestione del ciclo di vita.
- Propagazione di eventi a bean che implementano l'interfaccia **ApplicationListener**
- Accesso a risorse come URL e file

### Gestione del ciclo di vita

La gestione del ciclo di vita è basata su delle interfacce standard, ad esempio, l'interfaccia **ApplicationContextAware**. Un bean che implementa tale interfaccia avrà il metodo **setApplicationContext()** che verrà automaticamente invocato nel momento in cui il bean viene creato. Il bean riceverà in questo modo un riferimento al contesto su cui potrà effettuare invocazioni.

```
public class Publisher implements ApplicationContextAware {

    private ApplicationContext ctx;
    // Questo metodo sarà automaticamente invocato da IoC container
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        this.ctx = applicationContext;
    }
}
```

## Propagazione degli eventi

La gestione degli eventi è realizzata tramite la classe **ApplicationEvent** e l'interfaccia **ApplicationListener** che consente la ricezione degli eventi.

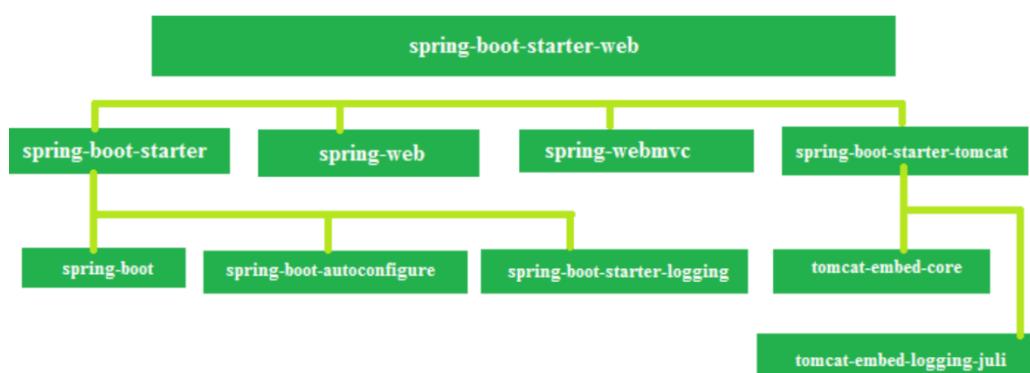
Se viene fatto il deployment, in un **ApplicationContext** "ac1", di un bean che implementa l'interfaccia **ApplicationListener**, il bean viene notificato ogni volta che un **ApplicationEvent** viene pubblicato in "ac1". Questa logica è quella del design pattern Observer.

Ci sono tre tipologie di eventi built-in (integrati) in Spring:

- **ContextRefreshEvent**: evento che avviene all'inizializzazione o refresh dell'application context.
- **ContextClosedEvent**: evento che avviene alla chiusura dell'application context.
- **RequestHandleEvent**: evento specifico per il web che avviene quando una richiesta HTTP è stata servita.

## Spring Boot

Spring Boot è una tecnologia di recente successo per accelerare lo sviluppo di nuove applicazioni Spring, sostanzialmente fornisce supporto alla configurazione e integrazione con DevOps. I principali package di Spring Boot possiedono anche con integrazione parte Web. Spring boot segue unPattern mvc.



# 10. Java Management Extensions (JMX)

## 10.1 Introduzione ai monitoring systems

Come già detto più volte, l'interesse del corso volge non solo alla fase di sviluppo, ma anche alla fase di deployment, configurazione ed esecuzione in ambiente reale dell'architettura enterprise. In questo caso l'attenzione si focalizza sul monitoraggio del sistema vi è quindi la necessità di controllo on-line e conseguenti azioni di gestione, non solo di network equipment, ma anche di componenti applicativi e di servizio.

Gli obiettivi del monitoraggio sono:

- La fault detection,
- La misura delle performance e riconfigurazione/re-deployment,
- L'identificazione dei colli di bottiglia.

I modelli utilizzati nel monitoraggio sono i seguenti:

- push o pull,
- azioni reattive o proattive,
- approccio ottimistico o pessimistico,
- con manager centralizzato o parzialmente distribuito o completamente distribuito.

Per il sistema di monitoraggio è importante avere protocolli e API standard per misurare le performance trattando in un certo senso il software come l'hardware dal punto di vista del monitoraggio e della misurazione.

Il protocollo **SNMP** viene impiegato nella gestione e nel management del mondo del networking, esso rappresenta uno standard, allo stesso modo di quanto accade nel modo del networking anche nel mondo distribuito in ambienti aperti e interoperabili si cerca di avere standard per il monitoraggio. Per fare un esempio, la **Distributed Management Task Force** (DMTF) è un'organizzazione per la standardizzazione di IT System Management in ambienti industriali e su Internet. Gli standard DMTF permettono la costruzione di componenti per System Management indipendenti dalla piattaforma e technology-neutral, abilitando così interoperabilità fra prodotti per la gestione di sistemi di diversi vendor.

Alcuni elementi fondamentali in DMTF sono:

- **Common Information Model (CIM)**: è un modello astratto per la rappresentazione degli elementi gestiti (ad esempio, computer o storage area network) come insieme di oggetti e relazioni. Il CIM svolge la stessa funzione del CMIB per SNMP, inoltre è estensibile in modo da consentire l'introduzione di estensioni product-specific.
- **Common Diagnostic Model (CDM)**: è un modello di diagnostica e definizione di come questo debba essere incorporato nell'infrastruttura di management.
- **Web-Based Enterprise Management (WBEM)**: è un insieme di protocolli per l'interazione fra componenti di system management (conformi a CIM e ai suoi profili) e la loro interrogazione. I profili in genere sono utilizzati in data-model molto estesi con diversi diritti e operazioni.

## 10.2 Java Management Extensions JMX

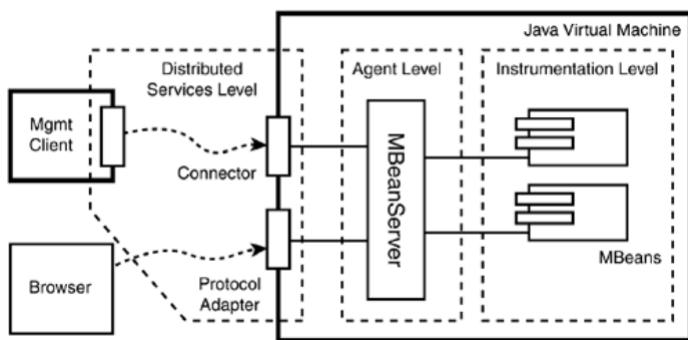
Il contesto nel quale si è inserita la specifica Java è quella del monitoraggio nell'ambiente distribuito, prima del framework **Java Management Extensions (JMX)** non vi era nessun approccio standardizzato in Java per far partire, gestire, monitorare e fermare l'esecuzione di componenti software.

I componenti software conformi alla specifica JMX vengono chiamati **MBean** ovvero **Managed Bean**, gli MBean sono gestiti attraverso un agente che svolge il ruolo di registry. Tale registry offre alle applicazioni di management, ovvero i clienti di tale agente, un modo di effettuare query, interrogare i bean e modificare i bean gestiti. L'utilizzo dell'agente consente di non avere collegamenti diretti tra componenti Mbean e cliente; quindi, l'agente rende trasparenti gli MBean, in modo che il cliente possa demandare all'agente intermedio alcune funzionalità di management, ciò produce un'ottimizzazione poiché i Managed Bean sono locali rispetto all'agente permettendo al cliente di comunicare attraverso internet.

### 10.2.1 Architettura JMX

JMX è organizzato secondo un'architettura a tre livelli:

- I componenti gestiti appartengono al livello **Instrumentation**
- Il livello agente è costituito dal registro per gli MBean (**MBeanServer**) e da alcuni servizi standard addizionali necessari poiché non c'è mai un collegamento diretto tra cliente ed MBean
- Il livello dei servizi distribuiti è costituito da adattatori e connettori (**Adaptor** e **Connector**), necessari per supportare l'accesso remoto al livello agente, verso web-server o altre interazioni.



### 10.2.2 Livello Instrumentation

Il livello Instrumentation definisce come creare risorse gestibili tramite JMX (MBeans), ovvero oggetti che offrono metodi per la gestione di un'applicazione, gestione di un componente software, gestione di un servizio, gestione di un dispositivo.

MBean non è altro che un componente che implementa una interfaccia di gestione, staticamente o dinamicamente. Nel primo caso, JMX implementa un'interfaccia Java standard statica e l'agente ne fa **inspection** tramite tecniche di **reflection** e convenzioni sui nomi, in questo modo è possibile interrogare l'Mbean. Nel secondo caso, quello dinamico, JMX offre un insieme di oggetti metadata attraverso i quali l'agente riesce a scoprire i metodi di management esposti. Le

interazioni dinamiche sono simili a quelle presenti in Corba. Questi metadati consentono di recuperare i metodi di management. I metadati risultano essere più dinamici rispetto alle interfacce, perché queste non sono modificabili.

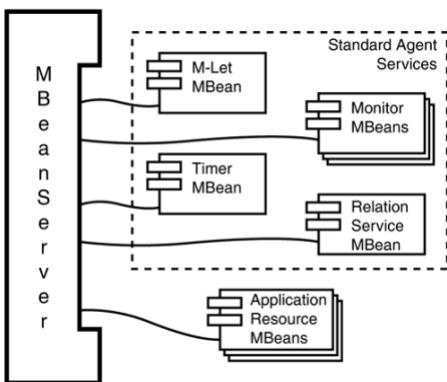
JMX definisce quattro tipi di componenti MBean:

- **Standard MBean:** è un Mbean statico, creato dichiarando esplicitamente una interfaccia Java con l'informazione di management che l'oggetto gestito implementa.
- **Dynamic MBean:** più dinamico, è un oggetto che implementa l'interfaccia **DynamicMBean** e offre la descrizione dei suoi veri metodi di management attraverso un insieme di oggetti metadata che tale interfaccia richiede di fornire.
- **Model MBean:** è un DynamicMBean esteso con descrittori addizionali che definiscono proprietà aggiuntive come behavioral properties, funzionalità orizzontali di persistenza, sicurezza, eccetera.
- **Open MBean:** (non di implementazione obbligatoria per essere conformi alla specifica) è un MBean in cui i tipi utilizzati nei metodi di management hanno il vincolo ulteriore di essere inclusi in un set predefinito di classi e tipi di base, sottoinsieme di funzionalità standardizzate.

Il supporto ai primi tre tipi di MBean è mandatory a partire dalla specifica JMX 1.0, si ricorda che JMX è una specifica, con diverse implementazioni possibili, come quella di Sun o IBM Tivoli.

### 10.2.3 Livello Agente

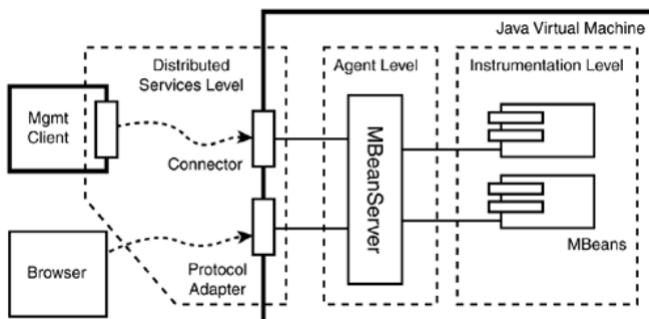
Il livello di agente è costituito da un server MBean e da un insieme di servizi di agente basati sul livello di Instrumentation, vi sono quattro servizi agente definiti nella specifica JMX: **M-Let**, **Timer**, **Monitoring** e **Relation**. Inoltre, il livello agente introduce il concetto di naming per gli oggetti, nomi che il lato cliente può utilizzare come riferimento indiretto alle risorse gestite.



Il server MBean è uno dei componenti chiave dell'architettura di management, esso opera come un canale di comunicazione che smista e delega tutte le invocazioni fra applicazioni di management e risorse gestite. Espone metodi per la creazione ed effettuazione di query, per invocare operazioni e per manipolare attributi su MBean. Il tipo di implementazione dei componenti MBean (Standard, Dynamic, Model, ...) è totalmente trasparente alle applicazioni client-side di gestione, l'Mbean Server nascondendo tale implementazione semplifica anche le cose lato client.

**MBeanServer** è un oggetto locale alla JVM dei componenti gestiti e non offre particolare supporto alla connessione remota verso di sé. Per questo servono connettori JMX o adattatori

di protocollo per accettare chiamate provenienti dall'esterno della JVM. Questi componenti, connettori o adattatori, sono spesso essi stessi degli MBean, registrati sull'agente, e forniscono una pluralità di differenti forme di connettività.



I connettori JMX sono strutturati in due componenti: lato server l'agente registra un server per le connessioni capace di ricevere invocazioni remote di metodo, invece, lato cliente, si può utilizzare una vista remota del server MBean per invocare operazioni su di esso.

La comunicazione può avvenire con una varietà di protocolli, in questo esempio si utilizza SOAP/HTTP, già visto in EJB come Web Services.

Gli adattatori di protocollo sono implementati solo lato server MBean, questi possono adattare operazioni server MBean del mondo JMX per interagire con protocolli preesistenti, o anche verso diversi modelli di informazioni, come SNMP Management Information Base, permettendo ad app di management legacy o a strumenti non-Java di interoperate con JMX.

#### 10.2.4 Livello servizi distribuiti

La specifica JMX Remote API definisce come si possa fare advertising e trovare agenti JMX usando infrastrutture di discovery e lookup esistenti, la specifica NON definisce un ulteriore servizio di discovery e lookup, ma la tecnologia JMX offre una soluzione standard per l'esportazione delle API di JMX instrumentation verso applicazioni remote, basata su RMI. Inoltre, JMX Remote API definisce un protocollo opzionale, non-mandatory e più efficiente, basato direttamente su socket TCP, chiamato **JMX Messaging Protocol** (JMXMP) per garantire la massima interoperabilità.

Gli MBean sono stati progettati per essere flessibili, semplici e facili da implementare. Gli sviluppatori di applicazioni, servizi di supporto e dispositivi possono rendere i loro prodotti gestibili (manageable) in modo standard, senza necessità di conoscere a fondo e di investire in sistemi complessi di management. Gli oggetti esistenti possono facilmente essere estesi per produrre MBean standard o essere oggetto di wrapping come MBean dinamici, rendendo così le risorse esistenti facilmente gestibili a basso costo.

### 10.3 Standard MBean

Lo standard MBean è il modo più semplice per rendere JMX-managed nuove classi Java. L'interfaccia statically-typed dichiara esplicitamente gli attributi tramite metodi getter e setter e operazioni di gestione. Per gli standard MBean è presente una convenzione sui nomi: quando un managed object viene registrato, l'agente cerca una interfaccia di management con lo stesso nome classe dell'oggetto + suffisso il MBean, nel caso se necessario, navigando l'albero di ereditarietà della classe.

```

public interface UserMBean{
    public long getId();
    public void setId(long id);
    public boolean isActive();
    public void setActive(boolean active);
    public String printInfo();
}
public class User implements UserMBean { ... }
public class Student extends User {
    /* anche questa classe può essere registrata come un UserMBean */
    ...
}

```

### 10.3.1 MBean Server registrazione

Per registrare un manageable object come un MBean è necessario creare prima un **ObjectName**. Il riferimento all'agente può essere ottenuto da una lista di implementazioni disponibili di MBeanServer o creandolo da zero. La registrazione di MBean consiste semplicemente nell'associare il manageable object con il suo nome di oggetto, una volta trovato l'MBean server.

```

ObjectName username = new ObjectName("example:name=user1");
List serverList = MBeanServerFactory.findMBeanServer(null);
MBeanServer server = (MBeanServer)serverList.iterator().next();
// Oppure per la creazione...
// MBeanServer server = MBeanServerFactory.createMBeanServer();
server.registerMBean(new User(), username);

```

### 10.3.2 Invocazione servizi di gestione

L'applicazione di management riferisce MBean passando un riferimento all' **ObjectName** all'agente, per ogni operazione invocata. Il server MBean cerca il riferimento Java corrispondente a MBean nel suo repository interno e invoca l'operazione corrispondente (o la modifica dell'attributo) sull'MBean, nascondendo tutte queste operazioni al cliente.

```

ObjectName username = new ObjectName("example:name=user1");
Object result = server.invoke( username, // nome MBean
    "printInfo", // nome operaz
    null, // no param
    null); // void signature

```

### 10.3.3 Meccanismo di notifica

L'architettura JMX definisce un meccanismo di notifica per MBean che consente di inviare eventi verso altri MBean o applicazioni di management. Gli MBean che vogliono emettere eventi di management devono implementare l'interfaccia **NotificationBroadcaster**, gli oggetti listener per gli eventi devono, invece, implementare l'interfaccia **NotificationListener** e devono effettuare la loro subscription presso Mbean (locale o remoto) che fa da broadcaster, questa subscription è fatta attraverso il livello di agente. Le operazioni di notifica svolte da broadcaster MBean sono parte della loro interfaccia di management JMX: le applicazioni possono effettuare query sul livello agente per avere informazioni sui tipi di notifica che possono emettere gli MBean di interesse, a tal fine, gli MBean broadcaster forniscono oggetti **MBeanNotificationInfo**.

La classe JMX Notification estende **EventObject** introducendo campi per il tipo di evento, numero di sequenza, timestamp, messaggio e dati utente opzionali, perciò le notifiche possono essere filtrate, grazie all'implementazione dell'interfaccia **NotificationFilter** che è subscribed presso broadcaster MBean, insieme con il listener che possiede il metodo **addNotificationListener()**. Il broadcaster deve controllare se la notifica supera il filtro prima di inviarla.

```
public interface NotificationFilter {
    public boolean isNotificationEnabled( Notification notification);
}
public interface NotificationBroadcaster {
    public void addNotificationListener( NotificationListener listener, NotificationFilter filter, Object handback)
            throws IllegalArgumentException;
}
```

Poiché l'implementazione di broadcaster MBean può diventare anche piuttosto complessa, è messa a disposizione la classe **NotificationBroadcasterSupport** che implementa l'interfaccia **NotificationBroadcaster**. In questo modo i broadcaster MBean possono: estendere tale classe per ereditare quella implementazione dei metodi di broadcasting, oppure delegare a questa classe il supporto alla gestione delle registrazioni e all'invocazione delle notifiche. Il meccanismo di notifica è generico e adatto a qualsiasi tipo di notifica user-defined. Comunque, JMX definisce la specifica classe **AttributeChangeNotification** per MBean che vogliono inviare notifiche sul cambiamento dei loro attributi di management, l'idea di questa classe è facilitare la creazione di amenti di notifica per il cambio degli attributi.

## 10.4 Dynamic MBean

Gli MBean dinamici implementano l'interfaccia generica **DynamicMBean** che offre metodi all'agente per fare il discovery di metodi e attributi di management (reale interfaccia di gestione), l'agente del server può usare questa interfaccia per recuperare gli attributi degli Mbean. I metadati che descrivono l'interfaccia di management sono completamente sotto la responsabilità dello sviluppatore, mentre nello Standard MBean i metadati vengono generati dall'agente stesso tramite introspezione. I casi possibili di utilizzo di Dynamic MBean sono ad esempio le situazioni in cui l'interfaccia di management può cambiare spesso, oppure per la necessità di abilitare facilmente il management su risorse esistenti.

In MBean dinamici l'interfaccia di management viene esposta tramite le classi di metadata definite in JMX API, i metadata sono ritrovati dinamicamente dall'agente come istanza della classe **MBeanInfo**, tale classe include tutti gli elementi di metadata, i quali ereditano caratteristiche comuni dalla classe **MBeanFeatureInfo**.

```

public class DynamicUser extends NotificationBroadcasterSupport implements DynamicMbean {

    // Attributi
    final static String ID = "id";
    private long id = System.currentTimeMillis();
    public Object getAttribute(String attribute)
        throws AttributeNotFoundException, MBeanException, ReflectionException {
        if (attribute.equals(ID))
            return new Long(id);
        throw new AttributeNotFoundException("Missing attribute " + attribute);
    }
    // Operazioni
    final static String PRINT = "printInfo";
    public String printInfo() {
        return "Sono un MBean dinamico";
    }
    public Object invoke(String actionPerformed, Object[] params, String[] signature) throws ... {
        if (actionName.equals(PRINT))
            return printInfo();
        throw new UnsupportedOperationException("Unknown operation" + actionPerformed);
    }
    // da definire all'interno della classe DynamicUser
    public MBeanInfo getMBeanInfo() {

        final boolean READABLE = true;
        final boolean WRITABLE = true;
        final boolean IS_GETTERFORM = true;
        String classname = getClass().getName();
        String description = "Sono un MBean dinamico";

        MBeanAttributeInfo id = new MBeanAttributeInfo
            (ID, long.class.getName(), "id", READABLE, !WRITABLE, !IS_GETTERFORM);
        MBeanConstructorInfo defcon = new MBeanConstructorInfo("Default", "Creates", null);
        MBeanOperationInfo print = new MBeanOperationInfo
            (PRINT, "Prints info", null, String.class.getName(), MBeanOperationInfo.INFO);

        return new MBeanInfo(classname, description,
            new MBeanAttributeInfo[] { id },
            new MBeanConstructorInfo[] { defcon },
            new MBeanOperationInfo[] { print },
            null);
    }
}

```

## 10.5 ModelMBean

I model MBean sono estensioni di MBean dinamici, essi forniscono un template generico per creare un'implementazione di gestione per risorse esistenti, separare l'implementazione di management dall'implementazione della risorsa, ed estendere metadata di gestione per fornire informazioni addizionali e proprietà behavioral per la gestione di queste funzionalità trasversali: come proprietà di caching, proprietà di sicurezza, proprietà di transazionalità, proprietà di persistenza. Tutte le implementazioni di JMX MBean server devono fornire almeno una implementazione dell'interfaccia **ModelMBean** tramite la classe **RequiredModelMBean**.

Per aggiungere politiche di gestione ai Model Mbean si utilizza l'interfaccia **Descriptor**, gli oggetti che implementano l'interfaccia Descriptor sono usati nei metadati di Model MBean per aggiungere politiche: Politiche specifiche per la particolare implementazione dell'agente JMX, la specifica JMX definisce alcuni comportamenti standard, implementazioni di Model MBean possono essere estese per supportare comportamenti custom. Un descrittore è una collezione di coppie nome-valore in base alle quali l'implementazione dell'agente adatta il suo comportamento, le classi di metadata di Model MBean estendono le classi corrispondenti usate con MBean dinamici e standard (implicitamente) e implementano l'interfaccia **DescriptorAccess**.

## 10.6 Servizi Standard a Livello di Agente

La specifica JMX definisce quattro servizi distinti a livello di agente che devono essere disponibili su ogni implementazione conforme alla specifica:

- **M-Let Service** che permette agli MBean di essere caricati dalla rete e inclusi nel livello di agente dinamicamente a runtime;
- **Timer Service** è uno scheduler che si occupa dell'invio di notifiche agli altri MBean;
- **Monitoring Service**: è un MBean che svolge il ruolo di osservatore per gli attributi di management degli altri bean e che notifica le modifiche avvenute;
- **Relation Service** permette di creare associazioni fra MBean e mantiene la loro consistenza.

### 10.6.1 M-let service

M-let service gestisce il loading dinamico di nuove classi Java dal server MBean, che possono trovarsi su macchina locale o su macchina remota, con l'idea di spostare delle configurazioni di una applicazione verso un server remoto. Come ogni altro standard MBean, l'interfaccia **MLetMBean** espone le operazioni di management considerate rilevanti per il servizio, come **addURL()** e **getMBeansFromURL()**. All'URL specificato da addURL() si trovano i file di testo M-Let che descrivono i componenti MBean tramite MLET tag.

```
<MLET CODE = class | OBJECT =
    serfile
    ARCHIVE = "archiveList"
    [CODEBASE = codebaseURL]
    [NAME = MBeanName]
    [VERSION = version] >
    [arglist]
</MLET>
```

Ad esempio, si vuole fare un'installazione JMX in diversi uffici di un'azienda e prevede che in una rete locale ci sia un MBean Server che abbia servizi A, B e C a livello agente. Senza M-let, bisognerebbe installare su tutti i nodi tutti i servizi A, B e C ma grazie a M-let si può velocizzare il processo. Un possibile file potrebbe essere:

```
<MLET CODE=com.mycompany.Foo
      ARCHIVE="MyComponents.jar,acme .jar"
</MLET>
```

### 10.6.2 Servizio di timer

Il servizio di Timer è basato sul meccanismo di notifica di JMX. **TimerMBean** è un broadcaster MBean in grado di emettere eventi, per ricevere notifiche dal timer il consumatore deve implementare l'interfaccia **NotificationListener** e registrarsi. Questo è analogo al servizio di Cron in Unix/Linux o a Task Scheduler Service su Windows NT. Per impostare il tutto è possibile

recuperare il servizio di timer quindi una volta registrato presso il server si può richiedere l'avvio del timer.

```
// fa partire il servizio di timer
List list = MBeanServerFactory.findMBeanServer(null);
MBeanServer server = (MBeanServer)list.iterator().next();
ObjectName timer = new ObjectName("service:name=timer");
server.registerMBean(new Timer(), timer);
server.invoke(timer, "start", null, null);

// configurazione di notification time
Date date = new Date(System.currentTimeMillis() + Timer.ONE_SECOND * 5);

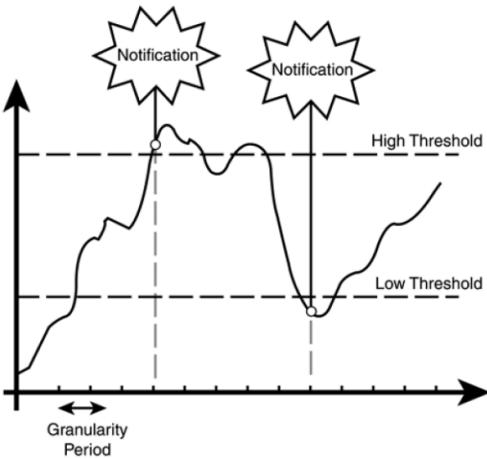
server.invoke(timer, // MBean
"addNotification", // metodo
new Object[] { // args
    "timer.notification", // tipo
    "Schedule notification", // messaggio
    null, // user data
    date}, // time
new String[] { String.class.getName(),
    String.class.getName(),
    Object.class.getName(), // signature
    Date.class.getName()} );

// registra il listener MBean
server.addNotificationListener(timer, this, null, null);
```

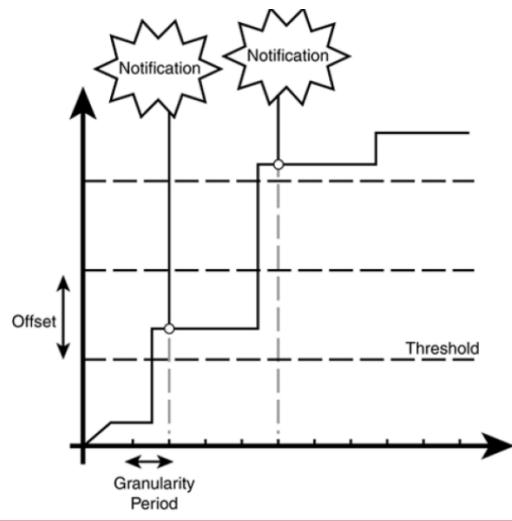
### 10.6.3 Servizio di monitoring

Il servizio di monitoring è un insieme di MBean che possono essere utilizzati per effettuare il monitoring degli attributi di risorse gestite. Le notifiche dei monitor differiscono dalle usuali notifiche di modifica di attributi perché si possono introdurre threshold e periodi di granularità. Vi sono tre implementazioni differenti: **counter monitor** traccia le variazioni di attributi che si comportano come contatori con variazioni discrete, **gauge monitor** per attributi integer e float, ad intervalli di granularità configurabile, con threshold, **string monitor** per informare in relazione a string matching/dismatching rispetto a valori attesi, si fissano dei valori o delle stringhe, le notifiche vengono scatenate in caso di matching o mismatching.

Si può riscontrare un problema se vi sono attributi “rumorosi”, in questi casi è impensabile ad ogni variazione di segnalare il nuovo stato al listener, per questo si cerca di fare monitoring con delle isteresi di tempo o di confidenza sugli attributi. Gli eventi sono sollevati solo quando vi sono variazioni significative. Le notifiche partono o dopo un periodo di tempo o al superamento di una certa soglia. Queste threshold servono per evitare un overhead del sistema di monitoraggio, che non deve occupare più del 10% del carico del sistema, così si vogliono evitare effetti di schizofrenia nelle azioni di gestione dipendenti dal monitoraggio.



Esempio per variazioni continue, dove vengono utilizzate soglie, in questo caso una alta e una bassa. L'emissione della notifica è sincronizzata con il periodo. Il tutto è sincronizzato dai granularity period quindi le notifiche sono date solo allo scattare di un nuovo periodo.



Esempio per variazioni discrete.

#### 10.6.4 Servizio Relation

Il Servizio di relation permette di definire relazioni fra MBean e di reagire a modifiche (per esempio nel caso di dipendenze). La consistenza delle relazioni viene mantenuta tramite la definizione di ruoli per gli MBean e associando o disassociando oggetti MBean a ruoli differenti nelle relazioni. Le notifiche sono emesse al momento della modifica nelle istanze di relazione (creazione, aggiornamento, rimozione, ...).

#### 10.7 JMX remote API

Dall'architettura mostrata all'inizio si può notare come sia presente una parte client e server con disaccoppiamento forte per ottenere la massima flessibilità delle interazioni dei client con Mbean che siano conformi a JMX. Esistono connettori offerti da RMI che offrono uno stub ovvero un end-point che può essere interrogato dal client. Inoltre, è disponibile un Registry RMI

dove reperire l'istanza dello stato. Per effettuare operazioni remote su MBean, un server per connettori RMI è a disposizione lato server: tramite chiamata alla classe **JMXServiceURL** si crea un nuovo URL di servizio (indirizzo per il server di connector). Il server di connector RMI è creato via **JMXConnectorServerFactory**, con parametri URL di servizio e MBeanServer. Il server di connector deve essere messo in esecuzione. L'URL (in formato JNDI) indica dove reperire uno stub RMI per il connettore, tipicamente in un directory riconosciuto da JNDI come RMI registry o LDAP: il connettore usa il trasporto di default RMI, il registry RMI in cui lo stub è memorizzato risponde alla porta 9999 su localhost, e l'indirizzo del server è registrato al nome “server”.

```
// lato servitore
MBeanServer mbs = MBeanServerFactory.createMBeanServer();
JMXServiceURL url = new JMXServiceURL("service:jmx:rmi:///jndi/rmi://" + "localhost: 9999/server");
JMXConnectorServer cs = JMXConnectorServerFactory.newJMXConnectorServer(url, null, mbs);
cs.start();
```

Il cliente crea un **RMI Connector Client** configurato per connettersi al **RMI connector** creato lato server: l'URL di servizio utilizzato deve fare match con quello usato alla registrazione del servizio di connector. Il connector client è restituito come risultato della connessione al connector server, il cliente ora può registrare MBean ed effettuare operazioni su di essi tramite MBeanServer remoto in modo trasparente alla distribuzione.

```
// lato cliente
JMXServiceURL url = new JMXServiceURL(service:jmx:rmi:///jndi/rmi://" + "localhost:9999/server");

JMXConnector jmxc = JMXConnectorFactory.connect(url, null);

MBeanServerConnection mbsc = jmxc.getMBeanServerConnection;

mbsc.createMBean(...);
```

Oltre agli usuali connettori standard RMI e RMI/IOP, si possono utilizzare connettori **JMXMP** (ad esempio per disporre di un livello di sicurezza maggiore tramite meccanismo SSL).

Per quanto riguarda gli adattatori di protocollo, sebbene non siano definiti da specifica, non esistono particolari implementazioni diffuse. Un adattatore di protocollo è un pezzettino software che sta lato MBean Server ed è in grado di ricevere richieste in un protocollo che non è conforme con JMX, e di tradurle nelle chiamate corrispondenti del MBean Server.

## 10.8 Esempio di utilizzo di MBean

```
package com.example.mbeans;

public interface HelloMBean {

    // operazioni (signature)
    public void sayHello();
    public int add(int x, int y);

    // attributi
    public String getName();
    public int getCacheSize();
    public void setCacheSize(int size);
}
```

```
package com.example.mbeans;

public class Hello implements HelloMBean {

    public void sayHello() {
        System.out.println("hello, world");
    }

    public int add(int x, int y) {
        return x + y;
    }

    /* metodo getter per l'attributo Name.
     * Spesso gli attributi sono utilizzati per fornire indicatori di monitoraggio
     * come uptime o utilizzo di memoria. Spesso sono read-only.
     * In questo caso l'attributo è una stringa */
    public String getName() {
        return this.name;
    }

    /* invece anche metodi getter e setter */
    /* invece anche metodi getter e setter */
    public int getCacheSize() {
        return this.cacheSize;
    }

    /* perché synchronized? Mantenere uno stato consistente per evitare modifiche concorrenti.
     * No notifiche concorrenti, non ci sono container che si occupano della sincronizzazione
     * quindi serve synchronized, prima non necessario con i container si occupano
     * internamente della sincronizzazione */
    public synchronized void setCacheSize(int size) {
        this.cacheSize = size;
        System.out.println("Cache size now " + this.cacheSize);
    }

    private final String name = "My First MBean";
    private int cacheSize = DEFAULT_CACHE_SIZE;
    private static final int DEFAULT_CACHE_SIZE = 200;
}
```

```

package com.example.mbeans;
import java.lang.management.*;
import javax.management.*;

public class Main {

    public static void main(String[] args) throws Exception {
        // ottiene il server MBean
        MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
        // costruisce ObjectName per MBean da registrare
        ObjectName name = new ObjectName("com.example.mbeans:type=Hello");
        // crea istanza di HelloWorld MBean
        Hello mbean = new Hello();
        // registra l'istanza
        mbs.registerMBean(mbean, name);
        System.out.println("Waiting forever...");
        Thread.sleep(Long.MAX_VALUE);
    }
}

```

```

package com.example.mbeans;
import javax.management.*;

public class Hello extends NotificationBroadcasterSupport implements HelloMBean {

    public void sayHello() {
        System.out.println("hello, world");
    }

    public int add(int x, int y) {
        return x + y;
    }

    public String getName() {
        return this.name;
    }

    public int getCacheSize() {
        return this.cacheSize;
    }

    public synchronized void setCacheSize(int size) {
        int oldSize = this.cacheSize;
        this.cacheSize = size;

        /* In applicazioni reali il cambiamento di un attributo di solito produce effetti di gestione.
         * Ad esempio, cambiamento di dimensione della cache può generare eliminazione o allocazione di entry */
        System.out.println("Cache size now " + this.cacheSize);
        /* Per costruire una notifica che descrive il cambiamento avvenuto: "source" è ObjectName di MBean
         * che emette la notifica (MBean server sostituisce "this" con il nome dell'oggetto);
         * mantenuto un numero di sequenza */
        Notification n = new AttributeChangeNotification( this, sequenceNumber++, System.currentTimeMillis(),
                "CacheSize changed", "CacheSize", "int", oldSize, this.cacheSize);
        /* Invio della notifica usando il metodo sendNotification() ereditato dalla superclasse */
        sendNotification(n);
    }

    ...
}

```

```

@Override
/* metadescrizione */
public MBeanNotificationInfo[] getNotificationInfo() {

    String[] types = new String[] { AttributeChangeNotification.ATTRIBUTE_CHANGE };
    String name = AttributeChangeNotification.class.getName();
    String description = "è stato cambiato un attributo!";
    MBeanNotificationInfo info = new MBeanNotificationInfo(types, name, description);

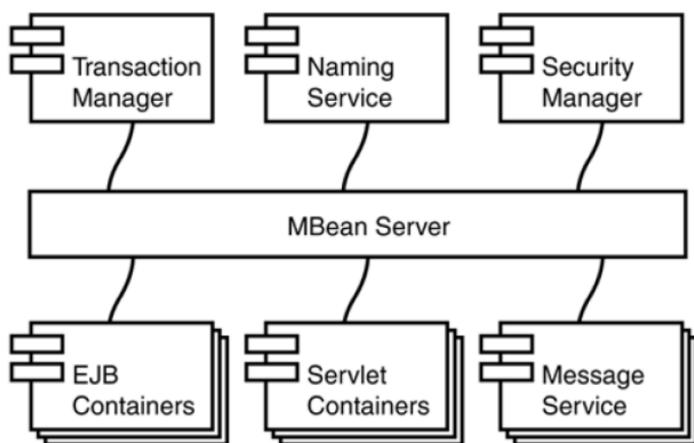
    return new MBeanNotificationInfo[] {info};
}

private final String name = "My first MBean";
private int cacheSize = DEFAULT_CACHE_SIZE;
private static final int DEFAULT_CACHE_SIZE = 200;
private long sequenceNumber = 1;
}

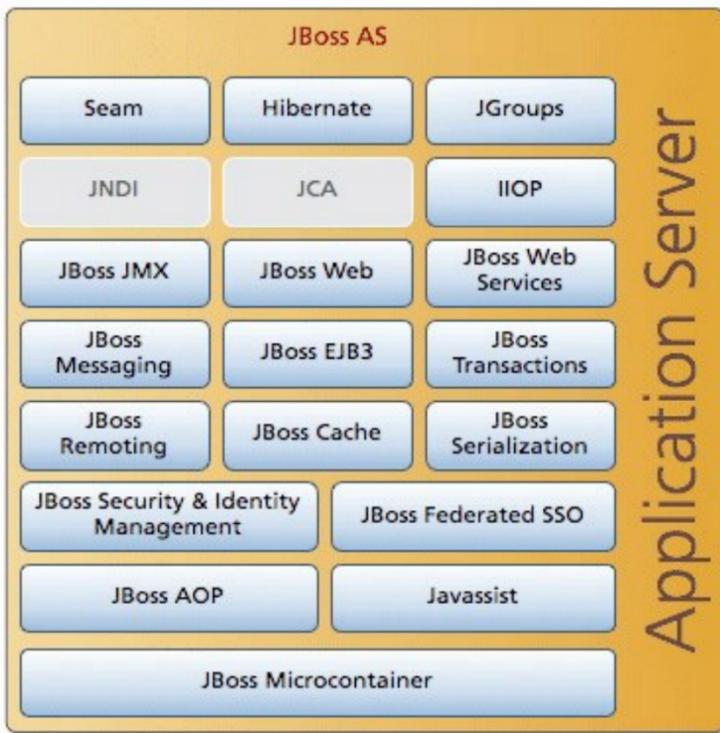
```

## 10.9 JMX in Application Server JBoss

L'application server **JBoss** è stato costruito al di sopra dell'infrastruttura JMX, questo è molto visibile in versione JBoss AS 4.3.\*. poiché a partire da questa versione JBoss utilizza un'architettura a microkernel basata su componenti MBean, con application server non monolitico. Sia le applicazioni realizzate su JBoss che l'application server sono facilmente manageable. La configurazione del server è altamente flessibile, infatti è possibile scegliere fra differenti implementazioni di servizio, si può fare l'embedding di differenti container nell'application server, anche a runtime (ad es. servlet container come Tomcat, Jetty, ...). Inoltre, se un'implementazione di servizio non offre una funzionalità richiesta da un'applicazione se ne può scegliere un'altra, mentre i servizi non necessari possono essere disattivati (shut down). Il nucleo dell'application server JBoss è JMX MBean server: questo rende l'application server estremamente semplice da estendere con nuove funzionalità. Aggiungere nuovi servizi o componenti application-specific si traduce nella creazione di nuovi MBean e nella loro registrazione al server MBean. Grazie a JMX questi componenti possono essere gestiti come MBean. Inoltre, queste implementazioni non essendo più monolitiche ma a microkernel consentono di aggiungere nuove funzionalità a run-time.

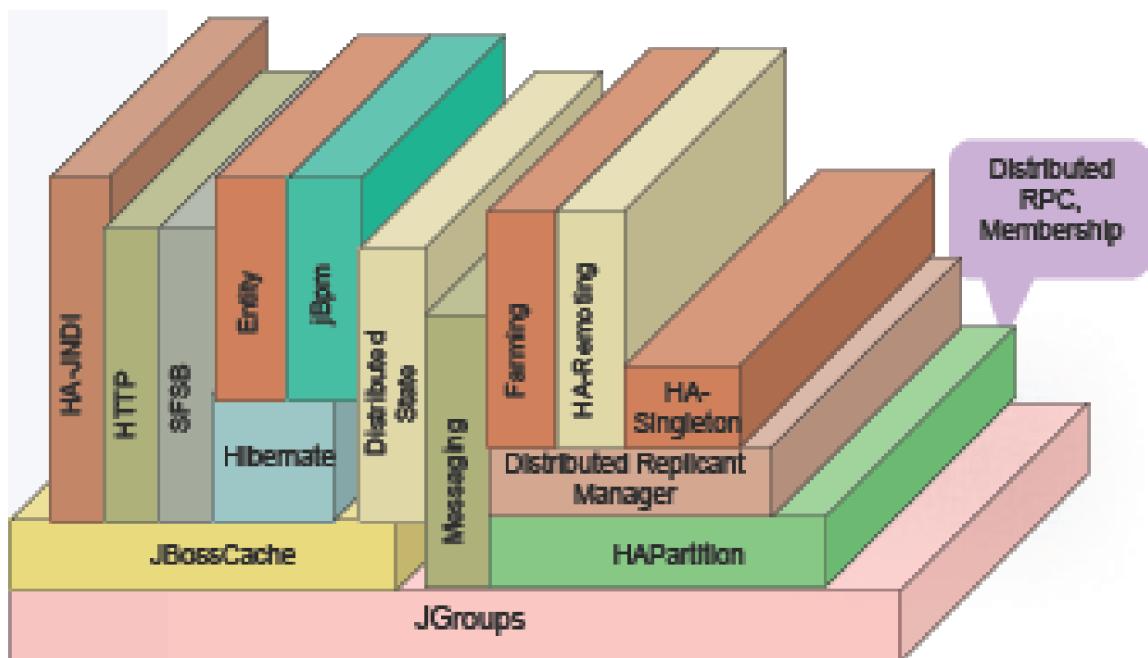


Oltre all'architettura a microcontainer, evoluzione del bus JMX delle versioni precedenti, sono presenti in forma di moduli una serie di servizi già conosciuti.



Questa architettura possiede il meglio dei due mondi, a container pensanti e leggeri. **JGroups** è un framework utile per garantire affidabilità nelle comunicazioni multicast, è importante quando si vuole comunicare in gruppi.

Architettura modulare e a stack, basata su comunicazione di gruppo (JGroups), caching (JBossCache) e supporto ad alta disponibilità (HAPartition)



Alta consistenza scambio di messaggi in multicast con costo alto che va a gestire la configurazione effettuata.

**Esempio:** uno dei componenti core caricati dal servizio M-let di JBoss è un'implementazione di ConfigurationService MBean: effettua bootstrap del server, fa il download e configura i servizi usando il file di configurazione XML jboss.jcml . Con questo MBean è possibile configurare e scaricare i vari servizi.

Utilizzato in prodotti di successo come lo stesso jBoss

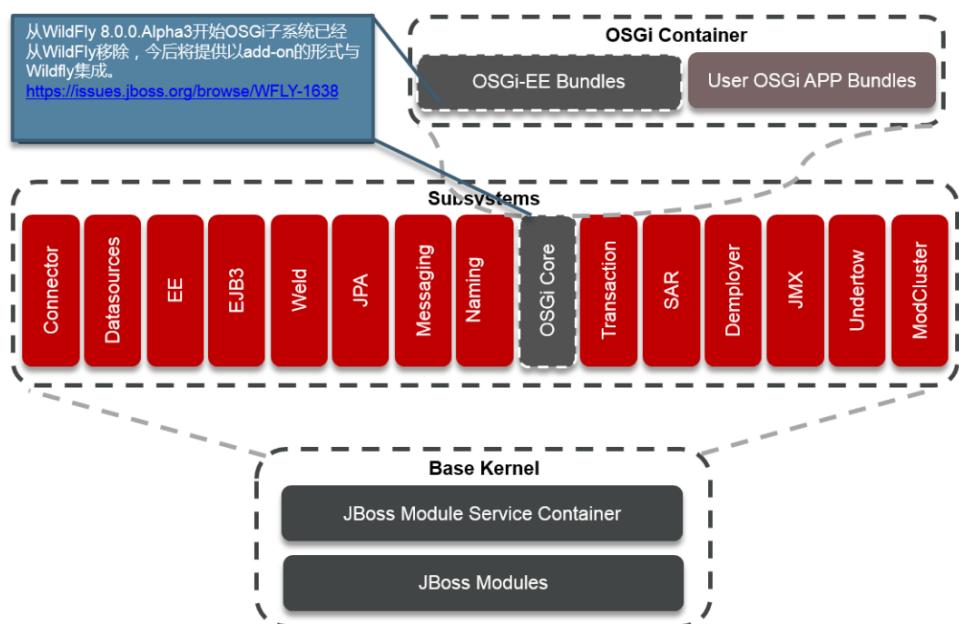
### Architettura di JBoss AS 7

Approccio di gestione modulare nel caricamento di servizi e librerie richieste in base a metadati di dipendenza, sia all'avvio del server che delle applicazioni (dipendenze implicite rispetto all'uso di package)

Alla base vi è un microkernel e al di sopra tutti i servizi, le dipendenze sono gestite con un unico file di configurazione, inoltre OSGI consente il caricamento di applicazioni a caldo ovvero dei plugin detti bundle.

Anche differenza  
pratica di:

non un file di con-  
figurazione per  
sottosistema, ma  
unico file  
(standalone.xml  
o domain.xml)



## 11. Clustering di applicazioni in WildFly/JBoss

Con il termine **cluster**, si fa riferimento ad un insieme di macchine, tipicamente vicine fra loro, che lavorano insieme in maniera coordinata. Un cluster può essere fisico o logico, un cluster è fisico se le operazioni di coordinazione avvengono a livello fisico (per esempio tramite circuiti, sensori, schede), il cluster è logico invece se queste operazioni avvengono via software.

Il cluster è quindi una struttura di base, formata da un insieme di macchine, sulla quale può essere effettuato il deployment di intere applicazioni, si parla in questo caso di **clustering** dell'applicazione.

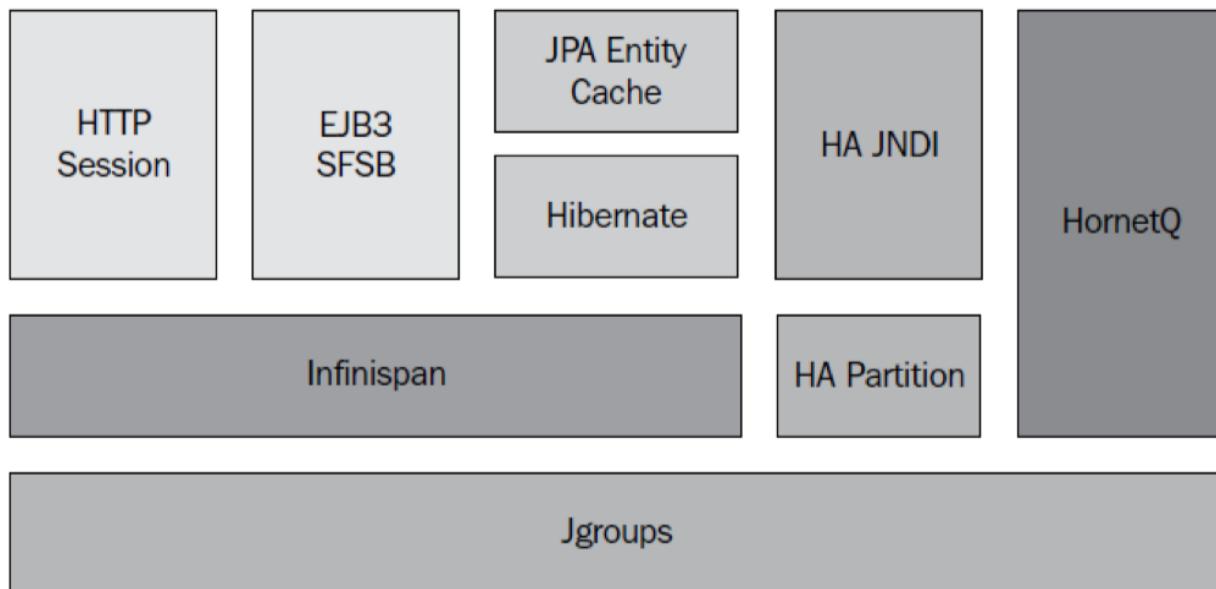
L'idea del clustering è quella di eseguire un'applicazione su server multipli in parallelo fornendo una visione singola ai clienti applicativi (usato, per esempio, dai motori di ricerca, da siti di e-commerce etc...).

Il clustering è fondamentale per avere applicazioni tolleranti ai guasti, bilanciate (load balancing, distribuisce il carico dell'applicazione tra più macchine) e scalabili (miglioramento di performance tramite semplice aggiunta di nuovi nodi al cluster).

**JBoss Clustering** è una soluzione open source con buona trasparenza (nel senso che non obbliga il programmatore ad occuparsi dei problemi di clusterizzazione). Un cluster JBoss è semplicemente un insieme di nodi, in cui ogni nodo è definito come un'istanza del server JBoss.

### 11.1 Architettura JBoss

JBoss offre un servizio per la gestione della replicazione dello stato applicativo chiamato **JBoss Cache**, che dalla versione 7 di JBoss è stato sostituito dal più moderno servizio **Infinispan**. L'architettura di JBoss è la seguente (non serve spiegarla):

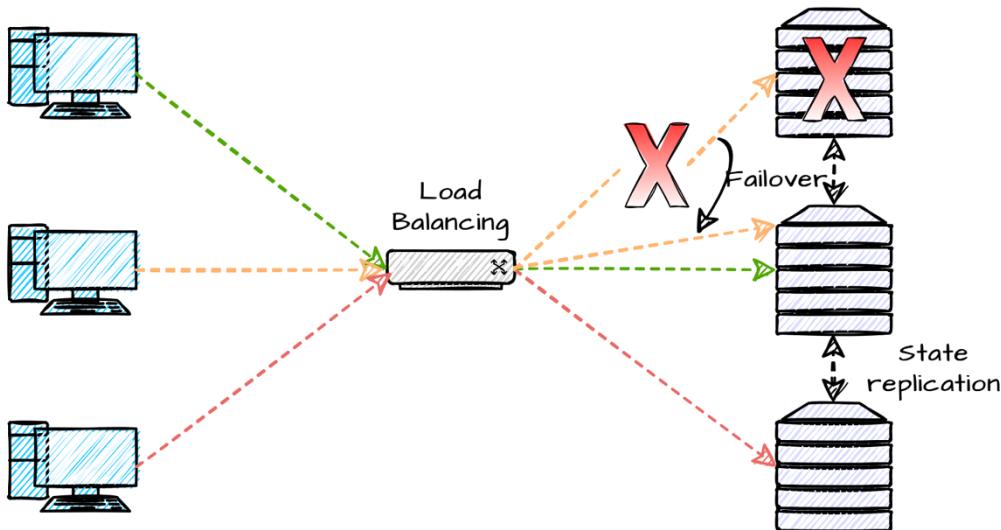


L'abilitazione del clustering service avviene lanciando, da linea di comando, il seguente codice:

```
run.bat -c all  
./run.sh -c all
```

Questo comando lancia l'application server, abilitando tutte le librerie necessarie, come **JGroups.jar** per abilitare il multicast di gruppo affidabile e **jboss-cache.jar** per la cache distribuita (la cache è una memoria molto veloce a cui si accede spesso, la cache distribuita è una cache non più residente su una sola macchina, ma distribuita su più macchine).

Quando una macchina subisce una caduta, il failover dirottta l'operazione che stava eseguendo quella macchina su un'altra macchina del cluster. La cache distribuita permette alla macchina su cui viene dirottato il lavoro di recuperare lo stato di quell'operazione e ricominciare il lavoro da dove si era interrotto, in questo modo non occorre ricominciare da capo l'operazione.



## 11.2 Comunicazione all'interno del cluster: JGroups

**JGroups** è lo strumento indispensabile che consente la comunicazione multicast in JBoss. Un canale JGroups tiene traccia automaticamente di chi fa parte del cluster sulla base di configurazione e nome del canale JGroups utilizzato. JGroups supporta lo scambio affidabile di messaggi all'interno del cluster. A default, la comunicazione multicast avviene con il protocollo UDP, ma JGroups offre anche la possibilità di utilizzare il protocollo TCP con connessioni punto-punto. UDP è ottimo per connessioni multicast locali, quando invece si vuole lavorare con multicast remoti si può scegliere TCP per avere una maggiore affidabilità con un costo maggiore della comunicazione.

A default JBoss utilizza quattro canali JGroups separati:

- Un canale usato dal servizio general-purpose di **HAPartition** (High Availability Partition)
- Tre canali creati da JBoss Cache, usati per supportare la replicazione dello stato.

### 11.2.1 Configurazione di JGroups

La comunicazione all'interno del cluster è configurabile tramite il file XML **cluster-service.xml**, questo file descrive la configurazione per la partizione di default del cluster. In questo file è inoltre possibile configurare lo stack di protocolli JGroups (la configurazione di default usa UDP con IP multicast).

Il file **cluster-service.xml** permette di configurare anche altri parametri come ad esempio:

- **PING**: protocollo per la scoperta dei membri del cluster.
- **MERGE2**: protocollo per la fusione di gruppi già scoperti (permette di unire al cluster sottogruppi creati a runtime).
- **FD**: è il protocollo per settare il timeout del failure detection.

Lo snippet sottostante mostra un esempio di file **cluster-service.xml**:

```

<mbean code="org.jboss.ha.framework.server.ClusterPartition"
name="jboss:service={jboss.partition.name:DefaultPartition}">

...
<attribute name="PartitionConfig">
<Config>
<UDP mcast_addr="${jboss.partition.udpGroup:228.1.2.3}"
mcast_port="${jboss.hapartition.mcast_port:45566}"
tos="8"

...
<! -- ping per scoprire i membri che appartengono al cluster-->
<PING timeout="2000"
down_thread="false" up_thread="false"
num_initial_members="3"/>

...
<! -- per fondere gruppi già scoperti -->
<MERGE2 max_interval="100000"
down_thread="false" up_thread="false"
min_interval="20000"/>

...
<! -- timeout per failure detection -->
<FD timeout="10000" max_tries="5"
down_thread="false" up_thread="false" shun="true"/>

...
<! -- questo protocollo verifica se un membro sospetto è realmente morto eseguendo nuovamente i
    <VERIFY_SUSPECT timeout="1500" down_thread="false"
    up_thread="false"/>

...
<pbcast.STATE_TRANSFER down_thread="false" up_thread="false"/>
```

## 11.2.2 HA Partition

L'High Availability Partition è un servizio general-purpose che realizza l'alta disponibilità, è utilizzato in JBoss per svolgere diversi compiti.

HA Partition è un'astrazione costruita sulla base dei canali JGroups, consente di effettuare e ricevere invocazioni RCP/RMI da e verso i nodi del cluster, supporta un registry distribuito che deve essere disponibile localmente a tutti i nodi nel cluster, offre un servizio di notifica ai listener che si sono registrati (in questo caso i listener sono i nodi del cluster), per comunicare cambiamenti di servizi nel registry o modifiche nell'appartenenza al cluster. Oltre questo, HA Partition è il nucleo di molti altri servizi di clustering come smart proxy lato client, farming, HA-JNDI.

Di seguito viene mostrato un esempio di configurazione di HA Partition:

```
<mbean code="org.jboss.ha.framework.server.ClusterPartition"
name="jboss:service=DefaultPartition">
    <attribute name="PartitionName">${jboss.partition.name:DefaultPartition}</attribute>
    <! – Indirizzo usato per determinare il nome del nodo -->
    <attribute name="NodeAddress">${jboss.bind.address}</attribute>
    <! -- deadlock detection abilitata o no -->
    <attribute name="DeadlockDetection">False</attribute>
    <! -- Max time (in ms) di attesa per il completamento del trasferimento di stato -->
    <attribute name="StateTransferTimeout">30000</attribute>
    <! -- configurazione protocolli JGroups -->
    <attribute name="PartitionConfig">...</attribute>
</mbean>
```

Per far parte dello stesso cluster, i nodi devono, semplicemente, avere lo stesso **PartitionName** e gli stessi elementi **PartitionConfig**.

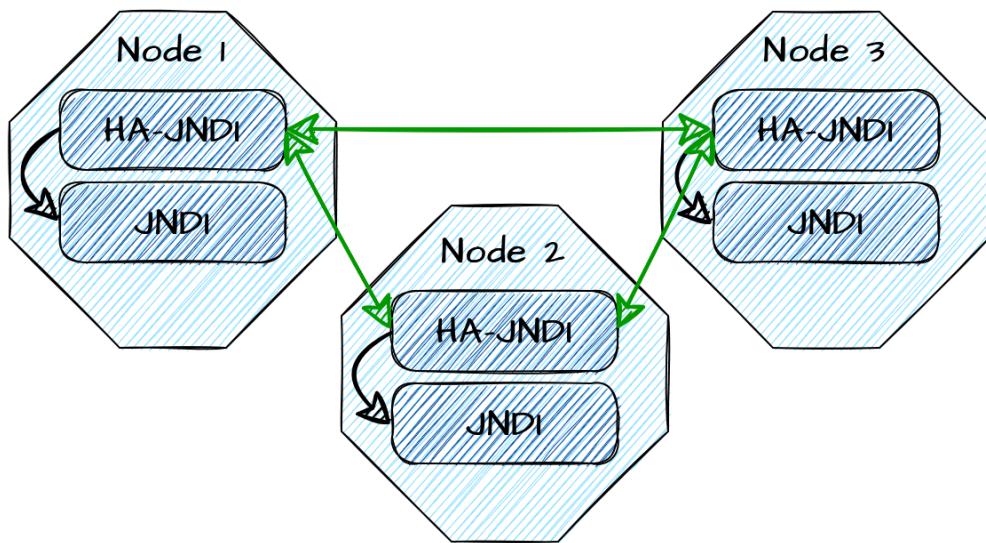
All'interno del cluster si ha la possibilità di avere un deployment omogeneo o disomogeneo. Nel **deployment omogeneo** si replica su tutti i nodi del cluster questo facilita la gestione ed è reso automatico attraverso farming. Nel **deployment disomogeneo** non vi è la replicazione su tutti i nodi del cluster, per questo è generalmente sconsigliato dagli sviluppatori JBoss per diverse ragioni, tra cui la mancanza di supporto a transazioni distribuite sul cluster. Il clustering omogeno è quello più utilizzato.

### Clustering HA-JNDI

Quando si porta un componente EJB su un cluster bisogna capire dove eseguire il servizio di nomi JNDI, se su un nodo o su più nodi.

Il cliente non usa più JNDI direttamente, ma interroga un **proxy HA-JNDI** che effettua le operazioni di lookup sul server JNDI. Lato server, il servizio HA-JNDI mantiene un albero JNDI (**context tree**) sul cluster, che deve essere disponibile fino a quando è presente almeno un nodo nel cluster. Ogni nodo dell'albero mantiene una sua copia locale di parte dell'albero JNDI (HA-JNDI), lo spazio dei nomi e quindi parzialmente replicato e partizionato e inoltre, ogni nodo mantiene una copia locale di JNDI con informazioni ulteriori su cui si può effettuare un binding. Quando viene effettuata un'operazione di lookup:

- Il primo passo consiste nel ricercare il nome nell'albero locale del nodo interrogato, se il binding non viene trovato nell'albero si interroga il servizio JNDI locale del nodo.
- Se anche nel servizio JNDI locale del nodo non si trova la corrispondenza, si interrogano gli altri nodi del cluster coordinando i componenti HA-JNDI di ogni nodo.
- Se anche così non si trova corrispondenza quel nome non è disponibile, viene quindi lanciata una **NameNotFoundException**, poiché a questo punto il nome non è presente in nessun JNDI del cluster.



I componenti HA-JNDI mantengono conoscenza reciproca e hanno una copia parziale dell'albero JNDI locale.

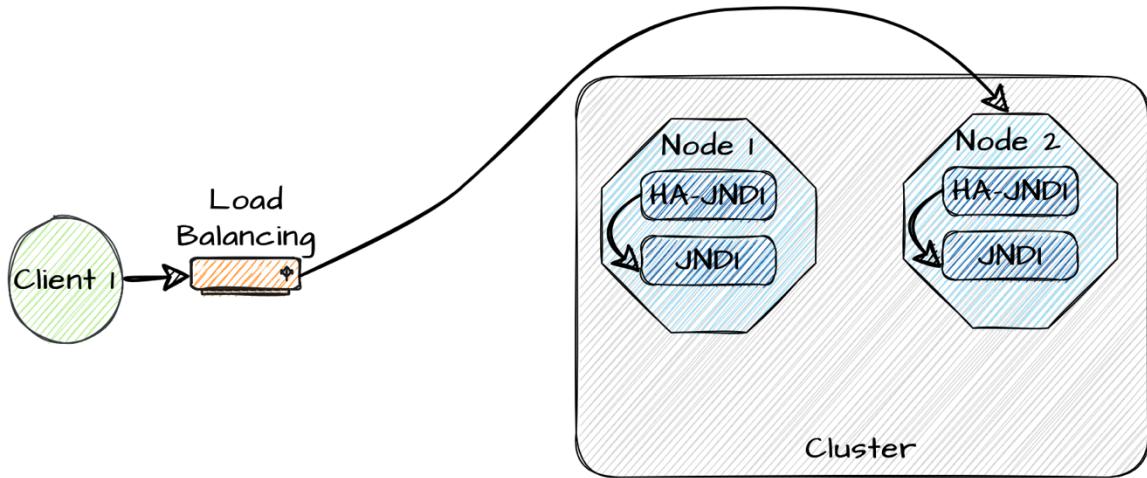
Questa ricerca su più livelli, con un livello locale e un livello distribuito con coordinamento forte, consente di limitare i colli di bottiglia e diminuire il coordinamento tra i nodi.

### 11.3 Comunicazione Client-to-Cluster

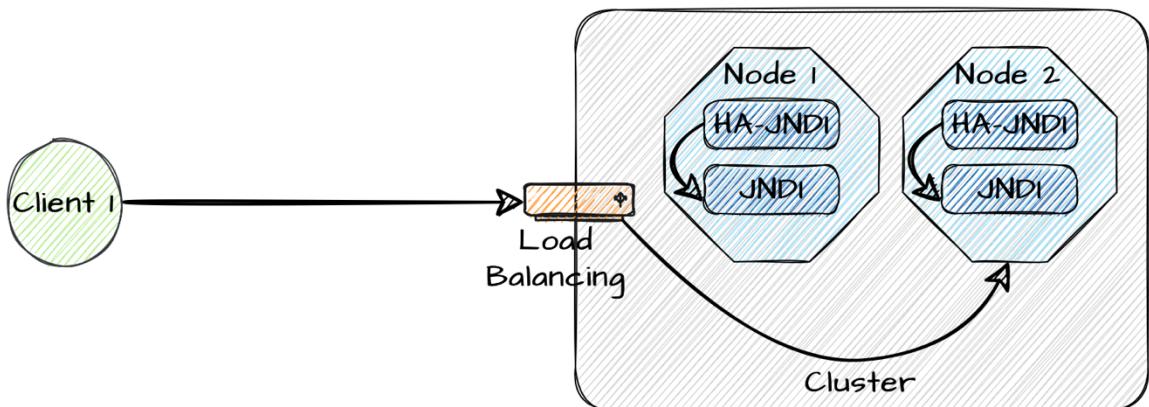
La comunicazione tra client e cluster può avvenire secondo due possibili modalità:

- **JBoss fat client:** l'idea di base è quella di avere, lato client, un **HA smart proxy** che contiene la logica di **load balancing** e capacità di resilienza in caso di guasti (**failover**). La logica di HA smart proxy può essere modificata e plugged-in dal programmatore, scegliendo quale politiche di failover e load balancing applicare. Dal punto di vista implementativo, lo smart proxy, si presenta come un grande stub RMI, detto HA-RMI, che i client EJB possono usare per interagire con il componente remoto.

Il fat client contiene al suo interno la lista dei nodi disponibili e le varie politiche di load balancing che si possono attuare, che sono: Random Robin, Round Robin, First Available All Identical Proxies. Se avviene una modifica nella composizione del cluster, all'invocazione seguente il server fa piggybacking della lista aggiornata con un protocollo dedicato, la lista dei nodi è mantenuta automaticamente lato server tramite JGroups. Vi è un piano di comunicazione interno a JGroups e un protocollo di comunicazione client to cluster. Client side le comunicazioni sono intercettate tramite meccanismi trasparenti che si occupano di load balancing e failover all'invocazione di metodi nell'interfaccia dello stub.



- **JBoss thin client:** In questo caso tutta la logica di load balancing e failover viene gestita lato server dal proxy locale al nodo, può essere realizzata tramite hardware o software a discrezione dello sviluppatore (**Apache mod\_jk**, **Apache mod\_cluster**). Questo consente ai clienti, di solito clienti web, di fare delle semplici richieste senza occuparsi del load balancing, sarà infatti questo modulo, hd o sw, ad occuparsi di smistare le richieste tra i vari nodi del cluster.



## 11.4 Replicazione dello stato

JBoss Cache è un framework per il supporto a cache distribuita (supporto alla replicazione dello stato). Per motivi di performance, realizza il caching di oggetti Java acceduti frequentemente, come ad esempio, sessioni HTTP, stati di session bean EJB 3.X, ed Entity EJB 3.X.

Ogni servizio di cache è definito in un **Mbean** separato, con il suo canale JGroups.

JBoss Cache è cluster-aware: lo stato è mantenuto consistente fra i vari nodi nel cluster, questo permette un elevata tolleranza ai guasti in caso di server crash, inoltre il servizio di cache si fa carico di invalidare e/o aggiornare lo stato nella cache.

Per esempio, per gestire il failover e il load balancing di session HTTP si usa JBoss per la replicazione dello stato. Lanciando JBoss con la configurazione **all** si include in JBoss la **session state replication**. Per quanto riguarda il load balancing invece, usualmente viene gestito da software esterno (**mod\_jk**, **mod\_cluster**) oppure a livello hardware.

In alternativa, per applicazioni Web è possibile configurare il cluster in modalità sticky session, lo stato è mantenuto in un solo nodo del cluster, quindi, non vi è replicazione. Questa strategia permette un aumento delle performance, ma di contro la caduta del server produce la perdita dello stato di sessione.

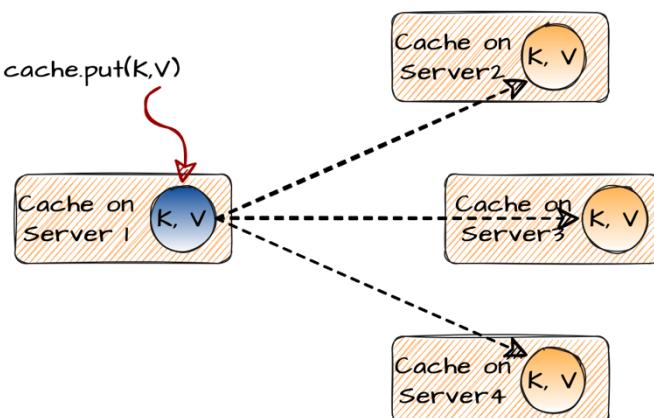
Il proxy mod-cluster ha maggiore conoscenza del cluster, partecipa a tutti i protocolli di coordinamento all'interno del cluster stesso. Rispetto al più semplice e tradizionale Apache mod\_jk vi è una configurazione dinamica del cluster, che è integrato con meccanismi di advertising su JGroups basati su multicast UDP, con basso costo di aggiornamento. I nodi cluster fanno discovery automatico di uno o più nodi load balancer disponibili. Il monitoraggio delle prestazioni attraverso multicast mette a disposizione diverse metriche per la misurazione del carico, numero di cpu, memoria utilizzata, numero di connessioni. Inoltre, si possono avere informazioni a livello applicativo utilizzando notifiche per intercettare eventi di re-deploy, undeploy delle applicazioni presenti su un certo nodo del cluster e prendere decisioni di management conseguenti.

Queste funzionalità non erano presenti su mod-jk. Ovviamente questi scambi aggiuntivi hanno un costo, che è comunque limitato in quanto il proxy è locale lato server e si fa intercettore e riconoscitore di queste informazioni che circolano all'interno del cluster.

## 11.5 Infinispan

Infinispan sostituisce JBoss Cache. Infinispan è un framework open-source basato su JGroups come supporto di comunicazione utilizzabile anche al di fuori di JBoss. Svolge il ruolo di infrastruttura di caching e replicazione per sessione HTTP, stato di stateful Session Bean, nomi JNDI e replicazioni del secondo livello di Hibernate. Infinispan ha quattro strategie di caching:

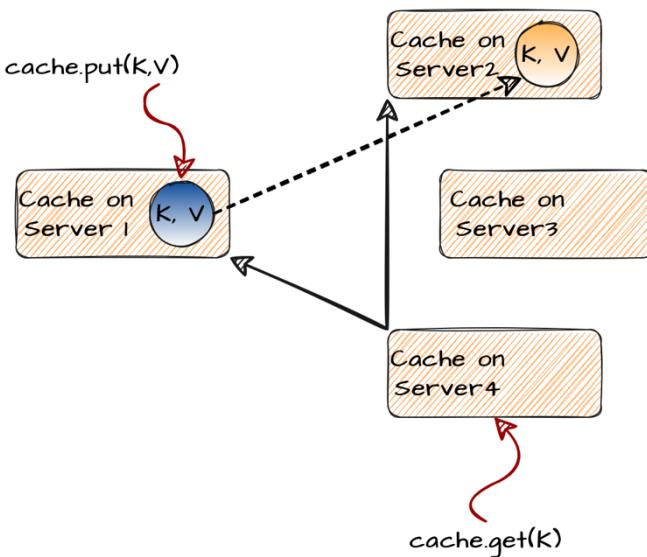
- **Locale:** l'oggetto è mantenuto solo in copia locale al nodo. Non c'è replicazione sul cluster, utilizzata tipicamente per secondo livello di Hibernate.
- **Replicazione:** è la strategia più onerosa perché con questa strategia Infinispan replica su tutti gli oggetti presenti in cache su tutti i nodi del cluster. Tipicamente è adatta per piccoli cluster, perché il carico di rete necessario per tenere informati i nodi è alto, di conseguenza è scarsamente scalabile.



- **Distribuzione:** strategia che prevede che il dato sia salvato in un sottoinsieme dei nodi del cluster. Se il nome è trovato nel nodo attraverso get non sono necessarie altre azioni,

altrimenti va cercato negli altri nodi del cluster in questo caso vi sono tempi di risposta più lunghi. Tutti gli oggetti in cache sono replicati solo su sottoinsieme **fisso** (configurabile) di nodi del cluster. Ovviamente ciò comporta minori performance quando oggetti non disponibili localmente, ma in generale è più scalabile. In questa modalità si sceglie un trade-off tra grado di replicazione e tempo di risposta, anche tenendo conto della dimensione della cache che deve contenere questi dati.

- **Invalidazione:** strategia nella fase di aggiornamento della cache, quando nella fase di scrittura si inserisce una entry che modifica il valore di una chiave preesistente, si invalidano le copie di quella entry disponibili in altri nodi del cluster. Per non avere letture inconsistenti, non avviene replicazione, ma rimozione di entry non valide. Tipicamente è utilizzata per dati comunque disponibili su data store persistente



Replicazione e invalidazione possono avvenire in modo bloccante o non-bloccante. Nel caso non-bloccante si crea una coda di modifiche/invalidazioni multiple. Questa strategia garantisce migliori performance in quanto le operazioni avvengono in batch, ma possono esservi errori riportati nei log ed eventualmente utilizzati per fare rollback se si trattano transazioni. Quando si agisce in modo non bloccante si mantiene il log delle operazioni da effettuare, quindi, è possibile in caso di fault rendere consistente la situazione sugli altri nodi.

Quando la cache si riempie, bisogna scegliere che oggetto rimuovere. Il processo di rimozione delle entry è detto **eviction**. In Infinispan la rimozione avviene solo a livello locale con varie strategie come: unordered random, FIFO, Last Recent Used LRU.

Anche nelle cache come nei DB, vi sono diversi livelli di isolation e locking delle risorse:

- **REPEATABLE\_READ** sono possibili phantom read.
- **READ\_COMMITTED** possibili non repeatable read ma maggiori performance.
- **Optimistic locking** e verifiche a posteriori in fase di commit.

## 11.6 Replicazione dello stato nel clustering di componenti EJB

- **Stateless Session Bean:** data la mancanza di stato, le chiamate possono essere bilanciate su ogni nodo del cluster, in questo caso si guadagna molto in scalabilità.
- **Stateful Session Bean:** lo stato è replicato e sincronizzato sul cluster, ad ogni modifica dello stato di uno di questi bean, entra in azione il Session State replication di JBoss Cache o di Infinispan. A seconda della strategia di Cache adottata (locale, replicazione e distribuzione), l'operazione di replicazione e sincronizzazione avrà un costo diverso.
- **Entity Bean:** per gli Entity Bean, non si prevede il supporto al clustering, in quanto essi non sono interrogabili remotamente e quindi non prevista abilitazione clustering. Si può decidere di replicare solo la cache di secondo livello Hibernate con Infinispan.
- **MessageDriven Bean:** fine, per i Message Driven Bean, la gestione è uguale ai Stateless Session Bean. Si usa HornetQ che è un supporto MOM creato da JBoss.

A seconda della tipologia di bean l'HA smart proxy sarà configurato in maniera opportuna. Nel migliore dei casi, è sufficiente l'uso di una annotazione specifica e proprietaria di JBoss, per informare il container EJB di JBoss che il componente considerato deve essere clustered. L'annotazione **@Clustered** è applicabile a:

- **Stateless Session Bean** con smart proxy a cui sono applicabili diverse politiche di load balancing, come Round Robin (default), First Available, First Available All Identical Proxies (tutti smart proxy per un bean verso stesso nodo), Random Robin.
- **Statefull Session Bean** con smart proxy, con diverse politiche di load balancing First Available (senza abilitazione esplicita replicazione stato).
- **Message Driven Bean** le politiche di load balancing sono Round Robin, Random, Custom (tramite implementazione interfaccia **ConnectionLoadBalancingPolicy**).
- **Entity Bean** non hanno problemi perché sono gestiti internamente quindi non possono essere gestiti dallo smart proxy lato client.

## 11.7 Configurazione di JBoss/WildFly

Le opzioni di configurazione di JBoss/WildFly prevedono una serie di file, che permettono di configurare il server in maniera personalizzata:

- **standalone.xml:** è il file di configurazione di default per server standalone (autocontenuto che contiene tutti i servizi). Contiene tutti i metadati di configurazione per subsystem, networking, deployment e socket binding. Usata automaticamente quando si fa partire un server standalone. Offre il supporto di Java EE Web-Profile ed alcune estensioni come RESTful Web Services.
- **standalone-full.xml:** fornisce il supporto per tutti subsystem eccetto high availability (HA). Supporto di Java EE Full-Profile e tutte funzionalità server.
- **standalone-ha.xml:** La configurazione standalone-ha.xml include tutti i subsystem di default più l'aggiunta mod\_cluster per il supporto al thin client e JGroups per standalone server, così che possa partecipare in un cluster high-availability. Questo è il profilo di default per funzionalità di clustering.
- **standalone-load-balancer.xml:** configura automaticamente l'uso di proxy front-end Undertow (Web server) per lavorare da load balancer. Undertow si connetterà ai nodi

server veri e propri usando le sue richieste built-in per clienti e proxy protocolli supportati:  
HTTP, AJP, HTTP2, H2C (clear text HTTP2).

## 12. BigData

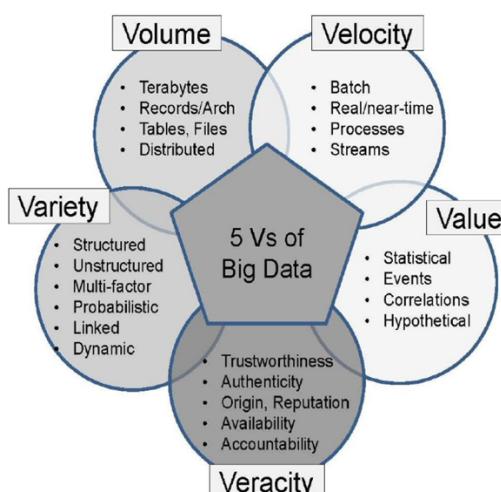
Molti sistemi sono sempre più spesso coinvolti nella gestione di enormi moli di dati, originati da sorgenti altamente eterogenee e con formati altamente differenziati. Per poter trattare l'enorme flusso di dati è necessario il supporto di un sistema distribuito che permetta di scalare orizzontalmente. Dove per **scalare verticalmente** si intende aumentare la capacità aggiungendo più risorse alla macchina, mentre **scalare orizzontalmente** si intende aumentare la capacità facendo cooperare più macchine.

Grandissime moli dati vengono continuamente generate, da dispositivi elettronici, sensori, smart city, industrie 4.0 ecc...

La necessità di collezionare, gestire e analizzare questi dati, nasce dall'interesse di grandi compagnie di profilare gli utenti, oppure, nell'ambito dell'industria 4.0, i dati possono essere analizzati per monitorare un macchinario, prevenirne un guasto e manutenerlo efficacemente.

Esistono molte definizioni di **BigData**, la più comune è quella che fa riferimento alle 6V:

- **Volume dei dati**
- **Varietà dei dati**: Indica le diverse tipologie di dati con cui si può avere a che fare, ad esempio, dati strutturati, non strutturati, semi-strutturati.
- **Velocità**: definisce la velocità con cui i dati vengono generati e come devono essere gestiti:
  1. **Streams**: flusso continuo di dati che vengono lavorati costantemente.
  2. **Real time (in senso stretto)**: i dati vengono processati rispettando un vincolo temporale. Un sistema real-time garantisce un certo tempo di risposta non necessariamente breve.
  3. **Interactive**: interazione con l'utente durante il processamento
  4. **Batch**: un insieme di dati viene collezionato per un periodo di tempo specifico e successivamente processato.
- **Valore**: Fa riferimento al valore dell'informazione che si può estrarre dai dati grezzi.
- **Veridicità**: Indica, il livello di affidabilità o inaffidabilità dei dati.
- **Variabilità**: Indica il fatto che il significato o l'interpretazione di un dato può variare in funzione del contesto in cui questo viene raccolto ed analizzato. Il valore, quindi, non risiede solamente nel dato, ma è strettamente collegato al contesto da cui si ricava.



I sistemi informativi moderni richiedono una gestione quality aware e un processing workflow ottimizzato per produrre nuovi servizi e incrementare il valore dei dati. Sono state definite delle architetture di riferimento per la gestione dei BigData.

Le aree applicative dei BigData sono le più svariate, dalla telefonia (social analysis, geomapping) all'ambito medico (monitoraggio dei pazienti) passando per l'ambito energetico (smart grid), per la sicurezza, i trasporti, il marketing ecc...

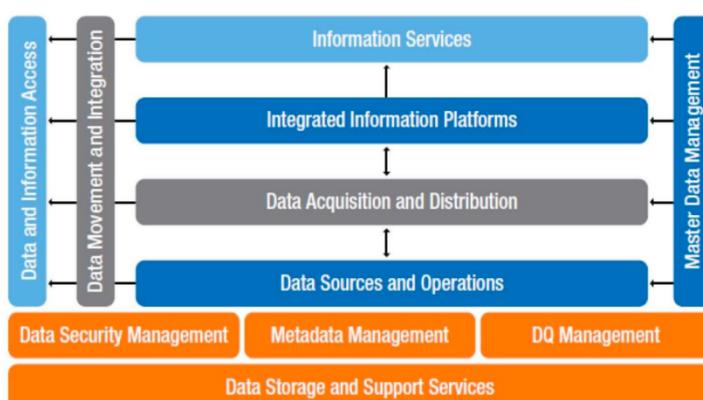
A causa dell'eterogeneità dei dati nasce la necessità di avere un modello dei dati comune, che si traduce nel bisogno di standardizzare l'architettura delle piattaforme di supporto.

## 12.1 Standardizzazione industriale

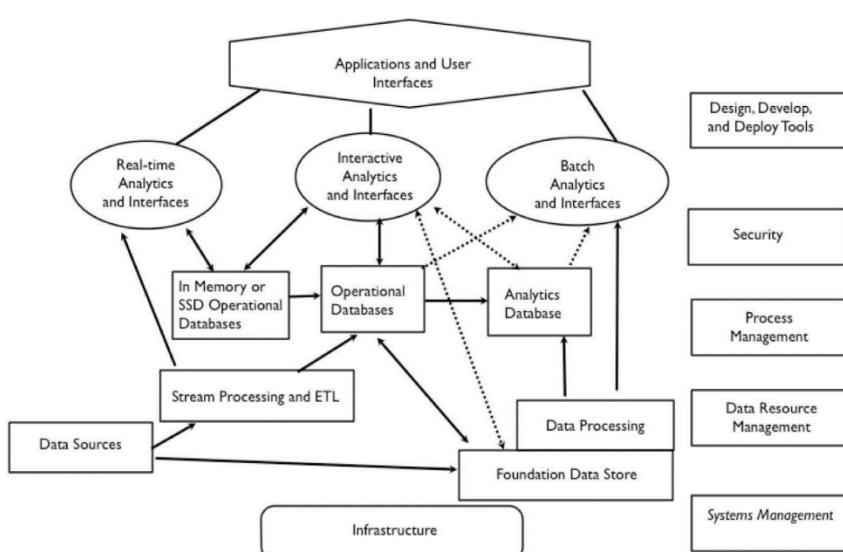
### 12.1.1 Open Data Center Alliance (ODCA)

**Open Data Center Alliance (ODCA)** propone l'idea di “**Information as a Service**”.

Definisce un'architettura a livelli, in cui al livello più basso (arancione) si trovano i servizi di base per la raccolta dei dati, l'estrazione dei metadati e la sicurezza. Scalando l'architettura verso l'alto si trovano una serie di facility (servizi) per la definizione delle sorgenti dati, l'acquisizione dei dati, l'integrazione e l'uso dei dati.



### 12.1.2 National Institute of Standards and Technology (NIST)



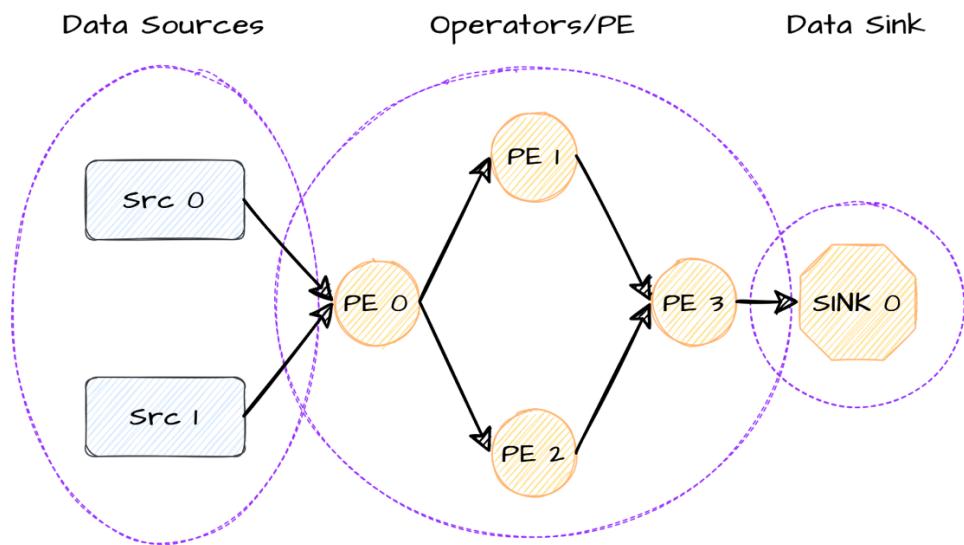
L'architettura proposta dal National Institute of Standards and Technology NIST prevede un'infrastruttura affiancata da una serie di funzionalità trasversali (System Management, Data Resource Management, Security...), anche in questo caso sono presenti i servizi basilari di Data Source e Data Processing, con la possibilità di eseguire stream processing e operazioni ETL (Extract/Transform/Load). La memorizzazione del dato può avvenire in cache, su un database classico o su un database orientato all'analisi. I dati memorizzati possono essere analizzati in maniera real-time, a batch o in maniera interattiva, ovvero con l'interazione dell'utente durante l'analisi.

## 12.2 InfoSphere Streams per Stream Processing

InfoSphere Streams di IBM è un sistema che permette di lavorare con grandi stream di Big Data. InfoSphere realizza alcuni dei concetti fondamentali dello stream processing, ad esempio, la possibilità di eseguire processing on-the-fly, filtrando e analizzando flussi di dati.

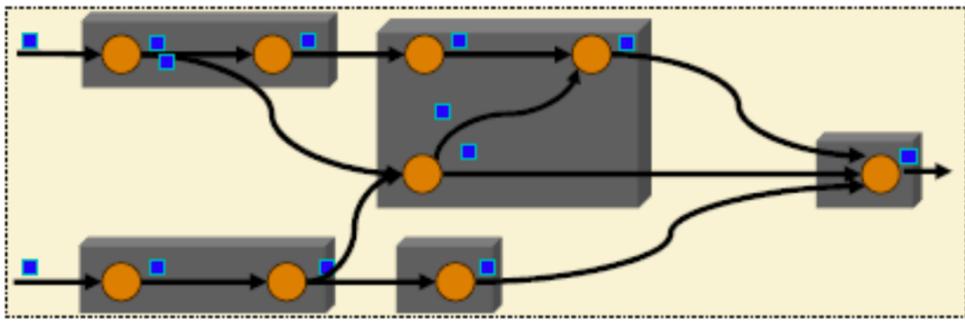
InfoSphere processa i messaggi in isolamento (uno per volta) o in finestre limitate, la natura dei messaggi può includere dati non convenzionali (spaziali, immagini, ecc...) e provenienti da sorgenti eterogenee (in termini di connettività, datarate, ecc...). InfoSphere ambisce a realizzare questi compiti con latenza dell'ordine dei millisecondi.

Il modello di programmazione utilizzato permette di definire grafi dataflow costituiti da datasource (input), operatori e sink (output), come quello mostrato nella figura sottostante:



Gli operatori possono essere fusi insieme a formare i **Processing Element** (pallini arancioni in figura) che rappresentano l'elemento minimo di computazione.

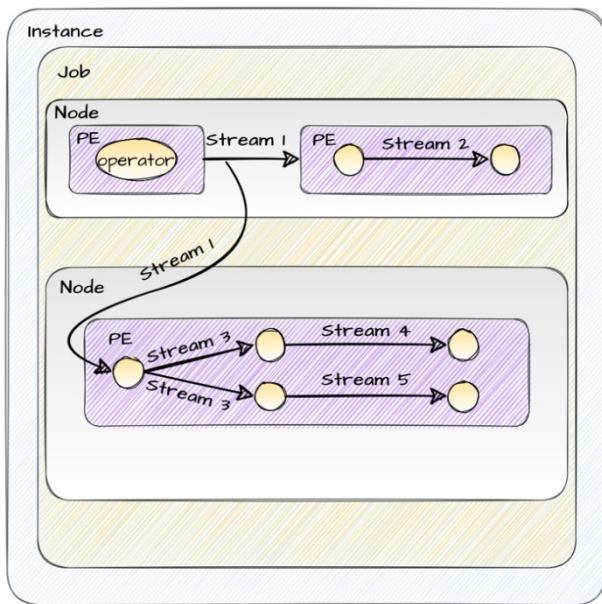
I **PE** sono ospitati all'interno di nodi (quadrati grigi in figura) e si occupano del processamento dei data sample (pezzi di informazione rappresentati in blu nella figura).



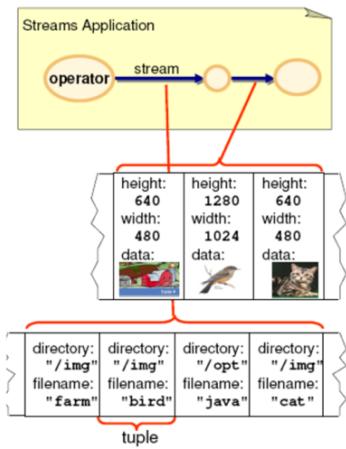
Il deployment dei PE sulle risorse fisiche (nodi) rappresenta un'operazione di cruciale importanza, infatti, bisogna tener conto del fatto che un Processing Element potrebbe avere pesanti carichi di lavoro, in questo caso sarebbe bene replicare il PE su altri nodi per alleggerire il suo carico. InfoSphere sgrava l'utente dal compito di dovere effettuare il deployment automatizzando il più possibile questa operazione. L'utente definisce i limiti minimo e massimo di risorse utilizzabili (e.g. questo PE deve essere replicato minimo 3 volte e massimo 5), InfoSphere si fa carico di gestire la replicazione dei Processing Element, gestendo inoltre l'attivazione e disattivazione dei PE a seconda delle necessità.

Un'applicazione InfoStream è, dal punto di vista logico, un grafo diretto costituito da Processing Element collegati fra loro. Un'applicazione Streams di cui è fatto il deployment su un'istanza di InfoSphere Streams prende il nome di **JOB**.

L'idea di base di InfoSphere è molto simile a quella dei sistemi a componenti, in cui lo sviluppatore deve occuparsi di scrivere solo la logica applicativa, in InfoSphere lo sviluppatore deve soltanto scrivere la logica applicativa del PE.



Nel linguaggio ad hoc di InfoSphere l'**operator** (pallini gialli in figura) è un componente fondamentale, il suo compito è quello di processare dati, chiamati **streams**, e generare nuovi streams. Gli streams sono definiti come una sequenza infinita di tuple strutturate, cioè un insieme di attributi e dei loro tipi. Un gruppo finito e sequenziali di tuple prende il nome di **finestra** e rappresenta un insieme di tuple che verranno processate dall'operator.



## 12.3 Batch Processing

Il Batch Processing consiste nell'esecuzione automatica e spesso periodica di un “job”, senza la necessità di intervento da parte di utenti esterni. Il codice di elaborazione viene eseguito su un insieme di input, chiamato **batch**. Di solito, il job leggerà i dati batch da un database e memorizzerà il risultato nello stesso database o in un database diverso.

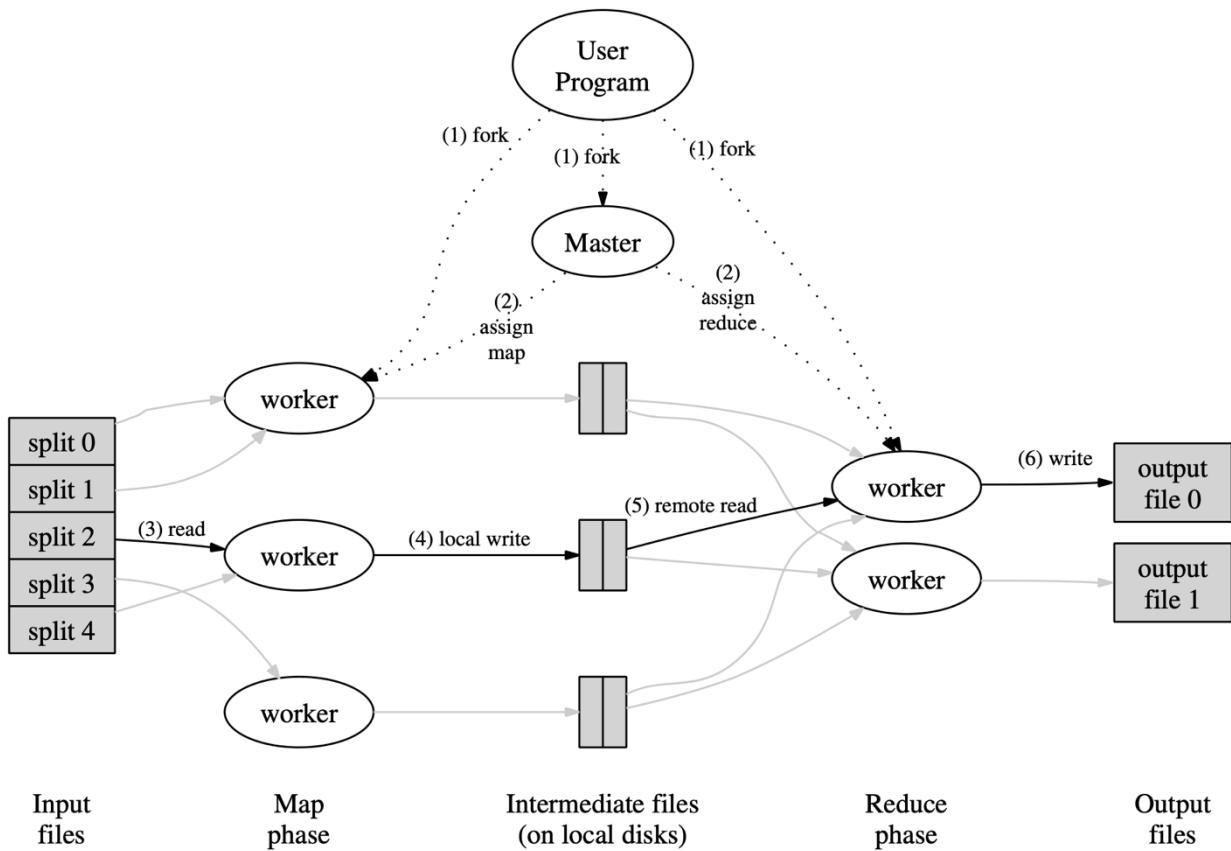
**MapReduce** è un modello di programmazione che è stato introdotto da Google nel 2004, oggi è implementato in vari sistemi di elaborazione e archiviazione dati (Hadoop, Spark, MongoDB, ...) ed è un elemento fondamentale della maggior parte dei sistemi di Batch Processing.

### 12.3.1 MapReduce

**MapReduce** è un modello di programmazione per l'elaborazione e la generazione di grandi set di dati. Gli utenti specificano la funzione **map** che elabora un sottoinsieme dei dati per generare un insieme di coppie **<chiave, valore>** intermedie, e una funzione **reduce** che unisce tutti i valori intermedi associati alla stessa chiave intermedia.

I programmi scritti in questo stile funzionale vengono automaticamente parallelizzati ed eseguiti su un grande cluster. Il sistema si occupa a runtime dei dettagli del partizionamento dei dati di input, della gestione dei guasti delle macchine e della gestione della comunicazione tra le macchine. Sia la funzione **map** che la funzione **reduce** sono funzioni senza stato che lavorano localmente a ciascuno nodo.

La figura mostra il flusso complessivo di un'operazione **MapReduce**:



Si supponga, per esempio, di avere  $M$  file di testo in ingresso a  $M$  macchine differenti che eseguono l'operazione di **map**. In questa prima fase, l'obbiettivo è quello di ottenere delle coppie chiave/valore, per esempio, contando il numero di occorrenze di ogni parola del file. Alla fine di questa fase si otterranno delle coppie chiave/valore intermedie, per esempio, dal file di testo si otterranno delle coppie del tipo <“parola”, numero>. A questo punto inizia la fase di **reduce**: come passaggio preliminare di questa fase si spostano tutte le coppie che hanno la stessa chiave su una stessa macchina (riutilizzando anche le macchine che hanno svolto la fase di map) e quindi a questo punto si esegue la funzione di **reduce**, che sostanzialmente riduce tutte le coppie chiave/valore aventi la stessa chiave ad un'unica coppia con quella chiave e con valore dato dalla somma di tutti i valori.

La fase di ridistribuzione (**reshuffling**), delle coppie intermedie nelle macchine che eseguiranno la funzione di **reduce**, e le fasi di lettura e scrittura su disco, sono quelle che comportano il maggior costo. Infatti, le funzioni di **map** e **reduce** sono eseguite localmente alla macchina. Per ottimizzare i trasferimenti da un nodo ad un altro, viene eseguita un'altra funzione che prende il nome di funzione di **combine** che ha il compito di aggregare i risultati parziali dei nodi prima di spostarli da una macchina all'altra.

Il modello MapReduce viene supportato da infrastrutture come quella di Google (non disponibile) o Apache Hadoop (open source). L'infrastruttura facilita la distribuzione dei dati sui nodi, nel caso di Hadoop attraverso **HDFS** (HaDoop File System), garantendo la suddivisione dei dati in sotto parti, ciascuna distribuita su un nodo, e supportando anche la replicazione. L'infrastruttura si occupa inoltre, della gestione di tutti i task (map, reduce e reshuffling), spostando la funzione di computazione li dove deve essere eseguita, in modo da limitare lo spostamento dei dati alla sola fase di reshuffling.

In caso di guasto, l'infrastruttura si fa carico di far ripartire i task che erano schedulati sulla macchina guasta, su un'altra macchina che aveva una copia dei dati di partenza. Chiaramente questo è possibile, in primo luogo, perché l'infrastruttura si occupa di replicare i dati e inoltre, perché le funzioni di map e di reduce sono funzioni senza stato.

### 12.3.2 Esempio di MapReduce

Si consideri l'esempio di un contatore di parole, che conti le occorrenze di ogni parola su un set di file di ingresso. Si supponga, per semplicità di avere due soli file di ingresso, così formati:

1. File1 = “hello world hello moon”
2. File2 = “goodbye world goodnight moon”

Il master assegna ad un nodo il compito di eseguire la funzione **map** del primo file. I risultati parziali ottenuti da questa operazione saranno i seguenti:

```
<hello, 1>
<world, 1>
<hello, 1>
<moon, 1>
```

Contemporaneamente, ad un altro nodo viene assegnato il compito di eseguire la funzione **map** del secondo file. I risultati parziali ottenuti saranno i seguenti:

```
<goodbye, 1>
<world, 1>
<goodnight, 1>
<moon, 1>
```

A questo punto inizia la fase di reshuffling, che ha l'obiettivo di portare sulla stessa macchina tutte le coppie che hanno la stessa chiave. Per ottimizzare i trasferimenti da un nodo ad un altro, viene eseguita un'altra funzione che prende il nome di funzione di **combine** che ha il compito di aggregare i risultati parziali dei nodi.

Il primo nodo avrà le seguenti coppie chiave/valore:

```
<moon, 1>
<world, 1>
<hello, 2>
```

Il secondo invece:

```
<goodbye, 1>
<world, 1>
<goodnight, 1>
<moon, 1>
```

Prima di passare alla funzione di Reduce, il framework raccoglie queste copie e riordina gli output dei map. Questa fase prende il nome di **shuffle and sort**. A questo punto, i risultati parziali vengono inviati alla funzione di reduce ottenendo il risultato finale:

```
<goodbye, 1>
<goodnight, 1>
<moon, 2>
<world, 2>
<hello, 2>
```

### 12.3.3 Apache Hadoop

Hadoop è un framework open source che supporta applicazioni distribuite con elevato accesso ai dati, permettendo alle applicazioni di lavorare con migliaia di nodi ed elaborare enormi moli di dati. Hadoop è ispirato al modello MapReduce introdotto da Google, e di fatto ne rappresenta una sua implementazione.

Apache Hadoop è un progetto che contiene tre sotto-progetti principali:

- **Hadoop Common**: un package di facilities comuni.
- **Hadoop Distributed File System (HDFS)**: un file system distribuito a supporto dello storage dei dati che facilita la scalabilità orizzontale.
- **MapReduce**: un framework per semplificare il processamento di enormi quantità di dati in parallelo su cluster di grandi dimensioni.

#### HDFS

Prende ispirazione da Google File System. Hadoop Distributed File System (HDFS) è un file system scalabile, distribuito, portatile, scritto in Java per framework Hadoop. HDFS può essere parte di Hadoop o un file system distribuito stand-alone general-purpose. HDFS è costituito da:

- **NameNode** che gestisce i metadata del file system.
- **DataNode** che memorizzano i veri dati.

HDFS memorizza file di grandi dimensioni in blocchi distribuiti sul cluster, garantisce affidabilità e fault-tolerance tramite replicazione su nodi multipli ed è progettato specificamente per deployment su hardware low-cost.

Un tipico cluster Hadoop integra funzionalità MapReduce e HDFS ed è costruito secondo un'architettura master/slave.

Il nodo master contiene:

- **Job tracker**: responsabile dello scheduling dei job task che eseguono MapReduce, del monitoraggio degli slave e della gestione dei job con fallimenti.
- **Task tracker**:
- **NameNode (HDFS)**
- **DataNode (HDFS)**

I nodi slave includono:

- **Task tracker:** eseguono MapReduce sotto il coordinamento del master.
- **DataNode (HDFS)**

#### 12.3.4 Google MapReduce vs Hadoop

I dati in input vengono suddivisi in “chunk” di misure appropriate, i chunk vengono elaborati in parallelo da più macchine che eseguono la funzione di **map**. I file intermedi generati nella fase di **map** vengono distribuiti, in base al valore di chiave, al più appropriato worker che esegue l’operazione di **reduce**.

Quindi scendendo nel dettaglio, quando il programma utente chiama la funzione MapReduce, si verifica la seguente sequenza di azioni:

- 1) Il file in input viene diviso in  $M$  parti, vengono avviate diverse copie del programma sul cluster di macchine.
- 2) Una delle copie del programma è il **master**. Il resto sono **worker** a cui viene assegnato il lavoro dal **master**. Ci sono  $M$  attività di Map e  $R$  attività di Reduce da assegnare. Il master seleziona i worker inattivi e assegna a ciascuno un’attività di mappatura o un’attività di riduzione.
- 3) Un worker a cui è assegnata un’attività di map legge il contenuto dell’input corrispondente. Analizza le coppie chiave/valore dai dati di input e passa ciascuna coppia alla funzione **map** definita dall’utente. Le coppie chiave/valore intermedie prodotte dalla funzione **map** vengono memorizzate nel buffer.
- 4) Periodicamente, le coppie memorizzate nel buffer vengono scritte sul disco locale, partizionate in regioni  $R$  dalla funzione di partizionamento. Le posizioni di queste coppie memorizzate nel buffer sul disco locale vengono restituite al master, che è responsabile dell’inoltro di queste posizioni ai reduce worker.
- 5) Quando un addetto alle riduzioni viene notificato dal master in merito a queste posizioni, utilizza chiamate a procedure remote per leggere i dati memorizzati nel buffer. Quando un reduce worker ha letto tutti i dati intermedi, li ordina in base alle chiavi intermedie in modo che tutte le occorrenze della stessa chiave siano raggruppate insieme. L’ordinamento è necessario perché in genere molte chiavi diverse sono associate alla stessa attività di riduzione. Se la quantità di dati intermedi è troppo grande per essere contenuta nella memoria, viene utilizzato un ordinamento esterno.
- 6) L’operatore di riduzione esegue un’iterazione sui dati intermedi ordinati e per ogni chiave intermedia univoca incontrata, passa la chiave e il corrispondente set di valori intermedi alla funzione **reduce** dell’utente. L’output della funzione viene aggiunto a un file di output finale.
- 7) Quando tutte le attività di mappatura e le attività di riduzione sono state completate, il master riattiva il programma utente. A questo punto la chiamata MapReduce nel programma utente ritorna al codice utente.

Dopo il completamento con successo, l’output dell’esecuzione di MapReduce è disponibile in  $R$  file di output (uno per ogni attività reduce, con nomi di file specificati dall’utente). In genere, gli utenti non hanno bisogno di combinare questi file di output in un unico file: spesso passano questi file come input a un’altra chiamata MapReduce o li usano in un’altra applicazione distribuita in grado di gestire l’input che è partizionato in più file.

Google MapReduce e Hadoop sono due diverse implementazioni (istanze) del modello MapReduce. Hadoop è open source, Google MapReduce no e in realtà non ci sono molti dettagli disponibili a riguardo.

Poiché lavorano con grandi set di dati, devono fare affidamento su file system distribuiti. Hadoop utilizza un file system distribuito standard l'**HDFS** (Hadoop Distributed File Systems) mentre Google MapReduce utilizza GFS (Google File System)

Hadoop è implementato in Java. Google MapReduce sembra essere implementato in C++.

### 12.3.5 Hadoop vs Spark

Apache Hadoop è software open source che consente agli utenti di gestire grandi set di dati (da gigabyte a petabyte) consentendo ad un cluster di computer di risolvere problemi su grandi quantità di dati. È una soluzione altamente scalabile ed economica che archivia ed elabora dati strutturati, semi-strutturati e non strutturati.

Apache Spark, anch'esso open source, è un motore di elaborazione dati per big data. Come Hadoop, Spark suddivide le attività di grandi dimensioni su diversi nodi. Tuttavia, tende a funzionare più velocemente di Hadoop e utilizza la memoria ad accesso casuale (RAM) per memorizzare nella cache ed elaborare i dati anziché un file system. Ciò consente a Spark di gestire casi d'uso che Hadoop non può.

Spark è un miglioramento di Hadoop per MapReduce. La differenza principale tra Spark e MapReduce è che Spark elabora e conserva i dati in memory per i passaggi successivi, mentre MapReduce elabora i dati su disco. Di conseguenza, per carichi di lavoro più piccoli, le velocità di elaborazione dei dati di Spark sono fino a cento volte superiori rispetto ad Hadoop.

### 12.3.6 MapReduce Scheduling e limiti di Hadoop

Considerando il fatto che, MapReduce, tipicamente lavora in batch, la politica adottata per lo scheduling è di tipo FIFO. Tuttavia, alcuni progetti aprono le porte alla possibilità di avere uno scheduling più dinamico.

Oltre a questo, vi è la possibilità di sfruttare nodi che hanno già terminato la loro attività di map o reduce, per eseguire map o reduce su un altro chunk di dati che era in attesa, si parla, in questo caso, di **capacity scheduler**. In questi sistemi non c'è possibilità di fare preemption.

Facebook ha cercato di lavorare in modo da ridurre la granularità dei task di map e reduce definendo dei gruppi di lavoro da mandare sullo stesso nodo per cercare di velocizzare l'esecuzione, si parla in questo caso di **fair scheduler**.

Con un sistema di batching che implementa MapReduce si pensi che il dato sia stabile, i sistemi di stream processing, invece, operano su dati che arrivano in maniera continua. Il **micro-batch** rappresenta un via di mezzo tra i due approcci, si potrebbe pensare di raccogliere in dati stream in un nodo e inviare delle finestre di dati ad una rete MapReduce che processerebbe quei dati in batch. Se la piattaforma di MapReduce è efficiente e i dati in ingresso non arrivano troppo velocemente si ottiene un processamento soft real-time.

## 12.4 Fault-tolerance in DSFS (Distributed Stream Processing System)

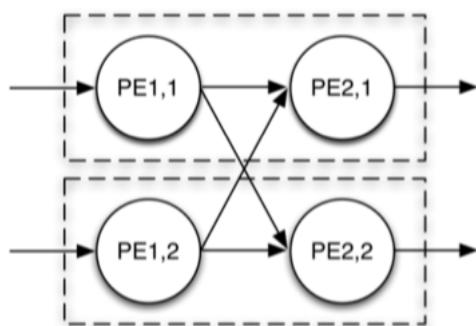
Un numero crescente di applicazioni ha la necessità di gestire, trasformare e analizzare stream di Big Data in modo scalabile ed efficace. Le applicazioni DSFS devono, auspicabilmente, eseguire senza interruzioni; tuttavia, la possibilità che si verifichi un guasto è molto alta soprattutto su larga scala, a causa del maggior numero di punti di failure. Si ha la necessità di fronteggiare efficientemente i guasti, in particolare, gli obiettivi sono quelli di garantire:

- **Availability:** Ovvero far sì che il sistema sia sempre disponibile, per esempio, rimpiazzando, prontamente, eventuali PE guasti.
- **Consistenza:** Mascherare gli effetti dei guasti sull'output, garantendo la gestione dei componenti stateful in maniera appropriata.
- **Costo:** Minimizzare i costi per la gestione dei fallimenti e mantenimento di fault-tolerance in assenza di guasti.

È possibile applicare diverse tecniche e strategie per garantire la fault-tolerance.

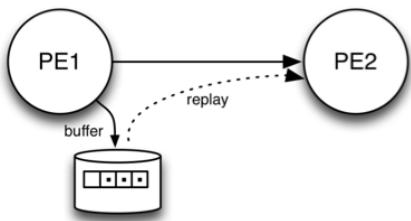
### 12.4.1 Active Replication

Questa tecnica consiste nell'avere due repliche fisiche, in esecuzione, per ogni PE nel grafo di flusso. Entrambi processano lo stesso input, mantenendo così lo stato consistente. Solo la copia primaria emette output, ma in caso di guasto di guasto del primario la copia secondaria può, con latenza minima, fare take over. Questo approccio garantisce una forte consistenza ma un overhead massimo.



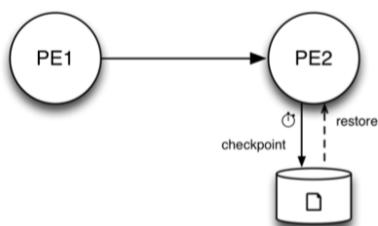
### 12.4.2 UpStream Backup

In questo caso il PE1 salva lo stato in uno storage persistente (con l'ipotesi che lo storage sia stabile e non si guasti). In caso di guasto si effettua il “**replayed**”, ovvero partendo dallo stato salvato dal PE1 il PE2 potrà rieseguire tutte le operazioni perse a causa del guasto. Questo approccio ha latenza molta alta e basso overhead.



### 12.4.3 Checkpointing

Si effettua, periodicamente, un checkpoint dello stato salvandolo su uno storage permanente. Vi è, quindi, un tradeoff tra la frequenza del checkpointing e la latenza (più alta è la frequenza minore è la latenza). Questa strategia comporta un basso overhead a runtime.

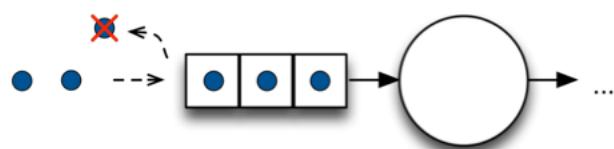


È possibile creare delle soluzioni ibride combinando alcune di queste strategie:

Technique	Latency	Consistency	Cost
Active Replication	Very Low	Strong	High
Upstream Backup	Very High	Strong	Low
Checkpointing	Low	Weak	Low
Checkpointing + Upstream Backup	High	Strong	Low

## 12.5 Gestione della variazione di carico

Molti flussi di dati nel mondo reale sono caratterizzati da una variabilità della dinamica di input (e.g. modifiche improvvise del datarate). Forti variazioni, dinamiche, del carico possono comportare un accodamento dei dati alle porte di input dei PE e conseguente perdita di dati.



Esistono diversi approcci per la gestione della variability del carico:

- **Over-provisioning:** Si allocano staticamente risorse sufficienti per gestire picchi di carico. Il cloud AWS di amazon nasce grazie a questo approccio, amazon ha fatto overprovisioning per gestire i picchi di carico (e.g black Friday) le risorse allocate sono state poi utilizzate per creare il cloud amazon.
- **Back-pressure:** Questo approccio consiste nel rallentare, all'indietro, i componenti che stanno procurando il superamento del traffico che si riesce a smaltire.
- **Load-shedding:** Invece di scartare tuple a caso, si cerca di fare dropping delle tuple di minore importanza.
- **Dynamic PE Relocation:** Consiste nello spostare i componenti di processing dai nodi overloaded a nodi sottoutilizzati

	<b>Latenza</b>	<b>Perdita Dati</b>	<b>Costo</b>
Over-provisioning	<b>Bassa</b>	<b>Nessuna</b>	<b>Alto</b>
Back-pressure	<b>Alta</b>	<b>Nessuna*</b>	<b>Basso</b>
Semantic Load Shedding	<b>Bassa</b>	<b>Sì</b>	<b>Basso</b>
Dynamic PE relocation	<b>Dipende</b>	<b>Nessuna</b>	<b>Dipende</b>

### 12.5.1 LAAR

È possibile riutilizzare temporaneamente le risorse normalmente dedicate a fault-tolerance per la gestione efficace di picchi di traffico. LAAR è una variante dinamica di tecniche di **Active Replication**: le repliche dei PE sono attivate/disattivate dinamicamente in dipendenza dal carico di sistema corrente e sui requisiti di garanzia di consistenza.

## 12.6 Digital Twins

Un gemello digitale è la rappresentazione virtuale di un'entità fisica, vivente o non vivente, di una persona o di un sistema anche complesso.

La componente digitale è in qualche modo connessa con la parte fisica, con la quale può scambiare dati e informazioni, sia in modalità sincrona, che asincrona.

Il gemello digitale può evolversi fino a diventare una vera e propria replica digitale di risorse fisiche potenziali ed effettive (gemello fisico), di processi, di persone, di luoghi, di infrastrutture, di sistemi e dispositivi che possono essere utilizzati per vari scopi.

La componente digitale può includere tutte le informazioni relative al ciclo di vita degli elementi fisici che rappresenta.

In termini generali le principali caratteristiche del gemello digitale sono:

- L'insieme dei dati e delle informazioni in qualunque modo riferibili alle entità rappresentate dal gemello digitale.

- La connessione tra gli elementi della componente fisica con la corrispondente parte virtuale.
- La possibilità di accesso ubiquitario a dati e risorse informatiche attraverso il web, con possibilità di ricerca e analisi delle informazioni (big data, machine learning, intelligenza artificiale).
- Lo scambio di dati e informazioni tra la componente virtuale e quella fisica, con utilizzo di sensori e attuatori.

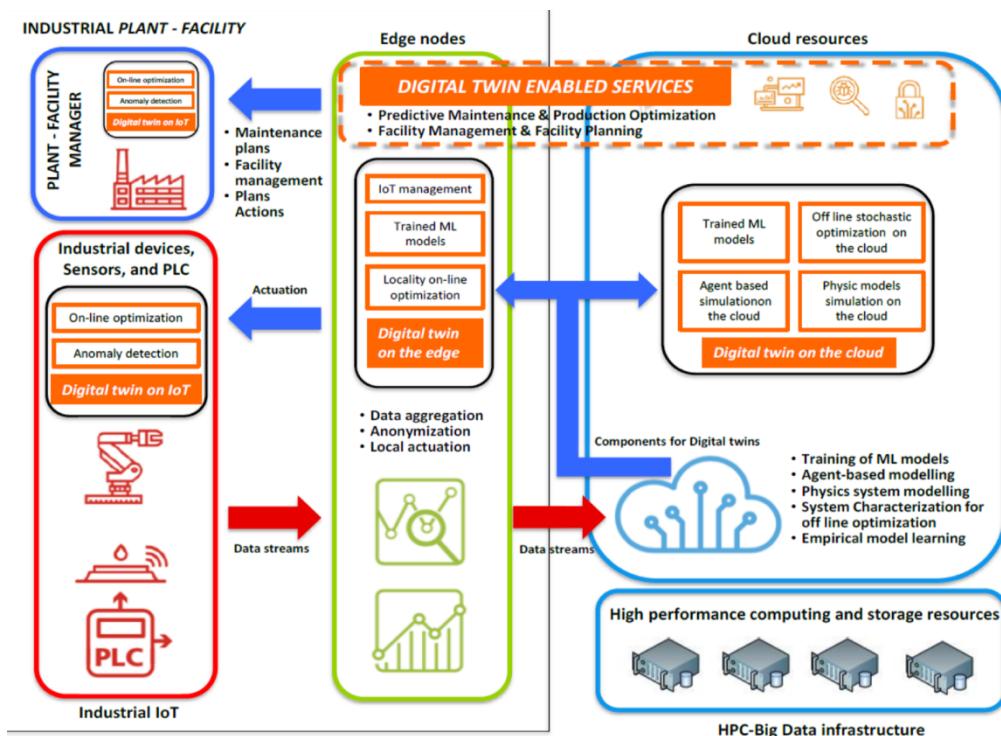
### 12.6.1 IoTwins

**IoTwins** è un progetto Europeo che ha lo scopo di abbassare le barriere all'adozione delle tecnologie Industria 4.0, consentendo di ottimizzare i processi e aumentare la produttività, la sicurezza, la resilienza e l'impatto ambientale.

L'approccio di IoTwins si basa su una piattaforma tecnologica che consente un accesso semplice ed economico alle funzionalità di analisi dei big data, ai servizi di intelligenza artificiale e all'infrastruttura edge cloud per la fornitura di gemelli digitali nei settori della produzione e del facility management.

L'obiettivo è quello di costruire un'architettura di riferimento per i gemelli digitali, allo scopo di rilevare e diagnosticare anomalie, determinare un insieme ottimale di azioni che massimizzano le metriche chiave delle prestazioni, per imporre la gestione on-line della qualità dei processi di produzione in condizioni di latenza e vincoli di affidabilità e per fornire previsioni per la pianificazione strategica e per creare nuovi servizi e modelli di business.

IoTwins propone un'organizzazione gerarchica e un'interoperabilità dei gemelli digitali:



## 13. Cenni di Node-JS

Node.js è una tecnologia Javascript che si utilizza lato server, supporta efficientemente l'I/O in modo asincrono non-bloccante, non utilizza thread/processi dedicati, garantisce scalabilità, ed esaspera il concetto di server stateless.

Node.js è molto utilizzato per lo sviluppo di applicazioni web, è possibile usarlo sia per chiamare le funzioni “core” su file system e networking ma attraverso l’uso di framework più larghi è usato per lo sviluppo delle applicazioni web stesse.

L’uso di JavaScript sia lato cliente che lato server offre diversi vantaggi, tra cui, il non uso del context switch. Se si sta programmando lato client si utilizza il DOM e non si accede al persistent storage mentre lato server si è più interessanti a lavorare con lo storage.

Per usare Node.js è necessario renderlo fruibile, è necessario appoggiarsi ad un run-time environment del linguaggio JavaScript supportato da Google Chrome V8 engine. Il codice JavaScript viene compilato con una buona efficienza a run-time. Vi è la massima concorrenza e scalabilità, di solito si evita di utilizzare più thread dedicati e concorrenti. Il funzionamento è sempre non bloccante, persino per chiamate I/O-oriented.

### 13.1 Event Loop

L’idea di base è gestire tutto con l’event loop: un ciclo infinito di elaborazione che si appoggia sull’utilizzo di uno stack o coda (dipende dall’implementazione). Sullo stack vengono depositate le operazioni da eseguire, che vengono processate una per volta e non appena eseguite vengono rimosse dallo stack. In sintesi, al posto dei thread, si usa un event loop con stack che riduce fortemente overhead di context switching. L’event loop utilizza il framework CommonJS, leggermente più simile a un vero linguaggio di programmazione OO.

Ha senso utilizzare questa strategia perché a seconda del tipo di operazione che bisogna eseguire, ad esempio l’accesso alle cache di primo o secondo livello, alla memoria, al disco o alla rete, varia il numero di cicli di CPU necessari per effettuare l’operazione. Quindi se un’operazione si blocca in attesa di queste risorse, risorse attive come i thread rimangono inutilizzate.

### 13.2 Thread vs Asynchronous Event Driven

Nelle applicazioni **event-driven** vi è un solo processo che effettua il fetching di eventi da una coda, li estrae e li processa. Nelle applicazioni **multi-threaded** tipicamente si lavora bloccando il chiamante in attesa che arrivi del lavoro da smaltire da un thread listener che poi delega la computazione a un pool di thread.

Il modello di base delle applicazioni **multi-threaded** è quello di incoming request, cioè si aspetta l’arrivo di richieste, invece nel caso di applicazione **event-driven** vi è una coda di eventi da processare che vengono processati dal singolo processo dell’event-loop.

In un server **multi-threaded** se una richiesta prevede diverse operazioni i thread si bloccheranno per molto tempo nell’attesa che tutte le operazioni vengano effettuate, nel caso **event-driven**, l’idea è quella di mantenere uno stato aggiornato man mano che arrivano nuovi dati per capire quando è il momento di processare l’evento.

Nelle applicazioni **multi-threaded** vi è context switching per passare da un thread bloccato in attesa di una risorsa ad un altro, ciò causa un notevole overhead, mentre nel caso dell'**event-driven** non c'è context switch, la CPU è sempre detenuta dall'event loop, questo in termini di scalabilità ha un grosso vantaggio relativo all'overhead creato dal cambio di contesto dal caricamento del processo e dallo scheduling di quello successivo relativo al context switch.

Nei sistemi **multi-thread** a causa degli accessi concorrenti a uno stato condiviso vi è anche il problema della sincronizzazione in accesso, quindi locking delle risorse, nelle applicazioni event-driven asincrone non esiste questo tipo di problema essendovi un unico processo. Se l'applicazione è event driven è necessario un supporto diverso che possiede primitive asincrone e non bloccanti.

Thread	Asynchronous Event-driven
Blocca applicazione e richieste con listener-worker thread	Un solo thread, che fa ripetutamente fetching di eventi da una coda
Usa modello incoming-request	Usa una coda di eventi e processa eventi presenti
Multi-threaded server potrebbe bloccare una richiesta che coinvolge eventi multipli	Salva lo stato e passa poi a processare il prossimo evento in coda
Usa context switching	NO contention e NO context switch
Usa ambienti multithreading in cui listener e worker thread spesso acquisiscono incoming-request lock	Usa framework con meccanismi per cosiddetto I/O asincrono (callback, NO poll/select, O_NONBLOCK)

### 13.3 Thread vs Eventi

Nelle applicazioni multi-thread per gestire una richiesta a un server web, si ha la necessità di invocare l'operazione **readRequest** sulla socket, poi si processa la richiesta e si manda una risposta, **sendReply**, che rappresenta l'operazione di scrittura:

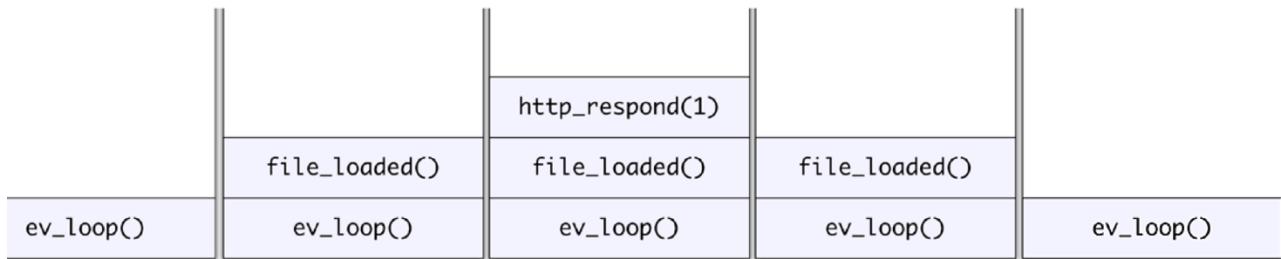
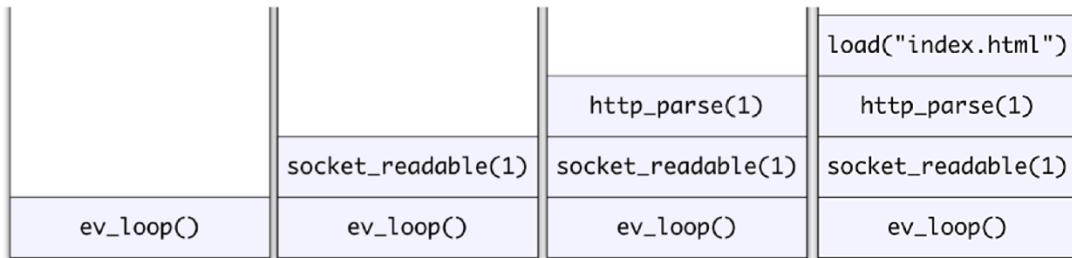
```
request = readRequest(socket);
reply = processRequest(request);
sendReply(socket, reply);
```

L'operazione bloccante è la **readRequest** su cui il thread si blocca in attesa. Sull'operazione di lettura il processo si sospende e quindi avviene il context switching.

Nel modello a eventi la prima invocazione è la **startRequest** sulla socket, che si mette in ricezione di una richiesta da un potenziale cliente. Quando la richiesta arriva, viene chiamato, a callback, il metodo di processamento. Quando l'evento di processing è terminato a callback viene invocato il metodo di replay. Il modello lavora a callback, il sistema processa le operazioni quando l'evento si verifica. In questa implementazione non c'è context switching, ma vi è un event loop che consuma le operazioni richieste una per volta.

```
readRequest(socket, function(request) {
  processRequest(request,
    function (reply) {
      sendReply(socket, reply);
    });
})
```

Di seguito è riportato un esempio di stack con event loop, sullo stack sono caricate di volta in volta le richieste che si vogliono eseguire, per esempio volendo servire una richiesta “index.html”:



Vi è sempre un evento di base che è l'event loop idle, in attesa di eventi da processare. Quando arriva una richiesta HTTP, l'evento di socket\_readable viene caricato sulla coda, si crea un nuovo un evento per il parsing della richiesta, per farlo è necessario caricare il file index.html. Quando il caricamento è avvenuto vengono gestite le richieste con la callback fino a svuotare la coda.

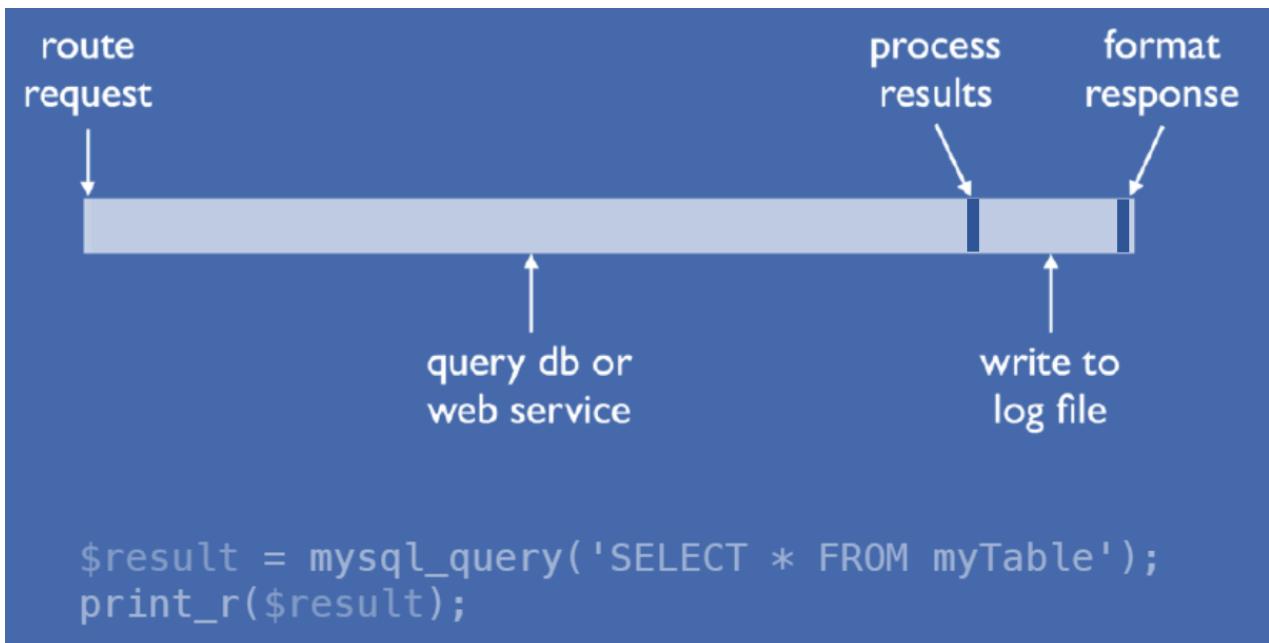
L'event loop è costituito da un while infinito che effettua la **pop()** (estrae dallo stack l'evento) e la **call()** all'arrivo di una richiesta:

```
while(true) {
    if (!eventQueue.isEmpty()) {
        eventQueue.pop().call();
    }
}
```

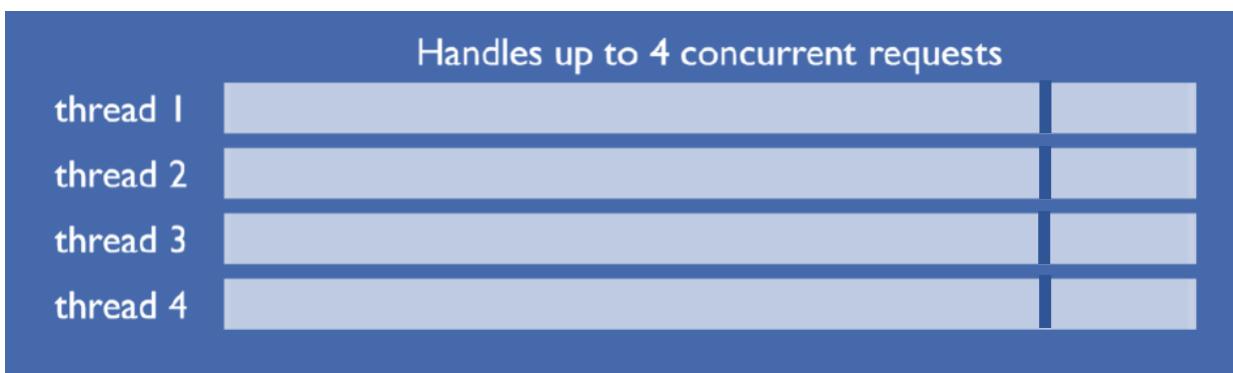
Se un'operazione di **call()** dura molto tempo è necessario che restituire subito il controllo. Quando l'operazione che ha richiesto molto tempo (es. lettura su DB) termina, questo provoca un **push()** di un evento sullo stack. In questo modo si ottiene la massima sincronicità e non serve più effettuare context switching.

### Operazioni I/O

In caso di operazione sincrona si ha un'attesa molto lunga (fino al punto evidenziato in blu scuro), quindi si inizia a eseguire solo dal momento in cui è presente il risultato e solo a quel punto si effettua il processing del risultato stesso.



Nel caso asincrono con thread in parallelo si ha lo stesso problema cioè vi è sempre molto attesa. Inoltre, creare nuovi thread ha in sé un costo elevato:

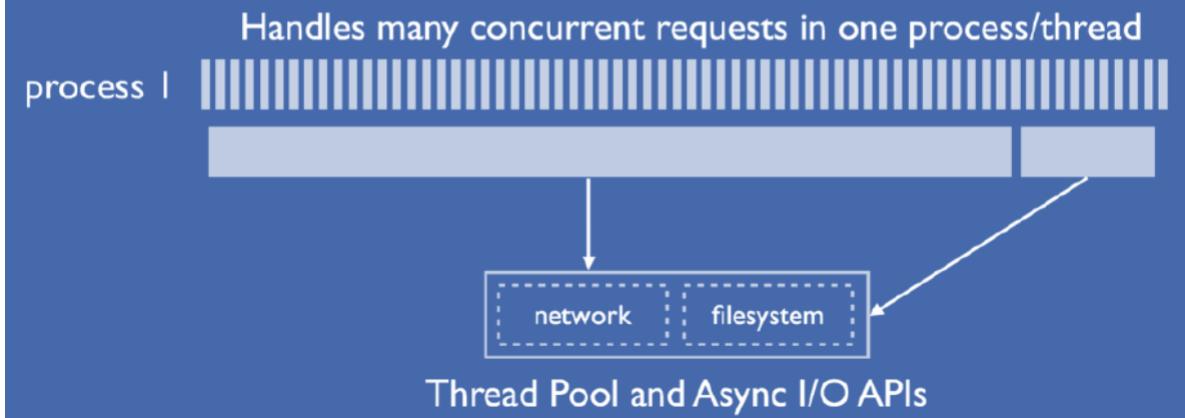


Se si gestisce tutto in modo asincrono non bloccante l'approccio diventa single-thread. Quando si interroga il db si passa la query e una funzione, nella function stessa si definisce la callback da invocare nel momento in cui arriva il risultato. La chiamata non è bloccante poiché nel momento in cui il risultato è disponibile invoca la doSomething per gestire l'evento. La cpu in questo modo è sempre occupata a eseguire funzioni, e non si paga l'overhead del context switch.

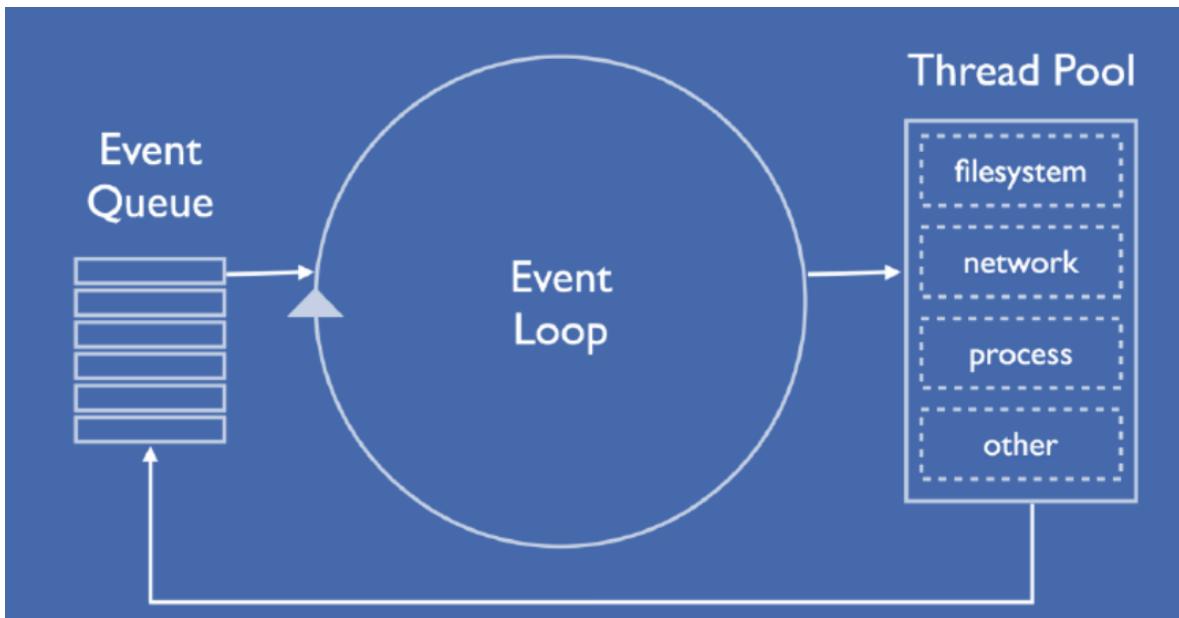
```

db.query("select..", function (result) {
    doSomething(result);
});
nextTask();

```



È importante che la funzione di callback non richieda molto tempo per l'esecuzione. I task “CPU intensive” vanno contro al modello stesso e possono portare a ritardi consistenti e blocchi, non garantendo più la concorrenza. Il tutto funziona bene dal momento che la computazione è parcellizzata in modo corretto, con task che eseguono in modo concorrente, senza che nessuno di questi si appropri per troppo tempo della risorsa e ritardi l'esecuzione di tutti gli altri.



Gestendo in questo modo gli eventi si possono associare alle operazioni di input output delle callback, così da rendere il servitore maggiormente scalabile non introducendo il context switch.

Per eseguire operazioni come operazioni su DB, letture di file da file system, ci sono dei thread dedicati ma il loro numero è molto inferiore a quello che si avrebbe adottando un approccio basato su thread. Infatti, applicazioni JavaScript (di scripting in generale) in attesa su richieste I/O tendono a degradare significativamente le performance. Per evitare blocking, Node.js utilizza la stessa natura event driven di JavaScript, associando callback alla ricezione di richieste I/O. Gli

script Node.js in attesa su I/O non sprecano molte risorse (popped off dallo stack automaticamente quando il loro codice non-I/O related termina la sua esecuzione).

I vantaggi di questo approccio sono l'utilizzo dello stesso JavaScript engine lato cliente e lato servitore, senza bisogno di DOM lato server. La gestione degli eventi avviene su una coda degli eventi. Ogni operazione esegue come una chiamata a funzione effettuata dall'event loop. Con l'Event loop viene effettuato il wrapping di tutte le chiamate bloccanti di SO (I/O su file e socket/network). Node.js usa un sistema di moduli (import/export). I moduli sono specializzati per il supporto a data management efficiente.

In particolare, Node.js è molto utilizzato per la scrittura di applicazioni web, è possibile usarlo sia per chiamare le funzioni core su file system e networking, ma anche su framework più larghi per facilitare lo sviluppo delle applicazioni web stesse, ed è associabile anche a devops e tecnologie frontend.

Viene esasperato il concetto di server stateless, perché il server non mantiene stato. Ogni evento viene messo sulla coda e l'event loop processa un evento indipendentemente dal precedente. Dato che è costoso adottarlo in questo modello, se ci fosse bisogno di stato bisognerebbe usare i cookie e quindi, lo stato viene mantenuto lato client.

### 13.3.1 Esempio di uso di Node.js in server-side script PHP

Nel primo caso viene effettuata l'invocazione e poi viene fatto il fetching del risultato in questo caso il processo si blocca finché non viene restituito il risultato.

```
<?php
$result = mysql_query('SELECT * FROM ...');
while($r = mysql_fetch_array($result)){
    //Do something
}
...
?>
```

Qui invece è presente una funzione annidata in un'altra per restituire il risultato solo quando questo è presente, lo script può dedicarsi a fare altro nel frattempo.

```
<script type="text/javascript">
mysql.query('SELECT * FROM ...', function (err, result, fields){
    //Do something
});

//Continue execution without waiting
</script>
```

### 13.3.2 Stili di programmazione: thread vs callback

In un approccio a thread la logica eseguita è sequenziale, quindi, al termine di tutti gli step viene stampato il risultato. Nel caso degli eventi si invoca lo step 1 che inserisce sulla coda una funzione da valutare che è r1, questa funzione è invocata in callback quando è disponibile l'evento r1. Successivamente si passa a r2 che a sua volta recupererà il risultato r2 e il tutto funziona ancora avendo un'esecuzione sequenziale dei vari step, ma contale esecuzione, questo pezzo di codice

è gestito in modo asincrono e quindi la logica applicativa procederà con la valutazione degli step quando disponibili, il processo non viene bloccato, il processo continua la computazione di altri eventi. All-done viene eseguito quando diventa possibile inserire nello stack la richiesta di valutazione della prima funzione, non quando è stato completato tutto. Se spostato dentro step 3 invece è corretto e il funzionamento è analogo alla programmazione a thread.

## Threads

```
r1 = step1();
console.log('step1 done', r1);
r2 = step2(r1);
console.log('step2 done', r2);
r3 = step3(r2);
console.log('step3 done', r3);
console.log('All Done!');
```

Works for **non-blocking**  
calls in both styles

## Threads

```
r1 = step1();
console.log('step1 done', r1);
r2 = step2(r1);
console.log('step2 done', r2);
r3 = step3(r2);
console.log('step3 done', r3);
console.log('All Done!');
```

## Callbacks

```
step1(function(r1) {
  console.log('step1 done', r1);
  step2(r1, function (r2) {
    console.log('step2 done', r2);
    step3(r2, function (r3) {
      console.log('step3 done', r3);
    });
  });
});
console.log('All Done!'); // Wrong!
```

## Callbacks

```
step1(function(r1) {
  console.log('step1 done', r1);
  step2(r1, function (r2) {
    console.log('step2 done', r2);
    step3(r2, function (r3) {
      console.log('step3 done', r3);
      console.log('All Done!');
    });
  });
});
```

## 13.4 Moduli

Il core di Node.js consiste di circa una ventina di moduli, alcuni di più basso livello come per la gestione di eventi e stream, altri di più alto livello come HTTP. Quindi con Node.js è possibile lavorare in maniera modulare, ogni modulo realizza funzionalità e logica applicativa consentendo di non scrivere sempre lo stesso codice.

Il modulo più usato è HTTP una volta creato il modulo con la require, si può richiedere la creazione di un http server che sia in grado di rispondere con un “hello world” e si può invocare questo server. Il tutto è gestito con funzioni non bloccanti ed è in grado di reagire generando una risposta http quando riceve una richiesta. A questo punto si può mettere in ascolto il servitore sulla porta 8000.

```

// Carica il modulo http per creare un http server
var http=require('http');
// Configura HTTP server per rispondere con Hello World
var server=http.createServer(function(request,response) {
response.writeHead(200, {"Content-Type":"text/plain"});
response.end("Hello World\n");
});
// Ascolta su porta 8000
server.listen(8000);
// Scrive un messaggio sulla console terminale
console.log("Server running at http://127.0.0.1:8000/");

```

Il core di Node.js è stato progettato per essere piccolo e snello, i moduli che fanno parte del core si focalizzano su protocolli e formati di uso comune. Per ogni altra cosa, si usa NPM.

## NPM

NPM è un package manager di grande successo e in forte crescita, che semplifica sharing e riuso di codice JavaScript in forma di moduli. In generale è preinstallato con distribuzione Node, esegue tramite linea di comando e permette di ritrovare moduli dal registry pubblico.

## FS

Il modulo di gestione dei file si chiama FS, è un modulo per file piccoli che possono stare in memoria. Una volta caricato il modulo con la `require()` si può effettuare la `read()`, questo metodo avvia la lettura passando la funzione che dovrà essere chiamata a callback al termine della lettura.

```

var fs = require("fs");
// modulo fs richiesto oggetto fs fa da wrapper a chiamate bloccanti sui file
// read() a livello SO è sincrona bloccante mentre
// fs.readFile è non-bloccante
fs.readFile("smallFile", readDoneCallback); // inizio lettura

function readDoneCallback(error, dataBuffer) {
    // convenzione Node per callback: primo argomento è oggetto
    // js di errore
    if (!error) {
        console.log("smallFile contents", dataBuffer.toString());
    }
}

```

Listener è la funzione da chiamare quando un evento associato viene lanciato, mentre Emitter è il segnale che un evento è accaduto. L'emissione di un evento causa invocazione di tutte le funzioni listener. In seguito a emit, i listener sono invocati in modo sincrono-bloccante nell'ordine con cui sono stati registrati. Senza listener non sono possibili operazioni.

```

myEmitter.on('myEvent', function(param1, param2) {
    console.log('myEvent occurred with ' + param1 + ' and ' + param2 + '!');
});
myEmitter.emit('myEvent', 'arg1', 'arg2');

```

## Modulo Stream

Node.js contiene moduli che producono/consumano flussi di dati (stream), questo può essere molto utile per elaborare contenuti di grosse dimensioni e spezzare il carico in arrivo. Questi stream possono essere elaborati e trasformati dinamicamente. Si possono costruire stream anche dinamicamente e si possono aggiungere moduli sul flusso.

Vi sono diversi tipi di stream: **Readable stream** (es: fs.createReadStream), **Writable stream** (es. fs.createWriteStream), **Duplex stream** (es. net.createConnection), **Transform stream** (es. zlib, crypto).

Per la lettura si crea un read stream e si lavora ad evento, su ogni evento è possibile registrare un listener che viene invocato all'arrivo di ciascun chunk di file:

```
var readableStreamEvent = fs.createReadStream("bigFile");
readableStreamEvent.on('data', function (chunkBuffer) { console.log('got chunk of', chunkBuffer.length);
//operazione eseguita ogni volta che arriva un chunk di dati

readableStreamEvent.on('end', function() {
    // Lanciato dopo che sono stati letti tutti i datachunk fine dello stream
    console.log('got all the data');
});

readableStreamEvent.on('error', function (err) {
    console.error('got error', err);
});
//gestione a evento dell'errore
```

Per quanto riguarda la scrittura è possibile creare uno stream output-file. Quando viene terminata l'operazione di scrittura viene emesso l'evento di fine end. Una volta invocata l'operazione di scrittura sarà anche emesso un evento di fine della scrittura che può essere recuperato e aggiunto al log-file per capire se la scrittura è andata a buon fine.

```
var writableStreamEvent = fs.createWriteStream('outputFile');
writableStreamEvent.on('finish', function () {
    console.log('file has been written!'); });
writableStreamEvent.write('Hello world!\n');
writableStreamEvent.end();
```

## Modulo NET

Questo modulo fa da wrapper per le chiamate di rete di SO, include anche funzionalità di alto livello, come:

```
var net = require('net');
net.createServer(processTCPconnection).listen(4000);
```

Crea una socket per una connessione TCP, fa binding del server su una porta (4000 in questo caso) e si mette in stato di listen per connessioni. Per ogni connessione TCP, invoca la funzione processTCPconnection.

```

var clients = []; // Lista di client connessi
function processTCPconnection(socket) {
  clients.push(socket); // Aggiunge il cliente alla lista
  socket.on('data', function (data) {
    broadcast("> " + data, socket); // invia a tutti i dati ricevuti
  });
  socket.on('end', function () {
    clients.splice(clients.indexOf(socket), 1); // remove socket
  });
} // invia messaggio a tutti i clienti
function broadcast(message, sender) {
  clients.forEach(function (client) {
    if (client === sender)
      return;
    client.write(message);
  });
}

```

La funzione processTCPConnection aggiunge la lista di clienti connessi alla socket. Dopo di che all'arrivo dei dati questi possono essere elaborati. In questo caso all'arrivo di un nuovo dato viene invocata la funzione di callback function, successivamente vengono inviati in broadcast tutti i dati ricevuti. Infine, tutti i dati vengono inviati ai vari clienti con la funzione broadcast. L'altro evento importante è la fine della connessione, bisogna rimuovere la connessione dalla lista dei clienti una volta terminata la connessione stessa, ancora una volta tutto è gestito in callback.

## Modulo Express

Express.js è il framework più utilizzato oggi per lo sviluppo di applicazioni Web su Node.js, è molto flessibile, ha ottime performance e consente di utilizzare diverse opzioni e motori di templating. Mette a disposizione eseguibili per la generazione rapida di applicazioni.

```

var express=require('express');
var app=express();
app.get('/',function(req,res) {
  res.send('Hello World!');
});
var server=app.listen(3000,function() {
  var host=server.address().address;
  var port=server.address().port;
  console.log('Listening at http://%s:%s',host,port);
});

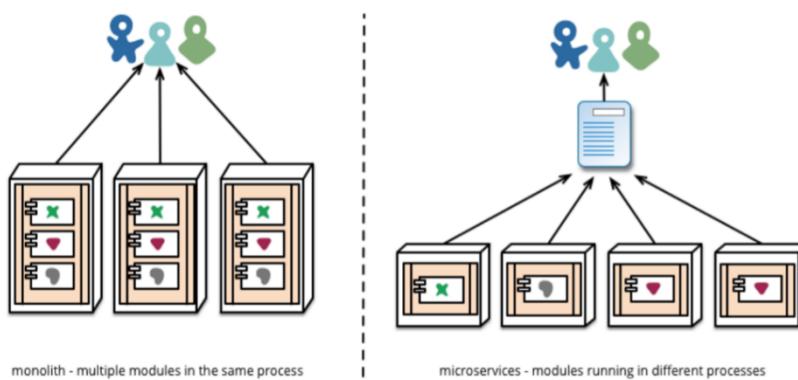
```

# 14. Docker e Orchestrazione Container

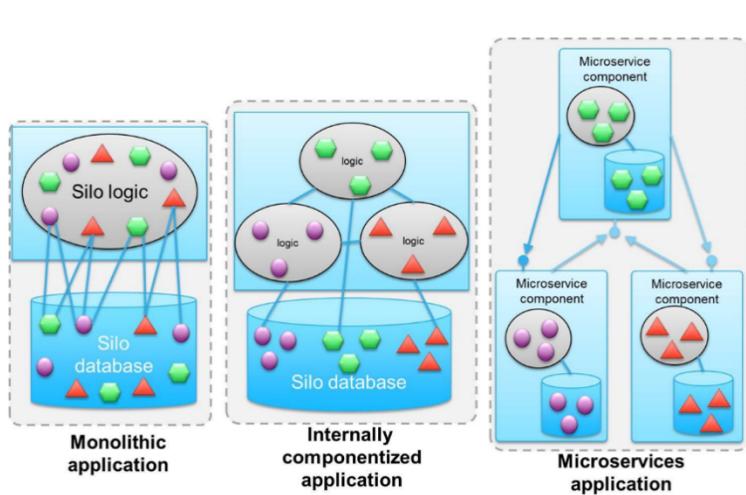
## 14.1 Micro-servizi, DevOps, Container

### Micro-servizi

Per avere applicazioni molto portabili e scalabili è importante esprimere le applicazioni in termini di micro-servizi, ovvero piccoli componenti che compongono un'applicazione distribuita, costituita da servizi "depliabili" separatamente che eseguono funzioni specifiche di business e comunicano con interfacce web. I micro-servizi sono piccoli blocchi di codice riutilizzabili che compongono l'applicazione. L'obiettivo è quello di rendere l'applicazione scalabile e meno sottoposta all'incremento del tempo necessario al deployment nell'ambiente dei DevOps.



I micro-servizi sono un cambio di paradigma rispetto alle applicazioni monolitiche che hanno un codice organizzato in un unico blocco, poiché portano all'isolamento e separazione di quelle funzionalità di base che composte insieme danno le stesse risposte dell'applicazione monolitica iniziale. Il vantaggio enorme introdotto da tale paradigma di programmazione è tangibile in termini di scalabilità dell'applicazione e relative performance, in termini di fault tolerance e manutenibilità dell'applicazione stessa. Al contrario, in una applicazione monolitica è necessario replicare tutta l'applicazione quando magari solo una funzionalità di questa è un collo di bottiglia per la performance. Con la suddivisione in micro-servizi si può replicare solo il micro-servizio che fa collo di bottiglia. Questo è molto interessante in un'architettura orizzontale dove si replicano sui vari host solo i servizi più richiesti.



Le applicazioni monolitiche hanno una così detta struttura a silos con una parte di applicazione e una parte di database. Vi è una fase intermedia detta Internally Componentized Application, visibile in framework come EJB dove i container accedono a un unico database, per poi passare all'architettura a micro-servizi dove ogni servizio ha la sua parte di database replicata. Questo concetto che può inizialmente sembrare contro intuitivo in realtà aumenta molto la scalabilità del sistema, poiché il database relazionale può diventare un collo di bottiglia negli accessi nelle letture e scritture consistenti. Inoltre, non vi è più un singolo punto di fallimento e quindi in caso di crash del database di un micro-servizio gli altri rimangono **up and running** senza essere affetti dal fallimento del singolo. Infine, si può utilizzare il tipo di storage migliore per la necessità del servizio, per esempio storage relazionale o non relazionale.

## Devops

I **DE**veloping **O**perationS (**DevOps**) facilitano lo sviluppo agile dell'applicazione consentendo di unire la parte applicativa a quella infrastrutturale, velocizzando il processo di cambiamento e correzione delle applicazioni. L'idea di mettere insieme micro-servizi e DevOps sopraggiunge per la necessità di preparare l'ambiente per controllare le nuove release, installarle, verificarle e poter facilmente tornare indietro, garantendo consistenza ed evitando crash e altri problemi.

I DevOps vanno nella direzione del continuo ciclo di sviluppo, seguendo le fasi di Design Build Deploy Test e Release senza mai fermarsi in un ciclo infinito. Per questo un'applicazione può continuamente essere aggiornata durante la sua esecuzione senza la necessità di interrompere o interferire con l'attuale versione. Il continuo processo di test e rilascio è per struttura più semplice da utilizzare in un'applicazione a micro-servizi.

## Container

Il container è un artefatto che si porta dietro tutto il necessario per l'esecuzione del micro-servizio, non sono necessari altri supporti. I micro-servizi possono essere contenuti in un container che diventa un modo per trasportare il micro-servizio da una parte ad un'altra in modo che siano auto-contenuti, ovvero contiene non solo la logica applicativa, ma anche la parte di deployment e operation garantendo la correttezza dell'esecuzione. Il container richiama sia il concetto di auto contenimento che di standardizzazione.



I container al contrario delle macchine virtuali consentono di non avere più il sistema operativo guest in ciascuna virtualizzazione, ma tale sistema operativo si trova al di sopra dell'infrastruttura. Per la gestione dei container vi è un container engine che poggia sul sistema operativo e garantisce l'isolamento dei container e dà la visione al container di avere accesso diretto alle chiamate del sistema operativo e alle risorse. Quindi, il container engine crea spazi

isolati per i container dandogli l'astrazione necessaria rispetto al sistema operativo. Dal punto di vista delle performance i container sono molto più veloci; per l'avvio di un container si parla di centesimi di secondi/secondi, mentre per una virtual machine di grosse dimensioni si parla anche di minuti. L'isolamento del container a livello di SO garantisce l'isolamento delle risorse, poiché il container vede il SO come se fosse suo. Dal punto di vista delle risorse, i container le percepiscono isolate anche se alcune risorse come networking e memoria non sono semplici da isolare.

	<b>Process</b>	<b>Container</b>	<b>VM</b>
<b>Definition</b>	A representation of a running program.	Isolated group of processes managed by a shared kernel.	A full OS that shares host hardware via a hypervisor.
<b>Use case</b>	Abstraction to store state about a running process.	Creates isolated environments to run many apps.	Creates isolated environments to run many apps.
<b>Type of OS</b>	Same OS and distro as host,	Same kernel, but different distribution.	Multiple independent operating systems.
<b>OS isolation</b>	Memory space and user privileges.	Namespaces and cgroups.	Full OS isolation.
<b>Size</b>	Whatever user's application uses.	Images measured in MB + user's application.	Images measured in GB + user's application.
<b>Lifecycle</b>	Created by forking, can be long or short lived, more often short.	Runs directly on kernel with no boot process, often is short lived.	Has a boot process and is typically long lived.

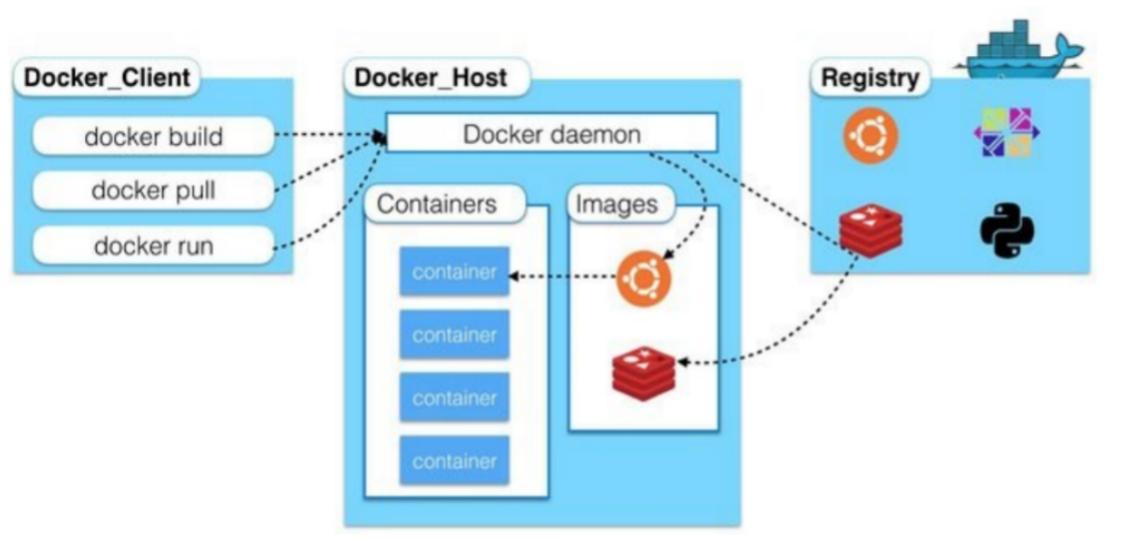
Dal punto di vista del marketing la containerizzazione ha avuto grande successo, poiché risolve a tutti gli effetti il problema “non funziona su questa macchina”. Il container, infatti, risolve tutte le dipendenze oltre che a fornire il servizio preposto, è leggero perché condivide il kernel con altri container ed esegue come un processo isolato. Inoltre, ha una maggiore efficienza rispetto alla virtual machine nei processi di lettura e scrittura. Con l'utilizzo di immagini si hanno servizi facilmente riproducibili e scalabili.

I container sono la giusta piattaforma per un micro-servizio, poiché sono soluzioni leggere e virtualizzate oltre ad essere corretti, auto contenuti e ben isolati, sono in grado allo stesso tempo di condividere le parti di supporto interne al kernel del sistema operativo e non sono specifici per una piattaforma, per questo sono estremamente portabili. Infine, possono ospitare i micro-servizi e le loro applicazioni con tutte le loro parti e le loro dipendenze.

## 14.2 Docker

Docker è un insieme di strumenti che facilitano la gestione di container. Offre molti strumenti non solo per ospitare container ma anche per gestirli, controllare le migrazioni dei componenti e le loro immagini. I micro-servizi possono essere ospitati e controllati dal container facilmente. Docker può consigliare come progettare e pacchettizzare componenti autonomi. Inoltre, i container offrono la possibilità di accedere con funzioni web ai micro-servizi ospitati. Per Docker basta utilizzare una piattaforma hardware, il sistema operativo, le librerie di sistema e le dipendenze dei processi durante la fase di sviluppo di test e produzione del software. Dal punto

di vista dell'ingegneria del software, in cui si è passati da un concetto di applicazione a quello di servizio, il processo di creazione viene basato nella realizzazione sui container e sullo sviluppo con i DevOps.



Docker è organizzato in tre parti:

- Un **registry-repository** da cui scaricare i componenti ovvero le immagini,
- Un **docker client** che comanda l'intero deployment ovvero richiede all'host certe operazioni,
- Un **docker host** che consente di scaricare l'immagini dal registry e adoperarle sul container dove eseguire l'applicazione.

Docker per la configurazione dinamica e statica utilizza orchestratori, che servono per gestire al meglio i micro-servizi in base al loro utilizzo e la loro richiesta. Docker consente la configurazione, il deployment, e la gestione del ciclo di vita dei container. L'architettura è client-server, il docker daemon realizza i servizi per gestire il ciclo di vita dei container, poi vi sono diverse API rest e Docker CLI.

Il Docker Client è la parte utilizzata dagli utenti per interagire con Docker Daemon, in particolare può utilizzare comandi, come **docker run** che avvia il container.

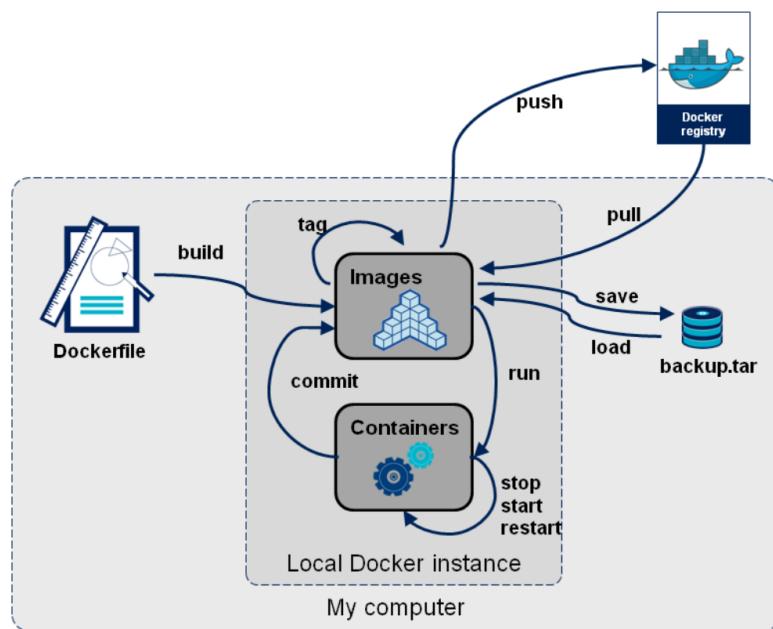
Il Docker Daemon ascolta le richieste dell'API Docker e gestisce i componenti Docker come immagini, contenitori, reti e volumi. I Docker Registry salvano le immagini mentre Docker Hub e Docker Cloud sono registry pubblici che chiunque può utilizzare. Di default Docker cerca immagini su Docker Hub, ma possono anche essere utilizzati registry personali. I registry sono i componenti di distribuzione per Docker.

Le immagini Docker sono organizzate per layer, sono componenti read-only per questo ad ogni nuova immagine si va ad aggiungere una nuova parte, lavorando per differenze, aggiungendo solo le modifiche rispetto all'immagine precedente. Le immagini sono identificate da Hash con convenzioni per il naming. Le immagini Docker sono utilizzate per creare container e sono definite come "build components" di Docker stesso.

I Docker container contengono tutto ciò che è necessario ad un'applicazione, sono simili a una directory in quanto il container in esecuzione vede un file system proprio e isolato rispetto al resto, con l'idea che all'interno di questo file system si trovano tutte le parti utili all'esecuzione.

Si possono usare registry pubblici o locali, i registry pubblici contengono tantissime immagini già pronte, il Docker CLI interagendo con il Docker Daemon può scaricare le immagini e a partire da quelle può istanziare i vari container, ciò può essere fatto sulla macchina locale in locale, oppure su un host remoto utilizzando un registry pubblico, tipicamente Docker Hub, oppure utilizzando dei propri registry privati.

### Gestione del ciclo di vita del container

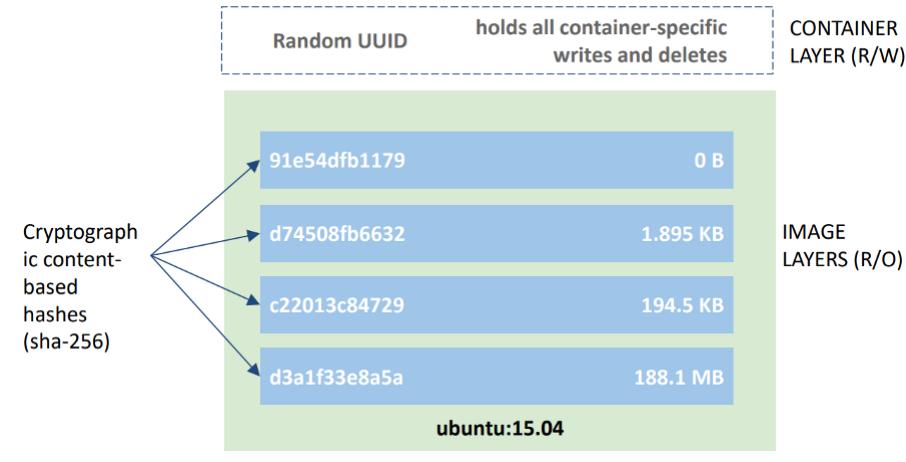


Il ciclo di vita parte dal registry locale o remoto da cui richiedere e scaricare immagini con operazione di pull, oppure inviare aggiornamenti delle immagini presenti con l'operazione di push. Inoltre, è presente un backup locale in cui salvare immagini con save o caricarle con load. Le immagini possono essere istanziate sul container con il comando di run. Il container può essere avviato con start, fermato con stop oppure riavviato con restart. Se vengono effettuati cambiamenti di configurazioni, aggiunte di librerie nel filesystem con la commit si può aggiornare l'immagine ed eventualmente aggiungere nuove informazioni attraverso tag. Il DockerFile in modo assertivo enuncia delle direttive che attraverso l'operazione di build consentono di creare nuove immagini.

### Immagini Docker

La gestione delle immagini in Docker è automatizzata e ottimizzata. Le immagini create sono organizzate su layer, poiché si utilizza una logica di composizione, che consente una volta creata un'immagine di accedervi in sola lettura. Ogni immagine è read-only, per ogni modifica in scrittura viene aggiunto un nuovo layer all'immagine senza modificare il layer precedente, con un approccio copy-on-write. In sintesi, si lavora per differenze ogni nuovo layer aggiunge le differenze rispetto al layer precedente, l'immagine finale è la somma di tutti i layer creati per quell'immagine.

Ogni layer è univocamente identificato da un hash con l'algoritmo SHA-256 crittograficamente sicuro per l'identificazione dell'immagine. Se sono già stati scaricati layer per un'immagine non sarà necessario scaricarli nuovamente, dovranno essere scaricati unicamente i layer mancanti per una determinata immagine. Nel Docker Daemon viene fatto caching di layer e questo evita di scaricare nuovamente gli stessi layer garantendo un grande vantaggio in termini di efficienza nell'utilizzo del disco e facilità nella modifica.



Sono presenti convenzioni per il naming delle immagini, per ciascuna devono essere indicati il nome dell'host nei registry e la porta, username, reponame ed eventuali tag.

[hostname[:port]]/[username]/reponame[:tag]

**Hostname/port** of registry holding the image. If missing, defaults to Docker Hub public registry (docker.io).

**Username**. If missing, defaults to **library** username on Docker Hub, which hosts official, curated images.

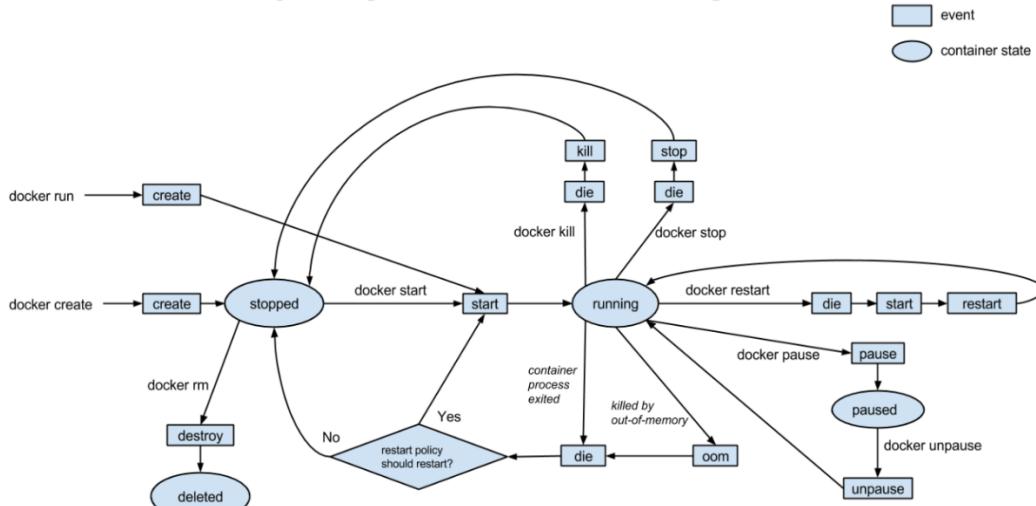
**Reponame**. Actual image repository.

**Tag**. Optional image specification (e.g., version number). If missing, defaults to **latest**.

Il **DockerFile** è un file di configurazione che serve per definire e creare nuove immagini. Il run nel DockerFile è una direttiva per lanciare le configurazioni e il software. Facilita la configurazione dell'immagine e la crea. Il DockerFile mostrato parte da due immagini preesistenti che vengono modificate, con l'aggiunta di layer software vengono aggiunti layer all'immagine stessa.



Docker non è solo un insieme di strumenti ma un framework che può lavorare ad eventi nella gestione dei container in esecuzione. Il programmatore può intercettare tutti gli eventi del container:



Docker possiede molti plugin, tra questi sono molto importanti i plugin di rete, per questo Docker consente di connettere risorse e container all'esterno e all'interno, offre anche virtualizzazioni in un certo senso è un Network as a Service. Per quanto riguarda la persistenza Docker prevede diverse modalità. Per esempio, consente di montare un'area di memoria come un file system ovvero uno storage persistente, che sarà quindi realizzato in memory, dà la possibilità di vedere il file system locale ospitante, oppure di avere una sandbox detta Docker Area che limita lo spazio di visibilità per il container.

Queste configurazioni avanzate vengono indicate nel Docker Compose. Docker compose parte da immagini già scaricate per comporle in un nuovo container. Questo può essere utile per mantenere separate in un'applicazione la parte di stato, dalla logica applicativa. Dal punto di vista del Docker Compose, container diversi vengono visti come un unico, il tutto può essere scorporato ed eseguito separatamente oppure interamente. Con il Docker Compose si agisce solo in locale quindi possono essere utilizzate immagini remote, ma prima devono essere scaricate in locale.

## Esempio applicazione a tre livelli per la gestione della collezione di libri

L'esempio mostrato è un'applicazione a tre livelli con client tier, application tier e database tier. L'application tier utilizza Spring Boot per sviluppare un'applicazione con una configurazione minima, con Spring Boot è possibile utilizzare Tomcat come server embedded. Per quanto riguarda la persistenza si può utilizzare JPA sempre nell'application tier che grazie alle annotazioni risulta semplice da configurare. L'unico file da specificare è application.yml dove scrivere l'URL per accedere al database, username, password ed eventuali parametri aggiuntivi.

```
Spring:
  profiles: container
  datasource:
    driverClassName: ${DATABASE_DRIVER}
    url: jdbc:mysql://${DATABASE_HOST}:${DATABASE_PORT}/${DATABASE_NAME}?useSSL=false&allowPublicKeyRetrieval=true
    username: ${DATABASE_USER}
    password: ${DATABASE_PASSWORD}
  tomcat:
    test-while-idle: true
    time-between-eviction-runs-millis: 60000
    validation-query: SELECT 1
  jpa:
    hibernate.ddl-auto: create-drop
    properties.hibernate.dialect: org.hibernate.dialect.MySQL5Dialect
```

L'applicazione può essere organizzata creando un container per Spring e un container per MySQL e i volumi. Il primo passo per definire l'immagine per ciascun container è scrivere il DockerFile per definire i passaggi di creazione dell'immagine e metterla in esecuzione. Una volta definito il DockerFile, viene scaricata la jdk e vengono aggiunte le parti che servono a configurare l'applicazione da distribuire. Per quanto riguarda invece l'immagine di MySQL si utilizza l'immagine di MySQL presente nel DockerHub. Rispetto al database va scritto uno script che verifichi che la connessione funzioni e che si connetta correttamente al database. Infine, notare come immagine pesi solo 116 MB, molto più leggera rispetto all'immagine di una Virtual Machine.

```
FROM openjdk:8-jre-alpine
MAINTAINER Dmitrij David Padalino Montenero

VOLUME /tmp
EXPOSE 8080
ARG JAR_FILE
ADD target/${JAR_FILE} app.jar
ADD wrapper.sh wrapper.sh

RUN apk add --update bash && rm -rf /var/cache/apk/*
RUN bash -c 'chmod +x /wrapper.sh'
RUN bash -c 'touch /app.jar'

ENTRYPOINT ["/bin/bash", "/wrapper.sh"]
```

```
#!/bin/bash
while ! exec 6</dev/tcp/${DATABASE_HOST}/${DATABASE_PORT}; do
  echo "Trying to connect to MySQL at ${DATABASE_PORT}..."
  sleep 10
done
java -Djava.security.egd=file:/dev/.urandom -Dspring.profiles.active=container -jar /app.jar
```

```
$ docker build -t librarydemo
$ docker login
$ docker tag librarydemo davidmonnua/springbootlibrarydemo:1.0
$ docker push davidmonnua/springbootlibrarydemo:1.0
```

## Docker SWARM

Docker Swarm è un orchestratore di container, è un engine che elabora degli script di configurazione di un cluster ed esegue in continuazione garantendo che ciò che viene scritto nello script sia realizzato nel sistema che sta eseguendo. Nella terminologia di Docker e Swarm si utilizza il termine servizio che sta ad indicare i container di produzione che istanziano un servizio concettuale. Un altro termine importante è Stack che sta ad indicare un gruppo di servizi in relazione tra di loro che condividono dipendenze e possono essere orchestrati e scalati insieme.

Docker Swarm è un gruppo di macchine fisiche o virtuali che eseguono Docker all'interno di un cluster. Le macchine su cui Docker viene eseguito possono essere fisiche o a loro volta virtual machine. Nell'esempio sottostante viene mostrata l'inizializzazione di un nodo master e un nodo worker.

### Virtual machines creation

```
$ docker-machine create -driver virtualbox myvm1
$ docker-machine create -driver virtualbox myvm2
$ docker-machine ls
NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS
myvm1 - virtualbox Running tcp://192.168.99.100:2376 v18.06.1-ce
myvm2 - virtualbox Running tcp://192.168.99.101:2376 v18.06.1-ce
```

### Cluster initialization

```
$ docker-machine ssh myvm1 "docker swarm init --advertise-addr 192.168.99.100:2377
Swarm initialized: current node 0unmutpbeeytxpquhiy1b6ok is now a manager.
To add a worker to this swarm, run the following command:
docker swarm join \
--token SWMTKN-1-4y8bpixrbno89dapkkwyjdw3268qfpz128tpf5hecjdti5hb1k-00oatopl9dzw3jejbietxa9vp 192.168.99.100:2377

$ docker-machine ssh myvm2 "docker swarm join --token SWMTKN-1-4y8bpixrbno89dapkkwyjdw3268qfpz128tpf5hecjdti5hb1k00oat

$ docker-machine ssh myvm1 "docker node ls"
ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS ENGINE VERSION
0unmutpbeeytxpquhiy1b6ok * myvm1 Ready Active Leader 18.06.1-ce
ktvyi0nypz31l645iej8torkx myvm2 Ready Active 18.06.1-ce
```

Nel caso dell'esempio proposto i servizi presenti sono un database SQL “mysqlDb”, un servizio “spring”, un “virtualizer”, poi vi sono il volume per la persistenza e la rete per interconnettere i container. Effettuare un comando di Docker Stack vuol dire disiegare l'applicazione con tutte le sue dipendenze.

Il file di deployment viene letto da Docker Swarm, inizialmente si definiscono i servizi, che sono il servizio del database “mysqlDb” a cui richiediamo l'ultima versione dell'immagine disponibile nel Docker Hub. Il Docker file non è necessario perché si utilizza l'immagine di un Docker Hub pubblico. Poi è necessario prendere i volumi, in modo da salvare il tutto in maniera persistente.

Dal punto di vista del deployment con placement si appone un vincolo sul fatto che i servizi “mysqldb” e il “visualizer” dovranno trovarsi sulla stessa macchina ovvero il nodo con il ruolo di manager. Poi vi sono alcune configurazioni di ambiente, infine il network deve essere gestito con il modulo webnet. Per la parte di application server si definisce il servizio “Spring”. In primo luogo, si richiama l’immagine creata in precedenza per Spring, qui è possibile dichiarare delle dipendenze, in questo caso rispetto al servizio “mysqldb”.

Deploy replicas consente di indicare quante repliche si desiderano per il deployment, oltre ad indicare la politica sul restart a seguito di un fallimento del container, infine è possibile limitare la quantità di risorse da allocare. Vi sono poi alcuni parametri di configurazione, come le porte, l’ambiente di configurazione per “mysqldb” e la rete a cui collegarsi webnet. Per il volume e il network si utilizza il default, ovvero quello locale, a questo punto il deployment è terminato.

Vi sono due macchine virtuali del cluster dette master e worker, MySQL e Visualizer sono sulla stessa macchina, mentre i cinque container Spring replicati sono distribuiti sulle due macchine, come distribuirli tra le due macchine se ne occupa Swarm. Il visualizzatore consente di mostrare il deployment di Docker Swarm. Per la gestione delle richieste che arrivano ai cinque container vi è Swarm load-balancer, che risiede sulle macchine del cluster e all’arrivo delle richieste le smista sui nodi in cui si trova l’application server Spring, in questo modo si può realizzare il load-balancing e scalare l’applicazione.

Swarm però non consente di monitorare in continuazione il funzionamento e scalare dinamicamente i container è, infatti, un orchestratore statico. Permette però cambiando il file di deployment Docker Compose File di indicare un numero diverso di repliche, istanziarne di nuove e quindi scalare l’applicazione. Quando Swarm va nuovamente a consumare il file di deployment modificato, questo avviene però in modo statico non dinamico.

```
$ docker-machine ssh myvm1 "docker stack deploy -c docker-compose.yml librarywebapp"
```

## 14.3 Kubernetes

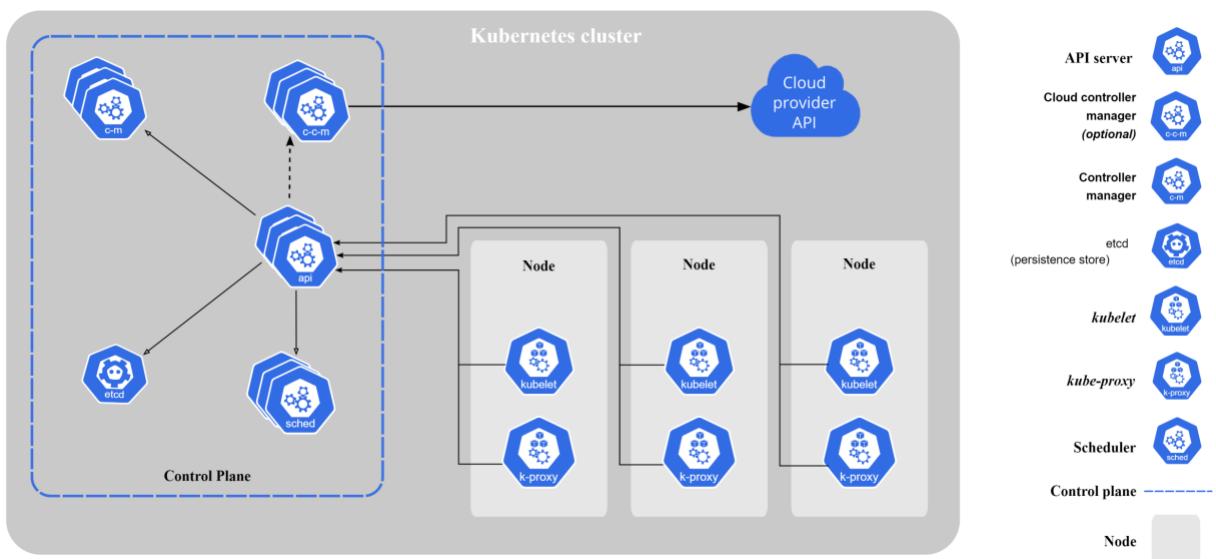
Kubernetes è un orchestratore di container compatibile con Docker, in cui la gestione dei container è dinamica e non statica, questo gli consente di offrire molte più funzionalità rispetto a Docker Swarm. È pensato per lavorare in ambienti cloud, infatti da una parte, quella interna, è un orchestratore di container, mentre la parte esterna si interfaccia con i Virtual Infrastructure Manager, ovvero con i controlli di API Rest, che dispongono i cloud provider per il recupero delle risorse che servono per formare l’infrastruttura fisica su cui distribuire i container. Questo strumento facilita molto la migrazione dei container da un ambiente locale a uno più largo.

Tra i benefici derivanti dall’utilizzo di Kubernetes vi sono la gestione fine grained e dinamica rispetto a tutte le problematiche di management come scaling e automatizzazione del rollout e del rollback. Una volta sviluppata una nuova versione è possibile mandare ovunque in produzione e tornare indietro senza problemi in caso di bug o errori. Kubernetes consente di effettuare il rollback facilmente. Poi vi è la gestione dello scaling con la possibilità di scaling intelligente fine grained e definibile dal programmatore e non solo con Round Robin. Infine, Kubernetes gestisce la fault tolerance e offre la possibilità di portare i carichi su cloud diversi.

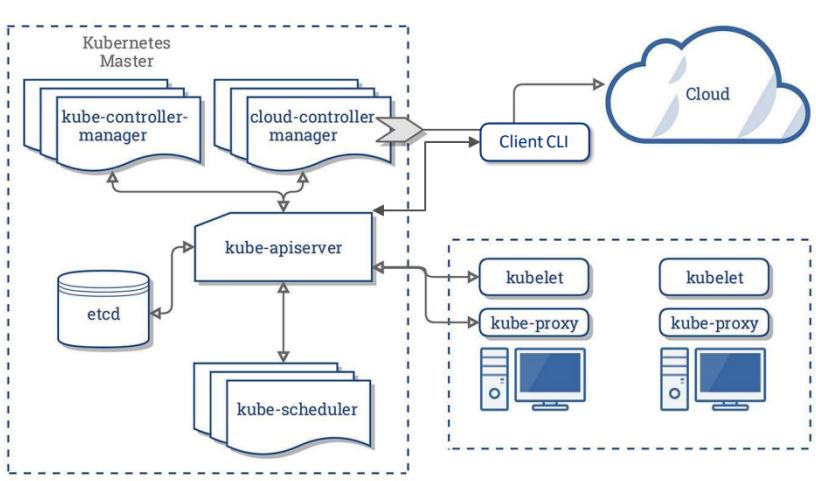
L'obiettivo principale di Kubernetes è nascondere la complessità di gestione di grandi quantità di container fornendo agli utilizzatori un set di API Rest. Kubernetes è estremamente portabile, infatti, si interfaccia con tutte le principali Cloud Platform private o pubbliche come Amazon AWS, Azure, Openstack, o resource manager come Apache Mesos. Per un orchestratore del genere è fondamentale fornire il deployment di applicazioni multi-container, garantire la continuità dei servizi grazie alla sua gestione di fault tolerance, rollback e rollout, scalare autonomamente e dinamicamente le applicazioni e rendere tutta l'architettura indipendente dall'infrastruttura sottostante.

### Architettura di Kubernetes

L'architettura di Kubernetes è abbastanza complessa. Prevede una serie di nodi worker chiamati Node all'interno dei quali è possibile istanziare i container. Questi nodi sono controllati da diversi componenti di controllo racchiusi nel Control Plane, ognuno di loro è dotato di un agente locale Kubelet per interagire con il Control Plane e una parte per la comunicazione detto kube-proxy. L'architettura ha componenti tipici del cloud come il componente API, oppure gli agenti locali su tutti i nodi presenti che consentono di gestire il ciclo di vita dei container. La parte di controllo attraverso il cloud controller manager parla con un cloud provider pubblico attraverso le Cloud Provider API.



L'architettura è di tipo master-slave. Il nodo master può essere uno o più di uno, poiché viene replicato al fine della fault tolerance, le repliche però agiscono come un componente solo. L'Etdc rappresentato come un database è l'unico componente che mantiene lo stato dell'infrastruttura, lo stato è mantenuto il più locale possibile e auto contenuto per evitare di creare delle dipendenze nel distribuito fra le varie macchine, in questo modo tutte le macchine worker possono agire anche nel caso in cui momentaneamente perdano la connettività con il controller per garantire il massimo disaccoppiamento e asincronicità possibile. Infine, vi è il client che può richiedere le funzionalità di Kubernetes per effettuare il deployment delle applicazioni.



## ETCD

L'**ETCD** è uno storage chiave-valore persistente distribuito fortemente consistente che fornisce un modo affidabile per memorizzare i dati, a cui è necessario accedere da a sistema distribuito o cluster di macchine. Gestisce le elezioni dei leader durante le partizioni di rete e può tollerare il guasto della macchina, anche nel nodo master.

## Controller Manager

Il Controller Manager Kubernetes è un demone che incorpora i loop di controllo principali gestiti con Kubernetes. In Kubernetes, un controller è un loop di controllo che osserva lo stato condiviso del cluster attraverso l'API server e apporta modifiche tentando di spostare lo stato corrente verso lo stato desiderato. Esempi di controller forniti oggi con Kubernetes sono il controller di replica, controller degli endpoint, controller dello spazio dei nomi e controller degli account di servizio.

## Cloud Controller Manager

I **Cloud Controller Manager** è un componente del Control Plane di Kubernetes che incorpora la logica di controllo specifica del cloud. Il Cloud Controller Manager consente l'interazione fra il cluster e il provider cloud API e separa i componenti che interagiscono con quella piattaforma cloud da componenti che interagiscono solo con il cluster. Disaccoppiando la logica di interoperabilità tra Kubernetes e l'infrastruttura cloud sottostante, il componente Cloud Controller Manager abilita il cloud provider a rilasciare funzionalità a un ritmo diverso rispetto al progetto Kubernetes. Il Cloud Controller Manager è strutturato utilizzando un meccanismo di plug-in che consente l'integrazione delle piattaforme di diversi provider cloud con Kubernetes.

## Kubelet

Il Kubelet è il principale "node agent" che esegue su ogni nodo, questo componente logico può registrare il nodo con l'APIserver. Il kubelet funziona attraverso la definizione di PodSpec. Un PodSpec è un oggetto YAML o JSON che descrive un pod. Di conseguenza il kubelet accetta una serie di PodSpec forniti attraverso vari meccanismi (principalmente tramite l'APIserver) e assicura che i container descritti in quelle PodSpecs siano running e in salute.

## Pod

Un **Pod** è un'astrazione che rappresenta un gruppo di uno o più application container e alcune risorse condivise per quei container. Tali risorse includono:

- Archiviazione condivisa, come volumi

- Networking, come indirizzo IP clust univoco
- Informazioni su come eseguire ciascun contenitore, come la versione dell'immagine del contenitore o specifiche porte da usare.

Un pod modella un host logico specifico dell'applicazione e può contenere diverse application container che sono strettamente accoppiati.

Per esempio, il pod potrebbe includere sia il contenitore che il Node.js e un container diverso che alimenta i dati da pubblicare dal Node.js web server. I container in un pod condividono l'indirizzo IP e lo spazio delle porte, sono sempre co-localizzati e co-schedulati, ed eseguiti in un contesto condiviso sullo stesso nodo.

I pod sono l'unità atomica della piattaforma Kubernetes. Alla creazione di una distribuzione su Kubernetes, si creano pod con container al loro interno (invece di creare direttamente i container). Ogni Pod è legato al nodo in cui è programmato e vi rimane fino alla cessazione o cancellazione. In caso di guasto di un nodo, i Pod identici sono schedulati su altri nodi disponibili nel cluster.

## **Service**

I **service** sono un modo astratto per esporre un'applicazione in esecuzione su un set di pod in comunicazione uno con l'altro per erogare il service. I pod vengono creati e distrutti per corrispondere allo stato del cluster. Ogni Pod ottiene il proprio indirizzo IP, tuttavia il set di pod in esecuzione in un momento specifico può essere diverso. I service sono astrazioni che definiscono un insieme logico di pod e una policy con cui accedervi.

## **Kube proxy**

Il proxy di rete Kubernetes viene eseguito su ciascun nodo worker. Espone i service definiti come indicato nell'API Kubernetes su ogni nodo e fa un semplice inoltro di flusso TCP, UDP e SCTP o Round Robin TCP, Round Robin UDP e inoltro Round Robin SCTP attraverso un insieme di backend. Gli IP e le porte del cluster di servizio sono ritrovati attraverso un ambiente compatibile con i Docker-links, variabili che specificano le porte aperte dal proxy service. C'è un componente aggiuntivo opzionale che fornisce DNS cluster per questi IP cluster. L'utente deve creare un servizio con l'apiserver API per configurare il proxy.

## **Scheduler**

Lo scheduler è la parte del controllo che si occupa dello scheduling, controlla la creazione dei Pod e trova il nodo migliore per ospitare quel Pod. Filtra i nodi per verificare quale soddisfi i requisiti di programmazione specifici per quel Pod. Se non ci sono nodi adatti, il Pod rimane non schedulabile. Per ogni nodo adatto e utilizzabile, lo Scheduler stima un punteggio eseguendo una serie di funzioni, il Pod è schedulato sul nodo adatto con il più alto punteggio, lo scheduler quindi notifica al server API questa decisione in un processo chiamato binding.

## **Volumi**

Kubernetes supporta diversi tipi di volumi. I due tipi principali sono: i volumi di tipo effimero, che hanno la durata di un Pod, quando un Pod cessa di esistere, Kubernetes distrugge i volumi effimeri. I volumi persistenti che esistono oltre la durata di un Pod, Kubernetes non distrugge i volumi persistenti. Per qualsiasi tipo di volume in un dato Pod, i dati vengono conservati durante i riavvii del contenitore. Al suo interno, un volume è una directory accessibile ai container in un

Pod. I Volumi hanno la massima interoperabilità con gli strumenti preesistenti quindi con tutti gli storage anche cloud.

## Network

Si ricorda che la gestione della rete può essere piuttosto complessa in Kubernetes, poiché vi sono i Pod che si mostrano internamente con indirizzi privati, che però attraverso i servizi di rete possono essere messi in comunicazione con altre macchine, anch'esse con un'interfaccia di rete. Tutti i mapping per far comunicare le macchine sono gestiti dal networking di Kubernetes.

## Modello dichiarativo

In Kubernetes, il modello dichiarativo funziona in questo modo:

- Dichiare lo stato (dal punto di vista del deployment) desiderato delle applicazioni (microservizi) in un file manifest.
- Pubblicare lo stato sul server API Kubernetes.
- Kubernetes memorizza lo stato nell'ETCD (cluster store) come stato desiderato dall'applicazione.
- Kubernetes implementa lo stato desiderato sul cluster.
- Kubernetes implementa i loop, attraverso il **Kube Controller Manager**, per assicurarsi che lo stato attuale dell'applicazione non vari dallo stato desiderato.

## Concetti di base di Kubernetes

### Pod

Nel mondo Kubernetes, l'unità atomica di programmazione è il Pod. Non è possibile eseguire un container direttamente su un Kubernetes cluster come in Docker, il container deve sempre essere eseguito all'interno di Pods. Anche se è possibile eseguire più contenitori all'interno dello stesso Pod, ogni Pod ospita solitamente un contenitore. I Pod sono anche l'unità minima di ridimensionamento. Qualora fosse necessario ridimensionare un'app, si aggiungono o si rimuovono Pod. Non si scala l'applicazione aggiungendo più contenitori a un Pod esistente. I Pod sono mortali e quindi sono inaffidabili. Quando un Pod si interrompe in modo imprevisto, Kubernetes non lo riporterà in vita, ma ne inizializza uno nuovo al suo posto.

### Service

Per superare l'inaffidabilità dei Pod entrano in gioco i servizi. I servizi forniscono una rete affidabile per una serie di Pod. I servizi hanno un front-end che consiste in un nome DNS stabile, un indirizzo IP e una porta. Sul back-end viene eseguito il bilanciamento dinamico su un set di Pod. I Pod vanno e vengono. Il Service osserva i cicli di vita dei Pod, si aggiorna automaticamente e nel frattempo continua a fornire l'endpoint di rete stabile. La mappatura tra Service e Pod viene eseguita tramite label e selettori di label.

### Deployment

I Pod non si auto-riparano, non si ridimensionano e non consentono facili aggiornamenti o rollback. I deployment si occupano di tutto questo, perciò i Pod sono eseguiti e gestiti grazie ai deployment. Un unico deployment può gestire solo un singolo tipo di Pod. Ad esempio, se è presente un'app con un Pod per il front-end web e un altro Pod per il back-end, ci sarà bisogno di due implementazioni. Dietro le quinte, i deployment sfruttano un altro oggetto chiamato set di repliche. Le distribuzioni utilizzano i set di repliche per fornire self-healing e scalabilità.

### Kubernetes Storage

Kubernetes ha un sottosistema di archiviazione ricco di funzionalità. Indipendentemente dal tipo di spazio di archiviazione di cui si dispone e da dove proviene, quando questo viene esposto nel cluster è detto Volume. Il sottosistema di volumes persistenti di Kubernetes è un insieme di oggetti API che consentono alle app di utilizzare spazio di archiviazione. Ad alto livello, i Persistent Volumes (PV) sono il modo utilizzato per mappare lo storage esterno sul cluster, le Persistent Volume Claims (PVC) sono come i biglietti che autorizzano le applicazioni (Pod) a utilizzare un PV.

### Daemons set

Sono utili quando è necessaria una replica di un particolare Pod in esecuzione su ogni nodo del cluster. Alcuni esempi includono i Pod di monitoraggio e la registrazione dei Pod.

### Jobs and Cronjobs

Sono utili quando è necessario eseguire un determinato numero di un particolare Pod e bisogna garantire che andranno a buon fine.

### Spazio dei nomi

Un cluster virtuale (un singolo cluster fisico può eseguire più cluster virtuali) destinato ad ambienti con molti utenti si diffondono in più gruppi o progetti, per isolare i problemi. Inoltre, a uno spazio dei nomi può essere assegnata una quota di risorse per evitare di consumare più delle risorse complessive presenti nel cluster fisico.

### Esempio

Per vedere in piccolo il funzionamento di Kubernetes si può utilizzare Minicube. Minicube è uno strumento che semplifica l'esecuzione di Kubernetes in locale, è un cluster con un singolo nodo all'interno di una Virtual Machine.

#### Cluster initialization

```
$ minikube start
$ kubectl cluster-info
Kubernetes master is running at https://192.168.99.100:8443 CoreDNS is running at
https://192.168.99.100:8443/api/v1/namespaces/kubesystem/services/kube-dns:dns/proxy
$ kubectl get nodes
NAME STATUS ROLES AGE VERSION
minikube Ready master 40d v.1.10.0
```

#### Launch Dashboard

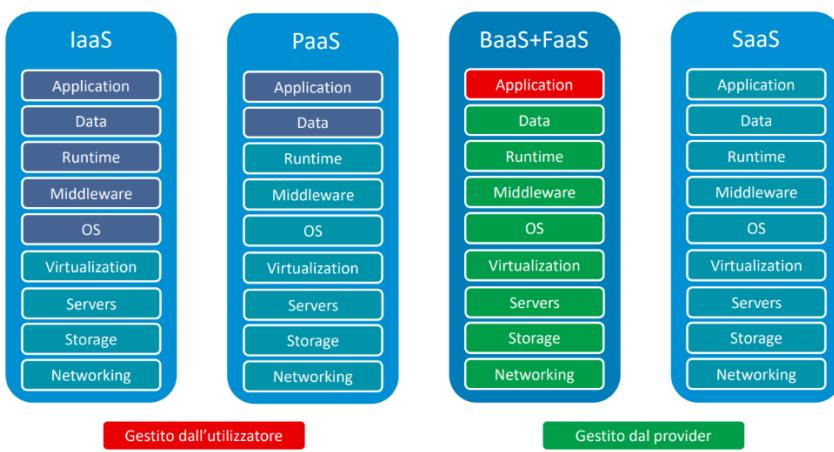
```
$ minikube dashboard
```

## 14.4 Applicazione Serverless e FaaS

In un'applicazione serverless il supporto esegue e con l'utilizzo di callback segnala quando la computazione può proseguire, eliminando la sincronicità forte che va contro la scalabilità del sistema. Il programmatore non deve gestire con degli strumenti il deployment delle applicazioni. Utilizzando questo modello per alcune tipologie di applicazioni per cui può essere rilevante rendere più semplice l'applicazione stessa e rimuovere il carico del deployment, è possibile

definire unicamente la logica applicativa. Il programmatore carica l'applicazione su una piattaforma e non deve fare altro. In questo senso è simile a map-reduce poiché il programmatore implementa unicamente le funzioni di map e reduce. Si elimina interamente l'attesa e non esistono processi idle, la parte di risorse è gestita in automatico dalla piattaforma. La parte più rilevante è la business logic, non la gestione o l'approvvigionamento dell'infrastruttura, non vi è alcun costo relativo al controllo attivo di risorse e alla scalabilità.

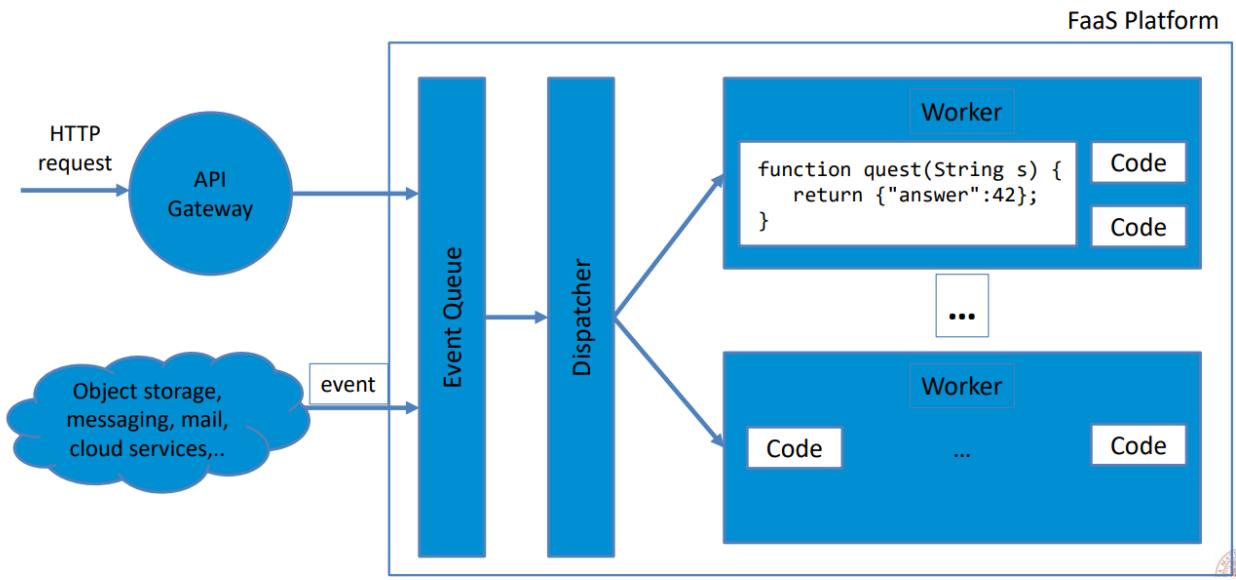
Analizzando le possibili soluzioni nel cloud FaaS si trova tra piattaforma PaaS e software SaaS, quindi come un Back-end as a Service e Function as a Service, BaaS + FaaS. Ci si trova in questa zona perché l'idea è quella di avere una serie di servizi di back-end che costituiscono il supporto e una parte di logica applicativa che non ha una conoscenza forte pregressa dei servizi di sistema.



Back-end as a Service significa fare outsourcing di tutti i servizi di sistema, con la logica di utilizzo di questi servizi molto semplice attraverso la disposizione di API per servizi come autenticazione, database e notifiche.

FaaS consente di dichiarare la logica applicativa. FaaS funziona a eventi, la funzione si attiva all'arrivo di un evento da gestire e computare, restituisce un risultato e infine la funzione scompare. Le funzioni possono avere stato se questo è mantenuto dal back-end, oppure se questo viene mantenuto lato client, ma non è mai mantenuto all'interno della FaaS stessa. Le funzioni FaaS, infatti, sono stateless eseguono in ambienti effimeri e con una semantica-event driven. FaaS scala automaticamente e con grana fine.

## Architettura di riferimento per piattaforma FaaS



Alla piattaforma FaaS arrivano eventi da diverse fonti e attraverso diverse richieste, il dispatcher fa dispatch presso i worker dell'evento, i worker eseguono la logica applicativa e restituiscono una risposta a seguito della computazione.

Alla piattaforma arrivano delle richieste, per esempio HTTP request, a questo punto l'API gateway consente di dirigere le richieste presso una coda di eventi a cui devono registrarsi, successivamente il dispatcher in base agli eventi da gestire carica le funzioni necessarie per la computazione sui worker.

Il vantaggio dal punto di vista del programmatore è che non deve avere conoscenza dell'architettura, mentre dal punto di vista del cloud vi è la massima portabilità e leggerezza. Il problema però, sono i vincoli legati alla piattaforma stessa poiché se una funzione utilizza una quantità eccessiva di risorse e la sua computazione impiega troppo tempo, fa crollare le prestazioni del sistema. Per questo Amazon e altri provider hanno dato limitazioni in termini di risorse (memoria) e tempo di utilizzo, quindi, ci sono vincoli temporali sulle esecuzioni oltre che sulle risorse che se non vengono rispettati portano alla uccisione di quella funzione, ciò non causa problemi in quanto tale funzione è effimera.

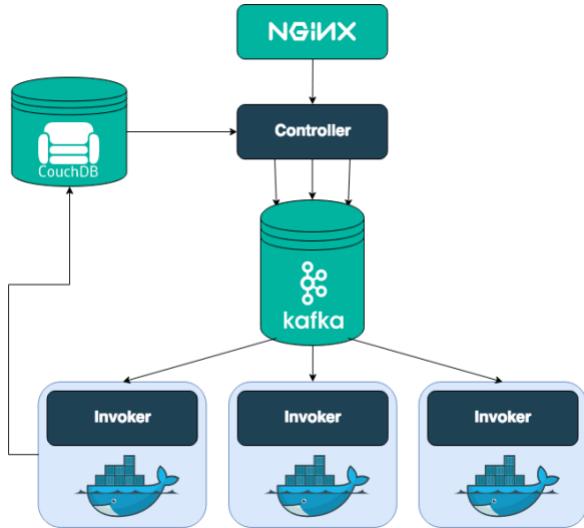
Le principali piattaforme FaaS sul cloud vanno nel senso dell'edge computing, le risorse cloud oggi non sono più unicamente disponibili in un punto geograficamente molto distante, ma sono distribuite in punti molto vicini a noi in micro-datacenter, quindi, con l'edge computing si riduce la latenza. Pensando in questo caso al cloud computing come batch e dividendo in task molto piccoli il lavoro, se non vi sono vincoli stringenti di QoS si possono sfruttare le FaaS e pagare meno.

Tra le piattaforme emergenti opensource per FaaS vi sono OpenFaaS, Openwhisk, Fission e Knative.

## Openwhisk, architettura interna

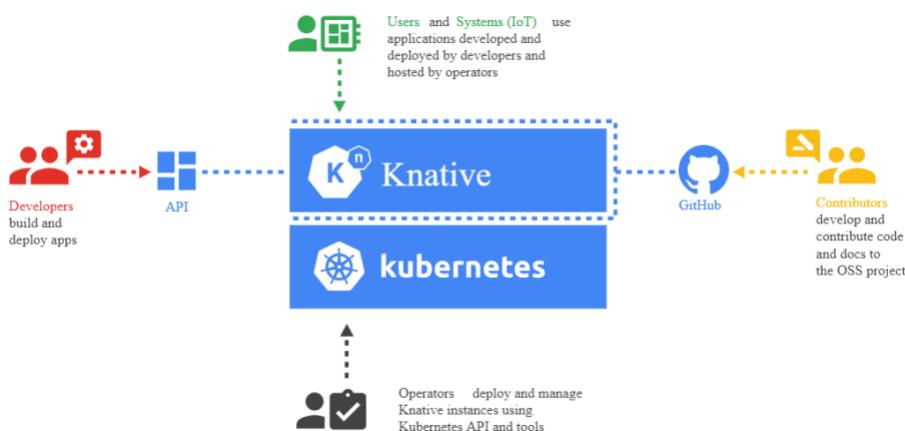
L'API gateway NGINX espone le API, è un punto di accesso e consente di eseguire nuove FaaS. Il controller, gestisce le richieste ed effettua il load-balancing. Apache Kafka è un middleware MOM pub-sub che permette le comunicazioni asincrone ad eventi per massima asincronicità e disaccoppiamento forte delle richieste in arrivo rispetto all'esecuzione delle funzioni. Invoker

esegue le funzioni (action). Couch DB viene utilizzato per il salvataggio di funzioni parametri e risultati, è molto scalabile in modo orizzontale. Kafka facilita la gestione a eventi, da notare come nelle nuove piattaforme si utilizzino sempre gli stessi modelli architetturali e strumenti già ben noti nel cloud.

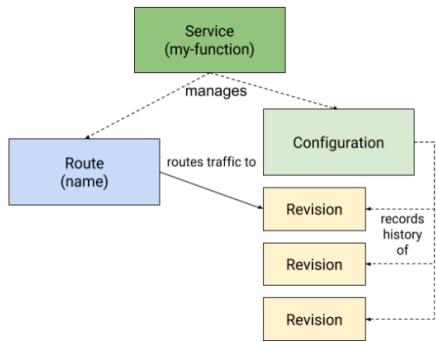


## Knative

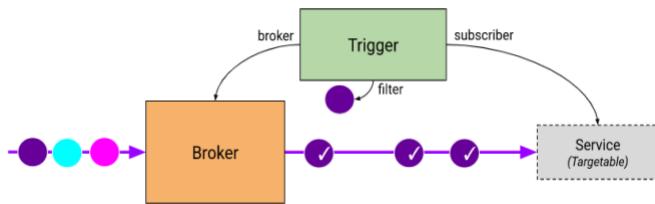
Knative è legato a Kubernetes, esso consente di spalmare i carichi di lavoro in workload che possono essere messi in esecuzione su Kubernetes su container che siano stateless, con il deploy e il management. Non è propriamente una piattaforma FaaS, bensì trasforma le risorse Kubernetes in workload serverless.



L'idea è che Knative facili un deployment veloce di container serverless con scaling automatico. Per il networking e il routing si usa Istio che serve a gestire la comunicazione tra Pod Kubernetes. Sono possibili snapshot di configurazioni a runtime.



I servizi vengono eseguiti runtime, questi generano eventi che fungono da trigger per altri servizi, offrendo elevato disaccoppiamento. Si ripensa completamente alla computazione, infatti, vi è totale indipendenza tra consumatori e produttori, i nuovi servizi vengono eseguiti a runtime. CloudEvents è lo standard utilizzato per la descrizione di eventi.



Dal punto di vista delle performance le FaaS non lavorano benissimo. Su applicazioni FaaS intensive ci si rende conto che non stanno utilizzando macchine di qualità, c'è variabilità estrema nei tempi di valutazione della stessa FaaS anche sulla stessa piattaforma.