PARTE 1

Cosa è un sistema distribuito?

Un sistema distribuito consiste di un insieme di macchine, ognuna gestita in maniera autonoma, connesse attraverso una rete. Ogni nodo del sistema distribuito esegue un insieme di componenti che comunicano e coordinano il proprio lavoro attraverso uno strato software detto middleware, in maniera che l'utente percepisca il sistema come un'unica entità integrata.

Quali sono le motivazioni tecnologiche ai sistemi distribuiti?

Lo sviluppo dell'informatica è sempre stato condotto dal rapidissimo sviluppo delle tecnologie hardware. Diverse "leggi" empiriche elaborate negli anni si sono provate fedeli nel prevedere la velocità di evoluzione. Ad esempio, ben conosciuta è la Legge di Moore che afferma che la densità dei transistor si raddoppia ogni 18 mesi. Questo sviluppo continuo ha condotto allo sviluppo di tecniche e metodi per la progettazione di sistemi software complessi, in grado di poter utilizzare al meglio queste componenti di sempre maggiori prestazioni. I sistemi distribuiti permettono di utilizzare al meglio quello che la tecnologia hardware continua a produrre. Un obiettivo importante è la riusabilità e l'integrabilità di soluzioni diverse: non è sempre necessario ripartire da zero ad ogni nuovo sistema distribuito, e la possibilità di integrare soluzioni pre-esistenti è una priorità importante per gli ambienti di sviluppo per sistemi distribuiti.

Quali sono le motivazioni economiche ai sistemi distribuiti?

I sistemi distribuiti rispondono in maniera completa alle esigenze dell'economia di mercato che è caratterizzata da numerose e frequenti acquisizioni, integrazioni e fusioni di aziende.
Quindi, la necessità di affrontare in tempi brevi l'integrazione dei sistemi informativi di aziende diverse, che si sono fuse insieme, richiede una infrastruttura versatile e agile, che permetta di poter essere operativi in pochissimo tempo.
Allo stesso tempo, spesso sistemi informativi di aziende che vengono separate dalla "casa madre" in un meccanismo di cosiddetto "downsizing" devono mantenere un certo livello di integrazione con le aziende del gruppo, in una sorta di federazione di sistemi che complica la gestione, prevedendo tre livelli di accesso al sistema informativo: dall'interno della azienda, dall'interno della federazione di aziende e dall'esterno.
Minimizzano il time to market permettendo di assemblare componenti preesistenti (i cosiddetti off the shelf) combinando sistemi hw/sw in un unico sistema distribuito. Il time to market è il tempo necessario per poter arrivare al prodotto finale, dalla ideazione, progettazione e realizzazione. Questo deve essere reso quanto più breve possibile, sia per il ricambio tecnologico continuo, ma anche perché le richieste dei consumatori variano significativamente in poco tempo.
I sistemi distribuiti sono capaci di poter reggere meglio dei sistemi centralizzati o Client-Server agli improvvisi picchi di carico e quindi rispondono anche a questa esigenza di assicurare la scalabilità del servizio che forniscono in "ogni" condizione.

Che cos'è la "legge" di Moore?

La legge di Moore afferma che la densità dei transistor si raddoppia ogni 18 mesi **e**d è il metro che ci permette di misurare il dinamismo dell'evoluzione tecnologica e di prevedere la velocità di evoluzione.

Perché nelle domande precedenti la parola legge viene messa tra virgolette?
Perché è un fenomeno empiricamente riscontrato, ovvero è una legge basata sull'osservazione.

PARTE 2

Cosa è un modello di riferimento e a cosa/chi serve?
Un modello di riferimento è un modello comune che serve come astrazione comune per produttori, sviluppatori e progettisti, in maniera da essere indipendente dalla specifica implementazione tecnologica. Può essere utilizzato come terreno comune per la comunicazione durante le fasi iniziali di progettazione, identificando termini e linguaggi da utilizzare.

Cosa sono i requisiti non funzionali di un sistema?
I requisiti non funzionali di un sistema indicano la qualità del sistema e non sono identificabili in una singola parte del sistema, ma sono globali e vanno considerati come fattori che hanno un impatto significativo sull'architettura. Non sono collegati direttamente alle funzionalità che il sistema deve realizzare.

Qual è la differenza sostanziale tra un sistema distribuito e uno parallelo/concorrente?
Le componenti di un sistema distribuito possono essere localizzate su macchine diverse, mentre in un sistema parallelo/concorrente tutte le componenti si trovano sulla stessa macchina.

Perché un sistema distribuito è concorrente, e perché la soluzione della concorrenza è più complessa rispetto ad un sistema centralizzato?
Un sistema distribuito è concorrente perché è possibile la contemporanea esecuzione di due o più istruzioni su macchine diverse. In un sistema centralizzato ci sono strumenti come lock e semafori che permettono di gestire in maniera più semplice la sincronizzazione rispetto ad un sistema distribuito.

Che significa che in un sistema distribuito manca un clock globale, e cosa comporta?
Non esiste un modo per poter determinare lo stato globale del sistema. E' impossibile riuscire a sincronizzare perfettamente gli orologi di tutti i processi e questo comporta l'impossibilità di ordinare, in modo preciso ed univoco, tutti gli eventi che occorrono all'interno del sistema. Tale risultato è dovuto alle differenze strutturali, di natura per lo più elettronica, dei vari dispositivi per la generazione del segnale di clock all'interno dei microprocessori.

Cosa significa che un sistema distribuito può tollerare malfunzionamenti parziali?
Ogni componente di un sistema distribuito può smettere di funzionare correttamente, in maniera indipendente dalle altre componenti e questo fallimento non deve invalidare le funzionalità che sono localizzate altrove nel sistema distribuito.

Perché un sistema distribuito è eterogeneo?
Un sistema distribuito è eterogeneo perché i vari processi possono essere fisicamente diversi. Infatti questi possono utilizzare diversi sistemi operativi, software scritti con differenti linguaggi di programmazione o utilizzare molteplici dispositivi hardware.

Cosa significa che un sistema distribuito asseconda l'evoluzione aziendale?
Significa che un sistema distribuito può cambiare anche in maniere sostanziale durante la sua vita, sia perché cambia l'ambiente sia perché cambia la tecnologia utilizzata. La flessibilità di un sistema

distribuito deve assicurare che la migrazione verso ambienti diversi, tecnologie differenti e applicazioni nuove può essere assecondata con successo e senza costi eccessivi.

Perché un sistema distribuito offre autonomia di gestione ai singoli nodi?

Un sistema distribuito non ha un singolo punto dal quale esso può essere controllato, coordinato e gestito. Quindi la collaborazione va ottenuta mettendo d'accordo le richieste del sistema distribuito con quelle del sistema che gestisce ciascun nodo, tramite politiche di condivisione e di accesso, formalmente specificate e rigidamente applicate.

Cosa significa che un sistema distribuito deve:

essere aperto: deve supportare la portabilità di esecuzione e la capacità di collaborare insieme tra diverse componenti, attraverso interfacce e servizi ben documentati e aderenti a standard noti e riconosciuti; questo aspetto è importante per poter far evolvere il sistema (si possono aggiungere nuove componenti) ma anche per evitare di rimanere legati a un singolo fornitore: se si usano standard aperti, si può cambiare fornitore senza particolari rischi per l'intera architettura.

essere integrato: deve incorporare al proprio interno sistemi e risorse differenti senza dover utilizzare sistemi ad hoc. Questo permette di trattare in maniera efficiente (economica) con il problema dell'eterogeneità hardware, software e di applicazioni.

essere flessibile: deve poter evolvere e far evolvere i sistemi distribuiti in maniera da integrare sistemi legacy al proprio interno. Un sistema distribuito dovrebbe anche poter gestire modifiche durante l'esecuzione in modo da poter accomodare cambi a run-time, riconfigurandosi correttamente.

essere modulare: deve permettere ad ogni componente di essere autonoma ma con un grado di interdipendenza (rapporto di reciproca dipendenza) verso il resto del sistema.

supportare la federazione di sistemi: deve unire diversi sistemi dal punto di vista amministrativo oltre che architetturale, per lavorare e fornire servizi in maniera congiunta.

essere facilmente gestibile: deve permettere il controllo, la gestione e la manutenzione per configurarne i servizi, la loro qualità (Quality of Service) e le politiche di accesso. La tolleranza ai malfunzionamenti è una delle principali richieste di qualità di servizio di un sistema distribuito. Un sistema distribuito è potenzialmente in grado di trattare con i malfunzionamenti, utilizzando (dinamicamente) componenti alternative per fornire funzionalità che alcune componenti non sono in grado temporaneamente di fornire.

essere scalabile: qualsiasi sistema distribuito accessibile da Internet può essere soggetto a picchi di carico non prevedibili e deve essere in grado di gestirli; ma si deve anche poter gestire che il sistema possa evolvere per accomodare evoluzioni del contesto aziendale che può crescere velocemente, aumentando notevolmente la platea di utenti che accedono ai servizi forniti dal sistema.

essere trasparente: deve mascherare i dettagli e le differenze dell'architettura sottostante che assicura la distribuzione dei servizi sulle componenti del sistema. Questa caratteristica risulta centrale per poter permettere l'agevole progettazione ed implementazione: il progettista/programmatore deve avere un certo grado di indipendenza dai dettagli della distribuzione dell'architettura.

Cosa è la trasparenza di accesso?

La trasparenza di accesso nasconde le differenze nella rappresentazione dei dati e nel meccanismo di invocazione per permettere l'interoperabilità tra oggetti. Questo significa anche che gli oggetti

devono essere accessibili attraverso la stessa interfaccia, sia che siano acceduti da locale che da remoto. In questa maniera un oggetto può essere facilmente spostato a run time da un nodo a un altro. In generale, questo tipo di trasparenza viene fornito di default dai sistemi, in quanto è il tipo di trasparenza necessario per assicurare l'interoperabilità in un ambiente eterogeneo.

Cosa è la trasparenza di locazione?
La trasparenza di locazione non permette di utilizzare informazioni riguardanti la localizzazione nel sistema di una particolare componente, che viene identificata ed utilizzata in maniera indipendente dalla sua posizione. Questo tipo di distribuzione fornisce una vista logica del sistema di naming, in modo da disaccoppiare il nome da una posizione all'interno della rete. Questo tipo di trasparenza è fondamentale per un sistema distribuito, in quanto senza di esso non si potrebbe spostare componenti da un nodo a un altro, poiché essi potrebbero essere riferiti secondo la loro posizione.

Cosa è la trasparenza di migrazione e perché dipende da trasparenza di accesso e di locazione?
Il compito di questo tipo di trasparenza è quello di nascondere la possibilità che il sistema faccia migrare un oggetto da un nodo a un altro, continuando ad essere raggiungibile ed utilizzabile da altri oggetti. Questo viene utilizzato per ottimizzare le prestazioni del sistema (bilanciando il carico tra nodi oppure riducendo la latenza per accedere a una componente) o anche per anticipare malfunzionamenti o riconfigurazioni del sistema (Ad esempio, se un nodo deve essere spento per aggiornamenti software, gli oggetti che si trovano sul nodo vengono spostati altrove, mantenendo lo stesso nome e tenendo attivi i servizi che stanno fornendo).
La trasparenza di migrazione dipende dalla trasparenza di accesso (che permette di accedere ad un oggetto, anche se locale, solo attraverso la propria interfaccia che viene usata da remoto) e dalla trasparenza di locazione (che nasconde la locazione fisica di un oggetto, permettendone l'accesso attraverso un sistema di naming logico fornito dal sistema).

Cosa è la trasparenza di replica e perché dipende da trasparenza di accesso e di locazione?
Con questo tipo di trasparenza, il sistema maschera il fatto che una singola componente viene replicata da un certo numero di copie (dette *repliche*) che vengono posizionate su altri nodi del sistema, e che offrono esattamente lo stesso tipo di servizio della componente originale. Ovviamente, il sistema si deve occupare di mantenere assolutamente coerente lo stato di tutte le repliche con la componente originale. Anche questo tipo di trasparenza dipende da quella di accesso e di locazione.
Le repliche vengono utilizzate per diversi scopi. Innanzitutto, per le prestazioni, facendo in modo di replicare componenti laddove (all'interno del sistema) maggiori sono le richieste per quel tipo di servizi, in modo da minimizzare la latenza per accedervi. Ma vengono anche utilizzate per poter far scalare il sistema in presenza di aumento del carico di lavoro.

Cosa è la trasparenza alla persistenza e perché dipende da trasparenza di locazione?
Questo tipo di trasparenza isola l'utente dalle operazioni che compie il sistema per rendere persistente (cioè in memoria secondaria) un oggetto durante una fase di non utilizzo.
Infatti gli oggetti di utilizzo raro non vengono mantenuti attivi (nello spazio di indirizzamento della memoria principale) ma vengono de-attivati, e memorizzati (con il loro stato) all'interno della memoria secondaria, mantenendo solamente un handle per la loro riattivazione, quando arrivano

richieste di operazioni da eseguire. In questo caso, l'oggetto viene riportato in memoria principale e reso attivo per rispondere alla richiesta. Gli oggetti che invocano servizi su un oggetto de-attivato non avvertono la differenza con le invocazioni su un oggetto attivo.
La trasparenza alla persistenza si basa sulla trasparenza di locazione, in quanto l'accesso indipendente dalla posizione fisica dell'oggetto permette una riattivazione dell'oggetto anche su nodi diversi da quelli su cui era stato de-attivato.

Cosa è la trasparenza alle transazioni(concorrenza)?
Un sistema distribuito è implicitamente concorrente, in quanto l'esistenza di diverse risorse accessibili da diversi nodi rende la concorrenza una regola.
La trasparenza alle transazioni (anche chiamata trasparenza alla concorrenza) nasconde all'utente le attività di coordinamento che vengono svolte per assicurare la consistenza dello stato degli oggetti in presenza della concorrenza. Sia l'utente che il progettista e lo sviluppatore sono ignari delle attività che vengono svolte per assicurare la atomicità delle operazioni e possono semplicemente ritenersi gli unici utenti all'interno del sistema.

Cosa è la trasparenza alla scalabilità e perché dipende da trasparenza di migrazione e di replica?
La scalabilità è uno dei principali motivi a favore di un sistema distribuito rispetto ad uno centralizzato. Un sistema viene detto scalabile quando è in grado di poter servire carichi di lavoro via via crescenti senza dover modificare la propria architettura e la propria organizzazione. Progettare un sistema scalabile è necessario visto che la platea di utenti ai quali potenzialmente un servizio su Internet viene offerto è smisurata. Un servizio deve potenzialmente poter scalare dalle poche decine alle centinaia di migliaia, o ai milioni di utenti senza che questo comporti la riprogettazione dell'intero sistema. Ovviamente, si dovranno acquisire nuove risorse, ma il sistema dovrà essere in grado di poterle utilizzare senza modifiche sostanziali.
La trasparenza alla scalabilità assicura che il progettista/sviluppatore non deve curarsi di come il proprio servizio scalerà al crescere delle richieste, ma sarà il sistema che provvederà, attraverso il meccanismo di replica e di migrazione, a fare in modo che le nuove risorse aggiunte al sistema vengano utilizzate per fare fronte al carico crescente.

Cosa è la trasparenza alle prestazioni e perché dipende da trasparenza di migrazione, di replica e di persistenza?
Il sistema rende il progettista/sviluppatore ignaro dei meccanismi che vengono utilizzati per ottimizzare le prestazioni del sistema, durante la fornitura di servizi. In particolare, il sistema può provvedere ad implementare politiche di bilanciamento del carico, spostando componenti da nodi carichi di lavoro verso nodi che hanno maggiori disponibilità di calcolo a disposizione, oppure politiche di minimizzazione della latenza, avvicinando (repliche di) componenti su nodi più vicini (in termini di topologia di rete) agli utenti che li usano più frequentemente, oppure politiche di ottimizzazione delle risorse di memoria, che prevedono la inattivazione di oggetti che non vengono usati frequentemente e che possono essere re-attivati se necessario. Per questo motivo la trasparenza alle prestazioni si appoggia sulla trasparenza di migrazione, di replica e di persistenza.

Cosa è la trasparenza ai malfunzionamenti e perché dipende dalla trasparenza alle transazioni?
La trasparenza ai malfunzionamenti nasconde ad un oggetto il malfunzionamento di oggetti con i quali sta interoperando. La trasparenza si estende agli utenti del sistema che non devono avere la sensazione di malfunzionamenti parziali all'interno del sistema, in quanto il sistema deve automaticamente riconfigurare la richiesta e fornire il servizio in maniera alternativa.
Si basa sulla trasparenza alle transazioni in quanto operazioni complesse eseguite come una transazione, se interrotte a causa di un malfunzionamento non vengono confermate (*commit*) e quindi non alterano lo stato della risorsa e possono essere ripetute su una replica.

PARTE 3

Qual è la differenza tra un processo e un thread? E quali le somiglianze?

I processi hanno memoria privata e la comunicazione avviene tramite strumenti come pype, socket , ovvero tramite un meccanismo di IPC Inter-Process Communication). I thread invece condividono le risorse, quindi condividono memoria, file aperti e così via. Entrambi eseguono istruzioni di un codice.

Perché dire che un thread è un "processo light-weight" è tecnicamente scorretto, anche se molto diffuso nella prassi comune?

È scorretto perché bisogna specificare che un thread è un processo light-weight che non partiziona la memoria e che quindi non ha memoria propria.

Quali sono esempi di applicazione che hanno bisogno di essere multithread?

Un browser che allo stesso tempo deve: scaricare dati, visualizzarli, reagire all'eventuale pulsante di stop premuto dall'utente.
Un' applicazione di rete che allo stesso tempo deve: chiedere dati a un'altra applicazione, fornire dati a chi li richiede, gestire l'input/output con l'utente.
I server che gestiscono i thread per gestire le richieste in concorrenza. Quando arrivano tante richieste esse devono essere servite quanto più velocemente possibile.
Applicazione streaming audio che deve poter: leggere l'audio dalla rete, decomprimerlo, gestire l'output, aggiornare il display.

Come usa un server il pool di thread per ottimizzare le prestazioni?

Un server inizialmente istanzia un pool di thread e li tiene in stato "sospeso" pronti per essere utilizzati, in questo modo non c'è bisogno di istanziare memoria per creare un nuovo thread ogni volta. Quando arriva una richiesta le viene associato uno di questi thread già pronti, così da servire la richiesta velocemente.

Come si può creare un thread in java e quali sono i vantaggi e svantaggi delle tue tecniche possibili?

Un thread in java si può creare in due modi:
il primo modo consiste nel creare una classe (Es. HelloThread) che estende la super classe Thread. In questa classe si definisce il metodo run(), che viene eseguito quando viene eseguito il thread. In questa classe è presente anche il main, il quale, quando viene eseguita la classe HelloThread, provvede ad istanziare un nuovo oggetto HelloThread e a lanciarlo eseguendo il comando start().
Vantaggi: è molto semplice da realizzare.
Svantaggi: non è possibile estendere più di una classe. Se la mia classe estende già un'altra classe, non posso estendere la classe Thread.
Il secondo modo prevede la creazione di una classe (Es. HelloRunnable) che implementa l'interfaccia Runnable. L'interfaccia Runnable vincola l'implementazione del metodo run all'interno della classe. Anche qui è presente il main. Quando viene creato un oggetto HelloRunnable, questo oggetto di per sé non è un thread, ma un oggetto che implementa Runnable. Nel main questo oggetto viene passato al costruttore della classe Thread, il quale crea un thread che a questo punto può essere lanciato con il comando start.
Vantaggi: è di più generale utilizzo ed è utilizzabile anche per l'approccio con executors.

Quando si invoca il metodo .start() su un thread cosa succede?

Il metodo .start() rende il thread disponibile per l'esecuzione, ovvero pone il thread nello stato di runnable.

Quando si invoca il metodo .run() su un thread cosa succede?
Il metodo .run() viene eseguito in modo sequenziale, come se stessimo chiamando un normale metodo di un oggetto. Il thread non viene eseguito.

Quando si invoca il metodo .start() su un thread, il thread viene eseguito: vero o falso?
Falso. il thread viene posto nello stato di runnable, sarà compito dello scheduling decidere a quale thread assegnare la CPU.

Quando si invoca il metodo .run() su un thread, il thread viene eseguito: vero o falso?
Falso. Il thread non esiste nemmeno perché non è stato eseguito il comando start.

Qual è la differenza tra un metodo run() e un metodo start() di un thread?
Il metodo start serve per creare un thread e renderlo disponibile per l'esecuzione, mentre il metodo run viene eseguito solo quando il thread è in esecuzione.

Cosa sono gli interrupt su un thread e quali metodi sono a disposizione per trattarli/gestirli?
Un interrupt è l'interruzione di un thread, un'indicazione che un thread dovrebbe fermare quello che sta facendo e fare qualcosa altro.
Per gestire un interrupt è possibile utilizzare il blocco try catch per catturare l'interrupt e gestirlo. Un thread che non invoca un metodo che lancia l'eccezione InterruptedException può controllare se è stato interrotto utilizzando *Thread.interrupted()*. In questo caso se è stato ricevuto un interrupt allora viene lanciata l'eccezione.

Quali sono gli stati del diagramma di stato di un thread?
Appena creato un thread è nello stato nuovo. Quando viene eseguito il metodo start il thread viene posto nello stato di *Runnable*, ovvero è disponibile per l'esecuzione. Nello stato di runnable il thread inizialmente è nel sottostato *Ready*. Qui lo scheduler può selezionare il thread per farlo eseguire e quindi lo pone nello stato di *Running*. Dallo stato di running lo scheduler può sospendere l'esecuzione del thread e porlo nello stato di ready. Dallo stato di ready un thread può essere posto in altri tre stati:
Timed Waiting: è lo stato di attesa temporizzata. Un thread viene posto in questo stato quando esegue comandi come sleep(sleeptime), join(timeout), wait(timeout), ovvero il thread decide di fermarsi volontariamente e uscire dallo scheduling per la quantità di tempo scelta. Quando è scaduto il tempo di attesa il thread esce da questo stato e torna allo stato ready.
Waiting: è lo stato di attesa non temporizzata. Un thread viene posto in questo stato quando esegue comandi come wait, join senza selezionare una quantità di tempo da attendere, ad esempio quando un thread A vuole aspettare la terminazione di un thread B. Un thread esce da questo stato quando arriva la notifica che quello che stava aspettando è terminato e ritorna nello stato ready.
Blocked: è lo stato di blocco. Un thread viene posto in questo stato quando cerca di utilizzare una risorsa che è già in uso da parte di un altro thread. Da questo stato il thread esce quando acquisisce il monitor, ovvero quando la risorsa diventa disponibile.
N.B dai tre stati precedenti un thread può uscire anche se è stato terminato e, in questo caso, non ritorna nello stato ready ma viene terminato definitivamente.

Un thread come passa dallo stato “New” a quello di “Runnable”?
Quando viene eseguito il comando start, un thread viene reso disponibile per l’esecuzione e quindi passa dallo stato di New allo stato di Runnable.

Un thread come passa dallo stato “Runnable” a quello di “Timed Waiting” e come se ne esce?
Un thread viene posto dallo stato Runnable a quello di Timed Waiting quando esegue comandi come sleep(sleeptime), join(timeout), wait(timeout), ovvero il thread decide di fermarsi volontariamente e uscire dallo scheduling per la quantità di tempo scelta. Un thread esce da questo stato in due modi:
il tempo di attesa è scaduto e il thread torna nello stato di ready;
il thread viene terminato, quindi non ritorna nello stato ready.

Un thread come passa dallo stato “Runnable” a quello di “Waiting” e come se ne esce?
Un thread viene posto dallo stato Runnable a quello di Waiting quando esegue comandi come wait, join, senza selezionare una quantità di tempo da attendere, ad esempio quando un thread A vuole aspettare la terminazione di un thread B. Un thread esce da questo stato in due modi:
arriva una notifica avvisando che quello che stava aspettando è terminato e così torna nello stato ready;
il thread viene terminato, quindi non ritorna nello stato ready.

Un thread come passa dallo stato “Runnable” a quello di “Blocked” e come se ne esce?
Un thread viene posto dallo stato Runnable a quello di Blocked quando cerca di utilizzare una risorsa che è già in uso da parte di un altro thread. Un thread esce da questo stato in due modi:
quando acquisisce il monitor, ovvero quando la risorsa diventa disponibile e così torna allo stato ready;
il thread viene terminato, quindi non ritorna nello stato ready.

Perché lo stato “Runnable” è strutturato in due sottostati “Ready” e “Running” e come si passa dall’uno all’altro?
Un thread che si trova nello stato Runnable può trovarsi nello stato di Ready o nello stato di Running. Quando un thread è nello stato di Ready, lo scheduler può selezionarlo per farlo eseguire e quindi porlo nello stato di Running. Quando un thread è nello stato di Running può essere sospeso dallo scheduler e tornare, quindi, nello stato di Ready.

Quali sono i possibili tipi di errore che vengono generati dai thread?
Ci sono due possibili errori: interferenza tra thread e inconsistenza della memoria.

Cosa è l’interferenza tra thread? E come si risolve?
L’interferenza tra thread avviene quando le operazioni effettuate da più thread per accedere alla stessa risorsa si intrecciano tra di loro. Questo problema si risolve con la sincronizzazione.

Cosa è la inconsistenza della memoria? E come si risolve?
L’inconsistenza della memoria avviene quando thread diversi hanno visione diverse dei dati. Ad esempio un thread A può avere nella sua cache un certo valore per una variabile C, mentre il thread B può avere un altro valore (perché non aggiornato) per la stessa variabile C. Non sempre i valori in

cache corrispondono con quelli in memoria. Questo problema si risolve stabilendo la relazione *happens-before*.

Cosa è la race condition?

La race condition si presenta quando il risultato di un'operazione dipende dall'ordine di esecuzione di diversi thread.

Perché i bug dovuti a race condition sono particolarmente complessi da trattare?

Perché l'uso del debugger può alterare l'ordine di esecuzione dei thread. Quindi può accadere che quando viene utilizzato il debugger non c'è interferenza tra thread e il programma funziona correttamente, invece quando non viene utilizzato può accadere che il programma non funzioni.

Cosa è la relazione "happens-before"?

La happens-before è una garanzia che la memoria scritta da un thread è visibile da un altro thread.

Come faccio a stabilire una relazione "happens-before"?

Un modo è quello di utilizzare la *sincronizzazione*. Anche due operazioni introducono una relazione happens-before:
Thread.start(): gli effetti del codice che ha condotto alla creazione sono visibili al nuovo thread;
Thread.join(): quando la terminazione di un thread A causa il return della join() di B, tutte le istruzioni di A sono in happens-before con le istruzioni di B che seguono la join.
Una altra maniera è rendere la variabile *volatile:* una scrittura a un campo volatile happens-before ogni successiva lettura della variabile (da parte di qualsiasi thread).

Cosa significa che una variabile in memoria è dichiarata "*volatile*"?

Significa che ogni volta che viene effettuata una scrittura su quella variabile in memoria, tale scrittura deve essere immediatamente comunicata a tutte le cache.

PARTE 4

Perché è necessaria la sincronizzazione?
La sincronizzazione è necessaria per risolvere i problemi derivanti dall'utilizzo di thread, ovvero l'interferenza tra thread e l'inconsistenza di memoria.

Perché è necessaria la sincronizzazione efficiente?
La sincronizzazione è necessaria che sia efficiente in modo da minimizzare la parte sequenziale in un codice e massimizzare la parte parallela, facendo sì che il programma sia più veloce.

Cosa sono i metodi sincronizzati?
I metodi sincronizzati sono un costrutto del linguaggio Java, che permette di risolvere semplicemente gli errori di concorrenza. Non è possibile che due esecuzioni dello stesso metodo sullo stesso oggetto siano interfogliate.

Cosa sono i metodi statici sincronizzati?
Un metodo statico sincronizzato previene l'esecuzione interfogliata di tutti gli altri metodi statici sincronizzati. In pratica si acquisisce il lock sull'oggetto ClassName.class.

Che relazione (dal punto di vista dell'accesso esclusivo) hanno un metodo sincronizzato dell'istanza e un metodo sincronizzato statico?
Metodi sincronizzati statici garantiscono accesso in mutua esclusione a metodi sincronizzati statici, mentre metodi sincronizzati di istanza garantiscono accesso in mutua esclusione a ai metodi sincronizzati di quella istanza.

Rispondere si/no e motivare la risposta per le seguenti domande:
Un thread che è in esecuzione di un metodo sincronizzato dell'istanza blocca l'esecuzione di:

un altro thread che esegue un metodo sincronizzato dell'istanza
Si, perché un metodo sincronizzato blocca l'esecuzione degli altri thread che tentano di eseguire un metodo sincronizzato.

un altro thread che esegue un metodo statico sincronizzato
No, perché un metodo sincronizzato non interferisce con il blocco di un thread che vuole eseguire un metodo statico sincronizzato. Sono due classi di sincronizzazioni totalmente separate.

un altro thread che esegue un metodo statico
No, perché un metodo sincronizzato dell'istanza blocca l'esecuzione solo dei thread che cercano di eseguire un altro metodo sincronizzato dell'istanza.

un altro thread che esegue un metodo dell'istanza
No, perché un metodo sincronizzato dell'istanza blocca l'esecuzione solo dei thread che cercano di eseguire un altro metodo sincronizzato dell'istanza. Un altro thread può eseguire un metodo dell'istanza non sincronizzato.

tutti i thread in esecuzione sulla JVM
No, perché vengono bloccati solo i thread che cercano di accedere ai metodi sincronizzati dell'istanza. Quindi altri thread, che vogliono eseguire metodi non sincronizzati, non vengono bloccati

tutti i thread in esecuzione su tutte le JVM in esecuzione sulla vostra macchina
No, principalmente perché ogni JVM è un processo diverso e i processi non condividono memoria. Quindi i thread generati da ogni JVM non interferiscono tra JVM diverse.

tutti i thread in esecuzione su tutte le JVM in esecuzione su tutti i vostri computer
NO.

tutti i thread in esecuzione su tutte le JVM in esecuzione su tutti i computer dell'Università
NO.

È possibile con i lock impliciti simulare i metodi sincronizzati di istanza?
Si, bisogna utilizzare il lock sull'oggetto *this*.

È possibile con i lock impliciti simulare i metodi sincronizzati statici?
Si, bisogna utilizzare il lock sull'oggetto ClassName.class.

È possibile usare una variabile volatile per eliminare l'interferenza tra thread (race condition)?
No, perché una variabile volatile garantisce solo la relazione *happens-before*, ma questa non ci permette di eliminare l'interferenza tra thread.

È possibile usare una variabile volatile per eliminare i problemi di inconsistenza della memoria?
Si, perché una variabile volatile garantisce la relazione *happens-before*.

Operazioni atomiche sono estremamente più efficienti delle operazioni non atomiche: vero/falso e perché?
Vero, perché le operazioni atomiche non sono interrompibili e si completano del tutto oppure per niente.

Se la variabile int a è volatile, allora la operazione a++ viene eseguita in mutua esclusione da parte di due thread: vero o falso e perché?

Falso, l'operazione a++ sulla variabile volatile non viene eseguita in mutua esclusione da due thread. L'operazione a++ è composta da tre operazioni che possono essere interfogliate tra altri thread.

Cosa è il deadlock?
Il deadlock è un problema che si presenta quando due thread sono bloccati, ognuno in attesa dell'altro.

Cosa è lo starvation?
È un problema che si presenta quando un thread non riesce ad acquisire accesso ad una risorsa condivisa, in maniera da non riuscire a fare progresso. Ad esempio quando c'è un metodo sincronizzato che impiega molto tempo se è invocato spesso, altri thread possono essere pervenuti dall'accesso.

Cosa è il livelock?

Il livelock è un problema che si presenta quando a thread A può reagire ad azioni di un altro thread B che reagisce con una risposta verso A. I due thread non sono bloccati ma sono occupati a rispondere alle azioni dell'altro.

Quale è la differenza tra deadlock e livelock?
Nel livelock i due thread sono in esecuzione ma sono occupati a rispondere alle azioni dell'altro e quindi non c'è progresso. Nel deadlock invece i due thread sono bloccati in attesa di essere eseguiti.

PARTE 5

Cosa è un design pattern?
Un design pattern (schema progettuale) è un concetto che può essere definito "una soluzione progettuale generale ad un problema ricorrente.

A cosa serve il design pattern del Singleton?
Il design pattern del Singleton serve a restringere l'istanziazione da parte di una classe ad 1 oggetto. Cioè di quella classe può esistere un solo oggetto. È una maniera per ottimizzare la gestione di alcune risorse, ad esempio nel nostro sistema chi gestisce la stampante deve essere un solo oggetto.

Cosa è la *lazy allocation*?
L'allocazione di un oggetto avviene solo quando utilizzato la prima volta

La soluzione al Singleton con il metodo getInstance() sincronizzato è corretta? Se no, perché? Se si, allora perché ne studiamo delle altre?
La soluzione con il metodo getInstance() sincronizzato funziona, ma la sincronizzazione serve solo la prima volta. I thread che entrano successivamente devono solo avere come risultato una instance, ma non possono eseguire il metodo in parallelo perché è sincronizzato.

Perché la soluzione al Singleton con un blocco synchronized subito dopo l'if non funziona?
Consideriamo due thread che eseguono il metodo getInstance in parallelo. Entrambi eseguono l'if ed entrambi lo superano perché instance inizialmente è nullo. Arrivati al blocco synchronized il thread A entra nel blocco mentre il thread B è nello stato di blocked. Il thread A crea la prima istanza ed esce dal blocco restituendo la prima istanza. Il thread B a questo punto entra nel blocco synchronized e crea un nuovo singleton che sovrascrive il primo.

Cosa è il double-checked locking? Perché non è corretto? Come si può modificare per renderlo corretto?
Il controllo sulla variabile instance viene effettuato prime e dopo la sezione critica. Fuori dalla sezione critica viene controllato che non esiste l'istanza. Se non esiste si entra nella sezione critica e si controlla che nel waiting non sia cambiata la situazione. Se non è cambiata la situazione viene creato un nuovo Singleton e si esce dalla sezione critica.
È scorretto perché non tiene conto dell'inconsistenza della memoria. La chiamata al costruttore "sembra" essere prima dell'assegnazione di instance, ma questo non è detto che accada. Un Singleton non inizializzato potrebbe essere assegnato a istance e se un altro thread controlla il valore poi lo può usare in maniera scorretta.
Soluzione: rendere la variabile instance *volatile*. Oppure utilizzare le classi statiche con l'idioma "Inizialization-on-demand holder". LazyHolder è inizializzata dalla JVM solo quando serve (alla prima getInstance()). Essendo un inizializzatore statico, viene eseguito una sola volta (al caricamento) e stabilisce una relazione happens-before tutte le altre operazioni sulla classe.

PARTE 6

Come funzionano i socket TCP in java?
I socket TCP sono gli endpoint di una comunicazione bidirezionale sulla rete che unisce due programmi, normalmente un client e un server.
Ad ogni socket viene assegnato un numero di porta che serve a identificare l'applicazione che è incaricata di dover trattare i dati, che è in esecuzione sul computer che li riceve. Quindi un socket viene univocamente definito dalla combinazione di indirizzo IP e numero di porta.

Quale è la differenza tra le classi ServerSocket e Socket in Java?
ServerSocket è il socket che apre il server e viene utilizzato per ricevere richieste di connessione. Quando il server riceve una richiesta assegna un socket alla connessione bidirezionale, restituendo l'oggetto Socket che viene utilizzato per la comunicazione tra client e server.
Il ServerSocket è unico mentre i Socket sono uno per ogni client connesso.

Cosa sono ed a cosa servono gli stream?
Lo stream è una sequenza ordinata di byte. Gli stream permettono di trasmettere istanze di classi Java (oggetti) tra client e server, attraverso un meccanismo di serializzazione. Gli stream I/O sono un'astrazione che Java fornisce al programmatore per trattare con una sequenza di dati che può essere "diretta a" / "proveniente da" diverse entità, quali file, periferiche, memoria e socket.

Cosa significa che gli stream vengono usati tipicamente come wrapper di altri stream?
Significa che ogni classe via via più specializzata prende come argomento per il costruttore una istanza di classi più alte nella gerarchia

Quali sono i metodi più importanti (e cosa fanno) di InputStream e di OutputStream?
I metodi più importanti sono getInputStream() e getOutputStream(), i quali restituiscono lo stream di input e di output associati all'oggetto chiamante. Questo ci permette di comunicare con l'oggetto.

Perché di solito i programmatori non usano InputStream o OutputStream direttamente?
Perché questi stream trattano solamente i byte e i programmatori non vogliono trattare i byte. Quindi questi stream vengono passati alle classi ObjectInputStream e ObjectOutputStream.

Come si fa a usare gli stream che sono associati ad un socket?
Dopo aver creato un oggetto Socket, viene creato un oggetto di tipo ObjectInputStream (ObjectOutputStream) al cui costruttore viene passato lo stream associato al socket tramite il metodo socket. getInputStream (socket. getOutputStream).

Quali sono i metodi offerti da ObjectInputStream (e ObjectOutputStream) per leggere (scrivere) un oggetto dallo (sullo) stream?
Per leggere abbiamo il metodo readObject(). È un comando bloccante, ovvero attende che sull' ObjectOutputStream corrispondente venga scritto un oggetto. Quando lo riceve, restituisce l'oggetto trasmesso (ed opportunamente deserializzato dallo stream). Visto che restituisce un object noi dobbiamo essere in grado di fare il casting.
Per scrivere abbiamo il metodo writeObject (), il quale prima di tutto serializza l'oggetto che vogliamo inviare e poi lo trasmette sullo stream.

PARTE 8

In che maniera la esperienza di Jim Waldo all'interno della progettazione di CORBA ha influito su alcuni requisiti di progettazione di Java RMI?

Avendo già effettuato ricerche sugli oggetti distribuiti in altri linguaggi e avendo fatto parte del team di progettazione di CORBA, Jim Waldo era a conoscenza di tutti gli errori fatti e quindi la proposta di Java RMI stabiliva obiettivi che, basati sulle esperienze precedenti, potevano garantire un'efficacie implementazione del modello ad oggetti.

Quale è il motivo dell'obiettivo di RMI come ambiente semplice? e come ambiente familiare al programmatore Java?

Un'ambiente semplice favorisce la diffusione, ma soprattutto impedisce che utilizzi errati e scorretti delle potenzialità del sistema possano creare problemi ai sistemi realizzati.
Un'ambiente familiare per il programmatore ne permette il facile utilizzi, poiché il programmatore già conosce e sa usare al meglio le caratteristiche del linguaggio di programmazione.

Quali sono i vantaggi dell'integrare il modello distribuito all'interno di un linguaggio di programmazione?

Il motivo fondamentale è il marketing. Inoltre permette di offrire un ambiente familiare allo sviluppatore Java, che può usare gli stessi strumenti, modelli e astrazioni che vengono utilizzati per oggetti locali. Ciò deve avvenire cercando di preservare la semantica degli oggetti che sono all'interno del linguaggio.

Perché la garbage collection rappresenta un utile strumento per la semplificazione dei compiti del programmatore?

Perché la garbage collection solleva il programmatore del doversi occupare della allocazione e deallocazione della memoria. Questo previene errori di memory leak.

Quali sono i rischi di memory leak? E perché sono problemi che sono particolarmente critici per i server?

Un memory leak si verifica quando, in caso di errori di programmazione, un programma può continuare ad allocare memoria per oggetti e non deallocarne mai, esaurendo in beve lo spazio di memoria a disposizione.
Questo è particolarmente pericoloso per applicazioni distribuite in quanto i server sono in esecuzione continua: anche il più piccolo memory-leak può portare n pochi giorni un server a esaurire la memoria a disposizione

Quali sono i vantaggi e gli svantaggi di un ambiente in cui la gestione della memoria è a carico del programmatore? e quando invece è a carico del sistema con un meccanismo di garbage collection?

Quando la gestione della memoria è a carico del programmatore l'esecuzione del programma è più efficiente ma in caso di errori di programmazione si verifica un memory leak.
Quando la gestione della memoria è a carico del sistema il garbage collection previene errori di memory leak, quindi il sistema è più stabile ma di contro c'è un calo di prestazioni. Inoltre c'è l'impossibilità di sviluppare un'applicazione real time.

Cosa sono le invocazioni unicast, multicast, di oggetti attivabili?
Un'invocazione unicast consiste nell' invocare un metodo e ottenere il risultato.
Un'invocazione multicast consiste nell'avere più oggetti server replicati che rispondono alle invocazioni e ottenere il risultato dell'invocazione dal primo oggetto che risponde. Si mette il meccanismo di replica a livello dell'oggetto.

A cosa può servire avere oggetti attivabili?
Un oggetto server attivabile è un oggetto che viene messo sul disco dalla JVM e viene attivato solo al momento della invocazione. Inoltre questo meccanismo fornisce riferimenti persistenti ad oggetti distribuiti, gestendo la loro esecuzione sulla base delle richieste ricevute. È inutile avere attivi degli oggetti se non vengono utilizzati.

Perché è importante poter sostituire livelli di trasporto (in RMI ma anche in altri sistemi) senza dover modificare gli strati superiori, fino alla applicazione?
Perché si deve essere aperti verso future espansioni che prevedano che il protocollo di trasporto che viene utilizzato possa essere modificato, senza intaccare l'intera applicazione.

Cosa significa che una applicazione Java viene eseguita in una sandbox?
Una sandbox è un ambiente dedicato, ristretto e controllato, all'interno del quale le operazioni che il programma può eseguire non risultano pericolose (sia accidentalmente che intenzionalmente).

Quali sono i 4 livelli di sicurezza forniti da Java?
I 4 livelli di sicurezza forniti da Java sono:

1. La sicurezza intrinseca del linguaggio
2. Il Classloader
3. Il Bytecode Verifier
4. Il Security Manager

Per ciascuno dei 4 livelli di sicurezza, fornire l'obiettivo e alcuni esempi di funzionamento

1) Sicurezza intrinseca del linguaggio
- Il linguaggio Java è fortemente tipizzato (strongly typed): tutte le variabili hanno un tipo definito a tempo di compilazione e solamente poche implicite conversioni (casting) vengono effettuate dal compilatore e dalla macchina virtuale a run-time. Tutte le altre operazioni di casting devono essere esplicitate dal programmatore.
- Java offre la gestione automatica della memoria attraverso un meccanismo di garbage collection.
- L'assenza di puntatori e l'impossibilità di poter fare aritmetica o assegnamenti con i riferimenti rende impossibile effettuare accessi illegali in memoria.
- L'accesso alla memoria reale non viene determinato a tempo di compilazione ma a tempo di esecuzione, quindi non si può conoscere in anticipo in che zona di memoria verranno memorizzati gli oggetti e quindi non si può scrivere codice per alterarli
- A tempo di esecuzione vengono controllati i limiti di array per prevenire accessi a elementi non esistenti.

2) Classloader
Si occupa di caricare la classe a tempo di esecuzione. Il suo compito principale è quello di caricare la classe in un namespace separato rispetto a quello delle classi locali, in modo che classi del linguaggio built-in, locali, non possono essere rimpiazzate da altre.

3) Bytecode Verifier
Il Bytecode Verifier controlla che il byte code della classe compilata sia conforme alle specifiche del linguaggio, che non ci siano stack underflow/overflow, e che non ci siano violazioni alle regole specificate dai modificatori di accesso. In poche parole verifica gli errori sintattici.

4) Security Manager
Si occupa di controllare che le operazioni che effettua il programma siano state permesse dal programmatore, ovvero da colui che sta eseguendo il codice. Si occupa di definire i confini della sandbox e utilizza un file di policy per applicare le politiche di sicurezza alle operazioni potenzialmente pericolose.

Perché è necessario che il Bytecode Verifier "ripeta" i controlli che (ovviamente?) il compilatore ha già effettuato?
Perché dopo che una classe java è stata compilata, il file .class che contiene il byte code può essere aperto con un editor di testo e quindi modificato, rendendo vani i controlli effettuati dal compilatore.

Quali sono i package in cui si struttura Java RMI e quali sono le loro funzioni?
Java RMI è contenuto in 5 package: *java.rmi* e *java.rmi.server* che contengono il meccanismo basilare di funzionamento delle invocazioni remote, *java.rmi.activation* per gli oggetti attivabili, *java.rmi.dgc* per la Distributed Garbage Collection e *java.rmi.registry* per il servizio di localizzazione.

Cosa è una interfaccia Remota?
Un'interfaccia remota per Java RMI deve estendere l'interfaccia *java.rmi.Remote* che è un'interfaccia cosiddetta marker, cioè un'interfaccia vuota che, in questo caso, serve solamente per poter segnalare che essa definisce dei metodi accessibili da remoto.

Come vengono distinti i metodi remoti?
Un metodo remoto è un metodo che lancia *java.rmi.RemoteException*. Questa è una checked exception, quindi deve essere gestita in una try/catch.

Cosa significa che la interfaccia remota aggiunge un ulteriore modificatore di accesso ai tradizionali valori in Java di public, private, etc. ?
Significa che i metodi remoti, dichiarati in una interfaccia remota, sono più accessibili dei metodi *public* che risultano accessibili, ma solamente da invocazioni all'interno della stessa macchina virtuale. Infatti un metodo remoto è accessibile da invocazioni all'esterno della JVM in cui si trova.

Come devono essere i parametri di un metodo remoto?
I parametri remoti di un metodo remoto devono essere dichiarati tramite la propria interfaccia remota, non utilizzando la classe dell'implementazione remota

Cosa si intende quando si dice che un oggetto locale è passato per copia invece che per riferimento?

Quando un oggetto locale viene passato per copia viene effettuata appunto una copia dell'oggetto in memoria. Avremo quindi due locazioni di memoria separate, puntate da due variabili, ma che contengono la copia dello stesso oggetto. In questo modo ogni modifica che si effettua ad una copia dell'oggetto non influisce sull'altra copia.

Quando un oggetto viene passato per riferimento esiste una sola copia in memoria dell'oggetto, che però è puntata da più variabili. In questo modo ogni modifica che si effettua all'oggetto, essendo unico, viene vista da tutte le altre variabili che puntano all'oggetto.

Cosa garantisce la integrità referenziale?
L'integrità referenziale garantisce che quando vengono passati più riferimenti allo stesso oggetto nella stessa invocazione, allora anche sulla macchina remota alla quale sono stati passati i riferimenti punteranno allo stesso oggetto.

Come si fa a localizzare un oggetto remoto?

Per poter invocare il metodo remoto di un oggetto remoto, l'oggetto client deve avere a disposizione il riferimento remoto. Questo può essere reperito in due maniere: **(a)** ottenuto come risultato di altre invocazioni (locali o remote) di metodi; **(b)** attraverso un servizio di naming.
Un oggetto remoto viene localizzato con il servizio di *naming*. Java RMI fornisce un semplice meccanismo di *name server* nella classe *java.rmi.Naming*. Tale classe fornisce metodi per ricercare (*lookup()*), registrare (*bind()*, *unbind()*, *rebind(*)) ed elencare (*list()*) gli identificativi registrati.

Perché è importante che il compilatore forzi la gestione della eccezione di RemoteException per i metodi remoti?
Perché RemoteException è una checked exception e quindi va gestita con una try/catch. Il programmatore deve esplicitare la gestione di questa eccezione.

Cosa è un metodo idempotente? e perché è utile come metodo remoto?
Un metodo idempotente è un metodo che se viene ripetuto tante volte con lo stesso parametro genera lo stesso risultato. È utile come metodo remoto perché se si verifica un'eccezione durante l'invocazione di un metodo remoto, non sappiamo se il problema è avvenuto durante l'inizio dell'invocazione, durante l'esecuzione o durante la risposta del metodo remoto.

PARTE 9

Perché gli oggetti remoti è bene che siano (quanto più possibile) simili agli oggetti locali?

Perché non è possibile la totale trasparenza, tra oggetti locali e remoti?
Perché il programmatore deve conoscere quale oggetto è locale e quale è distribuito, in modo da capirne e gestirne il comportamento diverso.

Cosa fa la classe RemoteObject? Cosa ridefinisce di Object?
La classe RemoteObject ridefinisce alcuni metodi di Object. I metodi ridefiniti sono *hashCode(), equals()* e *toString()*.
Il metodo **X.hashCode()** viene ridefinito in maniera che restituisca lo stesso codice per due stub diversi di oggetti remoti che si riferiscono allo stesso oggetto remoto. In questa maniera gli oggetti remoti, o meglio i loro stub, possono essere utilizzati come chiavi nelle tabelle hash.
Il metodo **X.equals()** restituisce un booleano che è vero se il riferimento remoto passato è uguale a quello di X. In effetti il confronto viene fatto sugli stub e quindi l'uguaglianza è effettuata sui riferimenti e non sul contenuto, visto che per poter controllare il contenuto di un oggetto remoto si dovrebbe fare una chiamata remota, che può generare una RemoteException che non viene lanciata dalla firma del metodo equals() come definito in Object().
Il metodo **toString()** su un oggetto remoto deve poter restituire informazioni addizionali circa la macchina sulla quale quell'oggetto remoto si trova, oltre alle consuete informazioni circa il nome della classe ed un codice hash.

Quale è la differenza nel passaggio di parametri remoti?

Argomenti locali a invocazioni remote sono passati per copia, mentre quelli remoti sono passati per riferimento.

Descrivere la architettura a layer di Java RMI con i compiti di ciascun layer e la maniera in cui comunicano tra di loro

Il sistema di Java RMI è strutturato su tre livelli (layer):

- Stub/Skeleton Layer che comprende gli stub lato client e gli skeleton lato server;
- Remote Reference Layer che specifica il comportamento della invocazione e la semantica del riferimento;
- Transport Layer che si occupa della connessione e della sua gestione.

Quali sono i vantaggi di una architettura a layer? e quali gli svantaggi?
Un'architettura di questo tipo permette di astrarre le funzionalità fornite da un livello sottostante o sovrastante, con il quale comunica attraverso un protocollo ben definito, ma che si basa sulle funzionalità offerte e non sulla loro implementazione. In questa maniera l'architettura può evolvere mantenendo la sua struttura tramite la sostituzione di un livello con un altro equivalente.

Cosa fa lo Stub&Skeleton/Remote-Reference/Transport layer?
Lo **Stub/Skeleton Layer** si occupa di essere l'interfaccia tra l'applicazione (cioè le classi Java scritte dal programmatore) ed il resto del sistema. L'interfaccia verso il basso, in direzione del remote

reference layer , consiste nel fornire uno *stream di marshal* di oggetti Java che vengono passati al remote reference layer. Gli oggetti che vengono passati al RRL vengono passati per copia.

Lo stub è incaricato di

1) Iniziare la connessione con la macchina virtuale remota, chiamando il RRL;
2) Effettuare il marshalling verso un stream di marshal, fornito dal RRL;
3) Attendere il risultato della invocazione;
4) Effettuare l'unmarshalling dei valori restituiti (o delle eccezioni verificatesi);
5) Restituire il valore verso l'oggetto client che ha richiesto la invocazione.

Lo skeleton è incaricato di effettuare il *dispatching* verso l'oggetto remoto, cioè curare che la invocazione sia effettuata sull'oggetto remoto in attesa di invocazioni. Quando uno skeleton riceve una invocazione in entrata, si occupa di:

1) Effettuare l'unmarshalling dal RRL (lato server) dei parametri per la invocazione;
2) Invocare il metodo sulla implementazione che si trova nella sua JVM;
3) Effettuare il marshalling del valore restituito (compreso di eventuali eccezioni) verso chi ha invocato il metodo.

Il **Remote Reference Layer** si occupa di interfacciare il livello di trasporto con quello di stub/skeleton fornendo e supportando la semantica della operazione di invocazione di un metodo. In poche parole si occupa della modalità di invocazione. Le modalità permesse sono:

1) Invocazioni *unicast*, vale a dire da un singolo client verso un singolo server;
2) Invocazioni *multicast*, vale a dire un singolo client fa una invocazione ad un insieme di server replicati, in maniera da poter garantire la ridondanza: se uno di questi è in esecuzione allora risponderà alla invocazione;
3) Invocazioni di *oggetti attivabili*: le invocazioni potrebbero essere effettuate ad un oggetto remoto che è persistente, vale a dire viene attivato se arrivano delle invocazioni;
4) Invocazioni con *riconnessione*: le invocazioni potrebbero tentare invocazioni alternative se l'oggetto remoto originariamente contatta non risponde alla invocazione.

Verso l'alto fornisce un riferimento ad un oggetto che implementa la interfaccia java.rmi.server.RemoteServer, che espone un metodo invoke() per effettuare l'inoltro della invocazione, che viene chiamato dallo stub.
Verso il basso interagisce con il livello di trasporto utilizzando l'astrazione di una connessione orientata ai flussi (stream di connessione).

Il **livello di trasporto** ha il compito di:

1) stabilire la connessione verso macchine con indirizzi IP remoti;
2) gestire le connessioni e monitorare il loro stato;
3) rimanere in ascolto per connessioni in arrivo;
4) gestire una tabella degli oggetti remoti che risiedono nello spazio di indirizzamento locale;
5) stabilire una connessione per le chiamate in entrata;
6) identificare l'oggetto dispatcher a cui inoltrare la connessione.

Il protocollo utilizzato da RMI è chiamato Java Remote Method Protocol.

In che maniera il marshalling è diverso dalla serializzazione?
Fare il marshalling di un oggetto in Java significa effettuare una serializzazione modificando la semantica dei riferimenti remoti (invece di un riferimento remoto viene inserito lo stub dell'oggetto remoto) e aggiungendo informazioni all'oggetto.

Cosa specializza di ObjectOutputStream lo stream di Marshal?

Modifica tre metodi:

- Il metodo *replaceObject()* che può definire un metodo alternativo per serializzare un oggetto sullo stream.
- Il metodo *enableReplaceObject()* che restituisce un booleano e stabilisce se la istanza deve oppure no specializzare il meccanismo di serializzazione, usando il metodo replaceObject().
- Il metodo *annotateClass(),* che permette di inserire informazioni addizionali sulla classe, viene usato per specificare il codebase e permettere quindi il caricamento dinamico. Nello stream si può far viaggiare un'annotazione che indica qual è il server web da cui si può caricare la risorsa.

A cosa serve rmic?

È uno stub compiler che eseguito sul file .class del nostro server, genera lo stub e lo skeleton.

PARTE EE

Quali vantaggi si ottengono sfruttando i containers di Java EE?

Java EE è un insieme di specifiche implementate da diversi container. I container sono ambienti runtime di java EE che forniscono servizi ai componenti che ospitano come ad esempio la gestione del ciclo di vita, dependency injection, concorrenza ecc. Questi componenti usano contratti ben definiti per comunicare con l'infrastruttura di java EE e con gli altri componenti

CONTAINER

L'infrastruttura di java EE è partizionata in domini logici chiamati container. Ogni container ha un ruolo specifico, supporta un insieme di API, e offre servizi ai componenti (sicurezza, accesso a database, gestione delle transazioni, naming directory, resource injection). I container nascondono la complessità tecnica e aumentano la portabilità. In base al tipo di applicazione si vuole creare, si dovranno capire le capacità e i vincoli di ogni container in modo da usarne uno o più. Java EE possiede quattro container differenti:

- Applet container sono forniti dalla maggior parte dei web browser per eseguire componenti di applet. Quando si sviluppano applet, ci si può concentrare sull'aspetto visivo dell'applicazione mentre il container fornisce un ambiente sicuro. L' applet container utilizza una sandbox dove il codice eseguito all'interno della sandbox non può essere eseguito al di fuori della sandbox.
- L'application client container (ACC) include un insieme di classi Java, librerie, e altri file richiesti per portare injection, security management e naming service alle applicazioni di Java SE.
- Il web container fornisce i servizi per la gestione e l'esecuzione delle componenti web (servlet, EJBs, filters, listeners, pagine JSF, web services). È responsabile della creazione di un'istanza, inizializzazione e invocazione di servlet e supporto dei protocolli HTTP e HTTPS.
- Il container EJB è responsabile della gestione dell'esecuzione di enterprise bean (session bean e message driven bean) contenente la logica di business delle applicazioni di java EE. Crea nuove istanze di EJBs, gestisce il loro ciclo di vita, e fornisce servizi come transazione, sicurezza, concorrenza, naming service, o la possibilità di essere invocati asincronamente.

Annotazioni e Deployment Descriptor: quali sono i vantaggi/svantaggi di entrambi gli approcci?

Il più grande vantaggio delle annotazioni è che riducono significativamente la quantità di codice che uno sviluppatore deve scrivere, e usando le annotazioni si può evitare il bisogno dei deployment descriptors. D'altra parte, i deployment descriptors sono file XML esterni che possono essere modificati senza dover modificare il codice sorgente e ricompilare. Se si usano entrambi, i metadata sono sovrascritti dai deployment descriptor quando l'applicazione o il componente viene deployato.

Qual è l'idea alla base del design pattern inversion of control?

L'idea è che il container prende il controllo delle nostre applicazioni e fornisce servizi tecnici (come transazioni o security management). Prendere il controllo significa gestire il ciclo di vita delle componenti, portando dependency injection e configurazioni alle nostre componenti. Il container prende in consegna il lavoro partendo dal contesto di esecuzione fino ad analizzare tutte le dipendenze necessarie.

Quali sono i vantaggi del "loose coupling, strong typing"?

Gli interceptor sono un modo molto potente per disaccoppiare preoccupazioni tecniche dalla logica di business. La gestione del ciclo di vita contestuale disaccoppia i bean dal gestire il proprio ciclo di vita. Con l' injection un bean non è consapevole dell'implementazione concreta di qualsiasi bean con cui sta interagendo. I bean possono usare notifiche di eventi per disaccoppiare i produttori dai consumatori o decorator per disaccoppiare preoccupazioni di business. In altre parole, loose coupling è il DNA su cui CDI è

stato costruito. E tutte queste agevolazioni sono distribuite in maniera typesafe. CDI non si basa mai su identificatori String-based per determinare come gli oggetti si adattano insieme. Invece, CDI usa annotazioni fortemente tipizzate (strongly typed) per legare i bean insieme. L'uso dei descriptor XML è minimizzato.

In che modo il ciclo di vita di un bean differisce da quello di un POJO?

Il ciclo di vita di un POJO è molto semplice: si crea una istanza di una classe usando la keyword new e si aspetta che il Garbage Collector se ne liberi per liberare memoria. Ma per mandare in esecuzione un CDI bean all'interno di un container non è permesso utilizzare la keyword new. Invece bisogna iniettare il bean e il container fa il resto, ovvero il container è l'unico responsabile della gestione del ciclo di vita del bean: esso crea l'istanza; esso si occupa di liberarsene. Allora come si fa ad inizializzare un bean senza chiamare un costruttore? Quando si inietta un bean il container è l'unico responsabile della creazione dell'istanza. Poi risolve le dipendenze e invoca i metodi annotati con @PostConstruct prima che il primo metodo di business venga invocato sul bean. Poi la notifica callback @PreDestroy segnala che l'istanza è in procinto di essere rimossa dal container

Quali sono i vantaggi derivanti dall'uso degli Interceptor?

Gli interceptor vengono utilizzati per interporsi tra le invocazioni di metodi di business. In questo aspetto è simile al aspect-oriented programming (AOP). AOP è un paradigma di programmazione che separa aspetti trasversali dal codice di business. La maggior parte delle applicazioni hanno in comune codice che viene ripetuto attraverso le componenti. Questo codice potrebbe riferirsi ad aspetti tecnici (loggare l'entrata e l'uscita di un metodo) o ad aspetti di business. Questi aspetti possono essere applicati automaticamente attraverso AOP all'intera applicazione. I Managed Bean supportano funzionalità simile ad AOP che forniscono l'abilità di intercettare invocazioni di metodi attraverso interceptor. Gli interceptor sono innescati automaticamente dal container quando un Managed Bean viene invocato. Gli interceptor possono essere concatenati e chiamati prima o dopo l'esecuzione di un metodo.

In che modo è possibile definire una sorta di priorità nell'esecuzione di una catena di Interceptor?

Utilizzando l'annotazione @Priority seguita da un valore che rappresenta la priorità che vogliamo assegnare all'interceptor. @Priority prende un intero che può essere qualsiasi valore. La regola è che l'interceptor con il valore di priorità più basso viene chiamato prima.

In che modo è possibile realizzare disaccoppiamento nelle applicazioni Java enterprise?

Attraverso l'utilizzo dell'annotazione @Inject degli oggetti di cui abbiamo bisogno. Visto che Java EE è un ambiente gestito non c'è il bisogno di costruire dipendenze a mano ma si lascia che il container inietti un riferimento per noi.

Qual è il meccanismo che permette di scegliere fra due diverse implementazioni di uno specifico bean?

Tramite l'utilizzo dell'annotazione @Qualifier. Sono annotazioni java che portano typesafe injection e disambiguano un tipo senza dover ripiegare sui nomi String-based. Un qualifier rappresenta della semantica associata con un tipo che è soddisfatto da una qualche implementazione di quel tipo. È un' annotazione user-definied, annotata con @Qualifier.

Perchè è stato introdotto il concetto di Interceptor Binding?

Perché lega la class interceptor al bean senza dipendenza diretta tra le due classi, ovvero riduce l'accoppiamento.

Qual è il vantaggio derivante dall'uso dei Decorator?

L'idea è prendere una classe e avvolgere un'altra classe attorno ad essa. In questo modo quando viene chiamata una classe decorator, si passa sempre attraverso il decorator circostante prima di raggiungere la classe target. I decorator sono utilizzati per aggiungere logica addizionale ai metodi di business. Gli interceptor non sono consapevoli dell'attuale semantica degli eventi che intercettano. Un decorator intercetta le invocazioni solamente per una certa interfaccia Java, e quindi è consapevole della semantica legata a questa.

Qual è il vantaggio derivante dall'uso degli Eventi?

Gli event permettono ai bean di interagire a tempo di compilazione senza dipendenza. Un bean può definire un evento, un altro bean può lanciare l'evento e un altro bean ancora può maneggiare l'evento. I bean possono essere in package differenti dell'applicazione. Questo schema segue il design pattern observer. L'event producer lancia gli eventi usando l'interfaccia Event. Un producer lancia eventi chiamando il metodo fire(), passando l'oggetto event, senza dipendere dall'observer. Gli eventi sono lanciati da un event producer e sottoscritti da eventi observer. Un observer è un bean con uno o più metodi observer.

Qual è la differenza fra una entità ed un oggetto?

Per essere un'entità una classe deve seguire le seguenti regole:

- La classe deve essere annotata con **@Entity**.
- L'annotazione **@Id** deve essere utilizzata per denotare una semplice chiave primaria
- La classe deve avere un metodo costruttore senza argomenti che deve essere public o protected. La classe può avere comunque altri costruttori.
- La classe deve essere una classe top-level. Una enum o un'interfaccia non possono essere designate come entità.
- La classe non deve essere final. I metodi e le variabili di istanza non possono essere final.
- Se un'istanza dell'entità deve essere passata per valore (invocazione remota RMI), la classe entity deve implementare l'interfaccia *Serializable*.

A cosa serve l'annotazione @GeneratedValue?

L'annotazione @GeneratedValue informa il persistence provider che deve autogenerare l'identificatore della classe entity.

Qual è l'elemento discriminante per distinguere una entità da un POJO?

L'annotazione @Entity

Qual è l'API fondamentale per la gestione delle operazioni sulle entità?

Java persistence API (JPA).

Java Persistence API has two sides. The first is the ability to map objects to a relational database. Configuration by exception allows persistence providers to do most of the work without much code, but the richness of JPA also allows customized mapping from objects to tables using either annotation or XML descriptors. From simple mapping (changing the name of a column) to more complex mapping (inheritance), JPA offers a wide spectrum of customizations. As a result, you can map almost any object model to a legacy database. The other aspect of JPA is the ability to query these mapped objects. In JPA, the centralized service to manipulate instances of entities is the entity manager. It provides an API to create, find, remove, and synchronize objects with the database. It also allows the execution of different sorts of JPQL queries, such as dynamic, static, or native queries, against entities. Locking mechanisms are also possible with the entity manager.

Quali sono le caratteristiche e le funzionalità più importanti della persistence unit?

Quale driver JDBC si dovrebbe usare? Come ci si connette al database? Qual è il nome del database? Quando la classe Main crea un EntityManager passa come parametro il nome di una persistence unit. La persistence unit indica all' entity manager il tipo di database da usare e i parametri di connessione, che sono definiti nel file persistence.xml, che devono essere acceduti. Un esempio d persistence unit definisce una connessione JDBC, specificando l'host, la porta e username e password.

Descrivere il ciclo di vita di una entità

Le entità sono POJO. Quando l'entity manager gestisce i POJO, questi hanno un persistence identity (una chiave che identifica univocamente un'istanza, simile ad una chiave primaria), e il database sincronizza il loro stato. Quando non sono gestite (ovvero sono distaccate dall'entity manager), possono essere usate come una qualsiasi altra classe java. Le entità hanno un ciclo di vita. Quando viene creata un'istanza di un oggetto entity con l'operatore new, l'oggetto esiste in memoria e il JPA non è a conoscenza dell'oggetto creato. Quando l'oggetto viene gestito dall'entity manager la tabella dell'oggetto mappa e sincronizza il suo stato, e quindi ora il JPA è a conoscenza dell'oggetto. Chiamando il metodo EntityManager.remove() si cancella il dato dal database, ma l'oggetto Java continua a vivere in memoria finché non viene eliminato dalla garbage collector.

Descrivere i tipi di relazioni in un database relazionale

Prima di tutto una relazione ha una direzione. Può essere unidirezionale (cioè un oggetto può navigare verso un altro) o bidirezionale (un oggetto può navigare verso un altro e viceversa). Per quanto riguarda le cardinalità abbiamo:

- Uno a uno
- Uno a molti
- Molti a uno
- Molti a molti

Definizione e funzionalità di un Persistence Context

Un Persistence Context è un insieme di istanze di entità gestite ad un certo tempo per una certa transazione dell'utente: solo un'istanza di un'entità con lo stesso persistent identity può esistere in un persistence context. Ad esempio, se una istanza Book con id 12 esiste nel persistence context, nessun altro book con questo ID può esistere all'interno dello stesso persistence context. Solo le entità che sono contenute nel persistence context sono gestite dall'entity manager, questo significa che qualsiasi modifica si verterà nel database. L'entity manager aggiorna o consulta il persistence context ogni volta che un metodo dell'interfaccia EntityManager viene chiamato. Ad esempio quando viene chiamato il metodo persist(), l'entity passata come argomento sarà aggiunta al persistence context se non esiste già. Ugualmente, quando un'entità viene cercata tramite la sua chiave primaria, l'entity manager controlla prima se l'entity richiesta è già presente nel persistence context. Il persistence context può essere visto come una cache di primo livello

Descrivere i vari tipi di query definiti da JPQL

- Dynamic queries: questa è la forma più semplice di query, che consiste in una query JPQL specificata dinamicamente a runtime.
- Named queries: sono statiche e non modificabili.
- Criteria API: JPA ha introdotto il concetto di query object-oriented.

- Native queries: questo tipo di query è utile per eseguire istruzioni SQL native invece di un'istruzione JPQL.
- Stored procedures query: si chiamano procedure memorizzate

Oltre alle invocazioni remote RMI, è possibile invocare beans in maniera alternativa?

Si, ci sono altri due modi: utilizzando l'annotazione @EJB, utilizzando CDI tramite l'annotazione @Inject.

Qual è il vantaggio dell'utilizzo di JNDI?

JNDI è maggiormente utilizzato per l'accesso remoto quando un client non è gestito dal container e non può utilizzare le dependency injection. Ma JNDI può essere anche utilizzato da client locali, anche se dependency injection risulta più semplice. L' injection è fatta a tempo di deploy. Se c'è una possibilità che i dati non saranno utilizzati, il bean può evitare il costo di risorse utilizzando una lookup JNDI. JNDI è un'alternativa all'injection; attraverso JNDI, il codice utilizza i dati solo se ne ha bisogno, invece di accettare il push di dati che potrebbero non servire.

Quali sono le differenze tra i tre differenti tipi di beans in termini di chiamate dai clients?

- **Stateless**: Il session bean non mantiene lo stato di conversazione tra metodi, e nessuna istanza può essere usata per nessun cliente. È usato per gestire task che possono essere conclusi con una semplice chiamata a un metodo.
- **Stateful**: Il session bean mantiene lo stato di conversazione, che deve essere mantenuto attraverso i metodi per un singolo utente. È utile per task che devono essere completati in vari passi.
- **Singleton**: Un singolo session bean è condiviso tra i clienti e supporta l'accesso concorrente. Il container farà in modo che esista una sola istanza per l'intera applicazione. Un' esempio è quello di fornire una cache disponibile a tutti i clienti.

Qual è il caso d'uso più comune per un singleton bean?

Fornire una cache disponibile a tutti i clienti, oppure popolare un database

Qual è il vantaggio dell'uso del: "Programmatic Authorization"?

Declarative authorization coprono la maggior parte dei casi di sicurezza di cui un'applicazione ha bisogno. Ma a volte c'è il bisogno di dettagliare l'autorizzazione (permettere l'accesso a un blocco di codice invece che l'intero metodo, permettere di negare l'accesso a un singolo individuo invece che a un ruolo dell'utente). Si può utilizzare la programmatic authorization per permettere o bloccare selettivamente l'accesso a un ruolo o a una parte. Questo perché si ha accesso diretto all'interfaccia Principal di JAAS, come lo ha anche il contesto EJB per controllare il ruolo dell'utente nel codice.

Quali sono le principali differenze fra Container-Managed e Bean-managed transactions?

Con le transazioni Container-Managed si delega la politica di demarcazione al container. Non si deve utilizzare esplicitamente JTA nel codice; si lascia che il container marchi i confini della transazione automaticamente tramite le operazioni di begin e commit che si basano sui metadati. Il container EJB fornisce servizi di gestione delle transazioni ai session bean e MDB. In un bean enterprise con transazioni container-managed, il container setta i confini delle transazioni.

In alcuni casi le transazioni CMT non forniscono il grado di demarcazione richiesta (un metodo non può generare più di una transazione). Per risolvere questo problema, EJB offre un modo programmatico per gestire le transazioni con BMT. BMT permette di gestire esplicitamente i confini di transazione (begin, commit, rollback) usando JTA

Quali sono le principali differenze fra il modello Point-to-Point ed il modello Publish-Subscribe?

Point-to-point: in questo modello la destinazione usata per tenere i messaggi è chiamata coda (queue). Quando si utilizza il modello point-to-point, un client invia un messaggio in una coda e un altro client riceve il messaggio. Una volta che il messaggio viene riconosciuto, il message provider rimuove il messaggio dalla coda.

Publish-subscribe: in questo modello la destinazione è chiamata topic. Quando si utilizza questo modello un client pubblica un messaggio in un topic e tutti gli iscritti a quel topic ricevono il messaggio.

Cosa sono i Message-Driven Beans?

Sono message consumer asincroni che vengono eseguiti all'interno di un EJB container. Sono stateless, quindi il container EJB può avere numerose istanze, eseguendole in modo concorrente, per processare messaggi in arrivo da vari producer. Anche se sono stateless bean, le applicazioni client non possono accedere gli MDB direttamente; l'unico modo per comunicare con un MDB è inviare un messaggio alla destinazione della quale l'MDB è in ascolto. In generale, gli MDB sono in ascolto di una destinazione e quando arriva un messaggio, lo processano. Possono anche delegare logica di business ad altri stateless session bean in modo sicuro e transazionale. Visto che sono stateless gli MDB non mantengono lo stato attraverso invocazioni separate da un messaggio ricevuto ad un altro. Gli MDB rispondono ai messaggi ricevuti dal container, mentre i stateless session bean rispondono alle richieste del client attraverso un'interfaccia appropriata.

Cosa sono gli Administered Objects?

Gli administered objects sono oggetti che sono configurati amministrativamente. Il message provider permette a questi oggetti di essere configurati e li rende disponibili nel namespace di JNDI. Come i JDBC datasources, gli administered objects vengono creati una sola volta. I due tipi di oggetti amministrati sono:

- Connection factories: usata dal client per creare una connessione ad una destinazione
- Destinations: punti di distribuzione che ricevono, mantengono e distribuiscono messaggi. Le destinazioni possono essere queue o topic.

Il client accede questi oggetti attraverso le interfacce portabili cercandole nel namespace JNDI o attraverso l'injection.

Qual è la differenza fra Synchronous Delivery e Asynchronous Delivery?

Synchronous: un consumer esplicitamente prende il messaggio dalla destinazione chiamando il metodo receive().

Asynchronous: un consumer decide di registrarsi a un evento che viene innescato quando arriva un messaggio. Il consumer deve implementare l'interfaccia MessageListener e ogni volta che arriva un messaggio il provider lo consegna chiamando il metodo onMessage().

Quali sono i principali meccanismi di affidabilità?

- Filtrare i messaggi: usando selettori si possono filtrare i messaggi che si vogliono ricever.
- Settare il tempo di vita del messaggio: viene settata un tempo di scadenza ai messaggi così che non possono essere consegnati se sono obsoleti.
- Specificare la persistenza dei messaggi: specifica che i messaggi sono persistenti nel caso di un guasto del provider.
- Controllare il riconoscimento: specifica vari livelli di riconoscimento dei messaggi.

- Creare iscritti duraturi:
- Settare priorità: setta la priorità di consegna di un messaggio.

Quali eccezioni si possono gestire con gli MDB?

- Application exceptions: sono eccezioni controllate che estendono Exception e non causano il roll back del container
- System exceptions: sono eccezioni non controllate che estendono RuntimeException e causano il roll back del container.

Il lancio di un JMSRuntimeException causerà il roll back del container, ma il lancio di un JMSException no. Se c'è bisogno di un rollback si deve chiamare esplicitamente il metodo setRollBackOnly().

Quali sono i requisiti di un Web Service?

- La classe deve essere annotata con @WebService o l'XML equivalente in un descriptor deployment.
- La classe può implementare nessuna o più interfacce che devono essere annotate con @WebService
- La classe deve essere public e non deve essere final o astratta
- La classe deve avere un costruttore public di default
- La classe non deve definire il metodo finalize().
- Per trasformare un SOAP web services in un EJB endpoint, la classe deve essere annotata con @Stateless o @Singleton
- Un service deve essere un oggetto stateless e non dovrebbe salvare lo stato del client attraverso i metodi chiamati.

Cosa rappresenta il file WSDL?

WSDL (Web Services Description Language) è il linguaggio di definizione dell'interfaccia che definisce l'interazione tra i consumer e i SOAP web service. È fondamentale per un SOAP web service dato che descrive il tipo di messaggio, la porta, il protocollo di comunicazione, le operazioni supportate, la locazione e cosa il consumer si aspetta di ricevere. Definisce il contratto a cui il service garantisce di adeguarsi. Si può pensare al WSDL come a un'interfaccia Java ma scritta in XML.

Quando UDDI risulta opzionale?

Quando si conosce l'esatta posizione del web service che si vuole invocare.

Consumers and providers that interact with one another over the Web need to be able to find information that allows them to interconnect. Either the consumer knows the exact location of the service it wants to invoke or it has to find it. UDDI provides a standard approach to locating information about a web service and how to invoke it. The service provider publishes a WSDL into a UDDI registry available on the Internet. Then it can be discovered and downloaded by potential consumers. It is optional as you can invoke a web service without UDDI if you already know the web service's location. UDDI is an XML-based registry of web services, similar to a Yellow Pages directory, where businesses can register their services. This registration includes the business type, geographical location, web site, phone number, and so on. Other businesses can then search the registry and discover information about specific web services. This information provides additional metadata about the service, describing its behavior and the actual location of the WSDL document.

Il ruolo del proxy nel meccanismo di invocazione di un Web Service