Appunti Programmazione Distribuita

Capitolo 1 (Libro Scarano)

Un sistema distribuito è un insieme di macchine gestito in maniera autonoma e indipendente connesse con una rete.

Sistema dove la funzionalità che si vuole realizzare si basa sulla condivisione. Ogni nodo coordina il lavoro attraverso un software detto "middleware", il quale permette al programmatore di percepire il sistema come un'unica entità.

Esempio: WWW è distribuito e non è centralizzato visto che non c'è nessuna autorità centrale.

Legge di Moore: Afferma che la densità dei transistor raddoppia ogni 18 mesi e quindi raddoppia la potenza di calcolo.

Un sistema distribuito permette di assecondare l'evoluzione della tecnologia, ma ha dei problemi da gestire come la gestione dei malfunzionamenti di componenti oppure latenza. Inoltre consente la comunicazione di processi tramite memoria distribuita e la definizione di applicazioni altamente scalabili che sfruttano un insieme di nodi interconnessi.

RM-ODP: Si basa ed integra il modello ISO/OSI ed ha l'obiettivo di gestire problemi di comunicazione in un sistema rispetto a problemi di connessione. Punta inoltre a standardizzare il concetto di trasparenza e portabilità all'interno di un sistema distribuito.

I sistemi distribuiti si caratterizzano per essere:

- Remoti: Le sue componenti devono poter essere sia locali che remote su macchine diverse
- Concorrenti: Nella loro natura possono fare esecuzioni contemporaneamente su macchine diverse
- Senza stato globale: Non esiste un modo per "fermare" in un dato istante tutte le macchine
- Malfunzionanti parzialmente: Ogni componente può smettere di funzionare senza differenze percepite
- Eterogenee: Deve sorpassare i vari tipi di hardware, software, protocolli e cosi via
- Autonomi: Non è centralizzato per il controllo
- Evolutivi: Devono assecondare l'evoluzione dell'ambiente in cui sono realizzati
- Mobili: I nodi e le risorse devono essere mobili a adattarsi alle prestazioni del sistema.

Un servizio distribuito punta ad essere:

- Aperto, uso di interfacce standard e riconosciute, capace di collaborare tra diverse componenti.
- Integrato, risorse differenti usate senza strumenti specifici
- Flessibile, gestire modifiche durante l'esecuzione e fare evolvere in modo da integrare i sistemi ereditati
- Modulare , ogni componente autonoma ma con un grado di interdipendenza dal sistema
- Facilmente gestibile per avere anche una buona QoS (vincoli di tempo, disponibilità, affidabilità e tolleranza ai malfunzionamenti).
- Sicuri, gli utenti non autorizzati non devono accedere ai dati sensibili.
- Scalabile ovvero gestire picchi di carico.
- Trasparenti ovvero che non vedo i dettagli implementativi e che mostra il sistema come un unica macchina.

Trasparenza di un sistema: I dettagli del sistema che offre la funzionalità sono nascoste agli utenti.

La trasparenza nei sistemi distribuiti è un requisito non funzionale importante che gli permette di essere "unico" a qualsiasi utente il che favorisce la produttività , questo anche riutilizzando delle applicazioni sviluppate.

Tipi di trasparenza:

- Accesso: Nasconde al livello più basso la rappresentazione dei dati e la loro invocazione, riesco ad usare quel sistema senza minimamente conoscerlo e significa che l'accesso ad un oggetto, sia da locale che da remoto viene fatto con gli stessi metodi. É fornito di default. (Esempio server Http che parte su local host)

- Locazione: Non so un oggetto dove sta precisamente, ma utilizzo un sistema di naming (disaccoppiare posizione da nome). Nella trasparenza di acceso non so se è in locale o remoto (esempio DNS viene coperto l'indirizzo IP)

- Migrazione: Si possono far migrare gli oggetti da un nodo ad un altro senza che l'utente se ne renda conto (troppe query sul DB e si cambia macchina). Per fare questo è necessario che ci sia trasparenza di locazione perché ho bisogno di essere spostato su un altra macchina e ho bisogno di quella di accesso perché deve accedervi nello stesso modo sia che sia remoto che locale.

- Replica: Se un server web è sovraccaricato ne mettiamo altri replicandolo così se avevo un flusso per una macchina adesso metto una macchina che mi indirizza il flusso su delle macchine replicate. Come Akamai che distribuisce e replica gli utenti in una sede vicino all'utente.

- Persistenza: Fa in modo che l'oggetto o la componente abbia uno stato, sia memorizzata su memoria secondaria senza che l'utente se ne renda conto, ad esempio Google Docs non ha il tasto salva ed è reso persistente in modo trasparente. Basato sulla trasparenza di locazione perché un oggetto può essere stato persistente su un nodo e viene riattivato su un altro nodo e deve essere accessibile da entrambi.

- Transizioni: Gestire la concorrenza (ACID come i database) e la consistenza dello stato degli oggetti. Per il sistema è faticoso gestire questa come le altre cose e potrebbe portare un sovraccarico di cui non ci rendiamo conto che comporta un costo.

- Malfunzionamenti: In presenza di malfunzionamenti su alcune componenti le altre componenti continuano a funzionare, anche se in modo limitato. Un altra cosa che garantisce la trasparenza alla transazione è fare un "rollback" per le transazione non complete. Serve trasparenza di replica (utilizzo altre macchine) e alle transazioni (se il malfunzionamento accade durante una transizione devo garantire lo stato).

- Scalabilità: In grado di poter servire carichi crescenti o decrescenti senza dover modificare l'organizzazione, scala aggiungendo risorse e usando la trasparenza di migrazione che è un scaling verticale (prende un macchina e ne aggiunge risorse) per lo scaling orizzontale utilizzo la replica che mi permette di scalare meglio rispetto alla verticale.

- Prestazioni: Il sistema deve funzionare con una QoS e avere delle buone prestazioni. E fa questo con il bilanciamento di carico e riducendo la latenza spostando sulla macchina sul bordo della rete dell'utente (essere da un "hop") e inoltre usa la memoria in maniera efficiente.

Avere troppa trasparenza a volte significa non avere troppo prestazioni, quindi in certi casi a parte di questa trasparenza a volta bisogna rinunciarci, per i costi.

Gli oggetti distribuiti sono al centro tra i sistemi distribuiti e programmazione ad oggetti ed hanno lo scopo di realizzare servizi distribuiti riutilizzabili. Questo viene integrato con il middleware che si trova tra applicazione e sistema operativo, il suo ruolo è quello di offrire le caratteristiche dei sistemi distribuiti.

Tipi di middleware:

- Infrastruttura: Come la JVM che scherma i dettagli della macchina sottostante che permette di eseguire il bytecode su qualsiasi piattaforma nascondendo il sistema operativo.

- Distribuzione: Automatizza le operazioni comuni per la comunicazione. Come il marshalling ovvero richiedere un servizio ad un altro nodo non potendo inviare parametri, oppure utilizzare uno stesso canale di comunicazione ed altro per la semantica e altro per la rete.

- Servizi comuni: Fornisce servizi comuni e riutilizzabili a tutti i sistemi riutilizzabili.

Il middleware permette il riuso, l'efficacia e l'efficienza facendo concentrare il programmatore solo sulla logica di buisness.

RPC: Ha definito il meccanismo di invocazione remota ed ha permesso che fossero invitati i dati su socket e si superassero le differenze di rappresentazione dei vari tipi.

Se ad esempio passo un intero da una macchina che lo vede con 4byte a una che lo vede in 8byte, il middleware si deve occupare di questa cosa. Inventa anche un altra cosa, affiancare a quello che invoca (client) dei programmi per forzare marshalling (comunicare i problemi nel passaggio dei dati). Qui mi interessa solo chiamare un programma e vedere l'altra macchina che esegue la procedura chiamata.

Problemi di RPC: I tipi di dato sono elementari ad esempio limitazioni per i puntatori, mancanza della gestione delle eccezioni , mancanza della concorrenza.

Passaggio da RPC a Oggetti distribuiti: C'è bisogno di aggiungere polimorfismo , ereditarietà.

Middleware implicito: Fornisce servizi comuni e trasversali ad ogni componente senza che questa debba esplicitamente chiederli.

Capitolo 2 (Libro Scarano)

Socket e struttura applicazioni distribuite

La comunicazione tra programmi su internet avviene grazie a TCP/IP e il modo in cui lo si fa è grazie alle socket che permettono di ricevere e trasmettere dati.

Socket: Astrazione canale di comunicazione che permette di far comunicare, identificato da una coppia indirizzo e porta. Su questo socket posso scrivere e leggere byte. Questo permette a due elaboratori di comunicare.

I due computer che comunicano tramite socket si identificano con il nome di client e server. Il server è in esecuzione e attende un client in esecuzione, il client d'altra parte sa dove un server sta ascoltando e si connette.

Il procedimento è che il server accetta la connessione e assegna un nuovo socket per comunicazione bidirezionale tra i due. Dopo fatto questo può tornare a mettersi in ascolto per altre connessioni.

Abbiamo quindi un cliente che usa i socket , un server con la server socket e il socket. Funzione accept() stabilisce la connessione tra il client e il server.

Gli strati che dobbiamo aggiungere per un applicazione distribuita sono lo stub e lo skeleton.

Stub: Oggetto che si trova sul client che rappresenta a tutti gli oggetti l'oggetto server verso il client, ha infatti gli stessi metodi che ci sono sul server. Non ha le reali implementazioni ma fa solo le invocazioni sui socket.

Skeleton: Si trova sul lato server e si occupa di effettuare l'invocazione del metodo richiesto sull'oggetto server restituendo il valore del metodo e comunicando allo stub che lo restituisce al client.

Tutto questo avviene perché sia lo stub che il server implementano un'interfaccia comune che è chiamata interfaccia remota dove sono definiti i metodi che devono essere implementati.

Per prendere il riferimento all'oggetto remoto si utilizza un servizio che ci permette di recuperare un oggetto grazie al solo nome.

Capitolo 3 (Libro Scarano)

Java RMI

Java RMI: Libreria java che permette lo sviluppo di applicazioni distribuite fornendo la possibilità di effettuare comunicazione remota tra programmi scritti in java. Il suo obiettivo è quello di assicurare la semplicità e l'integrazione dell'implementazione.

Java RMI fornisce anche un garbage distribuition in modo da preservare la modalità di gestione della memoria che permette al programmatore di non doversi occupare esplicitamente dell'allocazione e deallocazione della memoria.

Java RMI prevede due tipi di invocazione, unicast ovvero da un client verso un server e multicast ovvero verso diversi server.

Oggetto remoto: Oggetto i cui metodi possono essere acceduti da un'altro spazio di indirizzamento. La descrizione dei servizi che può utilizzare è in un interfaccia remota.

Invocazione remota di metodi: Invocazione di un metodo su un oggetto remoto.

L'interfaccia remota deve estendere Remote che è un interfaccia marker. Ogni metodo descritto in un interfaccia remota deve ,nella sua implementazione, rispettare questi vincoli:

- Deve dichiarare esplicitamente l'eccezione RemoteException.
- I parametri di un metodo remoto devono essere dichiarati tramite la propria interfaccia remota.

L'oggetto client utilizza esclusivamente l'interfaccia remota non la sua implementazione.

Per fare un implementazione remota si può:

- Far estendere alla classe che contiene l'implementazione UnicastRemoteObjcet

- Far estendere una classe che si occupi esplicitamente di esportare l'oggetto e di implementare le cose di RemoteObject e RemoteServer.

Per poter invocare il metodo di un oggetto remoto il client deve avere a disposizione il riferimento remoto che si può ottenere o come risultato di altre invocazioni o grazie a un servizio di directory.

Nel secondo caso RMI ci da un meccanismo di name server grazie a Naming che permette di gestire i riferimenti a oggetti remoti grazi ad un ID. Questa classe ci da i metodi di lookup() per ricercare , bind() per registrare e list() per elencare.

In RMI bisogna avviare il registro RMI applicativo presente nell'installazione Java (rmiRegistry).

Rmi.registry permette di implementare parzialmente la trasparenza di locazione dato che offre ad una applicazione Java RMI un servizio di naming per il Discovery di oggetti remoti.

Un metodo remoto può dichiarare solo parametri o valori restituiti che siano serializzabili.

Quando vengono passati come parametri più riferimenti allo stesso oggetto ad un'altra macchina con invocazioni diverse allora questi saranno oggetti distinti.

Questo accade per garantire il vincolo di integrità referenziale, infatti se li passiamo assieme client punteranno allo steso oggetto.

Modifiche a metodi di object, hashCode() per far si che stesso codice per oggetti remoti , equals() che vede se il riferimento remoto è uguale all'oggetto da cui lo usiamo e infine tostring() che da info anche sulla macchina.

RMI è formato quindi da:

Stub/Skeleton
Remote Reference Layer, che specifica il comportamento della invocazione e la semantica
Transport Layer: Si occupa della connessione della sua gestione.

Stub/Skeleton:

- Lo stub deve iniziare la connessione, effettuare il marshalling e attendere il risultato dell'invocazione, una volta arrivato effettuo l'unmarshalling e restituisce il valore all'oggetto client.

- Skeleton: Effettua l'unmarshalling e invoca il metodo sull'implementazione ed effettua il marshalling del valore restituito.

Remote Reference Layer: Si occupa di interfacciare il livello trasporto con quello di stub/skeleton dando e supportando la semantica delle operazioni. Da a stub/skeleton un riferimento ad oggetto che implementa RemoteServer e che espone un metodo invoke() per l'inoltro dell'invocazione. Poi comunica anche in basse con il trasporto layer.

Transport Layer: Stabilisce e gestisce la connessione. Rimane in ascolto per altre connessioni in arrivo e gestisce una tabella con oggetti remoti che sono nello spazio di indirizzamento locale. Stabilisce inoltre una connessione per le chiamate in entrata e identifica un oggetto dispatcher a cui inoltrare la connessione.

Meccanismo di lease GC: Ogni riferimento che ha assegnato ad un client ha un tempo di vita specificato, al termine del quale diventa weak e prendibile dal GC.

Capitolo 4 (Libro Scarano)

Processo creazione programma Java RMI

Specificare all'interno di un interfaccia quali sono i servizi che vogliamo offrire dal server. Le interfacce remote con RMI devono estendere Remote e tutti i metodi devono lanciare l'eccezione RemoteException

A questo punto il server viene implementato, dove un oggetto remoto in java è istanza di una classe che implementa interfacce remote o estende UnicastRemoteObject. Il costruttore deve essere scritto anche se può essere vuoto e deve lanciare RemoteException della superclasse.

RMI ci fornisce uno stub compilare rmic che eseguito sul .class del server genera lo stub e lo skeleton.

A questo punto RMI ci propone un servizio di naming Registry che deve essere lanciato prima di eseguire l'oggetto server. I client che vogliono accedere ai servizi offerti utilizzano la lookup() per recuperare il riferimento all'oggetto remoto.

Appena lanciato il server deve registrati sul servizio di naming.

Importante è anche il SecurityManager che serve a poter caricare classi dinamicamente sulla rete e che effettua un controllo a run-time di tutti gli accessi a risorse sensibili.

Ora possiamo lanciare RMI registro

Thread (Non Presente in libro)

Thread: Un'unità di esecuzione del programma che può essere lanciata dal thread principale per eseguire calcoli in parallelo, permettendo l'uso di più core.

Speedup: Indica di quante volte un programma parallelo è più veloce rispetto alla sua versione sequenziale. Secondo la legge di Amdahl, lo speedup è limitato dalla porzione sequenziale del programma. Prima di parallelizzare, è utile chiedersi se ci sono parti che possono essere eseguite in parallelo.

Concorrenza e contatori: In scenari di lettura e scrittura concorrente, le operazioni dei thread possono intrecciarsi (interleaving), portando a risultati imprevisti. È quindi necessaria una sincronizzazione della memoria.

Sincronizzazione: Esistono diversi approcci per sincronizzare le operazioni, tra cui la sincronizzazione a livello di metodo. La parola chiave synchronized garantisce la mutua esclusione, consentendo che solo un thread alla volta esegua un metodo.

Lock: Un meccanismo che permette di bloccare temporaneamente l'accesso alla memoria associata a un oggetto, evitando interferenze tra thread.

Problemi di synchronized: L'uso eccessivo di synchronized può causare blocchi e influire sulle prestazioni.

Volatile: Consente l'accesso diretto a porzioni di memoria specifiche per ogni thread, evitando il caching e migliorando la coerenza dei dati.

Operazioni atomiche: Le classi atomiche permettono operazioni thread-safe più veloci rispetto all'uso di synchronized.

Deadlock: Situazione in cui due o più thread si bloccano perché cercano di accedere a risorse in modo ciclico, generando uno stallo.

Starvation: Condizione in cui un thread non riesce mai a ottenere l'accesso a una risorsa e rimane bloccato in attesa.

Livelock: Situazione in cui due thread cercano continuamente di cedere il controllo all'altro, senza mai avanzare, rimanendo bloccati in un ciclo di attesa reciproca. A differenza del deadlock, i thread non sono fermi, ma nessuno progredisce.

FINE PARTE A Introduzione -> RMI

Capitolo 2 (Libro JAVA EE 7)

In un applicazione enterprise il controller gestisce il codice di buisness e fornisce servizi tecnici. Per

controllare si intende gestire il ciclo di vita delle componenti, fare l'iniezione delle dipendenze e la configurazione delle componenti.

POJO: Sono delle classi Java che vengono eseguite nella JVM.

JavaBeans: Sono dei POJO che seguono pattern specifici per interagire con altre componenti e sono eseguiti nella JVM

Managed-Beans: Sono oggetti gestiti dal container .

Enterprise JavaBean: Possono essere visti come dei Managed Bean con dei servizi extra, come quelli di sicurezza e transazioni.

Dependency Injection: Design pattern che disaccoppia componenti dipendenti, il container inietta oggetti dipendenti per noi.

Se vogliamo un CDI beans in un container non utilizziamo "new" ma iniettiamo il Bean e il container fa il resto. Con le iniezioni un Bean non è a conoscenza della concreta implementazione dei Bean con cui interagisce. Questo permette di diminuire l'accoppiamento. Questi vivono in un "ambito" ben definito che sono request, session , application e coversation. In tutti questi casi il client non controlla il ciclo di vita dell'istanza esplicitandola e distruggendola.

Deployment Descriptor: Descrive come una componente, un modulo o un'applicazione dovrebbe essere configurata. Con CDI è chiamato beans.xml ed è essenziale.

A tempo di deploy CDI controlla tutti i jar e war dell'applicazione e ogni volta che incontra un beans.xml il deployment descriptor gestisce tutti i POJO che diventano Beans CDI.

Un CDI Bean può essere qualsiasi classe che contiene la logica di buisness. Un Bean è un POJO che non eredita o estende nulla e può iniettare riferimenti ad altri Bean. Ha il suo ciclo di vita gestito dal container a può intercettare l'invocazione dei suoi metodi.

Un CDI Bean:

- Non è un classe interna non statica
- Ha la sua classe concreta o ha l'annotazione @Decorator
- Ha il costruttore di base con nessun parametro oppure dichiara un costruttore con @Inject.

@Inject: Abilità di iniettare Bean in altri Bean in un modo sicuro, ovvero senza annotazioni XML. Possiamo iniettare praticamente qualsiasi cosa grazie a @Inject.

Quando con @Inject non dichiariamo un qualificatore il container assume quello di Default.

@Default è un qualificatore esistente che informa CDI di iniettare l'implementazione di Default del Bean. Un Bean senza qualificatore è automaticamente il qualificatore @Default.

Qualifier: Durante l'inizializzazione, il container verifica che per ogni injection point sia disponibile esattamente un Bean compatibile. Se non esiste una specifica implementazione soddisfacente, il container segnala un errore e blocca il deploy dell'applicazione. Un qualifier è un'annotazione che aggiunge una semantica specifica a un tipo, consentendo di distinguere tra più implementazioni dello stesso tipo. Tuttavia, creare una nuova annotazione ogni volta che dobbiamo distinguere un'iniezione potrebbe rendere il codice eccessivamente verboso. Per evitare questa complessità, possiamo utilizzare i membri dei qualifier per parametrizzarli, riducendo la necessità di definire numerose annotazioni personalizzate.

Producers: Non possiamo iniettare direttamente classi come String, perché queste appartengono a un archivio che non include il file beans.xml. Tuttavia, se in una classe annotiamo un metodo o un campo con @Produces, possiamo rendere disponibili vari tipi e classi per l'iniezione utilizzando l'annotazione @Inject insieme a un qualifier.

Scopes: Ogni oggetto gestito da CDI ha uno scope ben definito e un ciclo di vita limitato ad un contesto specifico. Con CDI un Bean è limitato ad un contesto e vi rimane fin quando il Bean è distrutto dal container. Non c'è un modo per rimuovere un Bean da un contesto. Quando un bean di sessione o un POJO sono usate nelle web application loro non sono consapevoli del contesto dell'applicazione. Così CDI ha unito il livello web e dei servizi, assegnandogli degli ambiti significativi.

Scopes:

- Application: Si estende per l'intera durata di un applicazione. Il Bean è creato solo una volta per la durata dell'applicazione, ed è scartato quando l'applicazione è arrestata.

- Session: Si estende tra le varie richieste HTTP o tra le varie invocazioni dei metodi per una sessione di un singolo utente. Il Bean è creato per la durata di una sessione HTTP ed è scartato quando la sessione termina.

- Request: Corrisponde ad una singola richiesta HTTP o ad una invocazione di un metodo. Il Bean è creato per la durata del metodo al termine del quale viene scartato.

- Conversation: Si estende tra le multiple invocazioni tra i bordi delle sessioni con punti di inizio e di fine determinati dall'applicazione. Mantiene lo stato associato a un utente, si estende su più richieste ed è delimitato a livello di codice dall'applicazione. Può essere usato per processi lunghi dove c'è un preciso inizio e fine. Gli oggetti dello scope di richiesta hanno vita breve che perdura solitamente per una singola richiesta , mentre lo scope degli oggetti di sessione dura per l'intera durata della sessione dell'utente. Vi sono comunque alcuni oggetti del livello presentazione che possono essere usati tra più pagine ma non tra sessioni. Proprio per questa esigenza CDI fornisce questo scope. Questi oggetti hanno infatti un ben definito ciclo di vita che esplicita inizio e fine usando il contesto.

- Dependet pseudo-scopes: Stesso ciclo di vita del client.

Interceptors: Ci permettono di aggiungere operazioni trasversali ai Bean.

Dividiamo gli interceptor in:

- Interceptor al livello del costruttore (@AroundConstruct): Interceptor associati con un costruttore della classe obiettivo

- Interceptor al livello di metodi (@AroundInvoke): Interceptor associati con uno specifico metodo di business, cose da fare prima e dopo l'invocazione del metodo. Il metodo non deve essere né statico né final. Il metodo deve avere un parametro di InvocationContext e devono ritornare Object. L'InvocationContext permette agli interceptor di controllare il comportamento della catena di invocazioni. Se più interceptors sono intrecciati la stessa istanza di InvocationContext è passata nella catena. Il metodo proceed() di InvocationContext è estremamente importante perché dice al container che si dovrebbe procedere al prossimo intereceptors o all'invocazione effettiva del metodo.

- Interceptor di timeout: Interceptor che si inseriscono con metodi di timeout @AroundTimeout

- Interceptor delle chiamate dei cicli di vita: Interceptor che si interpongono sugli eventi del ciclo di vita dell'istanza dell'obiettivo. Aiutano a isolare del codice in una classe e invocarlo quando un evento del ciclo di vita è innescato.

Classi Iterceptors: Per specificare una classe interceptors dobbiamo sviluppare una classe separata e istruire il container di applicargli un Bean o metodo di Bean. Nella classe dove vogliamo usarlo possiamo o metterlo sopra une metodo o sopra la classe per indicare che vale per tutti i metodi.

Per diminuire l'accoppiamento possiamo definire una "interfaccia" con @InteceptorBinding che lega la classe interceptor ad un Bean senza una diretta dipendenza tra le classi. Una volta fatta l'interfaccia ci basta annotare la classe.

Decorator: Permettono di aggiungere codice da altre classi. Di base gli interceptor non sono a conoscenza della semantica delle azioni che intercettano e non sono appropriati per separare concetti collegati di business. I decoratori sono dei design pattern e l'idea è quella di prendere una classe e inserirgli una classe attorno. Sono utili per aggiunge una logica aggiuntiva a metodi businesss. Non sono in grado di risolvere cose tecniche che riguardano più tipi. I decoratori devono avere un punto di iniezione con lo stesso tipo del Bean che intendono decorare. Questo permette dal decoratore di inviare l'oggetto delegato. I decoratori implementano l'interfaccia che vogliono decorare.

Eventi: Permettono ai Bean di interagire senza dipendenze a tempo di compilazione. Un Bean può definire un enveto e un altro Bean può lanciarlo.

Capitolo 4

Entità: Sono oggetti che vivono brevemente in memoria ma in modo persistente in un database. Queste entità una volta mappate possono essere gestiste da JPA (Java Persistence API)

@Entity permette a chi offre la persistenza di riconoscerla come una classe persistente mentre @ID identifica l'identificatore unico di quell'oggetto. Con @GeneratedValue possiamo generare automaticamente l'id.

Per essere un entità una classe deve:

- Essere annotata con @Entity
- @Id deve essere utilizzato per denotare una chiave primaria semplice
- Deve avere un costruttore senza argomenti che può essere pubblico o protetto
- Deve essere di primo livello (no enum o interfacce)
- Non deve essere final

ORM: Delega a strumenti esterni o frameworks il compito di creare una corrispondenza tra oggetti e tabelle.

JPA mappa oggetti ad un database grazie ai metadata che abilitano il fornitore della persistenza a riconoscere un entità e ad interpretare il mappaggio, e far sincronizzare i dati di un attributo con la sua rispettiva tabella.

JPA ci perte di mappare le entità al database e di effettuare delle query con diversi criteri, il tutto senza dover conoscere le chiavi del DB o le colonne.

EntityManager: Ha il compito di gestire le entità, leggerle e scriverle in un dato database. I suoi metodi principali sono persist() e find(), con questi l'entitymanger nasconde le chiamate a JDBC.

Persistance unit: Indica all'entity manager il tipo di database da usare e i parametri di connessione che sono definiti nel persistente.xml .

Quando creiamo un istanza di un oggetto questo risiede in memoria e JPA non sa nulla su di lui, quando però la si gestisce con l'entitymanager quello oggetto si sincronizza con la sua tabella nel DB.

Per rimuovere un entità da un DB usiamo Entitymanager.remove() che però la eliminerà dal DB ma potremo ancora utilizzare l'oggetto nel memoria di Java.

Capitolo 5

Un associazione ha una direzione e può essere o unidirezionale o bidirezionale. Un'associazione ha anche una molteplicità. Una relazione ha un "possessore" che è implicito nelle associazioni unidirezionali, ma che deve essere invece specificato in quelle bidirezionali.

In JPA quando abbiamo un associazione tra due classi nel database avremo una tabella di riferimento. Questa tabella può essere modellata in due modi diversi con una chiave esterna con @JoinColumn o con una @JoinTable

Le cardinalità tra due entità possono essere one-to-one, one-to-many, many-to-one o many-to-many. Quando utilizziamo queste annotazioni abbiamo bisogno poi nella classe a cui ci riferiamo (in ogni caso tranne many-to-one) di specificare con mappedby da chi viene mappata.

@JoinColumn: Usata per personalizzare la colonna di join, ovvero la chiave esterna del proprietario della relazione. Il valore di default è la concatenazione del nome dell'entità_nome del riferimento alla chiave primaria. Può essere usata per cambiare la colonna della chiave esterna.

In @JoinTable abbiamo due attributi che sono @JoinColumn che sono distinti dal significato del possessore e posseduto della relazione. La @JoinTable permette di avere una tabella che tiene traccia di entrambi le chiavi, si crea un tabella di un oggetto che in realtà non esiste.

Capitolo 6

JPA ha due modalità di utilizzo:

1) Abilità di mappare oggetti in un database relazionale. La configuration by exception permette al fornitore della persistenza di fare il grosso del lavoro con poco codice, JPA permette inoltre di personalizzare il mapping con XML.

2) Abilità di fare query sugli oggetti mappati. Il servizio centralizzato per manipolare istanze di entità è l'entity manager. Fornisce un API per creare, trovare, rimuovere e sincronizzare oggetti. Permette inoltre di fare vari tipi di query come le dinamiche , statiche e native.

JPQL: Linguaggio definito in JPA per fare query sulle entità presenti nel database.

EntityManager: Punto centrale di JPA, controlla infatti il ciclo di vita delle entità cosi come fa query con un contesto persistente. È responsabile di creare e rimuove istanze di entità e di trovare entità in base alla loro chiave primaria. Quando un entity manager ottiene un riferimento d'una entità questa diviene "managed".

Una caratteristica importante di JPA è che le entità possono essere usate come oggetti normalmente su diversi strati di una applicazione ed essere gestiti da un entity manager quando abbiamo bisogno di caricare o inserire dati in un DB.

Application Managed: Significa che un applicazione è responsabile di esplicitare l'acquisizione di un EntityManger e di gestire il suo ciclo di vita.

In Java EE il modo comune di ottenere un entity manager è dal @PersistantContext o da JNDI. La componente che viene eseguita in un container non ha bisogno di creare o chiudere l'entity manager visto che il suo ciclo di vita è gestito dal container.

Persistant Context: Insieme di istanze di entità gestite e date per una transazione di un utente. Solo le entità che sono contenute in un persistence context sono gestite dall'entity manager, ovvero che le modifiche su di loro avranno effetti sul DB. Questo può essere visto anche come un primo livello di cache, è un piccolo spazio di vita dove le entità sono immagazzinate prima di andare nel database.

Persistance unit: Decide le impostazioni per connettersi al database e la lista di entità che possono essere gestite dal persistent context.

Per ritrovare un entità dal suo identificatore possiamo usare due metodi diversi. Il primo è find() di EntityManager che prende due parametri, la classe dell'entità e l'identificatore univoco. Il secondo modo è getReference() sempre di EntityManger, che restituisce però il riferimento ad una entità e non i suoi dati.

Un entità può essere rimossa con EntityManager.remove(). Anche se dopo questo metodo possiamo ancora utilizzare l'oggetto fin quando la garbage non lo distrugge.

Con JPA possiamo informare il fornitore di persistenza di rimuovere automaticamente gli "orfani" e di propagare le operazioni di rimozione.

Le associazioni che sono specificate come one-to-one o one-to-many supportano l'opzione di rimozione degli orfani (orphanRemoval = true affianco al tipo di associazione).

Con EntityManager.flush() il fornitore della persistenza può esplicitamente forzare a far defluire i dati al database senza però fare il committ della transazione.

Il metodo refresh() è usato per la sincronizzazione dei dati, ovvero sincronizza i nostro oggetto con quello che c'è nel DB.

Il metodo contains() dà un booleano che permette di verificare se una particolare istanza è gestita o meno dall'entity manager nel contesto di persistenza attuale.

Il metodo clear() svuota il contesto di persistenza, scollegando tutte le entità gestite. Questo significa che d'ora in poi le modifiche non verrano sincronizzate con il DB. Per sincronizzarle dobbiamo usare merge()

Queries:
- Dinamic query: Modo più semplice per fare una query che viene creata dinamicamente, quando servono. Sono onerose. (createQuery)
- Named query: Query statiche immutabili e per questo sono più efficienti, hanno un nome e la stringa che rappresenta la query. (createNamedQuery)
- Criteria, permettono di scrivere qualsiasi query in un modo object-oriented.
- Native: Eseguono SQL nativo
- Stored procedure queries

I metodi per eseguire una query sono getResultList() che da la lista di tutti i risultati mentre getSingleResult() se il risultato è unico. Per fare update o delete abbiamo excuteUpdate(). Una query può essere bloccata con setLockMode.

Il ciclo di vita di un'entità cade in quattro categorie, persistente, in aggiornamento, rimossa e in caricamento. Ogni ciclo di vita ha un pre e post evento che può essere intercettato da un entity manager per invocare un metodo di buisness. @PostLoad viene chiamata quando una entità con il database viene caricata con il metodo find() dal DB.

Le regole per usare le annotazioni di callback sono:

- I metodi non devono essere statici o final

- Un metodo può essere annotato con più annotazioni, comunque si può usare una sola di quella annotazione in tutta la classe.
- Il metodo deve lanciare solo eccezioni non controllate
- Non deve invocare EntityManager o operazioni di query

I metodi di callback funzionano bene quando le operazioni da fare solo legate all'entità, i listner invece servono proprio a estrarre la logica di buisness in una classe separata a farla condividere tra le varie entità.

Un listener è semplicemente un POJO nel quale si possono definire più callback. Per usarli nella classe usiamo poi @EntityRegister. Le classi listner devono avere un costruttore vuoto e quando invochiamo i meotoid questi devono avere accesso allo stato dell'entità

Capitolo 7

Le entità possono avere metodi per validare i loro attributi, ma questi non sono fatti per rappresentare operazioni complesse che spesso richiedono un'interazione con altri componenti. Lo strato di persistenza non è lo strato appropriato per i processi di buisness. Per separare lo strato di persistenza da quello di presentazione per implementare la logica di buisness, aggiungere la gestione delle transazioni e della sicurezza le applicazioni hanno bisogno di uno strato di business, per ottenerlo si usano gli EJB (Enterprise Java Bean).

Gli EJB sono delle componenti lato server che incapsulano la logica e si occupano della sicurezza e delle transazioni, inoltre sono un punto di ingresso per tecnologie del livello di presentazione come JSF (Java Server Faces) ma anche per servizi esterni come JMS.

Un container EJB è un ambiente che in fase esecuzione fornisce i servizi come il controllo delle transazioni, della concorrenza delle autorizzazioni di sicurezza.

Gli EJB di sessione sono:

- Stateless: Il Bean di session non contiene un stato di conversazione tra i metodi e qualsiasi istanza può essere usata per qualsiasi cliente.
- Stateful: Il Bean di sessione contiene uno stato di conversazione che deve essere trattenuto tra i metodi per un singolo utente.
- Singleton: Un singolo Bean è condiviso tra i client e supporta gli accessi concorrenti.

Questi Bean di sessione sono componenti gestite dal container quindi hanno bisogno di essere raggruppati in una archivio e fatti partire in un container.

Un embedded container è capace di eseguire applicazioni EJB in Java SE permettendo ai client di far partire la stessa JVM e class loader.

I servizi che offre un container sono:

- Comunicazione remota con un client: Un EJB può invocare metodi remoti con protocolli standard
- Iniezione delle dipendenze: Il container può iniettare più risorse in un EJB
- Controllo dello stato: Per gli stateful bean, il container gestisce la loro trasparenza
- Messa in comune (Pooling): Per gli stateless Bean il container crea un gruppo di istanze che possono essere condivise tra più clienti
- Ciclo di vita delle componenti: Il container è responsabile per gestire il ciclo di vita di ogni componente
- Messaggi
- Gestisce le transazioni: Con un controllo dichiarativo delle transazioni EJB usa le annotazioni per informare il container su come gestire le transazioni.

- Sicurezza: Autorizzazioni a ruoli
- Supporto alla concorrenza: Per i singleton
- Interceptor:
- Invocazione asincrona

Gli EJB sono managed objects infatti sono considerati Managed Bean. Quando un client invoca un EJB, questo non lavora direttamente con l'istanza di quell'EJB ma con una proxy su un'istanza.

EJB lite: Include una minima e potente parte delle caratteristiche di EJB.

Un EJB è fatto dai seguenti elementi:

- Una classe bean: La classe Bean contiene l'implementazione dei metodi di buisness e può implementare interfacce di buisness a piacere. In base al suo tipo inserisce la relativa annotazione (@Stateless, @Stateful, @Singleton).

- Interfacce di buisness: Queste interfacce contengono la dichiarazione dei metodo di business che sono visibili al client e sono implementati dalla classe bean. Un bean di sessione può avere interfacce locali, remote o nessuna.

Un classe di un bean di sessione è una classe Java standard che implementa la logica di business. I requisiti per svilupparne una sono:

- La classe deve essere annotata con (@Stateless, @Stateful, @Singleton)
- Deve implementare i metodi della sua interfaccia , se ne ha
- La classe deve essere definita come public ma non può essere final o astratta
- La classe deve avere un costruttore public senza argomenti
- La classe non deve avere il metodo finalize()
- I metodi di business non devono iniziare con ejb e non possono essere statici o final
- L'argomento e il valore di ritorno di un metodo remoto devono essere tipi legali di RMI

In base a da dove il client invoca un bean di sessione, la classe bean potrebbe dover implementare interfaccia locali o remote o nessuna delle due.

Una interfaccia di buisness è un interfaccia Java standard che non estende nessune specifica istanza di EJB. Queste possono avere due tipi di annotazioni.

- Remote: Denota un interfaccia di business remota e i parametri del metodo sono passati per valore devono essere serializzabili come parte del protocollo RMI
- Local: Denotano un interfaccia locale di business. I parametri dei metodi sono passati per riferimento dal client al bean.

Non si può marcare la stessa interfaccia con più di un'annotazione.

Se un bean espone un interfaccia, automaticamente perde la vista con nessuna interfaccia. Ha bisogno quindi poi di specificare esplicitamente che non espone un vista senza interfaccia con l'annotazione @LocalBean.

L'annotazione @LocalBean non è necessaria per i bean che non espongono alcuna interfaccia, perché in quel caso il container EJB espone automaticamente la classe del bean come tipo locale. Si usa @LocalBean solo quando il bean espone una o più interfacce, per specificare che, oltre a queste, il bean deve essere accessibile anche tramite la classe concreta stessa come tipo locale. In altre parole, serve a fornire un accesso diretto alla classe concreta senza passare esclusivamente per le interfacce dichiarate.

Stateless Bean: Sono i session bean più popolari, per la loro semplicità potenza ed efficienza e rispondono alla task comune di fare processi senza stato. I servizi stateless sono gli ideali quando si ha bisogno di implementare una task che può essere conclusa con una singola chiamata a metodo. I servizi stateless sono indipendenti, autonomi e non richiedono informazioni o lo stato tra una richiesta e un altra. Lo stato è mantenuto dall'oggetto ma non dal servizio. Questi seguono inoltre l'architettura dei servizi stateless e sono la componente di modello più efficiente perché possono essere raggruppati e condivisi tra i client. Per ogni EJB stateless il container mantiene un certo numero di istanze in memoria e le condivide tra i client. Quando un client invoca un metodo su uno stateless bean il container prende un istanza da quelle che ha raggruppato e la assegna al client. Quando finalmente la richiesta del client finisce l'istanza ritorna dove era per essere riusata. Questo significa che c'è bisogno di un piccolo numero di bean per gestire tanti client. Visto che uno stateless EJB vive in un container può usare qualsiasi servizio gestito dal container , tra cui la dipendenza delle iniezioni.

Stateful Bean: Gli stateful bean mantengono la stato conversazione. Sono utili per le attività che devono essere fatte in più step ognuno dei quali si affida allo stato mantenuto nei passi precedenti. Gli stateful bean hanno due tecniche importanti che sono quella della passivation e activation. La passivation è il processo di rimozione un istanza dalla memoria e salvarla in una locazione persistente. L'activation è il processo opposto , ovvero quello di recuperare lo stato e applicare all'istanza. Con l'annotazione @Remove facciamo si che un istanza sia permanente rimossa dalla memoria dopo aver eseguito un certo metodo.

Singleton bean: Session Bean che è istanziato una volta per tutta l'applicazione. Un singleton assicura che solo un'istanza di una classe esiste nell'intera applicazione e fornisce un punto di acceso globale per accedergli. Un caso comune di utilizzo è come sistema di cache dove l'intera applicazione condivide una singola cache. Per trasformare una classe in un singleton abbiamo bisogno di prevenire la creazione di una nuova istanza avendo un costruttore privato. Per ottenere l'istanza avremo bisogno del metodo getIstance(). Visto che è unico per più client per renderlo thread safe dovremmo utilizzare syncronized sul metodo getIstance(). Se annotiamo però semplicemente la classe con @Singleton non dobbiamo preoccuparci di tutte queste cose.

Gli EJB hanno bisogno di essere raggruppati in pacchetti prima che vegano messi su un container in fase di esecuzione. Un volta che li abbiamo raggruppati in un jar ad esempio possiamo metterli direttamente nel container.

I bean di sessione sono gestiti dal container e questo è un punto forte perché ci permettono di concentrarci solo sulla logica, però d'altro canto ci costringono ad eseguirli in un container. L'api embedded permette ai client di istanziare un EJB container sulla JVM. Questo fornisce un'ambiente gestito che supporta i servizi base di Java EE. Di base un container embedded cerca il percorso della classe del client per trovare l'insieme di EJB per l'inizializzazione. Una volta che abbiamo inizializzato il container l'applicazione per il contesto del container JNDI restituisce un contesto per recuperare l'EJB.

Il client di un session bean può essere qualsiasi tipi di componente. Per invocare un metodo su un session bean un client non istanza direttamente il bean ma ha bisogno del suo riferimento che può ottenere o con una iniezione o con JNDI.

@EJB è specifica per iniettare riferimenti di bean di sessione nel codice del client. L'iniezione delle dipendenze è possibile solo negli ambienti gestiti come un EJB container. Se il session Bean implementa più interfacce il cliente deve specificare a quale di quelle vuole il riferimento.

@EJB pero non ci da qualcosa di simile alle alternative di CDI, possiamo quindi usare @Inject. Il più delle volte possiamo scambiare @EJB con @Injcet senza differenze ma con i vantaggi del CDI.

Capitolo 8

Un client ottiene un riferimento ad un session bean o con l'iniezione delle dipendenze o con JNDI. Il container è colui che crea un istanza e la distrugge. Tutti i session bean passano per le fasi di creazione e distruzione

Ciclo di vita di Stateless e Singleton

Entrambi non mantengono uno stato conversazione con un client. Entrambi permettono l'accesso da qualsiasi client e il loro ciclo di vita è:

1) Un client richiede un riferimento al bean. Nel caso di un singleton potrebbe anche iniziare in fase di avvio del container.

2) Se la nuova istanza creata usa l'iniezione delle dipendenze il container inietterà tutte le risorse necessarie

3) Se l'istanza ha un metodo annotato con @PostCostruct il container lo invoca

4) L'istanza del bean processa la chiamata invocata dal client e rimane in una modalità pronta per aspettare le chiamate future. I bean stateless possono stare in quello stato fin quando non si libera dello spazio nella memoria, mentre i singleton vi rimangono fin quando non si arresta il sistema

5) Quando il container non ha più bisogno più dell'istanza invoca il metodo @Predestroy se c'è.

Differenze Stateful e Singleton su creazione e distruzione:

- Quando viene avviato un stateless bean, il container crea diverse istanze di esso e le conserva in un pool condiviso. Quando un client richiama un metodo su uno stateless bean, il container seleziona una delle istanze disponibili, delega l'esecuzione del metodo a quell'istanza e, al termine, la restituisce al pool

- La creazione di un Singleton bean dipende dal contesto: se è stato configurato per essere istanziato all'avvio o se dipende da un altro singleton già avviato, il container crea l'istanza durante il deploy. In caso contrario, l'istanza viene creata al momento in cui un client invoca per la prima volta un metodo di business. Poiché un singleton vive per l'intera durata dell'applicazione, viene distrutto solo quando il container si arresta

Ciclo di vita di Stateful

Gli stateful session bean non sono così diversi da stateless o singleton cambia solo il metadata. Il container genera un istanza e la assegna ad un solo client. Poi ogni richiesta da quel client è passata alla stessa istanza. Seguendo questi passi si potrebbe arrivare ad avere troppe relazioni uno a uno tra i client e bean. Se il cliente non invoca la sua istanza di bean per un lungo periodo di tempo il container deve pulirlo prima che la JVM finisca la memoria, deve preservare lo stato dell'istanza in uno Storage permanente e poi riprenderla quando è necessaria.

Il ciclo di vita è:

1) Il ciclo di vita di un stateful bean inizia quando un client fa richiesta di riferimento al bean. Il container crea una nuova istanza e la salva in memoria

2) Se la nuova istanza creata usa DI il container inietta le dipendenze necessarie

3) Se l'istanza ha un metodo annotato con @PostCostructed lo esegue

4) Il bean esegue la chiamata richiesta e rimane in memoria aspettando una successiva richiesta

5) Se il client rimane in attesa per tanto tempo il container invoca il metodo annotato con @PrePassitive se ve ne è, e mette l'istanza in uno store permanente

6) Se il client invoca un passitive bean il container lo attiva di nuovo end esegue se c'è il metodo @PostActivate

7) Se il client non invoca un passitive bean per un periodo specifico il container lo distrugge

8) In alternativa al punto 7 se il client chiama il metodo @Remove il container allora invoca il metodo annotato con @Predestroy e termina la sua vita.

Il container ci permette di fornire del codice quando gli stati di vita dei session bean cambiano.

Un metodo usato per questo deve:

- Non avere nessun parametro e ritornare void
- Non laniare un eccezione controllata
- Non può essere static o final
- Solo un annotazione di un dato tipo può essere presente sul bean
- Deve poter accedere all'ambiente dei bean

Per quanto riguardo il lato sicurezza l'obiettivo principale di EJB è controllare l'accesso al codice. In Java EE l'autenticazione è gestita dal livello web, il capo e i suoi ruoli sono poi passati al livello EJB, che controlla se l'utente autenticato può accedere al metodo in base al suo ruolo.

Autorizzazione dichiarativa: Può essere usate con le annotazioni o con XML. Include il dichiarare ruoli assegnare permessi o cambiare temporaneamente una identità di sicurezza. @RolesAllowed è usata per autorizzare un lista di suoli che possono accedere a un metodo. @DeclareRoles può essere usata per dichiarare ruoli nell'intera applicazione.

Autorizzazione Programmatica: La usiamo per bloccare o permettere in maniera selettiva l'accesso a un ruolo. I metodi che abbiamo sono isCallerInRole() che restituisce in boolean e vede se il chiamante ha un ruolo, mentre getCallerPrincipal() identifica il Principal che identifica il chiamante.

Capitolo 9

La gestione delle transazioni permette alle applicazioni di avere dati consistenti e processare quei dati in modo affidabile.

Una transazione è usata per assicurare che quei dati sono mantenuti in uno stato consistente. Rappresentano un gruppo logico di applicazioni che devono essere fatte come una singola unità, anche nota come unità di lavoro.

ACID: Si riferisce alle proprietà che definiscono una transazione affidabile, atomica , consistente, isolata e duratura.

L'isolazione di una transizione può essere definita usando delle condizioni di lettura:

- Dirty reads: Quando una lettura di una transazione rimuove i cambiamenti fatti dalla transazione precedente
- Repeatable reads: Quando il dato read sono assicurati di essere uguali se letto di nuovo durante la transazione
- Phantom reads: Quando nuovi record aggiunti al database sono rintracciabili dalle transazioni che sono iniziata prima dell'inserimento.

I Database usano diversi tecniche di blocco, per controllare come le transazioni accedono ai dati in modo concorrente. I meccanismi di locking impattano le read condition, i livello di isolamento sono comunemente usati in database per descrivere come il locking è applicato ai dati con la transazione.

I tipi di livelli di isolazione sono:

- Read uncomitted: La transazione non può leggere dei dati non committati
- Read committed: La transazione non può leggere dati non committati.
- Repeatable read: La transazione non può cambiare i dati che sono stai letti da una transazione diversa.
- Serializzabili: La transazione ha una lettura esclusiva

Una transazione di risorse locali è una transazione che tu hai con una specifica singola risorsa usando la sua specifica API.

Transaction manager: Componente essenziale per la gestione delle operazioni transazionali. Crea la transazione su una metà dell'applicazione e informa il gestore delle risorse che sta partecipando ad una transazione, e conduce il comitt o il ripristino sul manager delle risorse

Resource manger: Responsabili di gestire le risorse a registrarle con il gestore delle transazioni.

La risorsa è lo storage persistente dal quale leggiamo o scriviamo

JTS permette la gestione delle transazioni tra più risorse o risorse distribuite. JTS implementa inoltre OMG (Object Management Group) permettendo al gestore della transazioni di partecipare in transazioni distribuite tramite un protocollo di internet.

Quando sviluppiamo con gli EJB non dobbiamo preoccuparci della struttura interna del gestore delle transazioni visto che è astratto tutto da JTA.

Un EJB container è un gestore delle transazioni che supporta JTA e JTS per le transazioni degli EJB container. Le transazioni sono naturali per EJB infatti ogni metodo è incapsulato in una di esse

---

Capitolo 13

Molte delle comunicazioni viste fino ad ora sono sincrone: il chiamante e il chiamato devono essere attivi contemporaneamente e il chiamante deve attendere il completamento della risposta per proseguire.

Il Message Oriented Middleware (MOM) permette lo scambio di messaggi asincroni tra sistemi eterogenei. Funziona come un buffer tra produttori e consumatori di messaggi, che non devono essere disponibili contemporaneamente per comunicare.

- Provider: Software che immagazzina e invia messaggi.
- Producer: Invia i messaggi al provider.
- Location: Conserva i messaggi fino alla consegna.
- Consumer: Riceve e utilizza i messaggi.

In Java EE, JMS è l’API standard per creare, inviare, ricevere e leggere messaggi in modo asincrono.

L'architettura per lo scambio di messaggi include:

- Provider: Gestisce il buffering e la consegna dei messaggi, fornendo destinazioni per il loro immagazzinamento.
- Clients: Applicazioni o componenti Java che producono o consumano messaggi.
- Messaggi: Oggetti inviati o ricevuti dai client tramite il provider.

- Oggetti amministrati: Forniscono un'interfaccia per connettersi al provider (connection factories e destinations).

Tipi di destinazioni:

- Point-to-Point: Modello in cui un messaggio è inviato da un produttore a un singolo consumatore, attraverso una coda che conserva i messaggi fino al loro consumo o scadenza.
- Publish-Subscribe: Modello in cui un messaggio è inviato a più consumatori, che devono iscriversi a un topic per riceverlo. I messaggi non vengono recapitati ai consumatori che non erano iscritti al momento dell'invio o che erano inattivi.

Oggetti amministrati:

- Connection Factories: Creano connessioni verso destinazioni.
- Destinations: Conservano e distribuiscono messaggi. Possono essere code o topic configurati dal provider.

Message-Driven Beans (MDBs) sono consumatori asincroni di messaggi gestiti dal container EJB. Essendo senza stato, il container può creare più istanze per gestire richieste simultanee. Gli MDB ricevono messaggi attraverso destinazioni configurate, offrendo funzionalità come callback e gestione dei cicli di vita.

Una connessione JMS è rappresentata dall'oggetto Connection, mentre le transazioni sono gestite tramite Session, che raggruppa le operazioni di invio e ricezione in unità atomiche.

I messaggi sono suddivisi in:

- Header: Contiene informazioni standard per identificare e instradare i messaggi.
- Proprietà: Coppie nome-valore configurabili per filtrare i messaggi.
- Corpo: Contiene i dati, in diversi formati.

L'API semplificata di JMS include:

- JMSContext: Gestisce connessioni e contesti per inviare e ricevere messaggi.
- JMSProducer: Invia messaggi a code o topic.
- JMSConsumer: Riceve messaggi da code o topic.

Il client può consumare messaggi in due modi:

- Sincrono: Recupera esplicitamente i messaggi utilizzando il metodo receive(), restando in attesa di nuovi messaggi.
- Asincrono: Si registra con un MessageListener, che gestisce automaticamente l'arrivo dei messaggi.

Meccanismi per garantire una consegna affidabile:

- Filtraggio: Seleziona messaggi in base a criteri specifici.
- Scadenza: Imposta un limite di validità per i messaggi.
- Persistenza: Conserva i messaggi in caso di errori del provider.
- Controllo della conoscenza: Gestisce diversi livelli di conferma sulla consegna.
- Durable subscriber: Assicura la consegna ai consumatori iscritti, anche se temporaneamente inattivi.
- Priorità: Determina l'ordine di consegna.

Gli MDB differiscono dai bean di sessione poiché implementano l'interfaccia MessageListener, anziché interfacce locali o remote. Comunicano esclusivamente tramite messaggi inviati alle destinazioni configurate.

FINE PARTE B CDI->JMS

Capitolo 14

SOAP (Simple Object Access Protocol) è un protocollo per web service che garantisce un basso accoppiamento tra client e server, poiché il client non ha bisogno di conoscere i dettagli implementativi del servizio. Il consumer può invocare un servizio SOAP attraverso un'interfaccia che descrive i metodi disponibili. SOAP fornisce un metodo strutturato per connettere diversi componenti software, rendendolo ideale per integrare sistemi eterogenei.

I web service SOAP espongono una logica di business come servizio per i client. Utilizzano XML sia per il formato dei messaggi scambiati sia per descrivere le loro operazioni, consentendo una standardizzazione che facilita l'interoperabilità tra sistemi diversi. A differenza di oggetti o EJB, SOAP offre un'interfaccia standard che definisce il formato dei messaggi di richiesta e risposta, nonché il meccanismo per pubblicare e scoprire i servizi.

WSDL (Web Services Description Language) è il linguaggio che descrive l'interfaccia dei web service SOAP. Specifica il tipo dei messaggi, la porta, il protocollo di comunicazione, le operazioni supportate, la posizione del servizio e le risposte attese. WSDL consente a consumer e provider di condividere e comprendere i messaggi scambiati, utilizzando XML per descrivere cosa fa un servizio, come invocarlo e dove trovarlo. Mentre WSDL rappresenta l'interfaccia astratta del web service, SOAP ne implementa concretamente la comunicazione.

UDDI offre uno standard per localizzare informazioni sui web service e per invocarli. I fornitori di servizi pubblicano i documenti WSDL in un registro UDDI accessibile via internet.

Il documento WSDL, essendo il contratto tra il consumer e il servizio, può essere utilizzato per generare automaticamente il codice Java sia per il consumer che per il provider. Questo approccio, noto come "top-down" o "contract-first", parte dalla definizione del WSDL per sviluppare il servizio. L'approccio "bottom-up" parte invece da una classe Java già implementata, generando il WSDL corrispondente.

Un web service può essere rappresentato da una classe Java annotata con @WebService. Per essere considerata un web service, una classe deve:
- Essere annotata con @WebService.
- Poter implementare interfacce, anch'esse annotate come web service.
- Essere dichiarata pubblica e avere un costruttore pubblico.

Se la classe deve essere integrata in un container EJB, deve essere annotata come stateless o singleton.

Annotazioni in JWS

JWS utilizza annotazioni per mappare le classi Java a WSDL e per definire il processo di conversione tra l'invocazione di un metodo Java e una richiesta SOAP.

- @WebService: Se applicata direttamente a una classe, genera automaticamente l'interfaccia del web service. Tutti i metodi della classe sono esposti, tranne quelli esclusi con @WebMethod.
- @WebMethod: Permette di personalizzare il mapping di specifici metodi nel WSDL.
- @WebResult: Definisce il nome del valore restituito nel WSDL per un metodo.
- @WebParam: Personalizza i parametri dei metodi del web service, specificandone nomi e altre caratteristiche.
- @OneWay: Designa i metodi senza valore di ritorno come operazioni unidirezionali.
- @WebServiceContext: Fornisce un contesto a runtime al web service, ad esempio dettagli della richiesta o configurazioni dell'ambiente.

Cloud Computing

Punto di forza del Cloud Computing è la scalabilità estrema.

All'inizio l'elaborazione era on premise, ovvero il calcolo avviene solo su una macchina dove stava l'azienda. Nel tempo si è passati da calcolo ad alta prestazione (HPC) a calcolo per un alto numeri di richieste servite (HTC).

HPC (High Performance Computing) si concentra sull'esecuzione di calcoli complessi e intensivi in termini di prestazioni, utilizzando nodi omogenei organizzati in cluster con un controllo centralizzato. È progettato per massimizzare la capacità computazionale in un'unità di tempo.

HTC (High Throughput Computing) punta a gestire un alto volume di richieste, sfruttando nodi eterogenei in una rete peer-to-peer. I nodi collaborano in modo distribuito, mettendo a disposizione le proprie risorse di calcolo.

I sistemi distribuiti, come le grid, hanno obiettivi specifici:
- Efficienza, misurata in istruzioni eseguite per unità di tempo (FLOPS) o nella capacità di gestire molti task contemporaneamente (job throughput). Tuttavia, un sistema con elevate capacità di calcolo potrebbe non garantire un buon throughput.
- Qualità del servizio (QoS), che assicura standard definiti per adattabilità e flessibilità, sia per applicazioni scientifiche che di business.

L'utility computing è un modello in cui le risorse computazionali (calcolo, storage, rete) vengono fornite come un servizio on-demand, simile alle utenze domestiche come l'elettricità. Serve per scalare le risorse in base alle necessità, pagando solo per l'uso effettivo, senza dover gestire o mantenere infrastrutture fisiche. Nell'utility computing, il cliente accede a risorse calcolate su canone, con attributi come ubiquità (accesso geografico uniforme). Questo modello garantisce affidabilità tramite ridondanza: se un centro di calcolo non è disponibile, le richieste vengono reindirizzate altrove.

La classificazione dei sistemi si basa sull'architettura, sul controllo e sui modelli operativi.

- Computer Cluster: Utilizzati nei centri di calcolo, sono costituiti da nodi omogenei interconnessi tramite reti gerarchiche. Funzionano tipicamente su sistemi Unix o Linux e sono orientati all'HPC.

- Peer-to-Peer (P2P): Sistemi decentralizzati in cui i client si collegano autonomamente, entrando e uscendo senza obblighi. Esempi includono la blockchain, dove ogni nodo partecipa in modo indipendente. Utilizzano overlay network, con link virtuali che incapsulano i pacchetti in TCP/IP, e sono comuni nel file sharing.

- Data/Computational Grids: Precursori del cloud, le grid forniscono accesso a risorse distribuite senza il concetto di "pay-as-you-go" o scalabilità dinamica. L'utente si collega a un servizio e lo utilizza senza ulteriori possibilità di personalizzazione o crescita dinamica delle risorse.

I modelli di cloud computing condividono una base comune, con infrastrutture virtualizzate che offrono flessibilità e scalabilità in base alle esigenze degli utenti.

- Infrastructure as a Service (IaaS): Consente di noleggiare un'infrastruttura virtualizzata, dematerializzando l'hardware fisico. L'utente mantiene il controllo e la responsabilità completa su tutto, tranne sull'hardware, e può migrare i propri carichi di lavoro. Offre un pool di risorse virtuali per supportare workload diversi.

- Platform as a Service (PaaS): Fornisce un ambiente per sviluppare e distribuire applicazioni direttamente sul cloud. L'utente si concentra solo sullo sviluppo del software, lasciando al provider la gestione dell'infrastruttura e delle piattaforme sottostanti.

- Software as a Service (SaaS): Fornisce software pronti all'uso, come Gmail, in cui l'utente si limita a consumare il servizio senza necessità di sviluppo o gestione tecnica.

Perché il cloud?

1. I centri di calcolo centralizzano e ottimizzano risorse provenienti da altre strutture.
2. Permette la condivisione delle capacità di calcolo tra diversi utenti, soddisfacendo esigenze diversificate.
3. Separa i costi di manutenzione delle infrastrutture da quelli di sviluppo delle applicazioni.
4. Riduce significativamente i costi associati al calcolo.
5. Offre ambienti di programmazione scalabili e capacità di memorizzazione elevate.
6. Distribuisce i contenuti ai margini della rete, consentendo un accesso rapido e localizzato.
7. Garantisce maggiore privacy e sicurezza dei dati.
8. Supporta la scalabilità on-demand, permettendo di gestire picchi di carico in modo dinamico.

Elasticità: Il cloud consente di fornire rapidamente nuove risorse per gestire picchi di carico improvvisi. È progettato per scalare dinamicamente in base alle necessità. Per flussi di traffico regolari, è importante dimensionare la capacità per includere i picchi.

L'under-provisioning si verifica quando le risorse disponibili non sono sufficienti a gestire i picchi, causando rallentamenti o malfunzionamenti nei server.

Il cloud è dinamico e risponde velocemente alle richieste, rappresentando una soluzione efficace per la crescita delle esigenze di calcolo.

Esistono cloud pubblici, privati e ibridi. I cloud ibridi consentono di rilasciare dati non sensibili, bilanciando sicurezza e accesso.

L'obiettivo del cloud è spostare la computazione da dispositivi locali a data center su Internet, offrendo modelli "pay-as-you-go" e scalabilità delle prestazioni.

Modelli di costo:
- Capex (Capital Expenses): Spese in conto capitale che immobilizzano risorse finanziarie.
- Opex (Operational Expenses): Costi variabili basati sull'utilizzo e sul numero di utenti.

Il cloud favorisce la creazione di un ecosistema, in cui i provider collaborano con partner per ampliare i servizi offerti.

DaaS (Data as a Service): Modello che fornisce database come servizio gestito.

Seminario Spring

Spring è il più popolare tra i framework per lo sviluppo di applicazioni Java Enterprise. Mira a semplificare lo sviluppo di applicazioni Enterprise java.

Moduli Spring:

- Il modulo Core che fornisce le funzionalità fondamentali come IoC DI, unico ad essere sempre necessario

- Il modulo Data Access per l'integrazione con le sorgenti dati astraendo cose come JMS o JDBC

- Il modulo Web che fornisce le funzionalità necessarie per le applicazioni web (gestione richiesta e risposte)

- I moduli AOP e Instrumentation che offrono funzionalità trasversali

- Il modulo Test che fornisce funzionalità di test integrate con gli altri moduli.

Maven è uno strumento per la gestione dei progetti software, progettato per semplificare la build, la gestione delle dipendenze e la documentazione in modo centralizzato. Si basa sul concetto di POM (Project Object Model), un file XML che descrive le informazioni del progetto e le configurazioni necessarie per il processo di build.

Spring Boot semplifica la creazione di applicazioni Spring richiedendo una configurazione minima, rimuovendo ad esempio la configurazione degli XML.

Spring Boot fornisce un web server, che non rende più necessario configurare un web server su cui deployare l'applicazione (Tomcat di base).

Un progetto spring boot ha sempre un pom.xml , un dir java , una dir resources , una dir test e una classe per lo starting point dell'applicazione.

Le proprietà possono essere specificate all'interno del file application.properties o application.yml.

Spring IoC Container è una componente cruciale del modulo core, è responsabile della creazione, configurazione e gestione dell'intero ciclo di vita degli oggetti. Gli oggetti gestiti dal container sono detti Spring Beans e sono gestiti tramite la dependecy injection. I metadati possono essere rappresentati tramite XML o annotazioni.

Spring Sterotype fornisce annotazioni speciali usate per creare automaticamente i bean nel contesto dell'applicazione.

@Service: Contiene la logica di business
@Repository: Si occupa di operazioni CRUD e viene solitamente usata con implementazioni DAO
@Controller: Componente responsabile di gestire richieste degli utenti e restituire la risposta
@RestCotroller: Simile a @Controller ma si occupa solo di API backend

@Configuration: Indica che la classe ha metodi annotati con @Bean che definiscono come creare particolari Spring Bean. Spring al momento di creare il bean della classe @Configuration genererà i Spring Bean definiti dai metodi @Bean che potranno essere usati tramite DI.

Di default i bean sono gestiti come singleton, ma possono essere configurati per far si che ne venga generata una nuova istanza per ogni richiesta oppure per ogni sessione.

Spring Web MVC è il framework costruito sulla Servlet API di Java e fornisce il necessario per sviluppare applicazioni web.

Spring Web consente di creare bean controller per gestire le richieste HTTP in ingresso. Il mapping dei controller alle richieste HTTP avviene utilizzando @RequestMapping che indica il path base della richiesta che il controller deve gestire. Spring web fornisce anche dei bean che si occupano di convertire in tipi primitivi e oggetti i @RequestParam , @PathVariable , @RequestHeader , @RequestBody.

Buona pratica di sviluppo prevede che i controller si occupino solo della gestione delle richieste e risposte e che la logica di business dell'applicazione sia gestita dai servizi. I servizi sono forniti ai controller trami DI che è fatta con @Autowired. Implementare un servizio prevede la definizione di un'interfaccia e una classe per implementare i metodi definiti da annotare con @Service

In Spring ci sono due componenti in grado di intercettare richieste HTTP.

- I filtri che sono eseguiti prima di individuare il controller che gestirà la richiesta e prima della DispatcherServlet.

- HandlerInterceptor che esegue dopo la DispatcherServlet. Questi fornisco metodi per eseguire la logica per richieste prima che siano gestite dal controller , dopo che sono state gestite ma prima che vengano reindirizzate eventuali pagine e dopo il completamento della richiesta.

DispatcherServlet è la componente che si occupa di fare il forwarding delle richieste ai controller.

Spring Data offre un modello di programmazione basato su Spring per l'accesso ai dati. Spring Data JPA semplifica la creazione di repository per database relazionali, ottimizzando le operazioni comuni di accesso ai dati e riducendo lo sforzo di sviluppo. Hibernate è un ORM (Object Relation Mapping) che semplifica la gestione dello strato persistente. È un implementazione delle specifica JPA.

@Repository serve per definire l'interfaccia per la repo a cui Spring fornirà l'implementazione.

Spring Session gestisce le sessioni utente in modo centralizzato, consentendo di memorizzarle in database, Redis o altri storage, invece che nella memoria del server, migliorando scalabilità e gestione.

Redis fornisce uno store in-memory di coppie chiave valore. Offre una serie di strutture dati in memoria molto versatili che permettono di supportare un'ampia gamma di applicazioni con diverse necessità

Spring security è un framework per l'autenticazione e il controllo degli accessi. Il security filter chain definisce quali path URL sono da proteggere e quali non lo sono.

Spring cloud fornisce strumenti per consentire agli sviluppatori di sviluppare e gestire alcune delle operazioni comuni delle applicazioni distribuite.

Api gateway è il singolo punto d'ingresso per tutti i client ai dati o ai servizi del backend. Gestisce tutte le attività di accettazione ed elaborazione relative alle chiamate API.

Service Discovery permette a client, API gateway e altri servizi di scoprire la locazione delle istanze di un servizio.

Service registry è un database dei servizi, delle loro istanze e delle loro locazioni, è interrogato per individuare le locazioni delle istanze dei servizi.

Nel client-side service discovery, il client, prima di chiamare un servizio, consulta il service registry per ottenere le informazioni sulle istanze disponibili. Successivamente, esegue il load balancing e inoltra la richiesta direttamente a una delle istanze del servizio.

Nel server-side service discovery, il client invia la richiesta a un load balancer situato in una posizione nota. Il load balancer interroga il service registry per individuare le istanze disponibili e poi inoltra la richiesta a una di esse.

Spring Cloud Netflix integra Netflix OSS con applicazioni basate su Spring, fornendo strumenti per abilitare e configurare rapidamente pattern comuni in ambienti distribuiti. È particolarmente utile nello sviluppo di architetture a microservizi.

Ad esempio, consente di configurare un server Eureka per gestire il service registry, permettendo alle applicazioni di registrarsi automaticamente e di scoprire dinamicamente altri servizi.

Thymeleaf è un motore di template per Java utilizzato per generare e gestire contenuti HTML, XML, JS e CSS. I parametri vengono passati al template attraverso il Model, che viene popolato dal controller

durante la gestione delle richieste.
