Sommario

Lezione 1, Prologo ai sistemi distribuiti: [15](#_heading=h.30j0zll)

Middleware: è uno strato software che si occupa di connettere le macchine fra di loro in modo da far percepire all’utente il sistema come un'unica entità integrata. [15](#_heading=h.1fob9te)

Motivazioni dietro i sistemi distribuiti: [15](#_heading=h.3znysh7)

Legge di Moore [15](#_heading=h.2et92p0)

Leggi per le reti [15](#_heading=h.tyjcwt)

● Legge di Sarnoff: il valore di una rete di broadcast è direttamente proporzionale al numero degli utenti connessi. [15](#_heading=h.3dy6vkm)

● Legge di Metcafe: il valore di una rete di comunicazione è direttamente proporzionale al quadrato del numero di utenti. [15](#_heading=h.1t3h5sf)

● Legge di Reed: il valore di una rete sociale è direttamente proporzionale ad una funzione esponenziale in N. [15](#_heading=h.4d34og8)

● Reti di tipo Broadcast (Sarnoff): per le reti di questo tipo il vantaggio è legato semplicemente all’numero di utenti a cui è possibile vendere il servizio, reti più grandi permettono un bacino di utenza maggiore. [15](#_heading=h.2s8eyo1)

● Reti di comunicazioni e sociale (Metcafe e Reed): per reti di questo tipo il vantaggio invece è molto maggiore, infatti accorpando più reti il numero di comunicazioni che è possibile effettuare aumenta in modo notevole. [15](#_heading=h.17dp8vu)

Lezione 2, The Reference Model of Open Distributed Processing (RM-ODP) : [15](#_heading=h.3rdcrjn)

Caratteristiche di un sistema distribuito (Altissima probabilità di domanda): [15](#_heading=h.26in1rg)

● **Remoto**: Le componenti di un sistema distribuito devono poter essere locali o remote, quindi anche potenzialmente localizzate su macchine diverse. [15](#_heading=h.lnxbz9)

● **Concorrenza**: Un sistema distribuito è per sua stessa natura concorrente, in quanto la contemporanea esecuzione di due (o più) istruzioni è possibile su macchine diverse e non esistono strumenti come lock e semafori. [15](#_heading=h.35nkun2)

● **Assenza di uno stato globale**: Non esiste un modo per poter determinare lo stato globale del sistema, in quanto la distanza e l’eterogeneità (diversi clock) del sistema non permette di definire con certezza lo stato in cui si trova ciascun nodo. La mancanza di un clock globale rende impossibile riuscire a sincronizzare perfettamente gli orologi di tutti i processi e questo comporta l’impossibilità di ordinare, in modo preciso e univoco, tutti gli eventi che occorrono all’interno del sistema. [15](#_heading=h.1ksv4uv)

● **Malfunzionamenti parziali**: Ogni componente di un sistema distribuito può smettere di funzionare in maniera indipendente dalle altre componenti e questo non inficia le funzionalità che sono localizzate su altre macchine. [16](#_heading=h.44sinio)

● **Eterogeneità**: Un sistema distribuito per sua stessa definizione è eterogeneo (formato da parti diverse) sia del punto di vista hardware che software. [16](#_heading=h.2jxsxqh)

● **Autonomia**: Un sistema distribuito non ha un singolo punto dal quale esso puo’ essere controllato, coordinato e gestito. Quindi, la collaborazione va ottenuta mediando le richieste del sistema distribuito con quelle del sistema che gestisce ciascun nodo. [16](#_heading=h.z337ya)

● **Evoluzione**: un sistema distribuito può cambiare anche in modo sostanziale durante la sua vita, sia per motivi “ambientali” che tecnologici. La flessibilità di un sistema distribuito deve assicurare che la migrazione verso ambienti, tecnologie differenti e applicazioni nuove puo’ essere effettuata con successo e senza costi eccessivi. [16](#_heading=h.3j2qqm3)

● **Mobilità**: Così come appare naturale che gli utenti siano mobili, altrettanto naturale deve essere la mobilità dei nodi e delle risorse (ad esempio, dati) all’interno del sistema in modo da poter adattare al meglio le prestazioni del sistema (il famoso Akamai). [16](#_heading=h.1y810tw)

Requisiti non funzionali di un sistema distribuito (Altissima probabilità di domanda): [16](#_heading=h.4i7ojhp)

● **Aperti**: in modo da supportare la portabilità di esecuzione e l’interoperabilità (capacità di cooperare con altri sistemi) attraverso interfacce e servizi ben documentati e aderenti a standard noti e riconosciuti. In modo da permettere l’evoluzione e la possibilità di cambiare fornitore senza particolari rischi per la intera architettura. [16](#_heading=h.2xcytpi)

● **Integrati**: così da incorporare al proprio interno sistemi e risorse differenti senza dover utilizzare strumenti ad-hoc. Questo permette di trattare in maniera efficiente (economica) il problema della eterogeneità hardware e software. [16](#_heading=h.1ci93xb)

● **Flessibili**: Ovvero essere in grado di evolversi e avere la capacità di integrare sistemi legacy al loro interno. [16](#_heading=h.3whwml4)

● **Modulari**: in modo da permettere ad ogni componente di poter essere autonoma e indipendente verso il resto del sistema. [16](#_heading=h.2bn6wsx)

● **Supportino la federazione di sistemi**: in modo da unire diversi sistemi, dal punto di vista amministrativo oltre che architetturale, per lavorare e fornire servizi in maniera congiunta. [16](#_heading=h.qsh70q)

● **Facilmente gestibili**: in modo da permettere il controllo, la gestione e la manutenzione per configurarne i servizi, la loro qualità (Quality of Service) e le politiche di accesso. [16](#_heading=h.3as4poj)

● **Supporto per la qualità del servizio (Quality of Service):** per poter fornire i servizi con vincoli di tempo, di disponibilità e di affidabilità, anche in presenza di malfunzionamenti parziali. [16](#_heading=h.1pxezwc)

● **Scalabili**: perché qualsiasi sistema distribuito accessibile da Internet puo` essere soggetto a picchi di carico non prevedibili e deve essere in grado di gestirli. [16](#_heading=h.49x2ik5)

● **Sicuri**: utenti non autorizzati non possono accedere a dati sensibili. [16](#_heading=h.2p2csry)

● **Trasparenti**: mascherando i dettagli e le differenze della architettura sottostante che assicura la distribuzione dei servizi sulle componenti del sistema. [16](#_heading=h.147n2zr)

La trasparenza di un sistema distribuito [16](#_heading=h.3o7alnk)

● **Livello Base** [17](#_heading=h.23ckvvd)

o **Trasparenza di Accesso**: nasconde le differenze nella rappresentazione dei dati e nel meccanismo di invocazione per permettere la interoperabilità tra oggetti. Gli oggetti devono essere accessibili attraverso la stessa interfaccia, sia che siano acceduti da locale sia che siano acceduti da remoto. [17](#_heading=h.ihv636)

o **Trasparenza di Locazione**: non permette di utilizzare informazioni circa la localizzazione nel sistema (IP fisico) 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 (praticamente il DNS). [17](#_heading=h.32hioqz)

● **Livello di Funzionalità**: [17](#_heading=h.1hmsyys)

o **Trasparenza di Migrazione (locazione/accesso):** Nasconde la possibilità del sistema di spostare oggetti o dati da un nodo all’altro della rete, continuando ad essere raggiungibile e utilizzabile da altri oggetti. In modo da ottimizzare le prestazioni (AKAMI e le CDL Content Delivery Network). [17](#_heading=h.41mghml)

o **Trasparenza di Replica:** Con questo tipo di trasparenza, il sistema maschera il fatto che una singola componente viene replicata in 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. Le repliche vengono utilizzate per ragioni di: [17](#_heading=h.2grqrue)

o **Trasparenza alla Persistenza (Trasparenza/Locazione):** Questo tipo di trasparenza scherma l’utente dalle operazioni che compie il sistema per rendere persistente (cioè in memoria secondaria) un oggetto durante una fase di non utilizzo. [17](#_heading=h.vx1227)

o **Trasparenza alle Transazioni:** nasconde all’utente le attività di coordinamento che vengono svolte per assicurare la consistenza dello stato degli oggetti in presenza della concorrenza e nella gestione delle transazioni (ACID). [17](#_heading=h.3fwokq0)

● **Livello di Efficienza:** Scalabilità, prestazioni e malfunzionamenti. [17](#_heading=h.1v1yuxt)

o **Trasparenza di Scalabilità (migrazione/replica):** assicura l’operabilità del sistema in caso di picchi di carico, senza dover modificare l’architettura e l’organizzazione. (differenza tra scalabilità verticale e orizzontale). Tale trasparenza è basata sulla trasparenza di replica e migrazione. [17](#_heading=h.4f1mdlm)

o **Trasparenza alle Prestazioni:** Progettisti/sviluppatori ottengono alte prestazioni senza conosce i meccanismi utilizzati. Tale trasparenza viene ottenuta tramite il: [17](#_heading=h.2u6wntf)

o **Trasparenza ai malfunzionamenti:** in presenza di malfunzionamenti parziali il resto del sistema continua a funzionare, tale trasparenza è basata sulla trasparenza di replica ma anche sulla Trasparenza alle transazioni (facendo il rollback di eventuali transazioni iniziate ma non finite). [17](#_heading=h.19c6y18)

Il Middleware: (Lo chiede in modo veloce) [19](#_heading=h.3tbugp1)

● Middleware di infrastruttura: si occupa di comunicazione tra sistemi operativi diversi e gestione di concorrenza in maniere da essere portabile (per esempio la JVM). [19](#_heading=h.28h4qwu)

● Middleware di distribuzione: automatizza compiti comuni per la comunicazione come: [19](#_heading=h.nmf14n)

o **Marshalling**: invio di parametri per l’invocazione di servizi in remote. [19](#_heading=h.37m2jsg)

o **Multiplexing**: utilizzo dello stesso canale per più invocazioni/richieste. [19](#_heading=h.1mrcu09)

o **Gestione della semantica delle invocazioni**: unicast, multicast, attivazione on-demand (invocazioni a più oggetti). [19](#_heading=h.46r0co2)

o **Riconoscimento e gestione dei malfunzionamenti di rete**. [19](#_heading=h.2lwamvv)

● **Middleware per servizi comuni:** Implementa in automatico servizi comuni, come persistenza, transazioni, sicurezza, accesso, etc… (**accesso tramite Google per esempio).** [19](#_heading=h.111kx3o)

L’evoluzione del Middleware: [19](#_heading=h.3l18frh)

Passaggio tra RPC e Modello ad Oggetti distribuiti: [19](#_heading=h.206ipza)

CORBA (Common Object Request Broker Application): (Lo chiede) [19](#_heading=h.4k668n3)

Concetto di middleware implicito [20](#_heading=h.2zbgiuw)

Lezione 3, Programmazione concorrente e Thread in Java [20](#_heading=h.1egqt2p)

Motivazioni [20](#_heading=h.3ygebqi)

Le sfide [20](#_heading=h.2dlolyb)

Cosa ci riserva il futuro [20](#_heading=h.sqyw64)

Diversi tipo di programmazione concorrente [20](#_heading=h.3cqmetx)

● **Eseguita su calcolatori diversi** [20](#_heading=h.1rvwp1q)

● **Processi concorrenti sulla stessa macchina (multitasking)** [20](#_heading=h.4bvk7pj)

● **Programmazione concorrente nello stesso processo** (“processi lightweight” all’interno del processo: Thread): strumento efficace che permette di costruire programmi che eseguono più operazioni contemporaneamente. [20](#_heading=h.2r0uhxc)

Apache [20](#_heading=h.1664s55)

I Thread: (Lo chiede in modo veloce) [21](#_heading=h.3q5sasy)

Alcuni metodi della classe Thread [21](#_heading=h.25b2l0r)

Diagramma degli stati di un thread [22](#_heading=h.kgcv8k)

Problemi dei thread [22](#_heading=h.34g0dwd)

● **L’interferenza fra thread:** si ha quando più thread lavorano su una stessa variabile ed eseguendo le loro operazioni contemporaneamente si interfogliano. [22](#_heading=h.1jlao46)

● **Inconsistenza della memoria:** si ha quando thread diversi hanno visioni diverse di dei dati presenti in memoria. Tale problema può essere risolto creando una relazione happens-before (succede-prima, anche questo si ottiene con la sincronizzazione). [22](#_heading=h.43ky6rz)

● **Deadlock**: Quando due thread si attendono a vicenda per un tempo infinito senza fare nulla. Questo problema in Java si risolve tramite l’uso del metodo bow (inchino). [22](#_heading=h.2iq8gzs)

● **Livelock**: Quando due thread si attendono a vicenda compiendo in continuazione azioni che non portano a nulla. [22](#_heading=h.xvir7l)

● **Starvation**: Quando un thread in coda attende per un tempo infinito il suo turno, senza che gli venga mai concesso. [23](#_heading=h.3hv69ve)

● **Variabile volatile:** è una variabile utilizzata per rendere l’accesso ad una variabile asincrono in modo da risolvere il problema dell’inconsistenza della memoria (operazione atomica), ossia quando un thread modifica una variabile, tutti gli altri thread vedono immediatamente il nuovo valore. [23](#_heading=h.1x0gk37)

Lezione 4, Programmazione concorrente e Thread parte 2: [23](#_heading=h.4h042r0)

La legge di Amdahl [23](#_heading=h.2w5ecyt)

Lo Speedrun [23](#_heading=h.1baon6m)

Gli strumenti per la sincronizzazione dei thread sono un trade-off (o uno o l’altro) tra: [23](#_heading=h.3vac5uf)

Metodi sincronizzati [23](#_heading=h.2afmg28)

Problema del leaking [23](#_heading=h.pkwqa1)

Lock Intriseco [23](#_heading=h.39kk8xu)

Sincronizzazione dei metodi statici [23](#_heading=h.1opuj5n)

Lezione 5, Singleton e DoubleChecked Locking: [24](#_heading=h.48pi1tg)

Il Singleton: (Lo chiede in modo veloce) [24](#_heading=h.2nusc19)

● **Usare le synchronized:** questo metodo funziona solo in parte, eseguendo il metodo getInstance() sincronizzato il problema si risolve solo per una volta, successivamente tutti i thread possono accedervi contemporaneamente. [24](#_heading=h.1302m92)

● **Double e-checking lock**: Un'altra soluzione sta nell’effettuare un controllo sulla variabile di istanza per vedere se è stata inizializzata anche se così non è detto che funzioni al 100% perché la creazione del singleton può non essere ancora stata completata prima che un altro thread utilizzi il riferimento. [24](#_heading=h.3mzq4wv)

● **La soluzione definitiva** sta nel rendere la variabile **Instance,** **volatile.** Altrimenti, si possono usare le classi statiche con l’idioma **“Initialization-on-demand holder”** che essendo un metodo statico, viene eseguito una sola volta (al caricamento) e stabilisce una relazione **happens-before**. [24](#_heading=h.2250f4o)

Lezione 7, Socket TCP [24](#_heading=h.haapch)

I Socket [24](#_heading=h.319y80a)

● **Socket:** socket standard per comunicazioni. [24](#_heading=h.1gf8i83)

● **Server-socket:** svolge il ruolo appunto di server, istanziando un socket privato per ogni nuova richiesta da parte di un client. [24](#_heading=h.40ew0vw)

Gli Stream [25](#_heading=h.2fk6b3p)

Metodi della classe socket [25](#_heading=h.upglbi)

● **accept():** invocato su oggetti di tipo ServerSocket è un metodo bloccante, l’esecuzione si blocca fino a quando non si collega un client, a questo punto il metodo restituisce un socket che non è altro del socket di comunicazione (classe Socket). [25](#_heading=h.3ep43zb)

● **close():** chiude la connessione e il socket. [25](#_heading=h.1tuee74)

● **setSocketTimeout():** se entro un certo numero di millisecondi il server non riceve accept chiude le comunicazioni. [25](#_heading=h.4du1wux)

Lezione 8, Socket TCP 2 (with Curly): [25](#_heading=h.2szc72q)

Conversione da un oggetto locale ad un oggetto remoto: [25](#_heading=h.184mhaj)

● **Stub**: è un oggetto che si trova nel client è che svolge il compito di rappresentare il server per quest’ultimo, esso infatti espone gli stessi metodi implementati dal server in modo che il client li possa usare. [25](#_heading=h.3s49zyc)

● **Skeleton** (o client stub): è un oggetto che si trova nel server con lo scopo di comunicare a quest’ultimo le richieste di metodi che gli giungono dallo stub e di ritrasmettere i valori di ritorno provenienti dal server a quest’ultimo in modo che lo stub li possa a sua volta trasmettere al client. [25](#_heading=h.279ka65)

Problema dell’indirizzamento [25](#_heading=h.meukdy)

Lezione 10/11, java RMI (Java Remote Method Invocation): [25](#_heading=h.36ei31r)

Java RMI è costituito da 5 package: [26](#_heading=h.1ljsd9k)

Interfacce ed eccezioni remote [26](#_heading=h.45jfvxd)

● **Dichiarare l’eccezione java.rmi.RemoteException**: è una Checked-Exception (viene controllata dal compilatore e vista come un errore se non c’è). Questo con lo scopo di rendere il programmatore cosciente che si sta andando a lavorare con un oggetto remoto che ha tutta una serie di problematiche diverse rispetto ad un oggetto locale. [26](#_heading=h.2koq656)

● **Parametri dichiarati attraverso la propria interfaccia remota.** [26](#_heading=h.zu0gcz)

Meccanismo di invocazione remota: [26](#_heading=h.3jtnz0s)

● **Riferimenti remoti**: nel modello distribuito il client interagisce con il server tramite uno stub che rappresenta l’interfaccia remota dell’oggetto remoto in locale. Dal punto di vista della JVM infatti il tipo dello stub è uguale al tipo del server. Quindi un client può accedere tradizionalmente al tipo di un oggetto remoto controllando quale interfaccia remota implementa, attraverso instaceOf. [26](#_heading=h.1yyy98l)

● **Localizzazione e invocazione di oggetti remoti**: per poter invocare il metodo remoto, il client deve avere a disposizione il riferimento remoto. Questo può essere reperito in due modi: [26](#_heading=h.4iylrwe)

o **Ottenendo il riferimento come risultato di altre invocazioni (remote o locali) di metodi**; [26](#_heading=h.2y3w247)

o **Attraverso un servizio di directory:** ovvero tramite un name-server che Java RMI fornisce nella classe java.rmi.Naming, tale classe permette di gestire riferimenti remoti ad oggetti tramite un ID (stringa), oltre a fornire metodi per: [26](#_heading=h.1d96cc0)

Passaggio di parametri [27](#_heading=h.3x8tuzt)

Differenza tra il modello a oggetti locale e quello remoto [27](#_heading=h.2ce457m)

La sicurezza in java [27](#_heading=h.rjefff)

● **La sicurezza intrinseca di java:** ovvero tutte quelle caratteristiche del linguaggio che garantiscono la sicurezza: [27](#_heading=h.3bj1y38)

o **Linguaggio fortemente tipizzato**: ogni dato ha un tipo, e solo poche operazioni di casting vengono eseguite automaticamente dal compilatore. [27](#_heading=h.1qoc8b1)

o **Gestione automatica della memoria**: tramite il garbage collector. [27](#_heading=h.4anzqyu)

o **Assenza di puntatori**: insieme all’impossibilità di effettuare aritmetica sugli stessi, impedisce accessi illegali alla memoria. [27](#_heading=h.2pta16n)

o **Lazy allocation**: la memoria del programma viene allocata a tempo di esecuzione, questo rende impossibile sapere in anticipo dove il programma andrà a finire e quindi impossibile rimpiazzare quelle parti di memoria con software malevolo. [27](#_heading=h.14ykbeg)

● **Il Classloader**: carica le classi a tempo di esecuzione in modo da non permettere modifiche alle classi bult-in del linguaggio. [27](#_heading=h.3oy7u29)

● **Il Bytecode verifier**: controlla l’assenza di codice malevolo all’interno del bytecode e ne verifica la conformità con gli standard del linguaggio, questo è necessario perché il bytecode una volta generato è liberamente accessibile e quindi potrebbe essere manomesso. [27](#_heading=h.243i4a2)

● **Il Security Manager**: si occupa di definire i confini della sandbox. Viene interpellato dalla macchina virtuale per ciascuna operazione potenzialmente pericolosa e fornisce le autorizzazioni sulla base delle policy stabilite dall’utente. [27](#_heading=h.j8sehv)

I tre layer della architettura: [27](#_heading=h.338fx5o)

● **Stub/Skeleton layer**: che comprende gli stub lato server e gli skeleton lato client. Con le versioni più moderne di java, questi due componenti vengono creati automaticamente, semplificando di molto la vita di programmatori al netto di una piccola perdita di efficienza. [27](#_heading=h.1idq7dh)

● **Remote Reference Layer:** che specifica il comportamento dell’invocazione (unicast, multicast), in particolare, questo layer si occupa di fare da tramite tra il livello stub/skeleton e il livello di trasporto. [27](#_heading=h.42ddq1a)

● **Trasport Layer:** che si occupa della connessione e della sua gestione. Chiamato **Java Remote Method Control**, ma con il tempo questo protocollo e stato sostituito o comunque reso modificabile per garantire una migliore compatibilità di **Java RMI**. [27](#_heading=h.2hio093)

Marshalling [28](#_heading=h.wnyagw)

Lezione 13, Introduzione a Java EE (Java Enterprise Edition): [28](#_heading=h.3gnlt4p)

● I componenti di livello client vengono eseguiti sul computer client e sono di due tipi: [28](#_heading=h.1vsw3ci)

o **Client Web:** semplice pagina web che funge da UI, molto utile visto che tutti hanno un browser. [28](#_heading=h.4fsjm0b)

o **Client applicativo:** applicazione software installabile, che viene utilizzata in caso di necessità di un’interfaccia più ricca. [28](#_heading=h.2uxtw84)

I componenti di livello Web vengono eseguiti sul server Java EE. [28](#_heading=h.1a346fx)

● I **componenti di livello aziendale** eseguono la logica di business e si basano sull’utilizzo di java Beans. [28](#_heading=h.3u2rp3q)

● Il **software EIS** (Enterprise Information System) funziona sul server EIS. [28](#_heading=h.2981zbj)

Packaging [28](#_heading=h.odc9jc)

Java EE Container [29](#_heading=h.38czs75)

Ciclo di vita degli Enterprise java Beans: (Ci boccia la gente con sta cosa) [29](#_heading=h.1nia2ey)

Annotazioni e Deployment Descriptors [29](#_heading=h.47hxl2r)

● **Non ci sono import**: il container sa di cosa ha necessità e cosa deve importare. [29](#_heading=h.2mn7vak)

● **Stateless**: significa che l’oggetto è composto solo da metodi, per questo può essere utilizzato da tutti i programmi che hanno necessità di usare quei metodi. [29](#_heading=h.11si5id)

● **Remote e Local**: significa che il bean è sia remoto che locale. [29](#_heading=h.3ls5o66)

● **PersistenceContext**: è l’insieme delle componenti che vanno rese persistenti. [29](#_heading=h.20xfydz)

Lezione 14-15, Context and Dependency Injection: [29](#_heading=h.4kx3h1s)

Inversion control [29](#_heading=h.302dr9l)

loose coupling e strong typing [30](#_heading=h.1f7o1he)

Principio di Hollywood [30](#_heading=h.3z7bk57)

Pattern di dependecy injection [30](#_heading=h.2eclud0)

Managed Beans [30](#_heading=h.thw4kt)

Vantaggi dei managed beans: [30](#_heading=h.3dhjn8m)

● **Non devo gestire lo stato di uno oggetto**: ci pensa il container; [30](#_heading=h.1smtxgf)

● **Sono legati a contesti ben definiti**; [30](#_heading=h.4cmhg48)

● **Hanno un approccio type safe**: sto accedendo a delle risorse iniettate con una precisa struttura derivata dalla classe java; [30](#_heading=h.2rrrqc1)

● **Possono essere specializzati;** [30](#_heading=h.16x20ju)

● **Possono essere usati nel layer di presentazione;** [30](#_heading=h.3qwpj7n)

● **Ogni classe java può essere una bean CDI** (Content Dependence Injection); [30](#_heading=h.261ztfg)

Ciclo di vita dei managed bean [30](#_heading=h.l7a3n9)

● **New Instance**: prima di questa fase il bean non esiste, il container crea un oggetto tramite new; [30](#_heading=h.356xmb2)

● **Dependecy injection**: inietta le dipendenze (anche il bean può avere delle dipendenze); [30](#_heading=h.1kc7wiv)

● **Post contruct**: eseguo i metodi @PostCostruct e mi trovo nello stato di ready dove il bean è pronto a rispondere alle chiamate fino a quando sarà necessario; [30](#_heading=h.44bvf6o)

● **Pre destroy**: una volta che il bean diventa inutile vengono eseguiti i metodi annottati come @PreDestroy e il bean torna allo stato di inesistenza. [30](#_heading=h.2jh5peh)

Scope [31](#_heading=h.ymfzma)

Interceptor [31](#_heading=h.3im3ia3)

@Inject [31](#_heading=h.1xrdshw)

● **Property**: Quando viene eseguita sugli attributi; [31](#_heading=h.4hr1b5p)

● **Contructor**: Quando l’iniezione viene eseguita sul costruttore; [31](#_heading=h.2wwbldi)

● **Setter**: Quando l’iniezione viene eseguita sul setter; [31](#_heading=h.1c1lvlb)

Default Injection: se non dichiariamo nessun Qualifier, Java EE imposta automaticamente quello di Default. [31](#_heading=h.3w19e94)

Qualifier: è un meccanismo che permette di informare il container su quale classe iniettare utilizzando il sistema delle annotazioni (evitando l’uso di XML). [31](#_heading=h.2b6jogx)

Qualificatori con membri: per evitare di creare un grande numero di qualificatori all’interno del nostro codice è possibile “personalizzarli” inserendo dei membri, in questo modo abbiamo un unico qualificatore che agisce per ognuno dei nostri casi. [31](#_heading=h.qbtyoq)

Qualificatori Multipli: un altro metodo per ridurre il numero di qualificatori usati. [31](#_heading=h.3abhhcj)

Alternatives: a volte può essere necessario scegliere quale risorsa iniettare in base a un determinato scenario di utilizzo. Per fare ciò ci vengono in aiuto le alternative, che sono disattivate di default e nel caso in cui volessimo usarle devono essere attivate. [31](#_heading=h.1pgrrkc)

Producers: di norma all’interno di Java EE non è possibile iniettare tipi primitivi o POJO (java.util.data, java.util.String, ecc) perché sono impacchettati in file .jar e questo tipo di archivio non contiene deployment descriptor (beans.XML) come invece lo hanno i beans CDI. Quindi per iniettare questo tipo di oggetti dobbiamo utilizzare i produttori. [31](#_heading=h.49gfa85)

Disposer: alcuni tipo di oggetti e tipi di dati, una volta creati ed utilizzati necessitano di essere distrutti, per fare ciò si utilizzano i disposer. Praticamente con il producer creaiamo e con il disposer ne gestiamo la distruzione o la chiusura. [32](#_heading=h.2olpkfy)

Scopes: [32](#_heading=h.13qzunr)

● **Application Scope (@ApplicationScoped):** si estende per l'intera durata di un'applicazione. Il bean viene creato una sola volta per la durata dell'applicazione e viene eliminato quando l'applicazione è chiusa (sono bean di supporto per gestire o spostare dati). [32](#_heading=h.3nqndbk)

● **Session Scope (@SessionScoped):** Utilizzato per richieste HTTP o su metodi di invocazione per la sessione di un singolo utente. Il bean viene creato per la durata di una sessione http e viene scartato al termine della sessione. [32](#_heading=h.22vxnjd)

● **Request Scope (@RequestScoped):** utilizzato per una singola richiesta HTTP o un metodo invocazione. Il bean viene creato per la durata dell'invocazione del metodo e viene scartato quando il metodo termina. [32](#_heading=h.i17xr6)

● **Conversation Scope(@ConversationScoped):** si estende tra più invocazioni all'interno della sessione con i punti iniziale e finale determinati dall'applicazione. Le conversazioni vengono utilizzate su più pagine come parte di un flusso di lavoro a più fasi. [32](#_heading=h.320vgez)

● **Dependent pseudo-scope (@Dependent):** il ciclo di vita è uguale a quello del client. Viene creato ogni volta che viene iniettato e il riferimento viene rimosso quando il target di iniezione è rimosso. Questo è l'ambito predefinito per CDI. [32](#_heading=h.1h65qms)

Interceptor [32](#_heading=h.415t9al)

● **Constructor-level interceptors**: intercettore associato a un costruttore della classe target (@AroundConstruct); [32](#_heading=h.2gb3jie)

● **Method-level interceptors**: intercettore associato a un metodo business specifico (@AroundInvoke); [32](#_heading=h.vgdtq7)

● **Timeout method interceptors**: intercettore che si interpone sui metodi di timeout con @AroundTimeout (utilizzato solo con il servizio timer EJB); [32](#_heading=h.3fg1ce0)

● **Life-cycle callback interceptors**: intercettore che si interpone sul ciclo di vita dell'istanza target. (@PostConstruct e @PreDestroy). [32](#_heading=h.1ulbmlt)

Interceptor Binding [33](#_heading=h.4ekz59m)

Priorizzare gli intercettori [33](#_heading=h.2tq9fhf)

Decorators [33](#_heading=h.18vjpp8)

Eventi [33](#_heading=h.3sv78d1)

Lezione 17-18, Java Persistent API: [33](#_heading=h.280hiku)

L’utilità dei servizi [33](#_heading=h.n5rssn)

Caratteristiche trasversali [33](#_heading=h.375fbgg)

End to End principe [33](#_heading=h.1maplo9)

La persistenza in Java EE [33](#_heading=h.46ad4c2)

Object-Relational Mapping (ORM) [34](#_heading=h.2lfnejv)

Oggetti persistenti: Creazione di un’entità: [34](#_heading=h.10kxoro)

● **@Entity:** informa il container (o meglio l’EntityManager) che quella che segue è un’entità (oggetto persistente), e sarà esso a gestirla. [34](#_heading=h.3kkl7fh)

● **@id:** rappresenta la nostra chiave primaria, che identificherà in modo univoco ogni tupla dell’entità. [34](#_heading=h.1zpvhna)

● **Il resto:** sono attributi, costruttore e metodi getter e setter come una comune classe java. [34](#_heading=h.4jpj0b3)

Proprietà di un’entità: [34](#_heading=h.2yutaiw)

Configuration by Exception [34](#_heading=h.1e03kqp)

Struttura e trasformazione da oggetto a entità: [34](#_heading=h.3xzr3ei)

Definire query in Java EE [34](#_heading=h.2d51dmb)

Ciclo di vita delle entità: (Alta probabilità) [34](#_heading=h.sabnu4)

CallBacks: [35](#_heading=h.3c9z6hx)

Listeners [35](#_heading=h.1rf9gpq)

Validazione dei bean [35](#_heading=h.4bewzdj)

Relationship Mapping [35](#_heading=h.2qk79lc)

Gestione della cardinalità [35](#_heading=h.gjdgxs)

● **@OneToOne;** [35](#_heading=h.15phjt5)

● **@OneToMany;** [35](#_heading=h.3pp52gy)

● @**ManyToOne;** [35](#_heading=h.24ufcor)

● **@ManyToMany;** [35](#_heading=h.jzpmwk)

Concetto di persistence context [35](#_heading=h.33zd5kd)

Metodi del Entity Manager [35](#_heading=h.1j4nfs6)

● **Cercare un’entità all’interno del database**: per fare ciò esistono due metodi: [35](#_heading=h.434ayfz)

● **Rimuovere un entità all’interno del database:** per fare ciò si utilizza il metodo **Remove(),** rende l’entità **detached** all’interno dell’Entity manager (**l’entità non viene eliminata dalla memoria, ma torna ad essere un POJO, viene eliminata solo quando il GC la distrugge**). [36](#_heading=h.2i9l8ns)

● **Rimozione degli orfani**: con il temine **orfano** si fa riferimento a righe che non sono referenziate da nessuna altra tabella. JPA è in grado di rimuovere automaticamente tali entità se le viene esplicitamente chiesto in fase di definizione dell’oggetto (**cancellazione e modifica a cascata**). [36](#_heading=h.xevivl)

JPQL (Java Persistence Query Language): (Alta probabilità) [36](#_heading=h.3hej1je)

Query dinamiche: generate dinamicamente all’interno di un’applicazione ed eseguite. È la tipologia di query più utilizzata nonché la più versatile anche se ha degli svantaggi (non è possibile verificare sintatticamente il codice SQL all’interno di esse) [36](#_heading=h.1wjtbr7)

Named query: sono query statiche, inserite all’interno di una classe all’atto della sua creazione. Vengono utilizzate per compiti standard. Non possono essere modificate, ma gli si può passare un parametro. [36](#_heading=h.4gjguf0)

Criteria API (or Object-Oriented Queries): Sono query orientate agli oggetti, uniscono le Named e le Dinamiche e hanno il grande vantaggio che la sintassi viene verificata a tempo di compilazione in modo da evitare errori, anche se questo le rende meno leggibili. Sono inoltre fortemente tipizzate. [36](#_heading=h.2vor4mt)

Native query: sono query native del SQL. Non si usano a meno di casi molto rari e particolari. [36](#_heading=h.1au1eum)

Stored procedure query: sono memorizzate all’interno del database. Usate per compiti lunghi e ripetuti, oltre che per scopi statistici. Hanno il grande vantaggio che il codice è centralizzato e può essere utilizzato da diversi programmi. [36](#_heading=h.3utoxif)

Lezione 20, Enterprise JavaBeans, parte 1: [36](#_heading=h.29yz7q8)

Enterprise JavaBeans (EJB): è un componente di JEE server-side che si occupa di incapsulare la logica di business e di gestire transazioni e sicurezza. All’interno di JEE vengono definiti tre tipi di sessioni EJB: [36](#_heading=h.p49hy1)

● **Stateless**: Il bean non viene istanziato né usato da nessun client e non ci sono conversazioni di stato fra i metodi. Praticamente fornisce solo servizi. [37](#_heading=h.393x0lu)

● **Stateful**: l’opposto di stateless, viene utilizzato per operazioni che si dividono in più step. [37](#_heading=h.1o97atn)

● **Singleton**: è un bean che viene condiviso dai client e che supporta la concorrenza di accesso. [37](#_heading=h.488uthg)

● **La classe bean:** che contiene l’implementazione dei metodi di business e che può implementare nessuna o diverse interfacce. L’importante è che il bean sia annotato con @stateless, @stateful o @singleton. [37](#_heading=h.2ne53p9)

● **Un’interfaccia di business**: che contiene la dichiarazione dei metodi di business che sono visibili al client e implementati nella classe del bean. Un bean può avere un’interfaccia locale, remota o nessuna. L’utilizzo di un’interfaccia remota o locale dipende dal tipo di chiamata che si sta andando a effettuare. [37](#_heading=h.12jfdx2)

● **@Remote**: Denota un’interfaccia remota di business. I parametri sono passasti per valore e necessitano di essere serializzati. [37](#_heading=h.3mj2wkv)

● **@Local:** Denota un’interfaccia locale. I parametri sono passati per referenza dal client al bean. [37](#_heading=h.21od6so)

Stateless Beans [37](#_heading=h.gtnh0h)

Stateful Beans [37](#_heading=h.30tazoa)

Singleton [37](#_heading=h.1fyl9w3)

Packaging [37](#_heading=h.3zy8sjw)

Invocazione di EJB [37](#_heading=h.2f3j2rp)

Invocazione tramite iniezione [38](#_heading=h.u8tczi)

Invocazione con CDI (Context Dependecy Injection) [38](#_heading=h.3e8gvnb)

Invocazione diretta con JNDI [38](#_heading=h.1tdr5v4)

Lezione 21, Enterprise JavaBeans, parte 2: [38](#_heading=h.4ddeoix)

Ciclo di vita dei Session Beans: (Possibile domanda teoria allo scritto) [38](#_heading=h.2sioyqq)

Stateless beans e i singleton il ciclo di vita è molto semplice: [38](#_heading=h.17nz8yj)

● Tutto inizia quando il beans viene richiesto (tramite **injection** o **JNDI lookup**) e il container istanzia una nuova sessione. [38](#_heading=h.3rnmrmc)

● Se l’istanza appena creata utilizza a sua volta delle **injection**, il container soddisfa anche queste. [38](#_heading=h.26sx1u5)

● Se l’istanza implementa dei metodi **@PostConstruct**, vengono eseguiti dal container. [38](#_heading=h.ly7c1y)

● Il bean va nello stato di **ready** dove in caso di necessità esegue le richieste dei client, rimanendo in attesa quando non ne riceve. [38](#_heading=h.35xuupr)

● Il container non ha più necessità del bean. Quindi invoca i metodi **@PreDestroy** se ce ne sono e poi elimina il bean. [38](#_heading=h.1l354xk)

Stateful beans, il ciclo è un po' diverso: [38](#_heading=h.452snld)

● Tutto inizia quando il beans viene richiesto da un client (tramite **injection** o **JNDI lookup**) e il container istanzia una nuova sessione. [38](#_heading=h.2k82xt6)

● Se l’istanza appena creata utilizza a sua volta delle **injection**, il container soddisfa anche queste. [38](#_heading=h.zdd80z)

● Se l’istanza implementa dei metodi **@PostConstruct**, vengono eseguiti dal container. [38](#_heading=h.3jd0qos)

● Il bean esegue le richieste del client e va in memoria in attesa di richieste future (stato di **ready**). [38](#_heading=h.1yib0wl)

● Se il client rimane inattivo per un periodo di tempo, il container si prepara a rendere il bean **passivo** eseguendo prima i metodi annotati con **@PrePassivate**, successivamente lo sposta nella memoria di massa. [38](#_heading=h.4ihyjke)

● Se il client invoca un bean in stato **passivo** il container prima di risvegliarlo esegue i metodi annotati con **@PostActivate**, per poi spostarlo nella memoria centrale. [38](#_heading=h.2xn8ts7)

● Se il client non invoca un bean **passivo** per un certo periodo di tempo scatta un **timeout** e il bean viene distrutto. [38](#_heading=h.1csj400)

● In alternativa se il client invoca il metodo annotato con **@Remove** il container esegue i metodi annotati con @**PreDestroy** e distrugge il bean. [38](#_heading=h.3ws6mnt)

Autorizzazioni [39](#_heading=h.2bxgwvm)

● Dichiarativa Possono essere definite utilizzando **annotazioni** o un **XML Deployment descriptor.** [39](#_heading=h.r2r73f)

● Programmatica: permettono di realizzare un controllo sulle autorizzazioni a grana più fine, utilizzando due metodi forniti dall’interfaccia SessionContext: [39](#_heading=h.3b2epr8)

o **isCallerInRole():** Ritorna un booleano e controlla se il chiamante ha un ruolo assegnato. [39](#_heading=h.1q7ozz1)

o **getCallerPrincipal():** ritorna java.security.Principal che identifica il chiamante. [39](#_heading=h.4a7cimu)

Transazioni [39](#_heading=h.2pcmsun)

● **Atomicità**: una transazione è composta da una o più operazioni raggruppate in un’unità di lavoro, alla fine della transazione o tutte le operazioni sono andate a buon fine (commit) o si devono annullare quelle che sono state eseguite (rollback). [39](#_heading=h.14hx32g)

● **Consistenza**: alla fine della transazione tutti i dati in uscita devono essere consistenti. [39](#_heading=h.3ohklq9)

● **Isolata**: gli stadi intermedi della transazione non devono essere visibili esternamente all’applicazione. [39](#_heading=h.23muvy2)

● **Durabilità**: deve essere finita, ovvero una volta che si è conclusa i cambiamenti nei dati devono essere visibili alle altre applicazioni. [39](#_heading=h.is565v)

Lezione 23/24, Java Message Service [39](#_heading=h.32rsoto)

Message Oriented Middleware MOM [39](#_heading=h.1hx2z1h)

JMS è il set di API che si occupa di implementare questo sistema. In JMS esistono due tipi di destinazione ognuna applicata a uno specifico modello di messagistica: [39](#_heading=h.41wqhpa)

● **Point to point** all’interno del quale I messaggi viaggiano da un singolo producer a un singolo consumer. Questo modello si basa sul concetto di coda. Viene anche utilizzato nel caso in cui un producer si trovi a dover inviare tanti messaggi a diversi consumer (Multiple receivers). [39](#_heading=h.2h20rx3)

● **Publish-Sunbscribe**: in questo modello un singolo messaggio è inviato da un singolo producer per diversi consumer, anche contemporaneamente (viene chiamato sunscribers, perché si deve prima iscrivere al Topic per ricevere messaggi). In questo caso c’è però una dipendenza temporale, visto che i messaggi vengono consegnati solo dopo l’iscrizione (quelli prima vengono persi). [39](#_heading=h.w7b24w)

Java Messaging Service API in dettaglio: [39](#_heading=h.3g6yksp)

● **ConnectionFactory**: è un **administered object** (un oggetto che può essere creato e gestito solo dall’amministratore). Permette a un’applicazione di creare un oggetto **Connection**. [40](#_heading=h.1vc8v0i)

● **Destination**: è un altro **administered object**, che contiene tutte le informazioni sui provider come l’indirizzo di destinazione. Informazioni che sono nascoste ai client. [40](#_heading=h.4fbwdob)

● **Connection**: incapsula una connessione a un provider JMS. È una struttura thread-safe disegnata per essere condivisibile tramite il concetto di sessione. Prima di essere utilizzata, una connessione deve essere avviata con il metodo .start() (poi chiusa con il metodo stop()). [40](#_heading=h.2uh6nw4)

● **Sessioni**: Viene creata tramite il metodo createSession(). Il primo parametro specifica se la sessione implementa o meno le transazioni (se settato a true vuol dire che i messaggi non saranno inviati finché non sarà dato il commit(), se è settato su false() appena viene dato il metodo send() il messaggio viene inviato). [40](#_heading=h.19mgy3x)

● **Messaggi:** sono oggetti in cui vengono incapsulate informazioni, sono divisi in tre parti: [40](#_heading=h.3tm4grq)

o **Header**: contiene informazioni sul messaggio, che client e provider utilizzano per identificare il messaggio. [40](#_heading=h.28reqzj)

● **Properties**: è un meccanismo per inserire altri campi header all’interno del messaggio tramite applicazioni. [40](#_heading=h.nwp17c)

o **Bodies**: contiene i dati che si desidera inviare. È un campo molto versatile che può contenere molti dati differenti. [40](#_heading=h.37wcjv5)

Differenza tra Properties e Header: (Alta probabilità) [40](#_heading=h.1n1mu2y)

Meccanismi di affidabilità del broker [40](#_heading=h.471acqr)

● **Filtering messages:** permette di creare dei filtri in modo che il broker invii al client solo messaggi che rispettano determinati requisiti. [40](#_heading=h.2m6kmyk)

● **Time-to-live:** permette di definire un tempo (in millisecondi) dopo il quale il broker elimina i messaggi dalla destinazione se non sono stati ancora consegnati. [40](#_heading=h.11bux6d)

● **Message Persistence**: JMS supporta due metodi di consegna per i messaggi: [40](#_heading=h.3lbifu6)

o **Persistent**: si assicura che il messaggio sia recapitato una sola volta (Più costoso in termini di risorse). [40](#_heading=h.20gsq1z)

o **Nonpersistent**: si assicura che il messaggio sia recapitato ma non controlla quante volte. [40](#_heading=h.4kgg8ps)

● **Controllo degli Acknowledgment:** permette di ricevere una notifica dal consumer di avvenuta consegna del messaggio. Nelle sessioni transazionali, gli ack vengono automaticamente utilizzati (servono in caso di roll-back). Nelle sessioni non transazionali, devono invece essere specificati. [40](#_heading=h.2zlqixl)

● **Durable Consumer**: permette (all’interno del modello pub-sub) al consumer di ricevere i messaggi anche nel caso in cui non sia connesso al topic quando quei messaggi vengono inviati (è molto oneroso in termini di risorse, ma alcune volte necessario). [40](#_heading=h.1er0t5e)

● **Settare una priorità**: permette di settare una priorità ai messaggi, in modo che vengano inviati prima di altri con priorità più bassa (valore da 0 a 9). [40](#_heading=h.3yqobt7)

Message-Driven Beans [41](#_heading=h.2dvym10)

Ciclo di vita di un MDB [41](#_heading=h.t18w8t)

Lezione 26, Web Service and SOA [41](#_heading=h.3d0wewm)

SOAP: che offre un gran numero di servizi ma è meno efficiente. [41](#_heading=h.1s66p4f)

REST: versione migliorata di SOAP con performance migliori. [41](#_heading=h.4c5u7s8)

Software as Service [41](#_heading=h.2rb4i01)

Simple Object Access Protocol (SOAP): [41](#_heading=h.16ges7u)

● **Extensible Markup Lenguage(XML):** utilizzato da SOAP per costruire e definire. Rappresenta una scelta “terza” per garantire uno standard comune di comunicazione. Permette di definire delle regole per garantire che il file sia correttamente formattato. [41](#_heading=h.3qg2avn)

● **Web Services Descriptor Language (WSDL):** definisce i protocolli, le interfacce, i tipi di messaggi e le iterazioni tra consumer e provider, è come un’interfaccia di Java ma scritta in XML, serve a esporre i servizi del service provider. Contiene tutte le informazioni utili al consumer per identificare il provider, il protocollo di trasporto utilizzato e tutte le informazioni necessarie tramite una serie di elementi e attributi. [41](#_heading=h.25lcl3g)

● **Simple Object Access Protocol (SOAP):** è un protocollo di message-encoding utilizzato per definire la comunicazione dei web services. Si occupa in pratica di implementare lo scambio di messaggi XML tra provider e consumer. WSDL definisce il tipo di messaggi che devono essere scambiati e SOAP li implementa. [41](#_heading=h.kqmvb9)

● **HTTPS, SMTP, JMS**: come protocolli di trasporto; [41](#_heading=h.34qadz2)

● **Universal Descriptor Discover, and Integration (UDDI):** un servizio di registri e meccanismi di discover opzionale. Permette al consumer di indentificare il provider e di conoscere gli standard per comunicare con quest’ultimo. [42](#_heading=h.1jvko6v)

I tre pilastri dei Web Services sono: WSDL,SOAP,UDDI. [42](#_heading=h.43v86uo)

Writing SOAP Web Services: Per scrivere Web Services esistono due principali approcci: [42](#_heading=h.2j0ih2h)

● **Contract-first**: visto che il WSDL è il contratto tra consumer e service, può essere utilizzato per generare il codice java per consumer e provider. [42](#_heading=h.y5sraa)

● **Bottom-UP**: si parte da un codice normale e lo si rende web services tramite l’annotazione @WebService. [42](#_heading=h.3i5g9y3)

Java Architecture for XML Binding (JAXB) [42](#_heading=h.1xaqk5w)

WSDL Mapping [42](#_heading=h.4hae2tp)

● **JAXB**: utilizza annotazioni per determinare come marshal/unmarshal una classe to/from XML. [42](#_heading=h.2wfod1i)

● **JSON Web Signature (JWS):** utilizzato per mappare classi java in WSDL, per effettuare il marshal di un’invocazione di un metodo tramite SOAP e unmarshal per l’eventuale risposta del metodo. [42](#_heading=h.1bkyn9b)

@WebService: [42](#_heading=h.3vkm5x4)

WSDL mapping annotations: Utilizzate per personalizzare le firme dei metodi esposti. [42](#_heading=h.2apwg4x)

● @**WebMethod**: Permette di modificare il nome del metodo presente in WSDL o di escluderlo da quest’ultimo in modo che non sia visibile al client. [42](#_heading=h.pv6qcq)

● @**WebResult**: Controlla il nome del messaggio di ritorno in WSDL, permettendo di modificarlo. [42](#_heading=h.39uu90j)

● @**WebParam**: simile a @WebResult ma si occupa di parametri, permette infatti di modificare il nome di un parametro e il suo tipo. [42](#_heading=h.1p04j8c)

● @**OneWay**: viene utilizzata su metodi che non hanno un valore di ritorno (void), serve a informare il container in modo che possa ottimizzare la chiamata. [42](#_heading=h.48zs1w5)

SOAP Binding Annotations: Descrive come il web service è limitato a un protocollo di messagistica. Esistono due tipi di stili di programmazione che definiscono il SOAP Binding e che differiscono sostanzialmente per il contenuto del SOAPbody: [42](#_heading=h.2o52c3y)

● **Document**: il messaggio contiene un documento senza ulteriori regole di formattazione (Scelta di default). [42](#_heading=h.13acmbr)

● **RPC**: il SOAP contiene un elemento con il nome del metodo o la procedura remota che è stata invocata. [42](#_heading=h.3na04zk)

● **è inoltre possibile scegliere fra due tipi di serializzazione/deserializzazione:** [42](#_heading=h.22faf7d)

o **Literal**: i dati vengono serializzati in accordo allo schema XML. [42](#_heading=h.hkkpf6)

o **Encoded**: la codifica del SOAP specifica come gli oggetti devono essere serializzati. [42](#_heading=h.31k882z)

● **Questo ci porta ad avere 4 possibili tipologie di configurazioni:** [42](#_heading=h.1gpiias)

o Document/Literal (Configurazione di Default). [42](#_heading=h.40p60yl)

o Document/Encoded [42](#_heading=h.2fugb6e)

o RPC/Literal [43](#_heading=h.uzqle7)

o RPC/Encoded [43](#_heading=h.3eze420)

Handling Exceptions [43](#_heading=h.1u4oe9t)

Ciclo di vita [43](#_heading=h.4e4bwxm)

WebServiceContext [43](#_heading=h.2t9m75f)

Invoking SOAP Web Service [43](#_heading=h.18ewhd8)

Invocazione programmatica [43](#_heading=h.3sek011)

Invocazione tramite Injection [43](#_heading=h.27jua8u)

Invocazione con CDI [43](#_heading=h.mp4kgn)

Concetti rilevanti di Cloud Computing: [43](#_heading=h.36os34g)

High-Performance Computing (HPC) [43](#_heading=h.1lu2dc9)

Cluster architetture [43](#_heading=h.45tpw02)

High-Throughput Computing (HTC) [44](#_heading=h.2kz067v)

Reti P2P [44](#_heading=h.104agfo)

Computational and Data Grids [44](#_heading=h.3k3xz3h)

The Hype Cycle of New Technologies [44](#_heading=h.1z989ba)

Infrastrutture Virtuali [44](#_heading=h.4j8vrz3)

Cloud service models: I servizi offerti dal cloud possono essere divisi in tre categorie principali: [44](#_heading=h.2ye626w)

● **Infrastructure as a Service (IaaS):** l’utente può utilizzare il SO, sistemi di storage, applicazioni e i componenti della rete, ma non controlla direttamente l’infrastruttura cloud. Praticamente ha a disposizione una macchina da configurare. [44](#_heading=h.1djgcep)

● **Platform as a Service (PaaS):** permette di distribuire e gestire l’esecuzione di un’applicazione utilizzando risorse fornite dal provider cloud tramite un ambiente adeguato. [44](#_heading=h.3xj3v2i)

● **Software as a Service (SaaS):** permette di utilizzare software tramite un interfaccia web. [44](#_heading=h.2coe5ab)

Cloud Pubblici: costruito su internet, può essere utilizzato da chiunque pagando una certa quota mensile o giornaliera. [44](#_heading=h.rtofi4)

Cloud Privati: costruito all’interno di un dominio o all’interno di un intranet in possesso di una singola organizzazione, viene utilizzato per la gestione interna e per il funzionamento della stessa. [44](#_heading=h.3btby5x)

Cloud ibridi: utilizzato sia per scopi interni di un’azienda/organizzazione, che per offrire servizi al pubblico tramite internet. [44](#_heading=h.1qym8dq)

Model Cost: Grazie al cloud viene eliminata tutta quella parte di costi fissi per l’equipaggiamento che le aziende si trovavano a dover affrontare per la messa in opera della loro infrastruttura hardware. [44](#_heading=h.4ay9r1j)

Cloud Computing Economics: [44](#_heading=h.2q3k19c)

Concetti rilevanti di Microservice: [45](#_heading=h.158ubh5)

La morte dei Big Software [45](#_heading=h.3p8hu4y)

Micro Services [45](#_heading=h.24ds4cr)

Componenti come servizi [45](#_heading=h.jj2ekk)

Sono realizzati intorno alle competenze di business: [45](#_heading=h.33ipx8d)

legge di Conway: Qualsiasi organizzazione che progetta un sistema produrrà un design la cui struttura è una copia della struttura di comunicazione dell'organizzazione. [45](#_heading=h.1io07g6)

Smart endpoints and dumb pipes: [46](#_heading=h.42nnq3z)

Gestione decentralizzata del calcolo e dei dati [46](#_heading=h.2hsy0bs)

Infrastructure Automation [46](#_heading=h.wy8ajl)

Progettati per i malfunzionamenti [46](#_heading=h.3gxvt7e)

Progettazione evolutiva [46](#_heading=h.1w363f7)

Differenza tra MC e Web Services [46](#_heading=h.4g2tm30)

Concetti di Server-Less computing: [46](#_heading=h.2v83wat)

Benefici dello sviluppo Server-Less: [47](#_heading=h.1ade6im)

● **Sviluppo e deployment rapido**: i costi di sviluppo sono bassi e i tempi di sviluppo rapidi, con la possibilità di rilasciare subito e migliorare nel tempo. [47](#_heading=h.3ud1p6f)

● **Facilità di uso**: essenzialmente posso realizzare funzionalità complesse che si evolvono in maniera diversa e indipendente. [47](#_heading=h.29ibze8)

● **Costo minore**: tutto i costi di manutenzione e gestione vengono eliminati. [47](#_heading=h.onm9m1)

● **Scalabilità**: usare SL permette al provider di bilanciare il carico in modo molto più semplice e veloce, cosa che permette anche al provider di offrire servizi a costi minori. In poche parole, l’utente paga le invocazioni invece delle macchine. [47](#_heading=h.38n9s9u)

● **Nessun costo di manutenzione dell’infrastruttura**: che ovviamente sono a carico dei provider. [47](#_heading=h.1nsk2hn)

● **Si perde il controllo dell’infrastruttura hardware**: ci si lega al provider sia economicamente che progettualmente. [47](#_heading=h.47s7l5g)

● **Cold start**: Le applicazioni SL hanno un “avviamento freddo” perché il sistema che gestisce e ottimizza queste funzioni se vede che non sono utilizzate per un tot di tempo le mette da parte. Per risolvere questo problema si fa un Ping per riattivare la funzione ogni tanto, in modo da tenerla “sveglia” (ogni Ping ovviamente costa quindi se lo facciamo troppo spesso i costi aumentano). [47](#_heading=h.2mxhvd9)

● **Architettura condivisa**: si possono condividere le infrastrutture, ma se una va giù va giù anche l’altra. [47](#_heading=h.122s5l2)

# Lezione 1, Prologo ai sistemi distribuiti:

Un sistema distribuito è un sistema formato da un insieme di macchine autonome connesse alla rete, che comunicano tra loro per risolvere problemi. In un sistema parallelo, invece, gli elementi di elaborazione comunicano e cooperano per risolvere velocemente problemi di grandi dimensioni.

## Middleware: è uno strato software che si occupa di connettere le macchine fra di loro in modo da far percepire all’utente il sistema come un'unica entità integrata.

## Motivazioni dietro i sistemi distribuiti:

esistono motivazioni sia di tipo economico che tecnologico:

* **Economico:** rispondono in maniera precisa alle esigenze ed alle richieste della economia di mercato che è caratterizzata da numerose e frequenti acquisizioni, integrazioni e fusioni di aziende.
* **Tecnologico:** permettono una migliore gestione e integrazione delle tecnologie legacy (si cerca quando possibile di integrarle, perché è inutile cambiarle), permettendo di assecondare i rapidi e repentini cambiamenti dell’industria.

## Legge di Moore

è una delle leggi più famose dell’informatica (**legge perché per ora funziona**, domani non si sa, altrimenti l’avremmo chiamata teorema), essa afferma che il numero di transistor all’interno di un processore raddoppia ogni 18 mesi, per tenere il passo con la sempre crescente necessità di potenza di calcolo.

## Leggi per le reti

#### Legge di Sarnoff: il valore di una rete di broadcast è direttamente proporzionale al numero degli utenti connessi.

#### Legge di Metcafe: il valore di una rete di comunicazione è direttamente proporzionale al quadrato del numero di utenti.

#### Legge di Reed: il valore di una rete sociale è direttamente proporzionale ad una funzione esponenziale in N.

Queste leggi portano all’interessante conclusione che le reti sono naturalmente portate ad accorparsi, integrando i servizi in modo da fornirli agli utenti di tutte le reti che si stanno fondendo.

#### Reti di tipo Broadcast (Sarnoff): per le reti di questo tipo il vantaggio è legato semplicemente all’numero di utenti a cui è possibile vendere il servizio, reti più grandi permettono un bacino di utenza maggiore.

#### Reti di comunicazioni e sociale (Metcafe e Reed): per reti di questo tipo il vantaggio invece è molto maggiore, infatti accorpando più reti il numero di comunicazioni che è possibile effettuare aumenta in modo notevole.

# Lezione 2, The Reference Model of Open Distributed Processing (RM-ODP) :

Con lo scopo di facilitare lo sviluppo dei sistemi distribuiti è stato creato dall’**ISO/IEC** un modello di riferimento che integra il modello a 7 Layer **ISO/OSI**, ponendo maggiore attenzione sul problema della **comunicazione** piuttosto che sulla **connessione.** Tale modello punta ad astrarre e standardizzare anche il concetto di **portabilità** e di **trasparenza** all’interno di un sistema distribuito.

## Caratteristiche di un sistema distribuito (Altissima probabilità di domanda):

#### Remoto: Le componenti di un sistema distribuito devono poter essere locali o remote, quindi anche potenzialmente localizzate su macchine diverse.

#### Concorrenza: Un sistema distribuito è per sua stessa natura concorrente, in quanto la contemporanea esecuzione di due (o più) istruzioni è possibile su macchine diverse e non esistono strumenti come lock e semafori.

#### **Assenza di uno stato globale**: Non esiste un modo per poter determinare lo stato globale del sistema, in quanto la distanza e l’eterogeneità (diversi clock) del sistema non permette di definire con certezza lo stato in cui si trova ciascun nodo. La mancanza di un clock globale rende impossibile riuscire a sincronizzare perfettamente gli orologi di tutti i processi e questo comporta l’impossibilità di ordinare, in modo preciso e univoco, tutti gli eventi che occorrono all’interno del sistema.

#### **Malfunzionamenti parziali**: Ogni componente di un sistema distribuito può smettere di funzionare in maniera indipendente dalle altre componenti e questo non inficia le funzionalità che sono localizzate su altre macchine.

#### **Eterogeneità**: Un sistema distribuito per sua stessa definizione è eterogeneo (formato da parti diverse) sia del punto di vista hardware che software.

#### **Autonomia**: Un sistema distribuito non ha un singolo punto dal quale esso puo’ essere controllato, coordinato e gestito. Quindi, la collaborazione va ottenuta mediando le richieste del sistema distribuito con quelle del sistema che gestisce ciascun nodo.

#### **Evoluzione**: un sistema distribuito può cambiare anche in modo sostanziale durante la sua vita, sia per motivi “ambientali” che tecnologici. La flessibilità di un sistema distribuito deve assicurare che la migrazione verso ambienti, tecnologie differenti e applicazioni nuove puo’ essere effettuata con successo e senza costi eccessivi.

#### **Mobilità**: Così come appare naturale che gli utenti siano mobili, altrettanto naturale deve essere la mobilità dei nodi e delle risorse (ad esempio, dati) all’interno del sistema in modo da poter adattare al meglio le prestazioni del sistema (il famoso Akamai).

## Requisiti non funzionali di un sistema distribuito (Altissima probabilità di domanda):

Sono dei requisiti che anche se non esprimono una funzione specifica del sistema, contribuiscono a garantire la qualità del sistema stesso. In particolare, i sistemi distribuiti devono essere:

#### Aperti: in modo da supportare la portabilità di esecuzione e l’interoperabilità (capacità di cooperare con altri sistemi) attraverso interfacce e servizi ben documentati e aderenti a standard noti e riconosciuti. In modo da permettere l’evoluzione e la possibilità di cambiare fornitore senza particolari rischi per la intera architettura.

#### **Integrati**: così da incorporare al proprio interno sistemi e risorse differenti senza dover utilizzare strumenti ad-hoc. Questo permette di trattare in maniera efficiente (economica) il problema della eterogeneità hardware e software.

#### **Flessibili**: Ovvero essere in grado di evolversi e avere la capacità di integrare sistemi legacy al loro interno.

#### **Modulari**: in modo da permettere ad ogni componente di poter essere autonoma e indipendente verso il resto del sistema.

#### **Supportino la federazione di sistemi**: in modo da unire diversi sistemi, dal punto di vista amministrativo oltre che architetturale, per lavorare e fornire servizi in maniera congiunta.

#### **Facilmente gestibili**: in modo da permettere il controllo, la gestione e la manutenzione per configurarne i servizi, la loro qualità (Quality of Service) e le politiche di accesso.

#### **Supporto per la qualità del servizio (Quality of Service):** per poter fornire i servizi con vincoli di tempo, di disponibilità e di affidabilità, anche in presenza di malfunzionamenti parziali.

#### **Scalabili**: perché qualsiasi sistema distribuito accessibile da Internet puo` essere soggetto a picchi di carico non prevedibili e deve essere in grado di gestirli.

#### **Sicuri**: utenti non autorizzati non possono accedere a dati sensibili.

#### **Trasparenti**: mascherando i dettagli e le differenze della architettura sottostante che assicura la distribuzione dei servizi sulle componenti del sistema.

## La trasparenza di un sistema distribuito

permette di nascondere dettagli implementativi in modo da **semplificare le operazioni di sviluppo**. Nonché permettere un **maggiore riuso delle applicazioni sviluppate**.

Tale operazione viene fatta attraverso il **Middleware.**

La trasparenza all’interno di un sistema distribuito viene fornita attraverso vari livelli tra loro interconnessi:

#### **Livello Base**

##### **Trasparenza di Accesso**: nasconde le differenze nella rappresentazione dei dati e nel meccanismo di invocazione per permettere la interoperabilità tra oggetti. Gli oggetti devono essere accessibili attraverso la stessa interfaccia, sia che siano acceduti da locale sia che siano acceduti da remoto.

##### **Trasparenza di Locazione**: non permette di utilizzare informazioni circa la localizzazione nel sistema (IP fisico) 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 (praticamente il DNS).

#### Livello di Funzionalità:

##### **Trasparenza di Migrazione (locazione/accesso):** Nasconde la possibilità del sistema di spostare oggetti o dati da un nodo all’altro della rete, continuando ad essere raggiungibile e utilizzabile da altri oggetti. In modo da ottimizzare le prestazioni (AKAMI e le CDL Content Delivery Network).

##### **Trasparenza di Replica:** Con questo tipo di trasparenza, il sistema maschera il fatto che una singola componente viene replicata in 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. Le repliche vengono utilizzate per ragioni di:

* + - **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.
    - **Scalabilità:** il sistema in presenza di aumento del carico di lavoro.

##### **Trasparenza alla Persistenza (Trasparenza/Locazione):** Questo tipo di trasparenza scherma l’utente dalle operazioni che compie il sistema per rendere persistente (cioè in memoria secondaria) un oggetto durante una fase di non utilizzo.

##### **Trasparenza alle Transazioni:** nasconde all’utente le attività di coordinamento che vengono svolte per assicurare la consistenza dello stato degli oggetti in presenza della concorrenza e nella gestione delle transazioni (ACID).

#### **Livello di Efficienza:** Scalabilità, prestazioni e malfunzionamenti.

##### **Trasparenza di Scalabilità (migrazione/replica):** assicura l’operabilità del sistema in caso di picchi di carico, senza dover modificare l’architettura e l’organizzazione. (differenza tra scalabilità verticale e orizzontale). Tale trasparenza è basata sulla trasparenza di replica e migrazione.

##### **Trasparenza alle Prestazioni:** Progettisti/sviluppatori ottengono alte prestazioni senza conosce i meccanismi utilizzati. Tale trasparenza viene ottenuta tramite il:

* + - **Bilanciamento del carico:** su più macchine (**Migrazione/Replica**);
    - **Minimizzazione della latenza:** tramite sistemi come Akamai (**Migrazione/Replica**);
    - **Ottimizzazione delle risorse:** spostando risorse tra le varie memorie e ottimizzandoli (**Persistenza**)

##### **Trasparenza ai malfunzionamenti:** in presenza di malfunzionamenti parziali il resto del sistema continua a funzionare, tale trasparenza è basata sulla trasparenza di replica ma anche sulla Trasparenza alle transazioni (facendo il rollback di eventuali transazioni iniziate ma non finite).

L’eccesso di trasparenza può portare a degli errori progettuali, quindi non tutti i modelli la utilizzano al 100%. Inoltre, implementare ogni tipo di trasparenza ha un costo non indifferente, per questo prima di procedere si procede ad un’attenta **analisi costi/benefici**. Anche se alcuni tipi di trasparenza (**Locazione** ed **Accesso**) sono ritenuti fondamentali e pertanto vengono sempre implementati.

**Senza lezione specifica:**

# Il Middleware: (Lo chiede in modo veloce)

nato intorno agli anni 60/70 per offrire soluzioni scalabili ed eterogenee. Serve a integrare sistemi eterogeni, esso è uno strato software che si trova tra applicazioni ed hardware. Esistono tre principali strati del Middleware:

## Middleware di infrastruttura: si occupa di comunicazione tra sistemi operativi diversi e gestione di concorrenza in maniere da essere portabile (per esempio la JVM).

## Middleware di distribuzione: automatizza compiti comuni per la comunicazione come:

#### Marshalling: invio di parametri per l’invocazione di servizi in remote.

#### **Multiplexing**: utilizzo dello stesso canale per più invocazioni/richieste.

#### **Gestione della semantica delle invocazioni**: unicast, multicast, attivazione on-demand (invocazioni a più oggetti).

#### **Riconoscimento e gestione dei malfunzionamenti di rete**.

#### **Middleware per servizi comuni:** Implementa in automatico servizi comuni, come persistenza, transazioni, sicurezza, accesso, etc… (**accesso tramite Google per esempio).**

La funzione principale del middleware è quella di nascondere dettagli inutili per la progettazione effettiva del software in modo che il programmatore possa concentrarsi solo sull’effettivo sviluppo del programma. La scelta del middleware è molto importante, perché si deve scegliere qualcosa che duri nel tempo e che sia usato, affidabile e supportato**.**

## L’evoluzione del Middleware:

deriva dal **Remote Procedure Call (RPC)**,creato dalla **Sun** negli anni 80, con l’obbiettivo di invocare una procedura in esecuzione su un’altra macchina. Utilizza il paradigma di **sincronia dell’invocazione forzata** ovvero il client veniva bloccato fino a quando il server non rispondeva alla richiesta.

L’RPC si basa sull’implementazione del **Client Stub** e **Server Stub** che si occupano di forzare il Marshalling per risolvere problemi di incompatibilità. Tali parti di codice, con il passare degli anni, vennero automatizzate attraverso un linguaggio specifico creato ad-hoc chiamato **Interface Definition Language (IDL).** Tale sistema viene ancora oggi utilizzato per parti fondamentali di internet come il **DHCP** e il **DNS.**

## Passaggio tra RPC e Modello ad Oggetti distribuiti:

All’inizio degli anni 90 il modello RPC viene esteso in maniera de permettere l’invocazione di oggetti in remoto, questo ha portato a una vera e propria trasformazione del modello e all’integrazione al suo interno del paradigma della programmazione ad oggetti.

## CORBA (Common Object Request Broker Application): (Lo chiede)

sviluppato nel 1991 è la prima proposta significativa di ambiente di programmazione basato su sistemi distribuiti. Si basa su servizi offerti dall’**Object Request Broker (ORB)** che astrae il meccanismo di invocazione in modo completamente trasparente al client. Con il passare degli anni a causa di problemi iniziali di progettazioni dovuti a un’implementazione di tipo **reference** (tutti i produttori seduti al tavolo che facevano ognuno i propri interessi) Corba iniziò a mostrare alcune mancanze come l’assenza di **interoperabilità** tra ORB diverse (ogni produttore modificava la sua).

**Microsof.NET:** sulla quale sono basati tutti gli operativi Microsoft, si basa sul **Common Language Runtime (CLR)** linguaggio che permette l’integrazione di tutti i linguaggi supportati da Microsoft. Utilizza una macchina virtuale per risolvere un problema di eterogeneità livello software.

**Enterprise Java:** nasce con l’obiettivo di limitare la complessità è basato su un modello a componenti, modello che viene gestito da una componente server si chiama **Container (risposta al 90% delle domande XD).** 

## Concetto di middleware implicito

si basa su corba, i **primi middleware erano espliciti** (bisognava invocare qualcosa per connettersi), questo portava a dei problemi come: la complessità, mancanza di controllo sugli errori, mancanza di portabilità. Nel MW implicito servizi vengono forniti automaticamente dal sistema sulla base delle richieste specificate durante il **deployment,** dico al MW quali servizi voglio tramite un file **XML** e lui automaticamente li implementa.

# Lezione 3, Programmazione concorrente e Thread in Java

il nostro sistema è formato da tante componenti diverse, questo lo rende implicitamente concorrente.

## Motivazioni

La motivazione principale dietro la nascita dei **Thread** è l’effetto di **Thermal Noise**, tale effetto disturba la crescita secondo la legge di **Moore** del numero di transistor. Praticamente:

***Non si possono avere “tanti” transistor su un processore, che siano anche “facili da raffreddare” e che siano “veloci”: si deve rinunciare ad una di queste caratteristiche.***

Questo ha portato a un radicale cambiamento per quanto riguarda la progettazione di CPU, passando da un’architettura a **Single Core** ad una a **Multi Core.**

Questo cambiamento ha cambiato il modo in cui i programmatori intendessero il loro lavoro, infatti fino a poco tempo fa i miglioramenti della tecnologia comportavano un automatico miglioramento delle prestazioni software (CPU con clock maggiore eseguivano il codice con più velocità). Oggi invece un programma per essere più efficiente deve essere in grado di sfruttare al massimo il **Parallelismo** **delle applicazioni.**

## Le sfide

questo nuovo paradigma di programmazione ci porta inevitabilmente a scontrarci con sfide di natura “gestionale” in quanto avendo più processori che accedono e lavorano contemporaneamente con una singola memoria condivisa si vanno a creare tutta una serie di problemi di concorrenza e di sincronia.

## Cosa ci riserva il futuro

Architettura **master/slave** IBM, un core potente e tanti piccoli “schiavi” che eseguono operazioni più semplici.

## Diversi tipo di programmazione concorrente

tipo di programmazione che coinvolge diversi processi che vengono eseguiti concorrentemente:

#### **Eseguita su calcolatori diversi**

#### **Processi concorrenti sulla stessa macchina (multitasking)**

#### **Programmazione concorrente nello stesso processo** (“processi lightweight” all’interno del processo: Thread): strumento efficace che permette di costruire programmi che eseguono più operazioni contemporaneamente.

## Apache

è server molto famoso, nonché un ottimo esempio di applicazione **Multithread.** Per aumentare le performance Apache all’avvio instanzia un certo numero di thread che tiene in uno stato di **sospeso** pronti ad essere risvegliati ed usati.

## I Thread: (Lo chiede in modo veloce)

I thread sono processi all’interno dei processi, vengono anche detti **“processi leggeri”** (in quanto condivide molte caratteristiche con un processo, ma “leggero” sta a rappresentare che l’implementazione di un thread è molto meno onerosa di quella di un processo)e offrono diversi vantaggi, essi sono infatti più facili e veloci da creare rispetto al processo che li “ospita” e condividono con quest’ultimo risorse e memoria, cosa che rende estremamente efficiente la comunicazione a discapito di alcuni problemi di concorrenza.

Il concetto di processo è associato (ma comunque distinto) dal concetto di thread. Un thread è una parte di processo che viene eseguita in maniera concorrente ed indipendente all’interno di un processo (un processo è formato da uno o più thread). Una sostanziale differenza tra i due è che i processi solitamente sono indipendenti (non condividono aree di memoria e comunicano tra loro solo attraverso appositi meccanismi messi a disposizione dal sistema), mentre i thread condividono le informazioni di stato, la memoria e altre risorse di sistema. Un’altra sostanziale differenza è il costo per la creazione: la creazione di un processo è onerosa in quanto devono essere allocate nuove risorse necessarie alla sua esecuzione, mentre un thread essendo parte di un processo prevede una creazione in tempi ridotti e a costi minimi

In java un thread può essere creato in due modi:

* **Instanziando l’oggetto thread:** soluzione facile solo che ci vincola a non poter estendere altre classi, visto che java non permette l’ereditarietà multipla ed è necessario estendere la classe Thread.
* **Estrarre la gestione, passando un task ad un executor:** tramite l’interfaccia Runnable (metodo consigliato).

Un thread pool in programma indica un gestore software di thread utilizzato per ottimizzare e semplificare l’utilizzo dei thread all’interno di un programma. Una implementazione di thread pool è composta di solito da una coda di task e da un insieme di thread, entrambi inizialmente vuoti.

## Alcuni metodi della classe Thread

**.sleep(Int t):** Sospende (stato di **Wait**) l’esecuzione di un thread per un tot di millisecondi, Se il thread in pausa viene interrotto viene lanciata l’eccezione **InterruptException**.

**.interrupt():** Se lo stato del thread è su **wait**, questo metodo invia un **InterruptException**, se invece il thread si trova in uno stato diverso da wait viene interrotto e il flag **interrupt** viene settato a true.

.**Interrupted():** controlla lo stato del flag **interrupt** è utile per gestire le eccezioni.

**.join(int t):** Obbliga gli altri thread ad attendere il completamento del thread che invoca tale metodo prima di continuare. E’ inoltre possibile passare un tempo massimo in millisecondi di attesa, prima che gli altri thread continuino la propria esecuzione.

**.inAlive():** Controlla che un thread sia ancora in esecuzione.

**.getState():** Restituisce lo stato di un thread (runnable, terminated, ecc.…).

**.yield():** Cambia lo stato di un thread da **run** a **ready**.

**.start()** e **.run():** Start porta il thread in uno stato di ready, run invece contiene il codice che il thread dovrà eseguire una volta nello stato di running.

Il thread è nello stato New quando viene creato

Quando viene chiamato il metodo .start() il thread diventa eseguibile (Ready), con il metodo .run() viene eseguito (Running).

Nel Timed Waiting il thread viene messo in attesa, nel Waiting viene messo in attesa per un tempo indefinito. Dopo questi 2 stati il thread non viene eseguito ma torna solamente eseguibile.

Lo stato di Blocked non va a Terminated(ha completato la sua esecuzione) perché un thread bloccato non può morire a meno che non arrestiamo il programma.

## Diagramma degli stati di un thread

## Problemi dei thread

I thread basano la loro strategia di comunicazione sulla condivisione della memoria, questo tipo di tecnica è estremamente efficiente ma genera alcuni problemi come:

#### **L’interferenza fra thread:** si ha quando più thread lavorano su una stessa variabile ed eseguendo le loro operazioni contemporaneamente si interfogliano.

#### **Inconsistenza della memoria:** si ha quando thread diversi hanno visioni diverse di dei dati presenti in memoria. Tale problema può essere risolto creando una relazione happens-before (succede-prima, anche questo si ottiene con la sincronizzazione).

Entrambi questi problemi possono essere risolti tramite la **sincronizzazione** questo tipo di soluzione porta a sua volta a problemi di **Race Condition** (il risultato di un’operazione dipende dall’ordine di esecuzione dei diversi thread) questo a sua volta porta a problemi di **Liveness** come il:

#### **Deadlock**: Quando due thread si attendono a vicenda per un tempo infinito senza fare nulla. Questo problema in Java si risolve tramite l’uso del metodo bow (inchino).

#### **Livelock**: Quando due thread si attendono a vicenda compiendo in continuazione azioni che non portano a nulla.

#### **Starvation**: Quando un thread in coda attende per un tempo infinito il suo turno, senza che gli venga mai concesso.

#### **Variabile volatile:** è una variabile utilizzata per rendere l’accesso ad una variabile asincrono in modo da risolvere il problema dell’inconsistenza della memoria (operazione atomica), ossia quando un thread modifica una variabile, tutti gli altri thread vedono immediatamente il nuovo valore.

# Lezione 4, Programmazione concorrente e Thread parte 2:

## La legge di Amdahl

È una legge sull’utilità del calcolo concorrente, definisce il vantaggio che si ha se il programma è sviluppato ad-hoc per sfruttare la programmazione multithread.

## Lo Speedrun

è il tempo impiegato per svolgere un’operazione **X** con **N** processori dove **P** è la parte del programma che è possibile parallelizzare.

Questa legge ci dice che la parte sequenziale di un programma rallenta molto qualsiasi Speedrun speriamo di ottenere. Quindi è inutile spendere molti soldi in processori, conviene invece rendere preponderante la parte del programma parallelizzabile.

## Gli strumenti per la sincronizzazione dei thread sono un trade-off (o uno o l’altro) tra:

* **Facilità d’uso:** non tutti sono semplici da usare;
* **Efficienza:** possono creare colli di bottiglia e rallentare di molto l’esecuzione.

## Metodi sincronizzati

Permettono di risolvere facilmente gli errori di concorrenza **sacrificando efficienza**. Si usa la keyword **synchronized**  in questo modo l’esecuzione della porzione di codice successiva viene eseguita in mutua esclusione. Quando un thread si trova all’interno di una porzione di codice sincronizzata gli altri thread rimangono sospesi fino a quando il primo non ha completato. Questo crea una relazione **happens-before** e quindi risolve anche il problema **d’inconsistenza della memoria**. (I costruttori non posso essere sincronizzati, perché non servono).

## Problema del leaking

Nasce dal fatto che non si può sincronizzare il costruttore. Praticamente usiamo il riferimento all’oggetto prima che quest’ultimo sia effettivamente costruito.

## Lock Intriseco

È un’entità associata ad ogni oggetto, garantisce sia la **mutua esclusione** che il **happens-before**. Praticamente il thread prende il lock dell’oggetto che deve usare, lo usa e poi lo rilascia. Si usa **synchronized** su piccole porzioni di codice. Questo tipo di sincronizzazione viene detta “**a grana fine”**. **(Amdahl è felice)**

## Sincronizzazione dei metodi statici

Un metodo statico **synchronized** previene l’esecuzione interfogliata di tutti gli altri metodi statici sincronizzati. In pratica, si acquisisce il lock dell’oggetto **ClassName.class.**

**Attenzione: Metodi sincronizzati statici garantiscono accesso in mutua esclusione a metodi sincronizzati statici, mentre metodi sincronizzati di istanza garantiscono accesso in mutua esclusione ai metodi sincronizzati di quella istanza.**

**Metodi atomici:** i processori offrono la possibilità di eseguire operazioni non interrompibili (o si eseguono completamente o non si eseguono proprio). In java la cosa si implementa tramite l’uso di variabili **volatile**.

# Lezione 5, Singleton e DoubleChecked Locking:

## Il Singleton: (Lo chiede in modo veloce)

è un pattern utilizzato per limitare le istanziazioni possibili di una classe ad una sola, senza affidare questa limitazione al programmatore. Molto utile quando si devono utilizzare delle risorse I/O (stampanti, gestori di file, ecc…). L’utilizzo dei **singleton** non è sempre necessario, in quanto java istanzia un oggetto solo quando viene usato la prima volta (**Lazy Allocation**)**.** La situazione si complica invece con i thread dove potrebbero esserci problemi di **leaking**,che potrebbero portare alla creazione di due istanze. Per risolvere questo problema esistono due soluzioni possibili:

#### **Usare le synchronized:** questo metodo funziona solo in parte, eseguendo il metodo getInstance() sincronizzato il problema si risolve solo per una volta, successivamente tutti i thread possono accedervi contemporaneamente.

#### **Double e-checking lock**: Un'altra soluzione sta nell’effettuare un controllo sulla variabile di istanza per vedere se è stata inizializzata anche se così non è detto che funzioni al 100% perché la creazione del singleton può non essere ancora stata completata prima che un altro thread utilizzi il riferimento.

#### **La soluzione definitiva** sta nel rendere la variabile **Instance,** **volatile.** Altrimenti, si possono usare le classi statiche con l’idioma **“Initialization-on-demand holder”** che essendo un metodo statico, viene eseguito una sola volta (al caricamento) e stabilisce una relazione **happens-before**.

# Lezione 7, Socket TCP

Nel paradigma della programmazione ad oggetti la computazione avviene tramite un insieme di oggetti che mantengono degli stati precisi ed espongono i loro comportamenti (**metodi**). Questo tipo di comunicazione avviene di solito in modo locale, mentre per la programmazione distribuita abbiamo la necessità che tutto questo si svolga tra oggetti che non risiedono sulla stessa macchina (**Java Remote Method Invocation)**.

## I Socket

Sono degli **endpoint** di una comunicazione bidirezionale sulla rete che unisce due programmi. Ad ogni socket viene associato un **indirizzo IP** (identifica la macchina) e un **numero di porta** che indentifica il processo (programma) a cui il socket fa riferimento. All’interno del socket le due macchine coinvolte vengono dette **client** e **server.** Il procedimento di comunicazione prevede che:

1. Il **client richieda** la comunicazione al server (di cui conosce indirizzo e porta).
2. Il **server accetti** la comunicazione e istanzi un **socket di comunicazione** (sotto forma di thread) sulla porta fornitegli dal client.

Le **API** java implementano due principali tipi di socket:

#### **Socket:** socket standard per comunicazioni.

#### **Server-socket:** svolge il ruolo appunto di server, istanziando un socket privato per ogni nuova richiesta da parte di un client.

**Serializzazione:** è un processo per salvare un oggetto in un supporto di memorizzazione o per trasmetterlo su una connessione di rete. La serializzazione può essere in forma binaria o può utilizzare codifiche testuali (ad esempio il formato XML). Lo scopo della serializzazione è di trasmettere l'intero stato dell'oggetto in modo che esso possa essere successivamente ricreato nello stesso identico stato dal processo inverso, chiamato **deserializzazione**.

* **private static final long serialVersionUID = -414713378645982122L:** l’algoritmo di serializzazione usa questo numero per verificare che la classe appena caricata corrisponde ad un oggetto serializzato, **è buona norma utilizzare questa variabile sempre**.

## Gli Stream

la comunicazione tra client e server avviene tramite degli **stream** o flussi attraverso un meccanismo detto di **serializzazione.** Java offre una vasta gamma di interfacce e di classi per gestire i flussi.

## Metodi della classe socket

#### **accept():** invocato su oggetti di tipo ServerSocket è un metodo bloccante, l’esecuzione si blocca fino a quando non si collega un client, a questo punto il metodo restituisce un socket che non è altro del socket di comunicazione (classe Socket).

#### **close():** chiude la connessione e il socket.

#### **setSocketTimeout():** se entro un certo numero di millisecondi il server non riceve accept chiude le comunicazioni.

# Lezione 8, Socket TCP 2 (with Curly):

## Conversione da un oggetto locale ad un oggetto remoto:

per far ciò si aggiunge uno strato software composto da due componenti:

#### **Stub**: è un oggetto che si trova nel client è che svolge il compito di rappresentare il server per quest’ultimo, esso infatti espone gli stessi metodi implementati dal server in modo che il client li possa usare.

#### **Skeleton** (o client stub): è un oggetto che si trova nel server con lo scopo di comunicare a quest’ultimo le richieste di metodi che gli giungono dallo stub e di ritrasmettere i valori di ritorno provenienti dal server a quest’ultimo in modo che lo stub li possa a sua volta trasmettere al client.

Per garantire allo stub l’accesso ai metodi del server, sia **stub** che **skeleton**, implementano un’**interfaccia remota**.

## Problema dell’indirizzamento

quando non è possibile indirizzare un oggetto impiegato in modo diretto, viene implementato un servizio di naming ad un indirizzo ben noto, che si occupa di mantenere l’associazione tra l’ID dell’oggetto e l’indirizzo verso il quale deve essere aperto lo skeleton.

# Lezione 10/11, java RMI (Java Remote Method Invocation):

È una libreria di integrazione java creata da Jim Waldo, che offre la possibilità di invocare metodi appartenenti ad un oggetto remoto (non presente nella stessa JVM, i cui metodi sono esposti e utilizzabili tramite un’interfaccia), con un’architettura di solito basata su un **client** e un **server** (non è obbligatorio si può anche utilizzare un’architettura **peer to peer**). Il progetto nacque con due obiettivi principali:

* **Semplicità nella progettazione:** in modo da ridurre errori e abusi, favorendone allo stesso tempo la diffusione.
* **Integrazione con il linguaggio:** in modo da far sentire i programmatori a loro agio in un ambiente già conosciuto.Tale integrazione arriva al punto da rendere disponibile un **Garbage collector distribuito** (evita **memory-leak** e **grafi a dente di sega**)**,** in modo da rendere autonoma la gestione della memoria anche quando abbiamo a che fare con programmi distribuiti.

Altri obiettivi:

* **Invocazione trasparente di metodi remoti:** con lo scopo di offrire un meccanismo semplice e “trasparente” per invocare metodi remoti.
* **Non-trasparenza della natura locale/remota di un oggetto:** anche se questo obiettivo va contro il concetto di “trasparenza” è importante rendere visibile al programmatore la differenza tra oggetto remoto e oggetto locale.
* **Rendere minima la complessità del client e server:** in modo compatibile con gli altri obiettivi.
* **Preservare la sicurezza fornita da java:** Una delle sue caratteristiche principali sin dalla sua pubblicazione.
* **Modalità di invocazione:** deve prevedere che esistano diversi tipi di invocazioni, fornendo sia quello di tipo **unicast** che quella di tipo **multicast**. Deve inoltre permettere che l’oggetto server sia attivato anche solo al momento dell’invocazione.
* **Livelli di trasporto multipli:** deve essere aperto a modifiche future del protocollo di trasporto basato sui **socket.**

## Java RMI è costituito da 5 package:

* **Java.rmi e java.rmi.server:** meccanismi basilari per le invocazioni remote;
* **Java.rmi.activation:** per oggetti attivabili;
* **Java.rmi.dgc:** per il **Distributed Garbage Collection**;
* **Java.rmi.registry:** per il servizio di localizzazione.

## Interfacce ed eccezioni remote

Prima di definire un oggetto remoto è necessario creare un’interfaccia **remota** che estenda i metodi dello stesso, tale interfaccia deve implementare l’interfaccia **java.rmi.Remote** che è un’interfaccia **Marker** ovvero vuota, che serve solo a segnalare che l’interfaccia che stiamo andando a creare, definisce metodi accessibili da remoto. Ogni metodo che andiamo a dichiarare in un’interfaccia remota, deve a sua volta essere un metodo remoto e deve quindi soddisfare delle caratteristiche:

#### **Dichiarare l’eccezione java.rmi.RemoteException**: è una Checked-Exception (viene controllata dal compilatore e vista come un errore se non c’è). Questo con lo scopo di rendere il programmatore cosciente che si sta andando a lavorare con un oggetto remoto che ha tutta una serie di problematiche diverse rispetto ad un oggetto locale.

#### **Parametri dichiarati attraverso la propria interfaccia remota.**

Riassumendo, l’interfaccia remota definisce un livello ulteriore di accessibilità (oltre a public, protected, etc…), i metodi remoti infatti, sono più accessibili dei metodi public.

## Meccanismo di invocazione remota:

#### **Riferimenti remoti**: nel modello distribuito il client interagisce con il server tramite uno stub che rappresenta l’interfaccia remota dell’oggetto remoto in locale. Dal punto di vista della JVM infatti il tipo dello stub è uguale al tipo del server. Quindi un client può accedere tradizionalmente al tipo di un oggetto remoto controllando quale interfaccia remota implementa, attraverso instaceOf.

#### **Localizzazione e invocazione di oggetti remoti**: per poter invocare il metodo remoto, il client deve avere a disposizione il riferimento remoto. Questo può essere reperito in due modi:

##### **Ottenendo il riferimento come risultato di altre invocazioni (remote o locali) di metodi**;

##### **Attraverso un servizio di directory:** ovvero tramite un name-server che Java RMI fornisce nella classe java.rmi.Naming, tale classe permette di gestire riferimenti remoti ad oggetti tramite un ID (stringa), oltre a fornire metodi per:

* 1. **Ricerca:** lookup();
  2. **Registrare:** bind(), unbind(), rebind();
  3. **Elencare:** list(), gli identificativi registrati e contenuti all’interno del name-server.

Come abbiamo visto in precedenza l’invocazione dei metodi remoti avviene allo stesso modo dei metodi locali, con l’unica eccezione della gestione della **RemoteException**, di cui il client non sa nulla, tranne per il fatto che si è verificato un problema in un punto qualsiasi della comunicazione. Per questo motivo per garantire la **semantica delle operazioni** i metodi remoti devono essere **idempotenti,** ovvero che a ripetute applicazioni dello stesso metodo con gli stessi parametri devono restituire sempre lo stesso risultato.

## Passaggio di parametri

Un metodo remoto può dichiarare solo parametri o valori che siano **serializzabili**. Inoltre, il passaggio di dati a oggetti remoti avviene per **copia** e quindi non per **riferimento** come invece avviene per oggetti locali. Questo porta a dei problemi che Java RMI risolve garantendo che nel caso in cui **vengano passati, nella stessa invocazione due riferimenti allo stesso oggetto**, allora punteranno allo stesso oggetto, mantenendo così **l’integrità referenziale.**

## Differenza tra il modello a oggetti locale e quello remoto

Sono state apportate alcune modifiche agli oggetti della classe **Object:**

* **X.hashCode():** viene ridefinito in maniera che restituisca lo stesso codice per due stub diversi gli oggetti remoti che si riferiscono allo stesso oggetto remoto. In questo modo le chiavi possono essere utilizzate in tabelle hash.
* **X.equals():** restituisce un booleano che è vero se il riferimento remoto passato è uguale a quello di X. Il confronto viene effettuato sugli **stub.**
* **X.toString():** Restituisce informazioni addizionali come la macchina su cui l’oggetto si trova.

## La sicurezza in java

Per gestire la sicurezza java offre una **sandbox** (zona sicura), per garantire che le azioni svolte dai programmi siano controllate e ristrette, cosi di prevenire danni sia di natura intenzionale che non. La sicurezza java:

#### **La sicurezza intrinseca di java:** ovvero tutte quelle caratteristiche del linguaggio che garantiscono la sicurezza:

##### **Linguaggio fortemente tipizzato**: ogni dato ha un tipo, e solo poche operazioni di casting vengono eseguite automaticamente dal compilatore.

##### **Gestione automatica della memoria**: tramite il garbage collector.

##### **Assenza di puntatori**: insieme all’impossibilità di effettuare aritmetica sugli stessi, impedisce accessi illegali alla memoria.

##### **Lazy allocation**: la memoria del programma viene allocata a tempo di esecuzione, questo rende impossibile sapere in anticipo dove il programma andrà a finire e quindi impossibile rimpiazzare quelle parti di memoria con software malevolo.

#### **Il Classloader**: carica le classi a tempo di esecuzione in modo da non permettere modifiche alle classi bult-in del linguaggio.

#### **Il Bytecode verifier**: controlla l’assenza di codice malevolo all’interno del bytecode e ne verifica la conformità con gli standard del linguaggio, questo è necessario perché il bytecode una volta generato è liberamente accessibile e quindi potrebbe essere manomesso.

#### **Il Security Manager**: si occupa di definire i confini della sandbox. Viene interpellato dalla macchina virtuale per ciascuna operazione potenzialmente pericolosa e fornisce le autorizzazioni sulla base delle policy stabilite dall’utente.

## I tre layer della architettura:

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

#### **Stub/Skeleton layer**: che comprende gli stub lato cliente gli skeleton lato server. Con le versioni più moderne di java, questi due componenti vengono creati automaticamente, semplificando di molto la vita di programmatori al netto di una piccola perdita di efficienza.

#### **Remote Reference Layer:** che specifica il comportamento dell’invocazione (unicast, multicast), in particolare, questo layer si occupa di fare da tramite tra il livello stub/skeleton e il livello di trasporto.

#### **Trasport Layer:** che si occupa della connessione e della sua gestione. Chiamato **Java Remote Method Control**, ma con il tempo questo protocollo e stato sostituito o comunque reso modificabile per garantire una migliore compatibilità di **Java RMI**.

L’applicazione dell’utente si trova in cima a questi tre livelli.

## Marshalling

È un’operazione di serializzazione che modifica la semantica dei riferimenti remoti, aggiungendo informazioni all’oggetto. Il meccanismo di Marshalling di java si basa sulla specializzazione della classe **ObjectOutputStream.**

# Lezione 13, Introduzione a Java EE (Java Enterprise Edition):

**Java EE:** Nasce dalla necessità degli sviluppatori di lavorare con applicazioni distribuite, transazionali e portatili che sfruttino la velocità, la sicurezza e l'affidabilità della tecnologia lato server, con lo scopo di ridurre i tempi di sviluppo, il capitale investito e garantire la massima efficienza del prodotto creato.

Java EE, utilizza un modello di applicazione distribuito su più livelli.

## I componenti di livello client vengono eseguiti sul computer client e sono di due tipi:

#### **Client Web:** semplice pagina web che funge da UI, molto utile visto che tutti hanno un browser.

#### **Client applicativo:** applicazione software installabile, che viene utilizzata in caso di necessità di un’interfaccia più ricca.

## I componenti di livello Web vengono eseguiti sul server Java EE.

#### I **componenti di livello aziendale** eseguono la logica di business e si basano sull’utilizzo di java Beans.

#### Il **software EIS** (Enterprise Information System) funziona sul server EIS.

Molto importante è anche l’organizzazione in domini logici detti **container** ognuno con uno specifico ruolo, che contiene **componenti** e offre alcuni servizi. Questi container sono in grado di comunicare fra di loro tramite dei protocolli ben definiti.

La logica di Java EE si basa sulle componenti, che sono la naturale evoluzione degli oggetti (non contengono solo il software, ma anche le classi e i file necessari, per esempio il **deployment descriptor**). In Java EE sono definiti 4 tipi di componenti:

* **Le applet:** applicazioni che vengono eseguite all’interno di browser web e utilizzate per lo sviluppo di UI.
* **Applicazioni:** sono programmi eseguiti su un client. Sono in genere GUI o programmi di elaborazione batch.
* **Le applicazioni Web**: (composte da servlet, filtri servlet, listener di eventi Web, pagine JSP e JSF) sono eseguite in un contenitore Web e rispondono alle richieste HTTP dai client Web.
* **Applicazioni aziendali**: Implementano la logica di business e sono: Enterprise Java Beans, Java Message Service, Java Transaction API, chiamate asincrone, servizio timer, RMI / IIOP e vengono eseguiti in un contenitore EJB.

Tutto questo viene assemblato e gestito da un server Java EE. Infatti, in JEE è presente un ulteriore fase oltre la compilazione e l’esecuzione, questa fase detta **Packaging** è divisa a sua volta in due fasi **Assemblaggio** e **Deployment** in cui le componenti utili al programma (e quindi all’esecuzione) vengono prima unite fra di loro e successivamente affidate ad un container che le prende in carico e le esegue.

## Packaging

Per essere distribuite in un contenitore, le componenti devono prima essere impacchettate in un archivio formattato standard. In Java EE questo tipo di archivio e chiamato **Java Archive (jar)** ma ce ne sono anche altri (per esempio **Web Archive (war)).** All’interno del package sono contenute anche informazioni sugli oggetti e i file, chiamati **descrittori (**di solito queste informazioni vengono contenute in dei file **XML).**

## Java EE Container

Normalmente, le applicazioni multilivello web/client sono difficili da scrivere perché coinvolgono molte righe di codice intricato per gestire vari aspetti come: la gestione delle transazioni e dello stato, il multithreading, il pool di risorse e altri dettagli complessi di basso livello.

Java EE semplifica la scrittura delle applicazioni poiché la logica aziendale è organizzata in componenti riutilizzabili. Inoltre, il server Java EE fornisce servizi sottostanti sotto forma di **container** per ogni tipo di componente. Poiché non è necessario sviluppare questi servizi, si è liberi di concentrarsi sulla risoluzione del problema aziendale a portata di mano.

I **container** si basano su macchine virtuali standard edition e sul un grande numero di operazioni che quest’ultima riesce a realizzare. Vengono utilizzati sia sul server che sul lato client, dove rendono facile la gestione dell’applicazione verso servizi esterni.

## Ciclo di vita degli Enterprise java Beans: (Ci boccia la gente con sta cosa)

Viene completamente gestito dal container. Di norma gli oggetti nascono con una **new** e vengono distrutti quando non hanno più riferimenti a loro stessi e vengono gestiti dal garbage collector. In java EE invece il ciclo di vita viene gestito interamente (dalla **nascita** alla **distruzione**) dal container, infatti quest’ultimo istanzia l’oggetto quando serve e lo elimina quando diventa inutile (sostituendo completamente il garbage collector). Con questo metodo gli oggetti possono essere addirittura riutilizzati per altri scopi, per esempio se un bean diventa inutile e sta per essere distrutto mentre uno nuovo sta per essere creato, si riutilizza il vecchio sovrascrivendolo. Questo porta ad un enorme incremento delle performance.

## Annotazioni e Deployment Descriptors

Per dire di quali applicazioni, oggetti o classi si ha bisogno ci sono due metodi: usare le annotazioni, o utilizzare un **Deployment Descriptor (il file XML dei package).** In questo modo un **POJO (Plain Old Java Object)** puòdiventare un componente (uno degli obbiettivi di JEE).

#### **Non ci sono import**: il container sa di cosa ha necessità e cosa deve importare.

#### **Stateless**: significa che l’oggetto è composto solo da metodi, per questo può essere utilizzato da tutti i programmi che hanno necessità di usare quei metodi.

#### **Remote e Local**: significa che il bean è sia remoto che locale.

#### **PersistenceContext**: è l’insieme delle componenti che vanno rese persistenti.

**Extra:** Java EE si basa su standard, viene infatti chiamato **specifica ombrello** perché raggruppa un numerodi altre specifiche (o richieste di specifiche Java).

La piattaforma Java EE viene sviluppata tramite il **Java Community Process** (**JCP)**, ovvero tramite un gruppo di esperti (è la community stessa a decidere cosa implementare di Java EE e le aziende che hanno interessi diretti), le versioni inoltre vengono rilasciate con una cadenza molto lenta, in modo da permettere alle aziende di rimanere al passo. Java EE, **fornisce un ambiente aperto**, ovvero non ti blocca sull’utilizzo di software da parte di un singolo fornitore.

# Lezione 14-15, Context and Dependency Injection:

## Inversion control

si tratta di un design pattern utilizzato nella programmazione enterprise: il controllo del codice non è del programmatore ma del container, che offre servizi al codice, ed è quest’ultimo che decide quando istanziare un oggetto (**dependecy injection**) o fare un **contest injection**,ovvero il programmatore non è solo nello sviluppo, ma ha il container che lo “aiuta”. Questo è possibile tramite i **managed beans** ovvero dei bean creati per essere gestiti dal container.

## loose coupling e strong typing

tutto è tipizzato e niente viene convertito dal compilatore se non è esplicitamente richiesto tramite il casting, questo è un vantaggio per quanto riguarda la correttezza del codice. Allo stesso tempo la gestione e le esecuzioni delle classi non è vincolata da altre classi abbiamo un **basso accoppiamento.**

Ci permette, inoltre di inserire diverse componenti che vengono eseguite all’interno del programma tra cui:

* Interceptors;
* Decorator;
* Gestore degli eventi.

## Principio di Hollywood

quando vai a fare un colloquio come attore ti dicono “ti facciamo sapere”. Praticamente è il container che decide quando iniettare la risorsa, non la risorsa (l’attore).

## Pattern di dependecy injection

È una parte dell’**inversion of control e** il concetto è di richiedere l’inserimento di un oggetto di un certo tipo. Praticamente metto un salvaposto e poi il container inietta la risorsa da me richiesta.

## Managed Beans

Sono dei bean java progettati per essere gestiti dal container, con queste caratteristiche:

* Non è una **non-static inner class;**
* È una classe concreta o annotata come decoratore;
* Ha uno scope, un EL name, un insieme di Interceptor e un gestore del ciclo di vita (queste sono caratteristiche opzionali).
* Costruttore di default che non deve essere final;
* Attributi privati accessibili tramite getter e setter;
* Contengono metadati;

## Vantaggi dei managed beans:

#### **Non devo gestire lo stato di uno oggetto**: ci pensa il container;

#### **Sono legati a contesti ben definiti**;

#### **Hanno un approccio type safe**: sto accedendo a delle risorse iniettate con una precisa struttura derivata dalla classe java;

#### **Possono essere specializzati;**

#### **Possono essere usati nel layer di presentazione;**

#### **Ogni classe java può essere una bean CDI** (Content Dependence Injection);

## Ciclo di vita dei managed bean

si divide in quattro fasi gestite dal container:

#### **New Instance**: prima di questa fase il bean non esiste, il container crea un oggetto tramite new;

#### **Dependecy injection**: inietta le dipendenze (anche il bean può avere delle dipendenze);

#### **Post contruct**: eseguo i metodi @PostCostruct e mi trovo nello stato di ready dove il bean è pronto a rispondere alle chiamate fino a quando sarà necessario;

#### **Pre destroy**: una volta che il bean diventa inutile vengono eseguiti i metodi annottati come @PreDestroy e il bean torna allo stato di inesistenza.

Cosa molto importante è che per chi ha richiesto l’iniezione di questa risorsa queste fasi sono del tutto **trasparenti**, vengono eseguite tutte dal container.

## Scope

È il contesto del bean, ovvero descrive che tipo di vita ha il bean, ed è importante perchè permetterà al container di ottimizzare la vita di quest’ultimo.

## Interceptor

Si sovrappongono nelle invocazioni di un metodo di business. Viene inserito un metodo all’interno di un metodo, senza modificare il codice sorgente.

Vengono molto utilizzati nella **programmazione legata agli aspetti**. Se però i metodi sono tanti, andranno modificati tutti, e questo è un procedimento poco veloce.

La soluzione è far intercettare ognuno di questi metodi da un altro. In pratica **modifico la maniera in cui il container gestisce l’invocazione** dei metodi in modo che aggiunga delle caratteristiche a tutti o ad alcuni di questi metodi, senza modificare il codice sorgente, semplicemente modificando il codice XML, addirittura non serve ricompilarlo e possono essere create anche **catene di intercettori** per eseguire più operazioni.

La potenza è poter disaccoppiare la logica di business (quello che il metodo deve fare), dalle quello che fanno gli Interceptor, mantenendo però il controllo della tipizzazione.

## @Inject

È un’annotazione di iniezione che permette di iniettare codice all’interno del nostro. Il punto in cui si esegue l’iniezione viene detto **Injection Point.** In Java EE esistono tre meccanismi di iniezione:

#### **Property**: Quando viene eseguita sugli attributi;

#### **Contructor**: Quando l’iniezione viene eseguita sul costruttore;

#### **Setter**: Quando l’iniezione viene eseguita sul setter;

È importante notare che ognuno di questi metodi funziona e che la scelta fra questi va puramente in base ai gusti del programmatore.

## Default Injection: se non dichiariamo nessun Qualifier, Java EE imposta automaticamente quello di Default.

Se invece ci troviamo ad avere più risorse da iniettare, si utilizzano i **Qualifier.**

## Qualifier: è un meccanismo che permette di informare il container su quale classe iniettare utilizzando il sistema delle annotazioni (evitando l’uso di XML).

Questo tipo di sistema offre numerosi vantaggi, infatti posso rinominare la mia implementazione come voglio, rinominando i Qualifier, senza che il punto di iniezione cambi (questo garantisce il **basso accoppiamento**).

## Qualificatori con membri: per evitare di creare un grande numero di qualificatori all’interno del nostro codice è possibile “personalizzarli” inserendo dei membri, in questo modo abbiamo un unico qualificatore che agisce per ognuno dei nostri casi.

## Qualificatori Multipli: un altro metodo per ridurre il numero di qualificatori usati.

## Alternatives: a volte può essere necessario scegliere quale risorsa iniettare in base a un determinato scenario di utilizzo. Per fare ciò ci vengono in aiuto le alternative, che sono disattivate di default e nel caso in cui volessimo usarle devono essere attivate.

## Producers: di norma all’interno di Java EE non è possibile iniettare tipi primitivi o POJO (java.util.data, java.util.String, ecc) perché sono impacchettati in file .jar e questo tipo di archivio non contiene deployment descriptor (beans.XML) come invece lo hanno i beans CDI. Quindi per iniettare questo tipo di oggetti dobbiamo utilizzare i produttori.

## Disposer: alcuni tipo di oggetti e tipi di dati, una volta creati ed utilizzati necessitano di essere distrutti, per fare ciò si utilizzano i disposer. Praticamente con il producer creaiamo e con il disposer ne gestiamo la distruzione o la chiusura.

## Scopes:

Servono a gestire il ciclo di vita dei beans.

#### **Application Scope (@ApplicationScoped):** si estende per l'intera durata di un'applicazione. Il bean viene creato una sola volta per la durata dell'applicazione e viene eliminato quando l'applicazione è chiusa (sono bean di supporto per gestire o spostare dati).

#### **Session Scope (@SessionScoped):** Utilizzato per richieste HTTP o su metodi di invocazione per la sessione di un singolo utente. Il bean viene creato per la durata di una sessione http e viene scartato al termine della sessione.

#### **Request Scope (@RequestScoped):** utilizzato per una singola richiesta HTTP o un metodo invocazione. Il bean viene creato per la durata dell'invocazione del metodo e viene scartato quando il metodo termina.

#### **Conversation Scope(@ConversationScoped):** si estende tra più invocazioni all'interno della sessione con i punti iniziale e finale determinati dall'applicazione. Le conversazioni vengono utilizzate su più pagine come parte di un flusso di lavoro a più fasi.

#### **Dependent pseudo-scope (@Dependent):** il ciclo di vita è uguale a quello del client. Viene creato ogni volta che viene iniettato e il riferimento viene rimosso quando il target di iniezione è rimosso. Questo è l'ambito predefinito per CDI.

## Interceptor

Sono dei costrutti che permettono al container di intercettare la chiamata ad un metodo e inserire della logica prima di quella del business (permettono di aggiungere codice ai metodi senza toccarli). Esistono 4 categorie di interceptors:

#### **Constructor-level interceptors**: intercettore associato a un costruttore della classe target (@AroundConstruct);

#### **Method-level interceptors**: intercettore associato a un metodo business specifico (@AroundInvoke);

#### **Timeout method interceptors**: intercettore che si interpone sui metodi di timeout con @AroundTimeout (utilizzato solo con il servizio timer EJB);

#### **Life-cycle callback interceptors**: intercettore che si interpone sul ciclo di vita dell'istanza target. (@PostConstruct e @PreDestroy).

Ci sono diversi modi per definire un interceptors, il modo più semplice è inserirli all’interno del bean stesso.

Un **around-invoke method** deve avere le seguenti caratteristiche:

* Non deve essere **static** o **final;**
* Deve avere come parametro un **Javax.interceptor.InvocationContext** e deve ritornare un oggetto;
* Deve lanciare una **Checked Exception**;

**Funzionamento:** il container intercetta la chiamata, ed esegue **logMethod**, successivamente logMethod lancia **proceed()** per procedere con il prossimo intercettore se c’è altrimenti si procede con il metodo inizialmente invocato ovvero **createCustomer().**

**Class Interceptors:** si possono anche definire delle classi di intercettori da utilizzare su interi bean oppure su specifici metodi di specifici bean.

**Life-Cycle Interceptor:** inserendo all’interno della classe intercettore un metodo annotato come **PostConstruct** quest’ultimo verrà invocato prima del metodo PostConstruct della classe target.

**Chaining and Excluding Interceptors:** è possibile associare ad una classe target una catena di intercettori.

## Interceptor Binding

Per garantire il **loose-coupling** all’interno i java EE è stata inserito il **binding** degli intercettori, per questo è possibiledefinire un certo tipo di intercettore, implementarlo e poi utilizzarlo in modo disaccoppiato.

## Priorizzare gli intercettori

L'associazione degli intercettori ci offre un livello di riferimento indiretto, ma ci fa perdere la possibilità di ordinare gli intercettori. Per questo è possibile dare una priorità agli intercettori usando il **@Priority** insieme a un valore di priorità (il valore più piccolo viene chiamato prima).

## Decorators

Sono estremamente simili agli interceptors, anche se lavorano in modo diverso, infatti, se interceptors ignora la logica di business che viene dopo la sua esecuzione, il decoratore no, perché non viene svolto prima della **classe target** ma viene costruito intorno ad essa, modificandola e decorandola.

I decoratori devono avere un punto di iniezione delegato (annotato con @Delegate), con lo stesso tipo dei bean che decorano. Questo consente al decoratore di invocare l'oggetto e quindi invocare qualsiasi metodo di business su di esso. Anche i decoratori sono disabilitati e vanno attivati in caso di necessità.

## Eventi

Permettono ai beans di interagire senza dipendenze a tempo di esecuzione. Un bean può creare e gestire un evento. I produttori di eventi generano eventi utilizzando l'interfaccia javax.enterprise.event.Event. Un produttore genera eventi chiamando il metodo fire(), passa l'oggetto evento e non dipende dall'osservatore.

Un **osservatore** è un bean con uno o più metodi di osservazione, che in caso di attivazione dell’evento, eseguono il loro codice.

# Lezione 17-18, Java Persistent API:

## L’utilità dei servizi

Strutturare le applicazioni in servizi è utile perché permette di condividere questi ultimi, sia con gli utenti che con altri linguaggi o applicazioni, infatti non è necessario che la nostra applicazione sia strutturata allo stesso modo di quella che offre il servizio, basta solamente che entrambe si scambino dati tramite un protocollo comune e aperto.

## Caratteristiche trasversali

sono caratteristiche che vengono fornite a tutti i livelli dell’architettura java EE.

## End to End principe

La sicurezza di un sistema è data dalla somma del livello di sicurezza delle singole parti che lo compongono (lo stesso vale per l’efficienza). In parole povere se unisco dei moduli non è detto che siano efficienti o sicuri come lo sono presi singolarmente e non è detto che non generino problemi inaspettati anche se i singoli moduli nel loro dominio funzionano perfettamente.

## La persistenza in Java EE

I dati sono di solito memorizzati in database, uno strumento per assicurare la persistenza dei dati. La “vecchia” gestione dei database portava i programmatori a dover fare una distinzione tra la logica di questi ultimi e la logica della programmazione object oriented. Logicamente le due cose non sono molto diverse, infatti l’unica differenza sostanziale, sta nel fatto che le entità (DataBase) sono persistenti, mentre di oggetti (Object-Oriented) non lo sono (**per il Garbage Collector**).

## Object-Relational Mapping (ORM)

è una metodologia che permette di mappare oggetti in tabelle, unendo il mondo della programmazione ad oggetti con quello dei Database. JPA, modulo presente fin da subito in Java EE, è una astrazione ulteriore di JDBC che lo rende indipendente da SQL.

Quindi ORM permette di assegnare oggetti a tabelle e lo fa tramite l’**EntityManager** una componente JPA che si occupa di svolgere le cosiddette operazioni **crude** (creazione tabella, inserimento di tuple, ecc…), questo meccanismo si interfaccia bene con la gestione della persistenza di java.

## Oggetti persistenti: Creazione di un’entità:

un Entità è il nome che un **Oggetto Persistente** assume nella logica dei DB. In Java EE la creazione di un’entità si svolge utilizzando due annotazioni:

#### **@Entity:** informa il container (o meglio l’EntityManager) che quella che segue è un’entità (oggetto persistente), e sarà esso a gestirla.

#### **@id:** rappresenta la nostra chiave primaria, che identificherà in modo univoco ogni tupla dell’entità.

#### **Il resto:** sono attributi, costruttore e metodi getter e setter come una comune classe java.

## Proprietà di un’entità:

* Deve essere una **Classe Normale** con **Costruttore Public**;
* Se deve essere passata per valore deve implementare **Serializable;**

## Configuration by Exception

È un pattern che definisce l’importanza di fornire dei valori di default (**Covention**) quando ci sono molte configurazioni, come per JPA dove tutte le configurazioni sono già impostate a valori di default per il caso standard di utilizzo.

## Struttura e trasformazione da oggetto a entità:

All’interno della struttura di JPA troviamo un layer che si occupa dell’operazione di mapping (dopo il mapping l’oggetto diventa una riga della tabella).

Abbiamo un file XML (**Persitence.xml**) dove vengono racchiusi tutti quei dettagli che vengono nascosti al programmatore per garantire la trasparenza.

## Definire query in Java EE

All’atto di definizione di una entità posso definire anche delle query sullo stesso oggetto. È importante notare come la **FROM** e la **SELECT**, facciano riferimento a degli oggetti e non a delle tabelle.

## Ciclo di vita delle entità: (Alta probabilità)

Il ciclo di vita di un’entità viene gestito dall’EntityManager e si divide in 4 fasi:

* L’oggetto viene creato, quindi esiste in memoria (stato di **Exists in memory**);
* L’oggetto viene preso in carico da un **EntityManager** (stato di **Managed**), resta qui finché rimane persistente;
* L’oggetto viene rimosso (stato di **Removed**)dal gestore della persistenza ma non viene eliminato, continua a esistere nella memoria centrale (ridiventa un **POJO).**
* L’oggetto viene temporaneamente “abbandonato”, quindi **Rimosso dal Persistence Context (**è ancora nel DB solo che non viene gestito) ma non in modo definivo, **l’EntityManager** può tornare ad occuparsene (stato di **Detached**).

## CallBacks:

Come per gli oggetti, in JPA è possibile inserire all’interno del ciclo di vita delle entità della logica di business intercettando le varie fasi del ciclo di vita tramite metodi **Pre** e **Post.**

## Listeners

sono utilizzati per separare la logica di business da una specifica classe, in modo che possa essere condivisa da tutte le classi.

Una classe per essere utilizzata come listener deve:

* Avere un **costruttore pubblico senza argomenti.**
* La firma dei metodi di callback deve essere diversa tra di loro.
* Deve avere parametri di **tipo compatibile** con le entità che l’evento gli passerà.

## Validazione dei bean

Serve a garantire la formattazione dei dati inseriti nel DB (come campi non nulli o formattati in un determinato modo).

**Elementary Mapping (@Tables, @SercondaryTables, @id, @GeneratedValue)**

## Relationship Mapping

All’interno di JPA è possibile gestire tutto quello che riguarda le relazioni tra diverse entità (**chiavi esterne, join, relazioni bidirezionali o unidirezionali e cardinalità**), tramite delle specifiche annotazioni.

## Gestione della cardinalità

La cardinalità all’interno di JPA viene gestita attraverso 4 annotazioni, che permettono di realizzare sia relazioni unidirezionali che bidirezionali:

#### **@OneToOne;**

#### **@OneToMany;**

#### @**ManyToOne;**

#### **@ManyToMany;**

## Concetto di persistence context

È una sorta di cache dove l’Entity manager, “archivia” le entità prima che siano realmente inserite (**flush**), all’interno del database. È importante notare che all’interno del persistence context non possono esistere entità con lo stesso **id.** Inoltre, se un’entità è presente in due persistence context e viene flushata all’interno del db contemporaneamente, una delle due operazioni fallirà perché non possiamo scrivere contemporaneamente un oggetto (**sono** **transazioni**).

## Metodi del Entity Manager

Come ogni oggetto l’Entity manager espone diversi metodi che ci permettono di gestire i nostri oggetti persistenti.

**Persist():** rende persistente un oggetto (lo inserisce nel db).

#### **Cercare un’entità all’interno del database**: per fare ciò esistono due metodi:

* **Find():** cerca l’entità all’interno del database tramite un identificativo (**@id**), caricandone in memoria sia il riferimento che i dati associati a quest’ultimo (quindi tutto l’oggetto).
* **GetReference():** simile al metodo find tranne per il fatto che una volta trovato l’oggetto all’interno del db, carica in memoria solo il riferimento a quest’ultimo caricando i dati solo nel momento in cui questi vengono utilizzati (**lazy** **feching**).

#### **Rimuovere un entità all’interno del database:** per fare ciò si utilizza il metodo **Remove(),** rende l’entità **detached** all’interno dell’Entity manager (**l’entità non viene eliminata dalla memoria, ma torna ad essere un POJO, viene eliminata solo quando il GC la distrugge**).

#### **Rimozione degli orfani**: con il temine **orfano** si fa riferimento a righe che non sono referenziate da nessuna altra tabella. JPA è in grado di rimuovere automaticamente tali entità se le viene esplicitamente chiesto in fase di definizione dell’oggetto (**cancellazione e modifica a cascata**).

**Sincronizzazione con il database:** la prima sincronizzazione dei dati con il database avviene al primo **commit**, successivamente, JPA aggiorna automaticamente lo stato dell’istanza salvata nel db, ma per alcuni tipi di operazioni questo non è detto che sia sufficiente, per questo sono stati definiti i seguenti metodi:

**Flush():** forza il persistence provider a inserire i dati all’interno del database.

**Refresh():** è l’opposto del **flush**, praticamente sovrascrive lo stato attuale dell’oggetto caricato in memoria con lo stato dell’entità associata presente nel DB.

**Gestire il contenuto del Persistence Context:**

**Contains():** verifica se una data entità si trova all’interno del mio **Persistence** **Context** ritornando un boolean;

**Clear():** svuota il persistence context facendo decadere tutte le entità contenute all’interno di esso.

**Detach():** rimuove il **Persistent** **Context** facendolo diventare detached (decaduto);

**Merge():** Ricarica lo stato un entità all’interno del persistence context, in modo che ritorni ad essere gestita (in seguito a una clear per esempio).

# JPQL (Java Persistence Query Language): (Alta probabilità)

Nasce dalla necessità di dover gestire tutte quelle operazioni complesse che riguardano la gestione di dati all’interno di un database attraverso JPA. Molto simile a SQL, ha come unica differenza quella di lavorare con le entità e non con le tabelle. **JPQL** traduce le query in **SQL**, la traduzione di queste query è un overhead (operazione viene eseguita a tempo di compilazione o meglio a tempo di **deployment**). Le query vengono tradotte in base al database scelto.

All’interno di JPQL le query vengono suddivide in 5 categorie:

## Query dinamiche: generate dinamicamente all’interno di un’applicazione ed eseguite. È la tipologia di query più utilizzata nonché la più versatile anche se ha degli svantaggi (non è possibile verificare sintatticamente il codice SQL all’interno di esse)

## Named query: sono query statiche, inserite all’interno di una classe all’atto della sua creazione. Vengono utilizzate per compiti standard. Non possono essere modificate, ma gli si può passare un parametro.

## Criteria API (or Object-Oriented Queries): Sono query orientate agli oggetti, uniscono le Named e le Dinamiche e hanno il grande vantaggio che la sintassi viene verificata a tempo di compilazione in modo da evitare errori, anche se questo le rende meno leggibili. Sono inoltre fortemente tipizzate.

## Native query: sono query native del SQL. Non si usano a meno di casi molto rari e particolari.

## Stored procedure query: sono memorizzate all’interno del database. Usate per compiti lunghi e ripetuti, oltre che per scopi statistici. Hanno il grande vantaggio che il codice è centralizzato e può essere utilizzato da diversi programmi.

È importante notare che la query è un oggetto che deriva dalla classe query.

# Lezione 20, Enterprise JavaBeans, parte 1:

## Enterprise JavaBeans (EJB): è un componente di JEE server-side che si occupa di incapsulare la logica di business e di gestire transazioni e sicurezza. All’interno di JEE vengono definiti tre tipi di sessioni EJB:

#### **Stateless**: Il bean non viene istanziato né usato da nessun client e non ci sono conversazioni di stato fra i metodi. Praticamente fornisce solo servizi.

#### **Stateful**: l’opposto di stateless, viene utilizzato per operazioni che si dividono in più step.

#### **Singleton**: è un bean che viene condiviso dai client e che supporta la concorrenza di accesso.

Un EJB è composto da due elementi principali:

#### **La classe bean:** che contiene l’implementazione dei metodi di business e che può implementare nessuna o diverse interfacce. L’importante è che il bean sia annotato con @stateless, @stateful o @singleton.

#### **Un’interfaccia di business**: che contiene la dichiarazione dei metodi di business che sono visibili al client e implementati nella classe del bean. Un bean può avere un’interfaccia locale, remota o nessuna. L’utilizzo di un’interfaccia remota o locale dipende dal tipo di chiamata che si sta andando a effettuare.

Esistono due annotazioni per definire il tipo di interfaccia che si sta andando a implementare:

#### **@Remote**: Denota un’interfaccia remota di business. I parametri sono passasti per valore e necessitano di essere serializzati.

#### **@Local:** Denota un’interfaccia locale. I parametri sono passati per referenza dal client al bean.

## Stateless Beans

Sono i beans più usati all’interno di JEE. Hanno la particolarità che ogni operazione deve essere completata all’interno di una singola chiamata a un metodo. In parole povere offrono dei servizi che possono essere utilizzati senza che nulla venga istanziato.

Questo tipo di architettura permette di riutilizzare un bean su più client che si trovano ad eseguire le medesime operazioni. Per fare ciò il container crea per ogni **stateless EJB** un certo numero di istanze (**Pool**) in memoria condividendole tra i client, e siccome questo tipo di bean non ha uno stato ognuna di queste copie in memoria è equivalente.

## Stateful Beans

Sono beans che offrono metodi di business ai loro client ma non mantengono una conversazione con quest’ultimi. Sono di solito utilizzati in operazioni che si dividono in varie fasi, fungendo un po' da “memoria”.

Al contrario dei beans **stateless,** quelli **stateful** non possono essere condivisi tra più client, in quanto “personalizzati” o meglio in possesso di uno stato. Per questo motivo il container crea, per ogni client che ne ha bisogno, uno stateful bean **personale.**

## Singleton

Sono dei bean di sessione che vengono istanziati una sola volta per applicazione, tramite il pattern singleton. Ovviamente vista la natura di questa categoria di beans, il container può istanziare un unico singleton che verrà poi condiviso tra tutti i client.

## Packaging

Come quasi tutte le componenti Java EE, i beans necessitano di essere **impacchettati** (all’interno di un file **.jar** che contiene tutte le componenti utili al bean) prima di essere pronti al **deployment**. L’unione di tutte i vari bean e componenti viene inserito in un file **.war** che è pronto al **deployment.**

## Invocazione di EJB

Un enterprise java beans può essere invocato da diversi tipi di componenti Java EE, tramite tre diversi tipi di invocazione:

## Invocazione tramite iniezione

Con questo metodo di invocazione la situazione varia in base a sé il bean dispone o meno di un’**interfaccia:**

* Se il bean non ha interfacce quest’ultimo viene importato tramite un’annotazione.
* Se il bean dispone di numerose interfacce il client deve specificare quale vuole che sia referenziata.

In particolare, se il bean risiede su un **server differente** le **@EJB API** permettono di inserire come attributo l’indirizzo dell’interfaccia o del bean.

## Invocazione con CDI (Context Dependecy Injection)

Nella maggior parte dei casi possiamo sostituire l’annotazione **@EJB** con **@Inject** questo ci permette di ottenere tutti i vantaggi della **Context Dependecy Injection.**

Inoltre, per quanto riguarda l’iniezione di beans remoti visto che l’annotazione **@Injection** non prevede parametri dobbiamo prima produrre il bean remoto.

## Invocazione diretta con JNDI

È un meccanismo di naming ispirato a RMI, rappresenta un’alternativa all’iniezione, permettendoci di risparmiare risorse, infatti, **JNDI** inserisce i dati solo se sono necessari.

# Lezione 21, Enterprise JavaBeans, parte 2:

## Ciclo di vita dei Session Beans: (Possibile domanda teoria allo scritto)

## Stateless beans e i singleton il ciclo di vita è molto semplice:

#### Tutto inizia quando il beans viene richiesto (tramite **injection** o **JNDI lookup**) e il container istanzia una nuova sessione.

#### Se l’istanza appena creata utilizza a sua volta delle **injection**, il container soddisfa anche queste.

#### Se l’istanza implementa dei metodi **@PostConstruct**, vengono eseguiti dal container.

#### Il bean va nello stato di **ready** dove in caso di necessità esegue le richieste dei client, rimanendo in attesa quando non ne riceve.

#### Il container non ha più necessità del bean. Quindi invoca i metodi **@PreDestroy** se ce ne sono e poi elimina il bean.

## Stateful beans, il ciclo è un po' diverso:

#### Tutto inizia quando il beans viene richiesto da un client (tramite **injection** o **JNDI lookup**) e il container istanzia una nuova sessione.

#### Se l’istanza appena creata utilizza a sua volta delle **injection**, il container soddisfa anche queste.

#### Se l’istanza implementa dei metodi **@PostConstruct**, vengono eseguiti dal container.

#### Il bean esegue le richieste del client e va in memoria in attesa di richieste future (stato di **ready**).

#### Se il client rimane inattivo per un periodo di tempo, il container si prepara a rendere il bean **passivo** eseguendo prima i metodi annotati con **@PrePassivate**, successivamente lo sposta nella memoria di massa.

#### Se il client invoca un bean in stato **passivo** il container prima di risvegliarlo esegue i metodi annotati con **@PostActivate**, per poi spostarlo nella memoria centrale.

#### Se il client non invoca un bean **passivo** per un certo periodo di tempo scatta un **timeout** e il bean viene distrutto.

#### In alternativa se il client invoca il metodo annotato con **@Remove** il container esegue i metodi annotati con @**PreDestroy** e distrugge il bean.

## Autorizzazioni

Permettono di controllare l’accesso al business code, ed esistono due modi di gestirle in java EE:

#### Dichiarativa Possono essere definite utilizzando **annotazioni** o un **XML Deployment descriptor.**

#### Programmatica: permettono di realizzare un controllo sulle autorizzazioni a grana più fine, utilizzando due metodi forniti dall’interfaccia SessionContext:

##### **isCallerInRole():** Ritorna un booleano e controlla se il chiamante ha un ruolo assegnato.

##### **getCallerPrincipal():** ritorna java.security.Principal che identifica il chiamante.

## Transazioni

Sono operazioni che devono essere eseguite in sequenza e devono essere o completate o annullate per non andare a compromettere la consistenza dei dati. Le transazioni si basano su 4 proprietà **ACID**:

#### **Atomicità**: una transazione è composta da una o più operazioni raggruppate in un’unità di lavoro, alla fine della transazione o tutte le operazioni sono andate a buon fine (commit) o si devono annullare quelle che sono state eseguite (rollback).

#### **Consistenza**: alla fine della transazione tutti i dati in uscita devono essere consistenti.

#### **Isolata**: gli stadi intermedi della transazione non devono essere visibili esternamente all’applicazione.

#### **Durabilità**: deve essere finita, ovvero una volta che si è conclusa i cambiamenti nei dati devono essere visibili alle altre applicazioni.

All’interno di JEE le transazioni vengono interamente e automaticamente gestite dal container in ogni loro parte.

# Lezione 23/24, Java Message Service

È un servizio che permette di scambiare messaggi all’interno di JEE. Questo servizio si basa sul concetto di **broker** o **provider di posta**, una struttura software che offre due principali macro-vantaggi:

* **Disaccoppiamento temporale dall’invio del messaggio alla sua ricezione**
* **Implementazione di meccanismi di connessione molto flessibili**, che riducono di molto i compiti del programmatore.

## Message Oriented Middleware MOM

Si basano sull’ utilizzo di un broker o provider di posta (**Sistema asincrono**), che tiene “in caldo” i messaggi prima di spedirli al destinatario finale.

Il producerè la componente che “produce” il messaggio e lo inserisce all’interno di una **destinazione**, che rappresenta una sorta di casetta della posta dove il **consumer,** ovvero colui che “consuma” il messaggio sa di dover andare a controllare la presenza di nuovi messaggi.

## JMS è il set di API che si occupa di implementare questo sistema. In JMS esistono due tipi di destinazione ognuna applicata a uno specifico modello di messagistica:

#### **Point to point** all’interno del quale I messaggi viaggiano da un singolo producer a un singolo consumer. Questo modello si basa sul concetto di coda. Viene anche utilizzato nel caso in cui un producer si trovi a dover inviare tanti messaggi a diversi consumer (Multiple receivers).

#### **Publish-Subscribe**: in questo modello un singolo messaggio è inviato da un singolo producer per diversi consumer, anche contemporaneamente (viene chiamato sunscribers, perché si deve prima iscrivere al Topic per ricevere messaggi). In questo caso c’è però una dipendenza temporale, visto che i messaggi vengono consegnati solo dopo l’iscrizione (quelli prima vengono persi).

I messaggi possono anche trasportare job (per esempio quando usiamo i thread).

## Java Messaging Service API in dettaglio:

#### **ConnectionFactory**: è un **administered object** (un oggetto che può essere creato e gestito solo dall’amministratore). Permette a un’applicazione di creare un oggetto **Connection**.

#### **Destination**: è un altro **administered object**, che contiene tutte le informazioni sui provider come l’indirizzo di destinazione. Informazioni che sono nascoste ai client.

#### **Connection**: incapsula una connessione a un provider JMS. È una struttura thread-safe disegnata per essere condivisibile tramite il concetto di sessione. Prima di essere utilizzata, una connessione deve essere avviata con il metodo .start() (poi chiusa con il metodo stop()).

#### **Sessioni**: Viene creata tramite il metodo createSession(). Il primo parametro specifica se la sessione implementa o meno le transazioni (se settato a true vuol dire che i messaggi non saranno inviati finché non sarà dato il commit(), se è settato su false() appena viene dato il metodo send() il messaggio viene inviato).

#### **Messaggi:** sono oggetti in cui vengono incapsulate informazioni, sono divisi in tre parti:

##### **Header**: contiene informazioni sul messaggio, che client e provider utilizzano per identificare il messaggio.

##### **Properties**: è un meccanismo per inserire altri campi header all’interno del messaggio tramite applicazioni.

##### **Bodies**: contiene i dati che si desidera inviare. È un campo molto versatile che può contenere molti dati differenti.

## Differenza tra Properties e Header: (Alta probabilità)

Le properties sono definibili e possono essere aggiunte. Gli header invece non possono né essere aggiunti né specificati dal programmatore, in quanto standard e presenti all’interno di JMS. Inoltre, con le proprietà è il broker a effettuare i controlli, per esempio, se un messaggio non rientra nelle proprietà che desideriamo, non viene proprio inviato, questo porta a un notevole risparmio di risorse, visto che il client ha un notevole carico di lavoro in meno.

## Meccanismi di affidabilità del broker

JMS assicura un certo numero di servizi di affidabilità atti a garantire un recapito affidabile.

#### **Filtering messages:** permette di creare dei filtri in modo che il broker invii al client solo messaggi che rispettano determinati requisiti.

#### **Time-to-live:** permette di definire un tempo (in millisecondi) dopo il quale il broker elimina i messaggi dalla destinazione se non sono stati ancora consegnati.

#### **Message Persistence**: JMS supporta due metodi di consegna per i messaggi:

##### **Persistent**: si assicura che il messaggio sia recapitato una sola volta (Più costoso in termini di risorse).

##### **Nonpersistent**: si assicura che il messaggio sia recapitato ma non controlla quante volte.

#### **Controllo degli Acknowledgment:** permette di ricevere una notifica dal consumer di avvenuta consegna del messaggio. Nelle sessioni transazionali, gli ack vengono automaticamente utilizzati (servono in caso di roll-back). Nelle sessioni non transazionali, devono invece essere specificati.

#### **Durable Consumer**: permette (all’interno del modello pub-sub) al consumer di ricevere i messaggi anche nel caso in cui non sia connesso al topic quando quei messaggi vengono inviati (è molto oneroso in termini di risorse, ma alcune volte necessario).

#### **Settare una priorità**: permette di settare una priorità ai messaggi, in modo che vengano inviati prima di altri con priorità più bassa (valore da 0 a 9).

Tutte questi metodi possono ovviamente essere concatenati per definire un contesto.

## Message-Driven Beans

È un consumatore di messaggi asincrono, viene invocato dal container quando arriva un messaggio.

Hanno i seguenti requisiti:

* Annotato con **@javax.ejb.MessageDriven** o l’equivalente nel deployment descriptor.
* Implementa l’interfaccia **MessageListener**.
* Classe pubblica non final o abstract.
* Costruttore pubblico e senza argomenti (perché il container non sa che argomenti passargli).
* Non devono essere definite metodi **finaline()**.

## Ciclo di vita di un MDB

molto semplice, si passa dallo stato di esistenza a quello di pronto, tramite l’esecuzione dei metodi annotati con @PostConstruct, si rimane nello stato di pronto finché arrivano messaggi, si eseguono i metodi annotati con @PreDestroy e poi si elimina il bean.

# Lezione 26, Web Service and SOA

Non sono altro che applicativi sul web che offrono servizi. L’intera architettura si basa su due protocolli:

## SOAP: che offre un gran numero di servizi ma è meno efficiente.

## REST: versione migliorata di SOAP con performance migliori.

Il concetto principale dei web services è che il consumer non conosce nulla dell’implementazione del servizio che sta utilizzando, esso infatti si limita ad utilizzare l’interfaccia che quest’ultimo espone (**loose** **coupling**). I web services per questioni di standardizzazione utilizzano il protocollo HTTP (che è utilizzato a livello mondiale sulla porta 80). Tutto questo permette di esporre logica di business tramite servizi su internet e permette ai programmatori di evitare di scrivere codice già esistente, utilizzando servizi disponibili online e testati tramite delle interfacce in rete.

## Software as Service

Rappresenta un importante cambio di paradigma.

## Simple Object Access Protocol (SOAP):

È un protocollo che espone logica di business tramite servizi, garantendo un basso accoppiamento tramite un meccanismo che delega all’interfaccia la formattazione dei messaggi sia in response che in request. Praticamente il server **pubblica** un’interfaccia e il consumer **discover** utilizza i servizi del provider.

I Web Services si basano su molte tecnologie e protocolli per trasportare e trasformare dati dal **consumer** al **service provider** seguendo uno standard unico. I principali sono:

#### **Extensible Markup Lenguage(XML):** utilizzato da SOAP per costruire e definire. Rappresenta una scelta “terza” per garantire uno standard comune di comunicazione. Permette di definire delle regole per garantire che il file sia correttamente formattato.

#### **Web Services Descriptor Language (WSDL):** definisce i protocolli, le interfacce, i tipi di messaggi e le iterazioni tra consumer e provider, è come un’interfaccia di Java ma scritta in XML, serve a esporre i servizi del service provider. Contiene tutte le informazioni utili al consumer per identificare il provider, il protocollo di trasporto utilizzato e tutte le informazioni necessarie tramite una serie di elementi e attributi.

#### **Simple Object Access Protocol (SOAP):** è un protocollo di message-encoding utilizzato per definire la comunicazione dei web services. Si occupa in pratica di implementare lo scambio di messaggi XML tra provider e consumer. WSDL definisce il tipo di messaggi che devono essere scambiati e SOAP li implementa.

#### **HTTPS, SMTP, JMS**: come protocolli di trasporto;

#### **Universal Descriptor Discover, and Integration (UDDI):** un servizio di registri e meccanismi di discover opzionale. Permette al consumer di indentificare il provider e di conoscere gli standard per comunicare con quest’ultimo.

# I tre pilastri dei Web Services sono: WSDL,SOAP,UDDI.

## Writing SOAP Web Services: Per scrivere Web Services esistono due principali approcci:

#### **Contract-first**: visto che il WSDL è il contratto tra consumer e service, può essere utilizzato per generare il codice java per consumer e provider.

#### **Bottom-UP**: si parte da un codice normale e lo si rende web services tramite l’annotazione @WebService.

## Java Architecture for XML Binding (JAXB)

Prima di essere inviato, un oggetto scritto in java deve essere convertito in un file XML e, successivamente, dopo essere giunto a destinazione deve essere fatto l’opposto. In JEE questo processo viene svolto da **JAXB** tramite un insieme di API.

**Tutto questo è meno efficiente (perché aggiungiamo delle operazioni di parsing), ma non è nulla rispetto agli enormi vantaggi sui servizi che otteniamo.**

## WSDL Mapping

A livello di servizio il sistema è definito in termini di messaggi XML, operazioni WSDL e messaggi SOAP. Al livello di Java invece troviamo: oggetti, interfacce e metodi. Quindi occorre un servizio che converta gli oggetti java in XML e viceversa.

#### **JAXB**: utilizza annotazioni per determinare come marshal/unmarshal una classe to/from XML.

#### **JSON Web Signature (JWS):** utilizzato per mappare classi java in WSDL, per effettuare il marshal di un’invocazione di un metodo tramite SOAP e unmarshal per l’eventuale risposta del metodo.

## @WebService:

marca una classe java o un’interfaccia (utilizzando l’endpoint) in modo da farla diventare un Web Service.

Sono disponibili due diverse tipologie di annotazioni:

## WSDL mapping annotations: Utilizzate per personalizzare le firme dei metodi esposti.

#### @**WebMethod**: Permette di modificare il nome del metodo presente in WSDL o di escluderlo da quest’ultimo in modo che non sia visibile al client.

#### @**WebResult**: Controlla il nome del messaggio di ritorno in WSDL, permettendo di modificarlo.

#### @**WebParam**: simile a @WebResult ma si occupa di parametri, permette infatti di modificare il nome di un parametro e il suo tipo.

#### @**OneWay**: viene utilizzata su metodi che non hanno un valore di ritorno (void), serve a informare il container in modo che possa ottimizzare la chiamata.

## SOAP Binding Annotations: Descrive come il web service è limitato a un protocollo di messagistica. Esistono due tipi di stili di programmazione che definiscono il SOAP Binding e che differiscono sostanzialmente per il contenuto del SOAPbody:

#### **Document**: il messaggio contiene un documento senza ulteriori regole di formattazione (Scelta di default).

#### **RPC**: il SOAP contiene un elemento con il nome del metodo o la procedura remota che è stata invocata.

#### **è inoltre possibile scegliere fra due tipi di serializzazione/deserializzazione:**

##### **Literal**: i dati vengono serializzati in accordo allo schema XML.

##### **Encoded**: la codifica del SOAP specifica come gli oggetti devono essere serializzati.

#### **Questo ci porta ad avere 4 possibili tipologie di configurazioni:**

##### Document/Literal (Configurazione di Default).

##### Document/Encoded

##### RPC/Literal

##### RPC/Encoded

## Handling Exceptions

in java, quando qualcosa va storto viene generata un’eccezione che viene catturata e gestita da un metodo, con SOAP questo meccanismo non funziona perché producer e consumer non è detto che parlino lo stesso linguaggio e per di più sono separati da uno strato di rete. Per ovviare a questo problema SOAP riconosce automaticamente le eccezioni quando vengono generate e le converte in un **SOAP Fault** all’interno di **SOAP Message.**

## Ciclo di vita

Anche in questo caso è molto banale, con l’esecuzione di metodi annotati con @PostConstruct e @PreDestroy, dopo la creazione e prima della distruzione.

## WebServiceContext

SOAP dispone di un ambiente basato sul contesto a cui si può accedere iniettandone il riferimento tramite l’annotazione **@Resource,** permettendoci cosi di ottenere informazioni come l’endpoint della classe di implementazione, il contesto del messaggio, le informazioni di sicurezza ecc..

## Invoking SOAP Web Service

Invocare un Web Services è molto simile a invocare un oggetto con RMI: all’interno del client (che è un container a tutti gli effetti) possiamo trovare un **Consumer Proxy** che rappresenta il Web Service all’interno del client. Questo proxy ha come compito quello di convertire le richieste ai metodi effettuati dal client in messaggi SOAP, spendendoli via web al **Web Service Endpoint** che li elabora e invia la risposta come messaggio SOAP, che viene convertito in un istanza del tipo di ritorno scelto dal metodo e presentato al client dal **Consumer Proxy.** Il vantaggio che questo sistema offre è che sia il client che il server non hanno necessità di comprendere l’infrastruttura di rete né questi meccanismi che sono completamente invisibili al programmatore.

## Invocazione programmatica

Se il consumer viene eseguito al di fuori del container, bisogna invocare programmaticamente il SOAP Web Service, richiedendo prima un istanza del servizio (**CardValidatorService()**)in modo da poter invocare i metodi localmente.

## Invocazione tramite Injection

se il consumer viene eseguito all’interno di un container si può utilizzare l’**injection** per referenziare il SOAP Web Service client Proxy. In modo molto simile a **@Resource** o **@EJB.**

## Invocazione con CDI

Si può anche direttamente iniettare il riferimento a un Web Service utilizzando l’annotazione **@Inject**.

# Concetti rilevanti di Cloud Computing:

## High-Performance Computing (HPC)

Sistemi basati sulle pure performance e utilizzati per risolvere task singoli nel minor tempo possibile. La loro potenza viene misurata tramite il **Linpack Benchmark**.

## Cluster architetture

è costruita intorno a una rete a bassa latenza e ad alta banda che connette vari server tra di loro in modo che condividano potenza di calcolo prima di offrirla ai client tramite internet.

## High-Throughput Computing (HTC)

Sistemi in grado di poter eseguire il maggior numero di task possibili contemporaneamente, utilizzano un’architettura P2P, dove non è importante verificare con assoluta certezza che un task vada a buon fine.

## Reti P2P

all’interno di una rete P2P tutti gli host svolgono sia funziona di client che di server, quindi offrono e consumano risorse. La rete è una rete “libera”, ogni host si connette e disconnette in modo libero e indipendente, cosa possibile dal fatto che i dati sono ridondanti nella rete in modo da renderla sempre funzionale anche in mancanza di nodi.

## Computational and Data Grids

ricalca un po' il concetto delle reti idriche o fognare, io mi connetto a una griglia e ne utilizzo i servizi che propone senza interessarmi di dove quei servizi siano prodotti.

## The Hype Cycle of New Technologies

abbiamo prima di tutto una fase ascendente che porta all’esagerazione poi una fase discendente e successivamente un affermarsi della tecnologica in questione.

## Infrastrutture Virtuali

il processo di virtualizzazione delle infrastrutture ha portato a una riduzione di costi per l’hardware, a una stabilizzazione dei costi energetici e a un aumento dei costi di manutenzione negli anni.

## Cloud service models: I servizi offerti dal cloud possono essere divisi in tre categorie principali:

#### **Infrastructure as a Service (IaaS):** l’utente può utilizzare il SO, sistemi di storage, applicazioni e i componenti della rete, ma non controlla direttamente l’infrastruttura cloud. Praticamente ha a disposizione una macchina da configurare.

#### **Platform as a Service (PaaS):** permette di distribuire e gestire l’esecuzione di un’applicazione utilizzando risorse fornite dal provider cloud tramite un ambiente adeguato.

#### **Software as a Service (SaaS):** permette di utilizzare software tramite un interfaccia web.

## Cloud Pubblici: costruito su internet, può essere utilizzato da chiunque pagando una certa quota mensile o giornaliera.

## Cloud Privati: costruito all’interno di un dominio o all’interno di un intranet in possesso di una singola organizzazione, viene utilizzato per la gestione interna e per il funzionamento della stessa.

## Cloud ibridi: utilizzato sia per scopi interni di un’azienda/organizzazione, che per offrire servizi al pubblico tramite internet.

## Model Cost: Grazie al cloud viene eliminata tutta quella parte di costi fissi per l’equipaggiamento che le aziende si trovavano a dover affrontare per la messa in opera della loro infrastruttura hardware.

## Cloud Computing Economics:

il cloud computing si rileva estremamente conveniente dal punto di vista economico in alcuni casi. Per esempio quando la **domanda** per un servizio **varia nel tempo**, infatti se gli host del servizio decidono di **gestire i picchi** di carico di quest’ultimo si ritroveranno inevitabilmente ad acquistare delle macchine che per la maggior parte del tempo rimarranno inutilizzate, se d’altro canto decidono di **ignorare i picchi**, inevitabilmente con il passare del tempo gli utenti trovando un servizio lento e inaffidabile, **migreranno** altrove con conseguente perdita di ricavi. Altri vantaggi si hanno in situazioni in cui **non è ben nota a priori la quantità di traffico** che il servizio si ritroverà a gestire, problema che viene risolto dalla rapida **scalabilità** (allocando altre macchine), che il cloud possiede. Infine, un'altra situazione in cui il cloud è molto utile è quando ci troviamo ad avere bisogno di **semplice e pura potenza di calcolo**, che il cloud non ha difficoltà a offrire (evitando alti costi per l’acquisto di un gran numero di macchine performanti, l’utente paga **quanto consuma**).

# Concetti rilevanti di Microservice:

## La morte dei Big Software

Visto la sempre crescente complessità dei software odierni, con il passare del tempo le aziende si sono ritrovate a sviluppare software utilizzandone altri già esistenti con lo scopo di ridurre i costi oltre che per cercare di sviluppare programmi che si adattino a diversi campi di utilizzo. L’insieme di tutti questi fattori ha portato alla morte dei cosiddetti **Big Software** (o **Monolith**) ovvero di quei software estremamente voluminosi.

Un altro fattore che si è cercato molto di ridurre è il cosiddetto **Total Cost of Ownership TCO**, ovvero il costo derivante dal possesso di centri di calcolo (**il cloud risolve questo problema**). Altri fattori sono stati: i grandi costi di manutenzione anche dopo il rilascio e la natura strettamente accoppiata dei moduli componevano questi software (**per ovvi motivi tecnologici**).

La soluzione a questi problemi è il cosiddetto **Software** **Small**, ovvero un tipo di prodotto software basato su servizi sviluppati e mantenuti indipendentemente, in questo modo i vari moduli sono indipendenti e vengono mantenuti e ottimizzati dal team che li ha sviluppato.

## Micro Services

Sorgono dalla cenere dei big software. Nell’architettura a micro-servizi la gestione del software è in **mano al cliente** che ne gestisce lo sviluppo e il mantenimento.

Le applicazioni si sviluppano in tanti piccoli servizi in modo che: **comunichino tra di loro** nel modo più “leggero” possibile, risultino **indipendenti dalla piattaforma** (Aperti **all’Eterogeneità**) e siano costruiti intorno alla necessità di **business**.

L’intera architettura a micro-servizi di basa sul prendere un applicazione monolitica e dividerla in tanti piccoli servizi, ottenendo vantaggi soprattutto per quanto riguarda la **scalabilità**.

All’inizio è complicato sviluppare in micro-servizi, però quando la complessità aumenta questa difficoltà si attenua fino a ripagare del tutto gli svantaggi.

## Componenti come servizi

I micro-servizi sono WebService, sono indipendentemente deployabili e non c’è coesione. Anche in questo caso un producer espone qualcosa che può essere utilizzato di client.

Ci sono alcuni svantaggi in tutto questo: effettuando un’invocazione standard di metodi, vado sulla rete e questo porta inevitabilmente a dei cali di prestazione, che però sono nulla rispetto ai vantaggi che questa architettura offre.

## Sono realizzati intorno alle competenze di business:

## legge di Conway: Qualsiasi organizzazione che progetta un sistema produrrà un design la cui struttura è una copia della struttura di comunicazione dell'organizzazione.

In parole povere i nostri team sono composti da specialisti del medesimo settore o competenza, quest’ultimi produrranno una **architettura a Silos**, dove ognuno produce un pezzo differente e poi si mettono d’accordo di come farlo comunicare.

Nell’architettura a MS invece ogni team contiene specialisti in diversi settori o competenze (**Team Cross Functional**), con lo scopo di progettare una funzionalità che sia utile all’obbiettivo di business e di legarle nel modo più leggero possibile alle altre (vogliamo un sistema strettamente accoppiato).

I team sono indipendenti, piccoli (4-6 otto persone al massimo) e non lavorano su progetti ma sui prodotti in modo che il team sia responsabile sia durante lo sviluppo che durante la produzione.

## Smart endpoints and dumb pipes:

I team che sviluppano in micro-servizi utilizzano per la comunicazione i principi e i protocolli su cui si basa il World Wide Web.

L'infrastruttura scelta è in genere stupida (Estremamente semplice e rapida) mentre nei software Monolith la comunicazione tra i vari componenti avviene tramite invocazioni di metodi o chiamate di funzioni, strategie che risultano estremamente lente e complesse, è quindi necessario sostituire la comunicazione a grana fine con un approccio più grossolano.

## Gestione decentralizzata del calcolo e dei dati

All’interno dell’architettura MS è importante non evitare la standardizzazione: Non devo sicuramente obbligare nessuno a utilizzare uno specifico linguaggio o ecosistema, in modo da aumentare la produttività e l’efficienza dei singoli team, Ma devo anche assicurarmi che servizi scritti con linguaggi diversi **siano perfettamente compatibili**, per questo all’interno dei software MS è estremamente importante la **definizione dei contratti (interfacce)**, quello che c’è dietro quest’ultime invece importa.

Invece per quanto riguarda i dati, di solito nella progettazione monolitica quest’ultimi sono inseriti all’interno di un singolo database, nelle architetture MS invece ogni team è libero di scegliere il proprio DB in base alle proprie necessità. L’unico vincolo è che poi i vari DB siano in grado di riconciliarsi e di comunicare fra di loro.

## Infrastructure Automation

ovvero utilizzare delle infrastrutture che automatizzino tutta la fase di delivery e integrazione. Tramite test e deployment automatici.

## Progettati per i malfunzionamenti

All’interno di un’architettura MS si deve fare in modo di considerare che altri servizi possano non funzionare o non essere disponibili in determinati momenti ed è di fondamentale importanza che questo tipo di eventualità sia gestibile, evitando che l’intero ecosistema vada **down.**

## Progettazione evolutiva

è un concetto di **progettazione agile** che permette di avere frequenti cicli di release, in pratica si cerca di rendere totalmente indipendenti questi servizi rendendone facile la sostituzione o l’aggiornamento.

Per quanto riguarda il **Planning delle release,** in architetture MS si rilascia un servizio appena pronto in modo che ogni team funzioni alla sua velocità. È anche possibile fare del **versioning** dei servizi ma si cerca di evitarlo, perché con il passare del tempo le versioni che ci si porta “sulle spalle” aumentano, aumentando i costi di bug Fix e di gestione (si cerca di implementare una retrocompatibilità constante invece).

## Differenza tra MC e Web Services

sono due cose diverse, rappresentano quasi un vero e proprio paradigma di programmazione: sono distribuiti su team, lo sviluppo indipendente, la gestione del team, e tutto quello che sta scritto sopra. I WB rispetto i MS sono più a basso livello, sono l’implementazione in java dei MS se vogliamo.

# Concetti di Server-Less computing:

Un miglioramento dell’architettura MC è quella del **Server-less Computing,** banalmente all’interno del SL non c’è un server. Questo tipo di architettura, conosciuta anche come **Function as a Service,** si basa sulla creazione di funzioni che vengono eseguite all’interno di macchine cloud e presentate all’utente come MC.

Il vantaggio è notevole, prima di tutto non devo gestire il server con tutte le sue problematiche e costi, inoltre le funzioni vengono eseguite in container isolati, (molto più efficienti delle VM che abbiamo visto nel cloud, visto che il sistema operativo sottostante rimane lo stesso, indipendentemente dall’uso che si fa della macchina).

In pratica l’intero sistema si basa su una funzione che viene invocata al verificarsi di un evento, per esempio quando un client fa una richiesta, ma posso reagire anche a qualcosa di più specifico come l’inserimento di n-uple all’interno del DB, oppure ancora quando qualcosa viene modificato, quando vengono dati dei comandi vocali, o qualsiasi cosa che possa venerare un evento. Per ogni tipo di servizio offerto dal cloud, abbiamo una parte gestita dal programmatore e dall’altra gestita dal provider di servizi.

Questo tipo di architettura in API permette una scalabilità estremamente efficiente e facile da realizzare. In particolare, quanto più piccoli sono i task e le funzioni o MC, più posso scalare efficientemente.

## Benefici dello sviluppo Server-Less:

#### **Sviluppo e deployment rapido**: i costi di sviluppo sono bassi e i tempi di sviluppo rapidi, con la possibilità di rilasciare subito e migliorare nel tempo.

#### **Facilità di uso**: essenzialmente posso realizzare funzionalità complesse che si evolvono in maniera diversa e indipendente.

#### **Costo minore**: tutto i costi di manutenzione e gestione vengono eliminati.

#### **Scalabilità**: usare SL permette al provider di bilanciare il carico in modo molto più semplice e veloce, cosa che permette anche al provider di offrire servizi a costi minori. In poche parole, l’utente paga le invocazioni invece delle macchine.

#### **Nessun costo di manutenzione dell’infrastruttura**: che ovviamente sono a carico dei provider.

**Limiti dell’architettura Server-Less:**

#### **Si perde il controllo dell’infrastruttura hardware**: ci si lega al provider sia economicamente che progettualmente.

#### **Cold start**: Le applicazioni SL hanno un “avviamento freddo” perché il sistema che gestisce e ottimizza queste funzioni se vede che non sono utilizzate per un tot di tempo le mette da parte. Per risolvere questo problema si fa un Ping per riattivare la funzione ogni tanto, in modo da tenerla “sveglia” (ogni Ping ovviamente costa quindi se lo facciamo troppo spesso i costi aumentano).

#### **Architettura condivisa**: si possono condividere le infrastrutture, ma se una va giù va giù anche l’altra.