Programmazione Distribuita – Pietro

Sommario

01 – Introduzione

Yapping che inizia dalla scrivania di Vannaver Bush per arrivare all'IoT.

Un **sistema distribuito** è formato da un'insieme di macchine connesse attraverso una rete e gestite in maniera autonoma ed indipendente. Ogni nodo coordina il proprio lavoro attraverso uno strato software chiamato **middleware,** che permette all'utente di percepire il sistema come un'entità unica.

Si tratta di sistemi che rispondono a diverse motivazioni:

- ***economiche,*** in quanto sono un'ottima risposta ad un mercato pieno di continue acquisizioni, integrazioni e fusioni di aziende;
- ***tecnologiche,*** in quanto lo sviluppo dell'informatica è sempre stato trainato dal rapidissimo sviluppo delle tecnologie hardware, che raggiungono nuove vette nel giro di pochi anni.

    - Ci sono svariate leggi empiriche che descrivono lo sviluppo tecnologico, ad esempio la ***Legge di Moore,*** secondo cui il numero di transistor nei processori raddoppia ogni 18 mesi.

I sistemi distribuiti rispondono bene all'immensa platea di utenti di Internet, potendo gestire picchi di carico aggiungendo risorse in modo rapido, cioè scalando molto meglio di sistemi centralizzati o client-server.

Altre leggi utili descrivono il valore di una rete in base al numero di connessioni:

- ***Legge di Sarnoff,*** per cui il valore di una rete di broadcast è direttamente proporzionale al numero di utenti: V = a * N; è legata a televisione e radio.
- ***Legge di Metcalfe,*** per cui il valore di una rete di comunicazione è direttamente proporzionale al quadrato del numero degli utenti: V = a * N + b * N^2; sviluppata inizialmente per Ethernet.
- ***Legge di Reed,*** per cui il valore di una rete sociale è direttamente proporzionale ad una funzione esponenzile in N: V = a*N + b*N^2 + c*2^N.

Queste leggi consentono non solo di capire il valore delle reti, ma forniscono linee guida per le strategie di business, per la progettazione di prodotti e servizi e per l'analisi dell'evoluzione tecnologica.

I concetti fondanti di un sistema distribuito possono essere espressi tramite diverse keyword:

- **Remoto,** in quanto si possono avere, potenzialmente, componenti localizzate su macchine diverse.
- **Concorrenza,** in quanto per loro natura l'esecuzione di più istruzioni può essere eseguita su macchine diverse.
- **Assenza di uno stato globale,** in quanto non esiste un modo per stabilirlo, date la distanza e l'eterogeneità di ciascun nodo.
- **Malfunzionamenti parziali,** in quanto ogni componente può, indipendentemente dalle altre, avere un malfunzionamento, che però non influenzerà gli altri calcolatori.
- **Eterogeneità,** poiché per sua natura è eterogeneo per la tecnologia sia hardware che software.

- **Autonomia,** in quanto un sistema distribuito non ha un singolo punto dal quale può essere controllato, coordinato e gestito.
- **Evoluzione,** in quanto l'alta flessibilità consente sempre di cambiare ed adattarsi a nuovi ambienti o tecnologie.
- **Mobilità,** in quanto nodi e risorse possono essere spostate per adattare al meglio le prestazioni del sistema.

02 – Open Distributed Processing

Per facilitare lo sviluppo dei sistemi distribuiti, è importante la condivisione di un modello comune che sia indipendente dalla specifica implementazione, ma sufficientemente dettagliato ed informativo sui meccanismi implementati.

The Reference Model of Open Distributed Processing (RM-ODP) è un modello dell'ISO/IEC per la standardizzazione dei sistemi distribuiti che poggia su ISO/OSI, innestandosi sul settimo livello ed espandendo il concetto di connessione, che diventa *comunicazione.*

Un sistema distribuito dovrebbe soddisfare diversi requisiti non funzionali:

- **Apertura,** cioè uso di interfacce e standard noti e riconosciuti per facilitare l'interoperabilità ed evitare di legarsi ad un fornitore specifico.
- **Integrazione,** cioè incorporare sistemi e risorse differenti senza usare strumenti ad-hoc.
- **Flessibilità,** per poter integrare sistemi legacy e gestire modifiche a run-time.
- **Supporto di federazione di sistemi,** cioè supportare l'unione di diversi sistemi per fornire servizi congiuntamente.
- **Gestibilità,** in modo da permettere controllo e manutenzione per configurare servizi, quality of service e politiche di accesso.
- **Qualità dei servizi,** da garantire anche in situazioni di malfunzionamenti parziali.

- **Scalabilità,** cioè gestire picchi di carico aggiungendo risorse e senza cambiare l'architettura.
- **Sicurezza,** cioè evitare che utenti non autorizzati possano accedere a dati sensibili.
- **Trasparenza,** cioè mascherare dettagli e differenze del sistema sottostante.

In particolare, troviamo diversi tipi di trasparenza, strettamente interconnessi e suddivisi in tre livelli:

- **Trasparenza di accesso,** al livello di base, che nasconde le differenze nella rappresentazione dei dati e nell'invocazione per favorire l'interoperabilità tra oggetti tramite una stessa interfaccia, sia da remoto che da locale.
- **Trasparenza di locazione,** al livello di base, che indica la capacità di un sistema di nascondere ad utenti ed applicazioni la posizione fisica delle risorse, in modo da renderle indipendenti dai servizi da fruire.
- **Trasparenza di migrazione,** al livello di funzionalità, per cui un sistema può spostare oggetti da un nodo all'altro del sistema senza che gli utilizzatori ne siano a conoscenza. Poggia su accesso e locazione.
- **Trasparenza di replica,** al livello di funzionalità, per cui uno stesso oggetto viene replicato su altri nodi del sistema per motivi di prestazione. Poggia su accesso e locazione.
- **Trasparenza di persistenza,** al livello di funzionalità, per cui un oggetto è reso persistente senza che un utente se ne debba occupare tramite un meccanismo di attivazione-deattivazione. Poggia su locazione.
- **Trasparenza alle transazioni,** al livello di funzionalità, per cui il sistema garantisce le transazioni per offrire la coerenza del comportamento in presenza di accessi concorrenti. Non poggia su nulla.
- **Trasparenza alla scalabilità,** al livello di efficienza, se il sistema è in grado di servire carichi di lavoro crescenti integrando risorse, senza modificare architettura e organizzazione. Poggia su migrazione e replica.

- **Trasparenza alle prestazioni,** al livello di efficienza, se è possibile fornire alte prestazioni senza far conoscere i meccanismi usati. Poggia su migrazione, replica e persistenza.
- **Trasparenza ai malfunzionamenti,** al livello di efficienza, se il sistema riesce a fornire, anche solo parzialmente, i propri servizi nel caso di malfunzionamenti. Poggia su replica e transazioni.

03 – Middleware a oggetti distribuiti

Con **middleware** si indica uno strato software che separa le applicazioni dal sistema operativo, i protocolli di rete e l'hardware. Si occupa di fornire le astrazioni appropriate ai programmatori per rendere semplice la creazione di un sistema distribuito. Ne distinguiamo tre tipologie:

- **Middleware di infrastruttura,** che si occupano della comunicazione tra sistemi operativi diversi.
- **Middleware di distribuzione,** che automatizzano compiti comuni per la comunicazione come il multiplexing, il marshalling – che consiste nell'invio di parametri per le invocazioni remote – e la gestione della semantica delle invocazioni.
- **Middleware per servizi comuni,** come persistenza, transazioni e sicurezza.

I middleware rendono lo sviluppo efficace ed efficiente e favoriscono il riuso.

Possiamo vedere le **remote procedure call,** o RPC, come progenitrici del MW, in quanto permettevano l'invocazione di procedure come fossero locali, facilitando il compito del programmatore. Si trasmettevano i dati attraverso stream di byte su socket, in modo che le conversioni fossero automatiche e si forzasse la sincronia. Gli stub per client e server erano creati tramite un linguaggio specifico, *Interface Definition Language.*

Il passaggio al modello ad oggetti distribuiti richiese l'unione del modello RPC con la programmazione ad oggetti, il che richiese l'aggiunta di meccanismi come il polimorfismo, l'ereditarietà e la gestione delle eccezioni.

Common Object Request Broker Architecture, o CORBA, è uno standard che permette ad oggetti distribuiti eterogenei di comunicare e collaborare. Ricorreva a degli *Object Request Broker* per fornire diversi tipi di trasparenza a invocazione ed accesso a servizi di supporto.

.NET Framework è basato su una macchina virtuale, *Common Language Runtime,* ed eredita il meccanismo di comunicazione remota tra oggetti di DCOM e COM+.

Java Enterprise fornisce una netta separazione del layer di presentazione da quello di business, limitando la complessità di realizzazione di nuovi servizi. Si tratta di un *modello a componenti,* cioè blocchi di software riutilizzabili che mostrano i loro servizi tramite un'interfaccia.

Occorre ora fare una distinzione ulteriore tra i tipi di middleware:

- **Middleware implicito,** in cui vengono forniti servizi comuni e trasversali ad ogni componente tramite meccanismi di intercettazione delle richieste e delle interazioni tra gli oggetti. Il server fornisce i servizi sulla base delle richieste specificate durante la fase di deployment, in modo che tutto sia trasparente per lo sviluppatore.
- **Middleware esplicito,** come CORBA e .NET, che invece prevedono un uso esplicito dei servizi, portando ad avere codice difficile da scrivere e da manutenere.

04 – Socket

I protocolli TCP e UDP usano le porte per mappare i dati in ingresso con un particolare processo attivo ed ogni socket è legata ad un numero di porta. Una **socket** è definita come un identificativo univoco che rappresenta un canale di

comunicazione: è indipendente dal linguaggio di programmazione e client e server devono soltanto concordare su protocollo e numero di porta.

Java fornisce le API per i socket in ***java.net,*** che fornisce due classi principali: *ServerSocket,* che accetta connessioni ed è sempre in ascolto, e *Socket,* che si occupa dell'effettivo scambio di dati.

In Java uno **stream** è una sequenza ordinata di byte ed esistono molte classi sia per l'ottenimento di input che per l'invio di output.

05 – Threads [pt.1 – Introduzione e ciclo di vita]

La crescita esponenziale del numero di transistor descritta dalle legge di Moore si scontra con alcuni limiti fisici: la miniaturizzazione dei processori non può andare oltre un certo limite – 5 nanometri – ed in prossimità di quelle dimensioni è necessario tener conto degli effetti di disturbo della termodinamica. È quindi necessario un compromesso tra il numero di transistor, la loro velocità e la loro capacità di raffreddarsi.

Per ottenere miglioramenti tecnologici, ad oggi, la strada che si percorre è quella di aggiungere più transistor da organizzare in core multipli, ma per sfruttarli efficacemente c'è bisogno di software che sappiano sfruttare il parallelismo delle applicazioni.

La programmazione distribuita implica la conoscenza della **programmazione concorrente,** che possiamo dividere in tre tipologie:

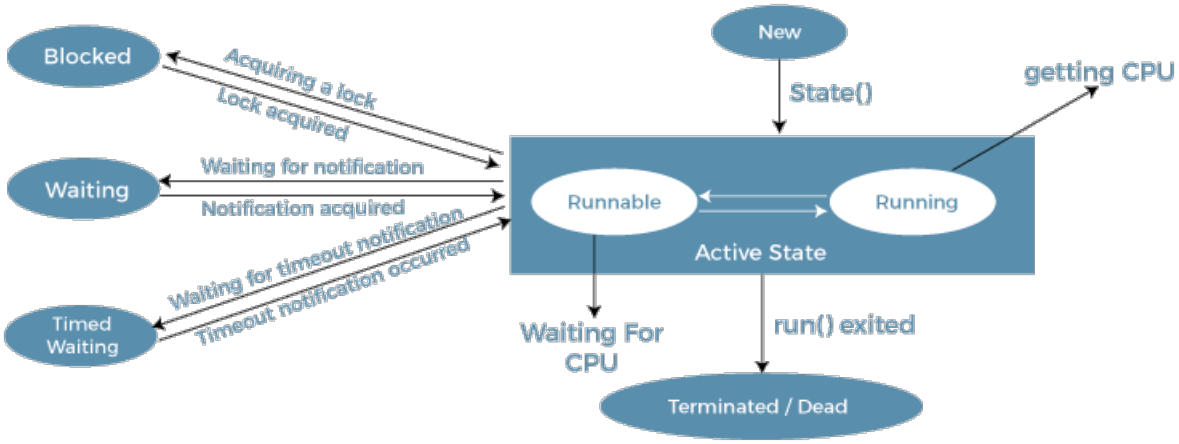
- concorrente *su calcolatori diversi;*
- concorrente *sulla stessa macchina* (multitasking tramite una fork);
- concorrente *nello stesso processo* (con i thread).

Il multitasking illude l'utente dandogli la sensazione di una macchina completamente dedicata, ma in realtà il sistema operativo esegue semplicemente più task contemporaneamente. Il multi-thread estende questo concetto ad un singolo processo, creando dei "processi lightweight" al suo

interno che condivideranno gli stessi dati e comunicheranno tramite memoria condivisa.

In Java, ogni applicazione ha almeno un thread utente, o *main thread,* che può crearne e farne partire altri, più alcuni thread di sistema che gestiscono memoria e segnali. I thread sono istanze di una classe Thread e possiamo gestirli sia istanziando un oggetto di volta in volta che ricorrendo ad un executor.

Ciclo di vita di un Thread in Java:



Life Cycle of a Thread

La comunicazione tra Thread è molto efficiente, ma è possibile incorrere in errori dovuti ad interferenza o ad inconsistenza della memoria; la soluzione sono i meccanismi di sincronizzazione.

- L'interferenza fra thread, o il loro interfogliamento, porta a condizioni di **race condition,** per cui il risultato di un'operazione dipende dall'ordine di esecuzione dei thread. Sono difficili da riprodurre e l'uso di un debugger può causare ulteriore confusione.
- Gli errori dovuti ad inconsistenza della memoria succedono quando thread diversi hanno visioni diverse dei dati. La soluzione è stabilire relazioni **happens-before**, che garantiscono che la memoria scritta da un thread sia visibile ad un altro thread. Possiamo farlo con la sincronizzazione, i metodi start() e join() o la keyword "volatile".

06 – Threads [pt.2 – speedup e deadlock]

La **legge di Amdahl** si usa per predire, a livello teorico, il massimo aumento di velocità che si ottiene usando più processori. Lo **speedup** S di un programma X è il rapporto tra il tempo impiegato da un processore per eseguire X rispetto al tempo impiegato da n processori per eseguire X. Indicando con *p* la parte del programma X che è possibile parallelizzare, allora con *n* processori la parte parallela avrà tempo *p/n,* mentre la parte sequenziale avrà tempo *1-p:*

$$S = \frac{1}{1 - p + \frac{p}{n}}$$

Qualsiasi parte sequenziale rallenta significativamente qualsiasi speedup, dunque per velocizzare un programma è necessario rendere la parte parallela predominante sulla sequenziale.

I **metodi synchronized** sono un costrutto del linguaggio Java che risolve in modo semplice gli errori di concorrenza, al costo di una certa inefficienza. Se un thread esegue un metodo sincronizzato per un oggetto, allora gli altri thread che hanno metodi sincronizzati dello stesso oggetto sono sospesi fin quando l'operazione del primo non termina. Una volta successo, tutti vedranno i cambiamenti dei dati e tra loro si stabilirà una relazione happens-before per determinare chi potrà operare successivamente.

Un **lock intrinseco,** o monitor lock, è un'entità associata ad ogni oggetto che garantisce sia accesso esclusivo che consistente. Un thread che vuole usare l'oggetto deve prima acquisirne il lock, stabilendo così relazioni happens-before: rilascerà il lock al termine, anche se ci sono state eccezioni.

È possibile specificare **azioni atomiche** in Java, cioè azioni non interrompibili che si completano del tutto oppure per niente. In particolare, *read* e *write* per variabili di riferimento, tipi primitivi e tutte le variabili *volatile;* una write di quest'ultima stabilisce una happens-before con tutte le letture successive.

Un **deadlock** è una situazione di stallo in cui ogni thread possiede delle risorse e cerca di acquisirne altre possedute dagli altri thread, bloccando il programma. Con **starvation** si indica una condizione per cui un thread non riesce ad acquisire l'accesso ad una risorsa condivisa per molto tempo. Con **livelock** si indica invece la situazione in cui un thread reagisce alle azioni di un altro e viceversa, senza che nessuno dei due proceda con la propria esecuzione.

07 – Threads [pt.3 – design pattern singleton]

Sintesi dello yapping iniziale:

Tanti esempi che mostrano il funzionamento dei metodi synchronized e la mutua esclusione ottenuta. Elenco delle combinazioni da osservare con due thread che cercano di accedere a due metodi:

- I metodi possono essere di istanza oppure statici.
- I metodi possono essere sincronizzati o meno.
- I metodi possono essere invocati da un solo thread o da entrambi.

Segue esempio sull'efficienza con gli array: un milione di celle, ciascuna inizializzata 10000 volte. A seconda della macchina usata si avranno risultati differenti e dei miglioramenti non lineari a causa degli overhead.

Fine yapping inutile.

Il **singleton** è un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, di cui fornisce poi un punto di accesso globale. È possibile implementarlo in due modi:

- Creare un unico costruttore privato nella classe singleton ed avere un metodo getter statico che restituisca l'istanza della classe. Tale istanza può essere creata preventivamente o alla prima chiamata del metodo ed il riferimento viene memorizzato in una variabile statica.

- Una ***lazy initialization*** che ne rimanda l'istanziazione fino al primo tentativo d'uso.

Segue una lunga catena di esempi su come istanziarli con i thread senza avere errori, che si conclude con un *LazyHolder* che esegue una sola volta l'inizializzazione quando richiesta la prima volta e che stabilisce una relazione happens-before con tutte le altre operazioni sulla classe.

08 – Introduzione Java EE

Java Enterprise Edition è un insieme di specifiche progettate per applicazioni enterprise, cioè un'estensione di Java SE per facilitare lo sviluppo di applicazioni distribuite, robuste ed altamente disponibili.

Nel **modello architetturale multilayer** per enterprise la logica dell'applicazione è separata in componenti che sono organizzate in layer; questi sono a loro volta mappati su diversi tier: generalmente client, server EE e database.

La Java EE infrastructure è partizionata in domini logici chiamati **container:** ognuno ha un ruolo specifico, offre determinati servizi, supporta un set di API e serve a nascondere i dettagli tecnici, migliorando così la portabilità. I domini sono collegati tramite frecce che specificano i protocolli usati.

A seconda delle applicazioni, è necessario capire funzionalità e limiti di ciascun container ed è possibile scegliere tra 4 tipologie differenti: *applet container, application client container* (ACC), *web container* ed *EJB container.* Java EE runtime environment definisce inoltre svariati tipi di componenti, che fanno riferimento a tre macro-categorie: client, web e business.

Con **Web Client** ci si riferisce a web browsers, applets e pagine dinamiche e si ha a che fare con un *thin client,* per cui non si hanno query al database né logica di business complessa.

Con **Application Clients** ci si riferisce a programmi eseguiti sul client che presentano un'interfaccia utente ricca; possono passare per lo strato Web o accedere direttamente allo strato di business.

Con **Web Components** ci si riferisce a:

- applicazioni eseguite in un web container che rispondono a richieste HTTP da web client;
- servlet, cioè classi che dinamicamente processano richieste e costruiscono risposte;
- pagine JSP, cioè documenti che eseguono servlet;
- JavaServer Faces, che forniscono pagine dinamiche basandosi sulle ultime due tecnologie.

I **Business Components,** come Session Beans o Java Persistence Beans, sono componenti eseguite in un EJB Container, il quale si occupa della logica di business transazionale e a cui si può accedere localmente o da remoto.

Il server Java EE fornisce servizi sottoforma di un container per ogni tipo di componente, dunque lo sviluppatore può concentrarsi sulla logica di business. I container rappresentano l'interfaccia tra una componente e le funzionalità di basso livello che questa supporta. Ciascuna componente deve essere assemblata in un Java EE module e deployata nel suo container per essere usata. È possibile personalizzare il supporto fornito dal Java EE server attraverso alcuni settaggi: *security* per configurare l'autenticazione e l'autorizzazione a componenti, *transaction management* per definire una transazione, *JNDI API lookups* per fornire un'interfaccia unica per accedere a servizi e risorse del server e *remote connectivity* per l'invocazione remota.

Un **Applet Container** è fornito dalla maggior parte dei browser per eseguire applet components ed impedisce l'accesso al computer locale per accesso a processi o files.

Un **ACC** (Application Client Container) è un insieme di classi e librerie che permettono di usare servizi di Java EE in applicazioni SE, come sicurezza e naming. L'ACC comunica con l'EJB container tramite RMI-IIOP e con il Web container con HTTP.

Un **Web Container** offre servizi per gestione ed esecuzione di componenti web come servlet e JSP ed è responsabile della loro inizializzazione, invocazione e gestione del ciclo di vita.

Un **EJB Container** è responsabile della gestione dei bean e gestisce il ciclo di vita degli EJB; fornisce transazioni, sicurezza, concorrenza, distribuzione, servizio di naming ed invocazioni asincrone.

Java EE definisce differenti tipi di moduli con il proprio packaging format, basato sul proprio *Java Archive* (jar) format:

- Application client module contiene classi Java ed altre risorse in un jar che può essere eseguito in un ambiente Java SE o in un ACC. Nel jar troviamo una directory META-INF che contiene meta-informazioni che descrivono l'archivio, in particolare Manifest.mf è usato per definire dati legati ad extensions e packages. Nel caso di deploy in un ACC, il deployment descriptor si troverà in application-client.xml.
- EJB module contiene almeno un session bean o un message-driven bean in un file jar, con il deployment descriptor in ejb-jar.xml.
- Web application module contiene servlet, JSP, JSF, HTML, ecc., ed ha un jar con estensione .war; il deployment descriptor è in WEB-INF/web.xml.
- Enterprise module contiene moduli EJB, moduli Web applications ed altre librerie ed ha un archivio jar con estensione .ear; permette il deployment coerente in un unico passo ed il descriptor è application.xml.

Metadata come annotations e deployment descriptors realizzano, in Java EE, il *declarative programming,* in quanto specificano come raggiungere un certo obiettivo. I metadati personalizzano i servizi offerti dai container, associando informazioni addizionali a classi, interfacce, costruttori, metodi, parametri, etc.

Il meccanismo prediletto attualmente è quello delle annotazioni, che riduce la quantità di codice da scrivere ma che richiede di apportare modifiche al sorgente e ricompilare. Nel caso siano presenti entrambi, XML ha precedenza sulle annotazioni.

[Slide 53 in poi yapping sulla storia degli standard, la nascita e le tecnologie coinvolte]

09 – Context e Dependency Injection

La prima versione di Java EE ha introdotto il concetto di **inversione del controllo** (IoC), nel senso che il container prende il controllo del business code in modo da poter fornire servizi senza che il programmatore debba scriverli. La gestione totale del programma avviene configurando il contesto di esecuzione, cioè l'ambiente, e risolvendone le dipendenze con altre componenti: si parla quindi di **Context e Dependency Injection.**

Tutte le componenti di Java EE possono essere resi *CDI Managed Beans* ed il concetto alla base di tutto è il *"loose coupling, strong typing",* per cui il disaccoppiamento consente di inserire interceptors, decorators e gestione degli eventi in tutta la piattaforma in completa armonia. Viene però mantenura una forte tipizzazione, usando le annotazioni con parametri in modo da poter legare in modo "safe" i beans.

Indichiamo con *POJO* un "Plain Old Java Object", cioè una classe java eseguita su una JVM e che può essere usata come JavaBeans per Java SE nel caso in cui rispetti delle convenzioni specifiche riguardo naming, costruttore e metodi getter/setter.

Dependency Injection è un design pattern che disaccoppia componenti dipendenti facendo sì che il container esegua un *inject* degli oggetti dipendenti al posto nostro. Un modo diverso di vedere la DI è come inverso della Java Naming and Directory Interface (JNDI), in quanto JNDI fornisce un riferimento ad un certo oggetto su richiesta, mentre DI fa sì che il container inietti la dipendenza nell'oggetto che ne ha bisogno.

In un POJO, il ciclo di vita consiste semplicemente in una creazione di una nuova istanza con *new* e si attende poi che il garbage collector liberi memoria, ma ciò non è possibile all'interno di un container. In un CDI bean il container, dopo aver usato *new,* deve risolvere le dipendenze necessarie ed invoca tutti i metodi che hanno l'annotazione *@PostConstruct;* prima della deallocazione invoca quelli annotati con *@PreDestroy.* I Beans CDI possono essere stateful, cioè in esecuzione in uno scope ben definito che a client diversi può risultare diverso.

Gli **Interceptors** si frappongono tra invocazioni di metodi di business e sono utili quando si vuole seguire il paradigma della *Aspect-oriented Programming* (AOP), in cui si separano dal codice di business i "cross-cutting concerns", ad esempio:

- Technical concerns, come log di ingresso/uscita o durata di un metodo.
- Business concerns, come, ad esempio, l'invio di un nuovo ordine di acquisto se la quantità disponibile di una certa merce in magazzino arriva a livelli minimi.

Il container si occupa di chiamare gli interceptor prima/dopo l'invocazione di un metodo, assicurando i servizi agli EJB tramite una loro catena configurabile.

Ogni Java EE specification ha un xml deployment descriptor opzionale che descrive come una componente/applicazione deve essere configurata. Nel caso dei CDI diventa invece obbligatorio, in quanto *beans.xml* è necessario ad abilitare CDI e configura interceptors, decorators e alternatives. Durante la fase di deployment, CDI controlla tutti gli application jar ed i war file alla ricerca dei beans.xml da usare per trasformare i POJOs in CDI Beans.

È possibile eseguire tre tipologie di injection: su una proprietà, un costruttore o un setter. Nel caso esista una sola implementazione di una certa classe, verrà eseguita una *default injection* tramite l'annotazione *@Inject,* ma nel caso ve ne sia più di una è necessario risolvere l'ambiguità. Per fare ciò si usano i **Qualifiers,** delle annotazioni aggiuntive che servono proprio a specificare quale

delle molteplici implementazioni si stia cercando di inserire. Ci sono casi in cui questi potrebbero diventare molti, per cui si può rielaborare la classe per avere un ***qualificatore con parametri*** che snellisca il tutto. Infine, nel caso bisogni scegliere tra due implementazioni alternative della stessa interfaccia, è possibile usare l'annotazione *@Alternative* per la classe e si modifica coerentemente il beans.xml per segnalarlo.

Se vogliamo iniettare classi come java.util.Date, presenti nel file rt.jar e che non hanno un beans.xml, la soluzione è usare un **producer.** Nel caso sia necessario prevedere in modo esplicito la distruzione degli oggetti così iniettati si useranno i **disposer.**

In CDI un bean è legato ad un contesto e rimane attivo al suo interno fin quando il container non lo distrugge. Troviamo diversi tipi di scope:

- **Application scope**, che si estende per l'intera durata di un'applicazione ed implica che il bean venga creato una sola volta e distrutto quando l'applicazione termina. È utile per classi helper o oggetti che memorizzano dati condivisi dall'intera applicazione, ma presenta problemi di efficienza e concorrenza.
- **Session scope,** che viene creato per la durata di una sessione utente.
- **Request scope,** che viene creato per la durata di un'invocazione di un metodo o di una singola richiesta HTTP.
- **Conversation scope,** che si estende attravero invocazioni multiple all'interno di sessioni con punti di partenza e fine identificati dall'applicazione. È l'unica ad essere programmabile e possiamo vederla come una via di mezzo tra request e session.
- **Dependent pseudo-scope,** cioè il default scope per CDI.

Gli **Interceptors** non conoscono l'implementazione concreta dei bean CDI con cui interagiscono e vengono definiti aggiungendoli al bean stesso. Ne esistono quattro tipologie:

- Associati ad un costruttore della classe target, con l'annotazione @**AroundConstruct.**
- Associati ad uno specifico business method, con l'annotazione @**AroundInvoke.**
- Di timeout, con l'annotazione @**AroundTimeout.**
- Di callback del ciclo di vita, con le annotazioni @**PostConstruct** e @**PreDestroy.**

È possibile concatenare degli interceptors passandone una lista: l'ordine di invocazione sarà determinato dall'ordine con cui sono stati specificati nell'annotazione @*Interceptors.*

Per rendere gli Interceptors loosely coupled e smettere di scriverli direttamente nel codice è possibile usare l'annotazione @*InterceptorBinding* per definire un **binding;** una volta definito, lo si attacca all'Interceptor. Quest'operazione richiede di specificare una priorità per ciascun Interceptor se li si vuole eseguire in ordine, in quanto non diventerebbe più possibile ordinarli in una lista.

Come per le alternative, anche l'uso degli interceptors va abilitato esplicitamente.

I **Decorators** aggiungono logica ai metodi di business (dunque ne conoscono l'implementazione) e sono pertanto complementari agli interceptors. Anche questi vanno abilitati.

Gli **eventi** permettono le interazioni tra beans senza nessuna dipendenza a tempo di compilazione e, tramite il design pattern ***Observer,*** è possibile anche che i beans che definiscono, lanciano e gestiscono l'evento in questione siano in package o layer separati.

10 – Java Persistence API [pt.1 – ORM e Persistence Unit/Context]

I database servono a garantire la persistenza dei dati, cioè la loro permanenza anche dopo che il programma termina ed il Garbage Collector ha ripullito la memoria.

La **Java Persistence API** (JPA) è un'astrazione su JDBC che lo rende indipendente da SQL, fornisce *Object-Relational Mapping* (ORM) ed ha un linguaggio apposito con cui specifica le query, Java Persistence Query Language (JPQL).

Nel modello JPA, un'**entità** è un POJO che viene dichiarato, istanziato ed usato come altre classi Java. Viene mappata in una tabella tramite annotazioni, in particolare usiamo *@Entity* sulla classe e *@Id* al suo interno per definire l'ID univoco dell'oggetto. È inoltre necessario:

- prevedere un costruttore senza argomenti, né private e né final;
- nessun metodo/attributo persistente deve essere final;
- nel caso voglia essere passata per valore o come oggetto remoto, deve implementare *Serializable.*

Il principio di **ORM** è quello di delegare a tools esterni o frameworks, come JPA, il compito di creare una corrispondenza tra oggetti e tabelle e, nel nostro caso, utilizza come metadati le annotazioni ed i descrittori XML (le informazioni sul database sono nel file *persistence.xml)*.

Viene in genere seguita l'idea della *Configuration by exception,* per cui le regole di default vengono applicate dal container se non altrimenti specificate. In altre parole, fornire una configurazione è un'eccezione alla regola.

A gestire il tutto troviamo l'**EntityManager,** responsabile per le *operazioni CRUD,* che comunica con entità e database sottostanti e permette di fare query in formato JPQL. È possibile eseguire query dinamiche create a run-time, query

statiche definite a compile-time e query con native SQL o stored procedures. Per le query statiche si usano le annotazioni *@NamedQuery* e/o XML metadata.

Per creare un EntityManager si usa una EntityManagerFactory, che per essere creata ha bisogno di una **Persistence Unit** (PU), la quale indica il tipo di database da usare ed il modo di connettersi come definito nel persistence.xml. Per ciascuna PU è possibile specificare nome, classe a cui si riferisce, tipo di db, posizione tramite URL e modalità di autenticazione.

Quando si crea un'istanza di una entity con new, l'oggetto esiste in memoria, ma JPA non sa niente di lui. Quando diventa *managed* dall'entity manager, la tabella corrispondente mappa e sincronizza il suo stato, mentre quando è vista come un normale POJO è *detached.* Il metodo *remove()* cancella i dati soltanto dal DB, l'oggetto Java rimane in memoria fino all'intervento del Garbage Collector.

Quando si dice "gestito dall'applicazione" ci si riferisce ad una situazione in cui l'applicazione è responsabile per l'istanza specifica di EntityManager ed il suo ciclo di vita. Quando si dice invece "gestito dal container" vuol dire che l'applicazione è una servlet o un EJB e quindi ci si affida a risorse iniettate.

Due ulteriori aspetti principali di JPA sono:

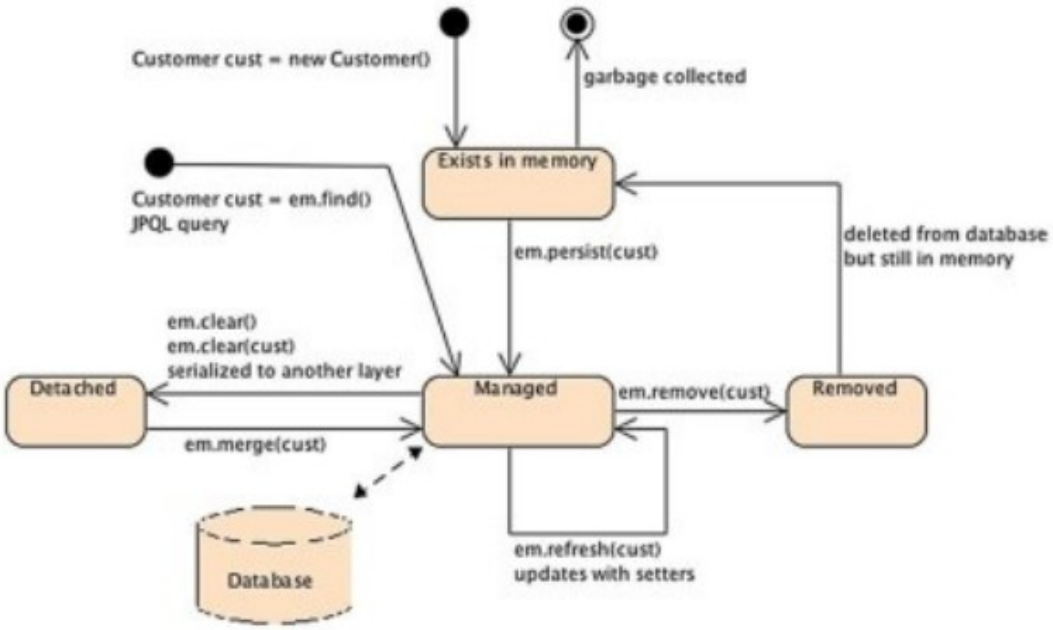
- *Transaction e Looking mechanisms,* forniti dalla Java Transaction API (JTA) per gestire l'accesso concorrente ai dati.
- *Callbacks e listeners,* per agganciare business logic nel ciclo di vita di un oggetto persistente.

Il **Persistence Context** è un insieme di istanze di entità gestite in un certo tempo per una transazione utente e l'EntityManager lo aggiorna o consulta quando viene chiamato un metodo. Possiamo vederlo come una cache dove l'EM memorizza le entità prima di scriverle nel database.

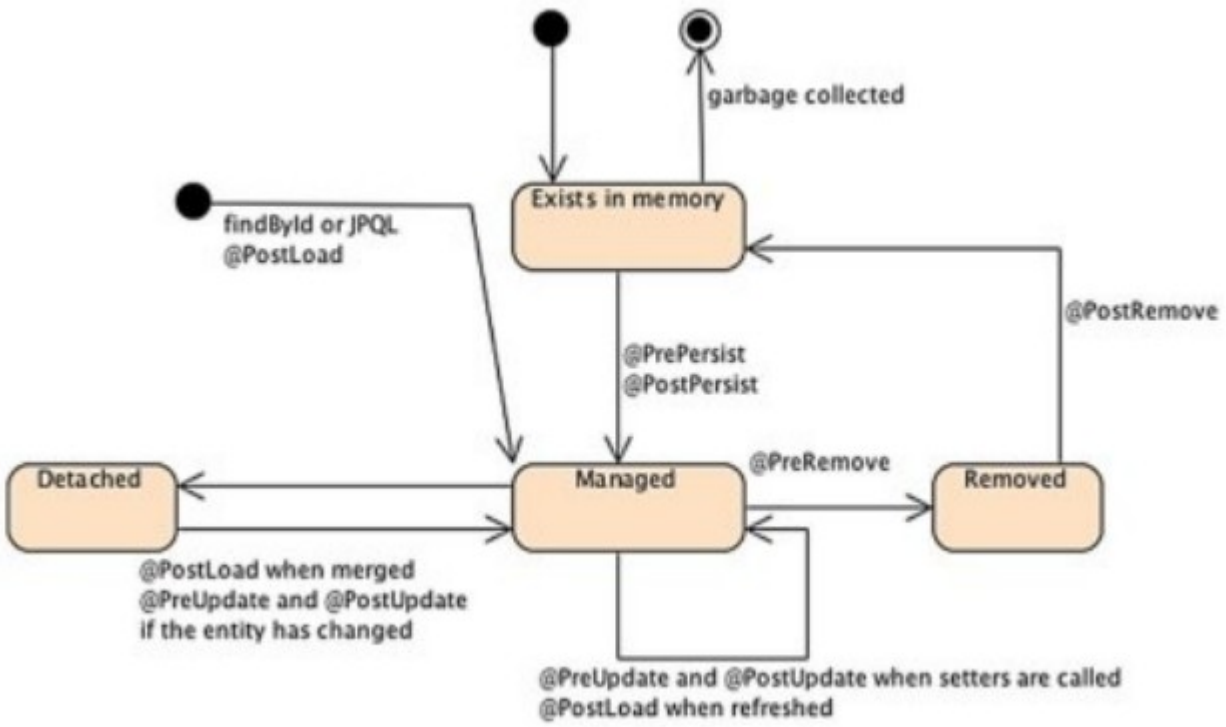
11 – Java Persistence API [pt.2 – Query e ciclo di vita]

JPQL permette di interrogare entità persistenti indipendentemente dal database usato e restituisce un'entità o una collezione di esse, cioè POJOs facili da gestire in Java. Le query sono molto espressive (lo schema tipico è: SELECT, FROM, WHERE, ORDER BY, GROUP BY, HAVING) e ne abbiamo diversi tipi:

- **Query dinamiche,** che vengono specificate a run-time e sono pertanto molto costose.
- **Named Query,** che sono statiche e non modificabili, dunque meno flessibili ma più efficienti. Bisogna notare che i nomi hanno scope relativo al persistence unit, dunque devono essere univoci.
- **Criteria API,** che sono object-oriented.
- **Query native,** per eseguire SQL nativo al posto di JPQL. Non sono portabili e anche qui c'è bisogno che il nome sia unico.
- **Query da stored procedure,** cioè sono definite nel database stesso. Sono utili per compiti ripetitivi e ad alta intensità di uso dei dati, potendo ottimizzare le prestazioni e fornendo sicurezza ulteriore, ma si perde in portabilità.



Il ciclo di vita delle entità ricade in quattro categorie di stati: *persist, updating, removing* e *loading;* ciascuna ha pre-eventi e post-eventi che possono essere intercettati dall'EM quando si invoca un metodo di business:



I metodi di callback sono inglobati all'interno della definizione di entità, ma, se si vuole estrapolare questa logica per applicarla a più entità, è possibile definire dei POJOs detti **entity listeners:** l'entità interessata annoterà *@EntityListeners.* I listeners devono avere un costruttore pubblico senza argomenti e specificare il tipo dell'entità nei metodi di callback: nel caso di *Object* può essere chiamato su diverse entità. Bisogna definirli nel file persistence.xml.

12 – EJB [pt.1 – Introduzione e tipologie]

La logica di business viene eseguita in un layer apposito, il **business layer,** che ha il compito di interagire con servizi esterni, inviare messaggi asincroni, orchestrare componenti del DB verso sistemi esterni e servire il layer di

presentazione. Poggia sugli **Enterprise JavaBeans,** delle componenti lato server che gestiscono transazioni, sicurezza e comunicazioni tra componenti interne ed esterne. Possono inoltre essere:

- ***Stateless,*** se il session bean non contiene conversational state ed ogni istanza può essere usata da ogni client.
- ***Stateful,*** se il session bean contiene un conversational state che deve essere mantenuto attraverso i metodi per un singolo user.
- ***Singleton,*** se il session bean è condiviso da vari client e supporta accessi concorrenti; in questo caso il container si assicura che ne esista una sola istanza per l'intera applicazione.

Un EJB si compone di una classe, annotata con stateless, stateful o singleton, ed un'interfaccia di business locale o remota; è anche possibile che non ve ne sia nessuna per indicare che l'accesso è solo in locale. Necessita di un costruttore pubblico senza parametri e non possono esserci metodi final o static.

Se l'architettura ha client che risiedono all'esterno dell'istanza di JVM in cui risiede il container EJB, allora è necessaria un'interfaccia remota. È possibile usare invocazioni locali quando i bean sono in esecuzione nella stessa JVM, mentre è possibile usare sia locali che remote sullo stesso session bean. La vista senza interfaccia è una variante della vista locale che espone tutti i metodi pubblici di business della classe bean localmente senza l'utilizzo di un'interfaccia separata.

Automaticamente, alla creazione di un EJB avviene una creazione di un nome JNDI, il cui scope può essere global, app o module.

Gli **Stateless Beans** sono i bean più semplici e popolari e sono usati per completare task che richiedono una singola invocazione di un metodo, senza nessuna conoscenza delle precedenti interazioni. Il container ne mantiene un pool in modo da assegnarli ai client che li richiedono: un piccolo numero di EJB può servire molti client ed il loro ciclo di vita è gestito totalmente dal container.

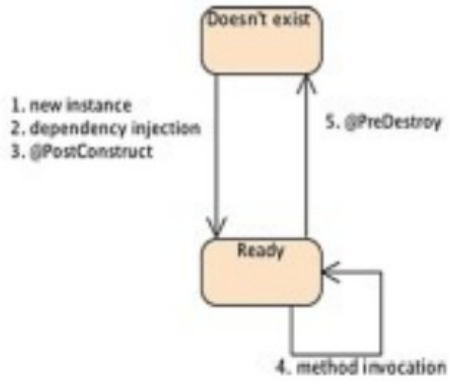
Gli **EJB Stateful** mantengono lo stato della conversazione con il client, dunque hanno con lui una relazione uno-a-uno: per ridurre il carico di lavoro, il container ricorre automaticamente a tecniche di attivazione e passivazione per serializzare l'EJB in memoria di massa e riattivarlo all'occorrenza.

Un **Singleton Bean** viene istanziato una sola volta per applicazione ed esempi di utilizzo comuni sono la gestione di una cache di oggetti tramite Hashmap. Necessitano di un costruttore privato, in modo da evitare che ne venga creato un altro, e metodi di accesso sincronizzato per ottenere la cache.

Per porre gli EJB in un container è necessario inserire classi, interfacce, superclassi, ecc, in un unico package chiamato *Enterprise Archive.* Questo file *EAR* può contenere opzionalmente anche un deployment descriptor chiamato ejb-jar.xml e serve a raggruppare in modo coerente degli EJB di cui va effettuato il deploy assieme.

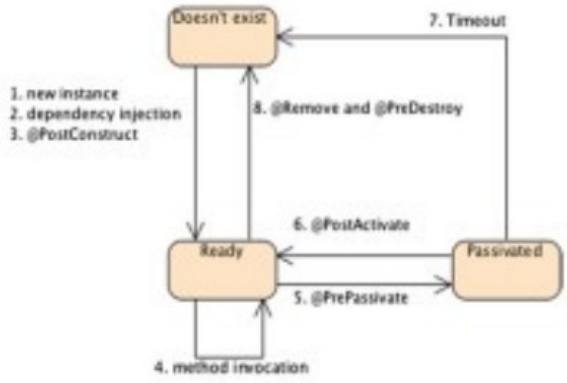
Il client può interagire con l'EJB tramite un riferimento, che può essere sia iniettato, con le annotazioni @*EJB* e @*Inject,* o acceduto con un lookup JNDI. Per bean del tipo "no-interface" basta la prima annotazione, nel caso di più interfacce bisogna specificare a quale ci si vuole riferire e nel caso di EJB remoto si può specificare il nome JNDI dopo l'annotazione @*EJB.* È possibile usare JNDI anche per l'accesso locale, per evitare Inject troppo costose.

13 – EJB [pt.2 – Cicli di vita, autorizzazioni e transazioni]



Stateless e Singleton bean life cycle

1. Il ciclo di vita inizia quando il client richiede un riferimento al bean. Il container crea una nuova istanza
2. Alla creazione, vengono anche iniettate le dipendenze richieste
3. Se l'istanza appena creata ha un metodo annotato con @PostContruct il container lo invoca
4. Invocazioni dei metodi da parte dei client
5. Il container invoca il metodo annotato con @PreDestroy, se esiste, e termina il ciclo di vita della istanza del bean



Stateful bean life cycle

1. Il ciclo di vita inizia quando il client richiede un riferimento al bean. Il container crea una nuova istanza e la memorizza in memoria
2. Alla creazione, vengono anche iniettate le dipendenze richieste
3. Se l'istanza appena creata ha un metodo annotato con @PostContruct il container lo invoca
4. Invocazioni dei metodi da parte dei client
5. Se il client resta idle per un certo periodo di tempo, il container invoca il metodo annotato con @PrePassivate e rende passivo il bean in uno storage permanente
6. Se un client invoca un bean «passivated», il container lo attiva riportandolo in memoria ed invoca il metodo annotato con @PostActivate, se esiste
7. Se un client non invoca il bean all'interno del session timeout period, il container lo elimina
8. Alternativamente al passo 7, se un client chiama un metodo annotato con @Remove, il container invocherà il metodo annotato con @PreDestroy, se esiste, e termina il ciclo di vita del bean

L'autenticazione viene gestita dal layer di presentazione o dall'application client, che passano l'utente autenticato al layer EJB, il quale verifica se questi abbia accesso al metodo in base al suo ruolo.

L'**autorizzazione dichiarativa** può essere definita usando annotations oppure nel deployment descriptor e richiede di definire ruoli, assegnare permessi a metodi o bean interi o ancora cambiare temporaneamente una security identity.

L'**autorizzazione da programma** permette una grana più fine nel controllo, come accesso ad blocco del programma o per uno specifico utente, ed usa le API di *Java Authentication and Authorization Service* (JAAS).

Una **transazione** rappresenta un insieme di operazioni logiche che devono essere realizzate come una singola unità di lavoro: se tutte vanno a buon fine si esegue un commit, altrimenti si esegue un rollback.

JTA garantisce i suoi servizi tramite un *Transaction Manager,* cioè una componente core che gestisce le operazioni, ed un *Resource Manager,* che è invece responsabile della gestione delle risorse e della loro registrazione con il TM. Nel caso siano coinvolti più DB, JTA effettua il coordinamento usando *Extended Architecture* per *Distributed Transaction Processing:* si tratta di una tecnica di commit a due fasi in cui prima si notifica ad ogni RM che sta per avvenire un commit tramite il comando 'prepare' e, se tutti sono pronti, la transazione può procedere.

Nelle transazioni **container-managed** il container gestisce le transazioni, il programmatore non sviluppa codice JTA ed i servizi vengono forniti ai session beans. È possibile anche avere transazioni **bean-managed.**

[Non ho voglia di dedicare un capitolo al terzo blocco di slide degli EJB, perché è una lezione pratica su un HelloWorld ed un esempio leggermente più complesso]

14 – Messaging [pt.1 – Introduzione]

Message-oriented middleware (MOM) è un provider che permette lo scambio di messaggi asincroni fra sistemi eterogenei e può essere visto come un buffer che produce e consuma messaggi. È loosely coupled in quanto i produttori non sanno chi c'è all'altra estremità del canale di comunicazione e non richiede che produttore e consumatore siano disponibili contemporaneamente.

Quando un messaggio viene inviato, il software che memorizza il messaggio e lo invia è detto *provider* o *broker,* il sender del messaggio è chiamato *producer* e la componente che riceve il messaggio è detta *consumer.* Ogni componente interessata ad un messaggio in una particolare destinazione può consumarlo.

In Java, l'API che gestisce questi concetti è **Java Message Service** (JMS), che prevede interfacce e classi per connettersi ad un provider e creare, inviare e ricevere un messaggio. In un EJB container, *Message-Driven Beans* (MDBs) possono essere usati per ricevere messaggi in container-managed way.

Qui i provider gesticono buffering e delivery dei messaggi, mentre con clients si indicano sia applicazioni Java che componenti che producono o consumano messaggi. I messages sono gli oggetti inviati dai clients, mentre gli *administered objects* sono oggetti, come ad esempio le connection factories, forniti attraverso JNDI lookups injection.

La comunicazione asincrona viene garantita fornendo una destinazione dove i messaggi possono essere mantenuti fino all'instradamento, ne esistono due tipi:

- **Point-to-Point model** (P2P), in cui il messaggio viaggia da un singolo producer verso un singolo consumer. Ogni messaggio viene inviato ad una specifica coda, che li mantiene fin quando il consumer non li consuma o scadono. Una coda può avere consumers multipli, ma solo uno di loro potrà leggere il messaggio. Questo modello non garantisce, inoltre, che i messaggi siano instradati in un particolare ordine. Viene detto *pull-based* in quanto i messaggi non sono consegnati direttamente, ma vengono prelevati dalle code. Il receiver notifica l'avvenuta ricezione ed il processamento del messaggio tramite un acknowledge.
- **Publish-subscribe model** (pub-sub), in cui si ha invece una comunicazione uno-a-molti, in quanto un messaggio viene inviato da un singolo producer e raggiunge, potenzialmente, diversi consumers, qui chiamati subscribers. Questi devono sottoscrivere (e disdire) ad un topic in modo dinamico ed il topic conserva i messaggi fin quando non vengono distribuiti a tutti i subscribers. È un modello *push-based* in quanto i messaggi sono automaticamente mandati in broadcast al consumer, senza che questi ne abbia fatto richiesta.

Gli **administered objects** sono oggetti che non si configurano tramite codice ma amministrativamente: è il provider che ne permette la configurazione e li rende poi disponibili nello spazio dei nomi JNDI. Ne abbiamo due tipi:

- **Connection factory,** usate dai clienti per creare una connessione ad una destinazione.
- **Destinazioni,** cioè punti di distribuzione del messaggio che ricevono, mantengono e distribuiscono messaggi.

I **Message-Driven Beans** (MDBs) sono message consumer asincroni eseguiti in un EJB container. Questi si occupa dei servizi e può contenere numerose istanze di beans, in concorrenza, per processare messaggi di producers diversi. Gli MDBs sono stateless e sono, in genere, in ascolto su una destination sola, in modo da consumare e processare un messaggio appena arriva.

Java Messaging Service API (JMS) è un insieme di standard Java API che permette alle applicazioni di creare, inviare, ricevere e leggere messaggi in maniera asincrona, definendo un insieme di interfacce e classi per comunicare con altri message providers. Possiamo vederlo come un analogo di JDBC, in quanto uno permette la connessione a diversi DB e l'altro a diversi providers.

Una **Connection** è un'astrazione di una connessione attiva di un JMS client con uno specifico JMS provider: incapsula una connessione, che generalmente è rappresentata come connessione socket TCP, e crea una sessione. Per usare una ConnectionFactory, il client deve eseguire una JNDI lookup o usare injection.

Una **Destination** è un administered object che contiene informazioni di configurazione provider-specific, come un destination address. Anche in questo caso è necessario un lookup JNDI ed i meccanismi sottostanti sono nascosti tramite un'interfaccia. Nel caso di p2p la destinazione è rappresentata dall'interfaccia Queue, mentre per pub-sub è Topic.

Un **Messaggio** è formato da:

- ***Header,*** obbligatorio, che contiene informazioni sul messaggio come ID, destinazione, priorità, expiration, timestamp, tipo, persistent/non, ecc.
- ***Properties,*** opzionale, formato da una coppia name/value per consentire al client di chiedere solo messaggi con un certo criterio.
- ***Body,*** opzionale, che contiene la parte informativa.

Altre interfacce utili:

- **JMSContext,** che contiene una active connection ad un MS provider e single-threaded context per inviare e ricevere messaggi.
- **JMSProducer,** creato da JMSContext per l'invio di messaggi.
- **JMSConsumer,** creato da JMSContext per ricevere messaggi.

Il messageProducer è l'oggetto che ha il compito di inviare messaggi ad una Destination: implementa l'interfaccia MessageProducer e viene generato da una sessione con il metodo createProducer() passandogli il nome logico della Destination. È possibile avere il produttore fuori dal container o dentro e in quest'ultimo caso c'è differenza se c'è CDI o meno:

- se è fuori dal container: si ottengono una connection factory ed una coda con JNDI, si crea un JMSContext con la factory e lo si usa per creare un JMSProducer, che provvederà all'invio con il metodo send().
- [esempi in codice]

I messageConsumer sono gli oggetti che ricevono i messaggi provenienti da una destination e possono essere:

- **sincroni,** se il ricevitore preleva esplicitamente il messaggio con una receive();
- **asincroni,** se il ricevitore registra l'evento di arrivo di un messaggio tramite un MessageListener che invoca il suo metodo onMessage().

15 – Messaging [pt.2 – Affidabilità ed MDB]

JMS definisce dei livelli di affidabilità per assicurarsi del corretto instradamento dei messaggi, anche nei casi in cui il provider sia sotto carichi elevati o vada addirittura in crash:

- **Filtering messages,** che usano i selector per ricevere solo i messaggi che si desiderano. Si usa una stringa per fare selezione su headers, metadati e/o proprietà custom.
- **Setting message time-to-live,** per scegliere un expiration time in modo da non instradare messaggi se obsoleti. Il producer imposta un tempo in millisecondi, oltre il quale il provider rimuove il messaggio.
- **Specifying message persistence,** per specificare la persistenza di messaggi e proteggersi da possibili malfunzionamenti del provider. Si tratta del valore di default, ma è possibile renderlo non persistent per migliori prestazioni.
- **Controlling acknowledgment,** cioè controllo degli ack a vari livelli, che può essere automaticamente fatto dalla sessione con un *auto_acknowledge* o esplicitamente dal client con un metodo di Message.
- **Creating durable subscribers,** cioè assicurare l'instradamento di messaggi verso un 'unavailable' subscriber in un pub-sub model. Si ha creando dei durable customer che si registrano presso un broker JMS con un nome univoco, in modo da poter ricevere i messaggi arrivati mentre era disconnesso.
- **Setting priorities,** cioè dare una priorità ai messaggi. Varia da 0, bassa, a 9, alta, ed è specificata nell'header.

Un **Message Driven Bean** (MDB) è un consumatore di messaggi asincrono, simile ad un EJB stateless, che tramite CDI può accedere ad altri EJB, JDBC, EntityManager, ecc. Viene preferito ad un JMS Client per la possibilità di far gestire al container transazione, multithread, sicurezza, ecc.

Deve essere annotata con *@MessageDriven,* implementare l'interfaccia del listener, essere public e presentare un costruttore senza argomenti per l'instanziazione automatica.

Il suo ciclo di vita è simile a quello dei bean stateless (stati "doesn't exist" e "ready", con onMessage() che riaggiorna ready) ed è possibile inserire interceptors.

Seppur nascano come asincroni, con il container che risveglia gli MDB all'arrivo di un messaggio, è possibile – ma non consigliato – renderli sincroni: in questo caso bloccheranno le risorse del server, facendo busy waiting e non consentendo al container di liberarli. Si incorre in problemi di prestazione, in quanto il pool di istanze del container potrebbe svuotarsi: in questo caso il container potrebbe peggiorare la situazione creando nuovi istanze e consumando ancora più memoria.

È infine possibile, per un MDB, diventare un message producer tramite le API JMS.

È possibile definire transazioni anche per lo scambio di messaggi e, in quanto EJB, si può scegliere tra bean-managed o container-managed.

16 – Web Services [pt.1 – Introduzione e SOAP]

L'**architettura orientata ai servizi** (SOA) fornisce il contesto metodologico e di business in cui utilizzare al meglio le tecnologie basate sui servizi. Questi ultimi sono *self-contained processes* deployati su una piattaforma middleware standard e che possono essere descritti, pubblicati, localizzati ed invocati attraverso la rete. I servizi spaziano dal rispondere a richieste semplici all'eseguire complesse logiche di business e sono scritti in modo da essere indipendenti dal contesto in cui sono usati.

Affinché si possa parlare di SOA, i servizi devono essere neutrali rispetto alla tecnologia, essere loosely coupled e supportare location transparency.

I **web services** sono la tecnologia più rappresentativa nell'ambito middleware, in quanto si tratta di SOA che usano Internet come communication medium. Si tratta di applicazioni che possono essere implementate con diverse tecnologie, come SOAP e REST, alla cui base abbiamo il concetto di "loosely coupled", in quanto il client non conosce nulla riguardo l'implementazione del servizio.

Un servizio accessibile via Web usa un'interfaccia universale, un browser, ma non è necessariamente un WS, i quali sono invece utilizzabili automaticamente sia da altre applicazioni sia da utenti.

Il concetto di **Software as a Service** è comparso per la prima volta con il modello software *Application Service Providers:* sono companies che combinano software ed infrastrutture con servizi di business e professionali per fornire una soluzione completa ai customer, sotto forma di servizio richiesto su sottoscrizione. Si tratta di entità terze che eseguono il deploy, mantengono e gestiscono applicazioni e forniscono servizi software-based tramite rete.

Con *service provider* si indica l'organizzazione che fornisce i servizi, con *service consumer* il fruitore, sia individuo che applicazione, e con *service registry* una repository di service descriptions (WSDL) pubblicati dal provider.

Web Services Description Language (WSDL) ha il ruolo di Interface Definition Language tra i consumer ed il servizio e serve a specificare l'interfaccia del Web Service, in modo da rispettare il contratto. Fornisce informazioni sul tipo del messaggio, la porta, il protocollo, le operazioni supportate, la posizione e cosa si aspetta il consumatore come valore di ritorno.

Simple Object Access Protocol (SOAP) è un lightweight protocol per lo scambio di informazioni basato su XML e che utilizza HTTP come protocollo di scambio. La struttura dei messaggi è la seguente:

- **SOAP envelope,** che identifica il documento XML come messaggio SOAP.

- **SOAP header,** opzionale, che contiene informazioni aggiuntive per il message processing.
- **SOAP body,** in cui c'è il contenuto vero e proprio del messaggio e vi troviamo chiamate e risposte.
- **SOAP fault,** che contiene informazioni su eventuali errori verificatisi durante il processing.

UDDI è un servizio di localizzazione dei servizi che mantiene un registro in cui i provider possono pubblicare la descrizione dei loro servizi in WSDL, in modo da renderli scopribili e scaricabili. È opzionale quando si conosce già la locazione del servizio ed è attualmente in disuso, in quanto si perde un po' il concetto di 'loosely coupled'. È basato su XML e contiene anche informazioni ulteriori sui servizi, come disponibilità o costo.

L'implementazione di riferimento per i WS è *Metro*, prodotto dalla community di GlassFish. Implementa JAX-WS, che permette di costruire e consumare WS in Java e di mascherare SOAP al programmatore. JAX-WS facilita inoltre sia l'implementazione top-down che quella bottom-up, permettendo anche di annotare un POJO per renderlo Web Service.

I vantaggi principali del fare in modo che un WS sia anche un EJB riguardano flessibilità ed efficienza, poiché in questo modo lo stesso servizio può essere fornito con due modalità:

- *Dipendente dalla piattaforma,* con richiesta diretta RMI o simili, per client Java che operano all'interno dello stesso ecosistema, con alte prestazioni.
- *Indipendente dalla piattaforma,* via WS, per client esterni o su piattaforme diverse, con maggiore interoperabilità ma minore efficienza.

17 – Web Services [pt.2 – SOAP binding]

I Web Services sono definiti in termini di XML e scambiano messaggi SOAP tramite operazioni definite da WSDL. Ciò significa che è necessario tradurre gli oggetti Java in operazioni WSDL e viceversa e di questo si occupa **Java Architecture for XML Binding** (JAXB) tramite delle annotazioni che consentono di fare marshal e unmarshal classe-XML. **JWS** usa invece le annotazioni per mappare classi Java in WSDL e per definire marshal – da metodo a richiesta SOAP – e unmarshal – da risposta SOAP a valore di ritorno del metodo.

JAX-WS e **WS-Metadata specifications** definiscono due diversi tipi di annotazioni:

- Nel package *javax.jws* troviamo le WSDL mapping annotations, che permettono il mapping WSDL/Java.
- Nel package *javax.jws.soap* troviamo le SOAP binding annotations, che permettono la personalizzazione di SOAP binding.

Il **SOAP binding** descrive come il WS è legato al protocollo di messaging e ne esistono due tipi: *Document,* scelto di default e preferibile per dati complessi, e *RPC,* che contiene un elemento con il nome del metodo da invocare e preferibile per operazioni semplici. Bisogna inoltre scegliere tra due formati di serializzazione/deserializzazione: *Literal,* se usiamo un XML schema, o *Encoded,* se usiamo regole di codifica definite da SOAP.

Poiché è possibile che consumer e service siano scritti in linguaggi diversi, gli errori sono notificati tramite un SOAP fault, che viene automaticamente convertito da JAX-WS a partire da un'eccezione Java.

I WS SOAP hanno un ciclo di vita simile a quello dei beans, con soltanto gli stati "doesn't exist" e "ready".

Possono accedere al loro environment context iniettando una reference di *javax.xml.ws.WebServiceContext* con l'annotazione *@Resource.* Il WS può così accedere ad informazioni come il message context o la security della richiesta.

SOAP WS permette di definire metadati anche usando file xml, in particolare troviamo *webservices.xml* nella sotto-directory WEB-INF, che sovrascrive o estende le annotazioni.

Usando WSDL per descrivere il servizio ed alcuni stub (proxy) generati dal sistema, si può facilmente usare WS senza usare direttamente HTTP o SOAP: è simili a quanto fatto per RMI, con la differenza che si possono usare anche altri linguaggi. I proxy sono la rappresentazione Java di un WS endpoint, come servlet o EJB, ed instradano le chiamate locali Java al remote WS tramite HTTP.

Poiché JAX-WS è disponibile in Java SE, un SOAP WS consumer può essere sia un qualunque tipo di Java code in esecuzione nella JVM che una qualunque componente Java EE in esecuzione in un container.