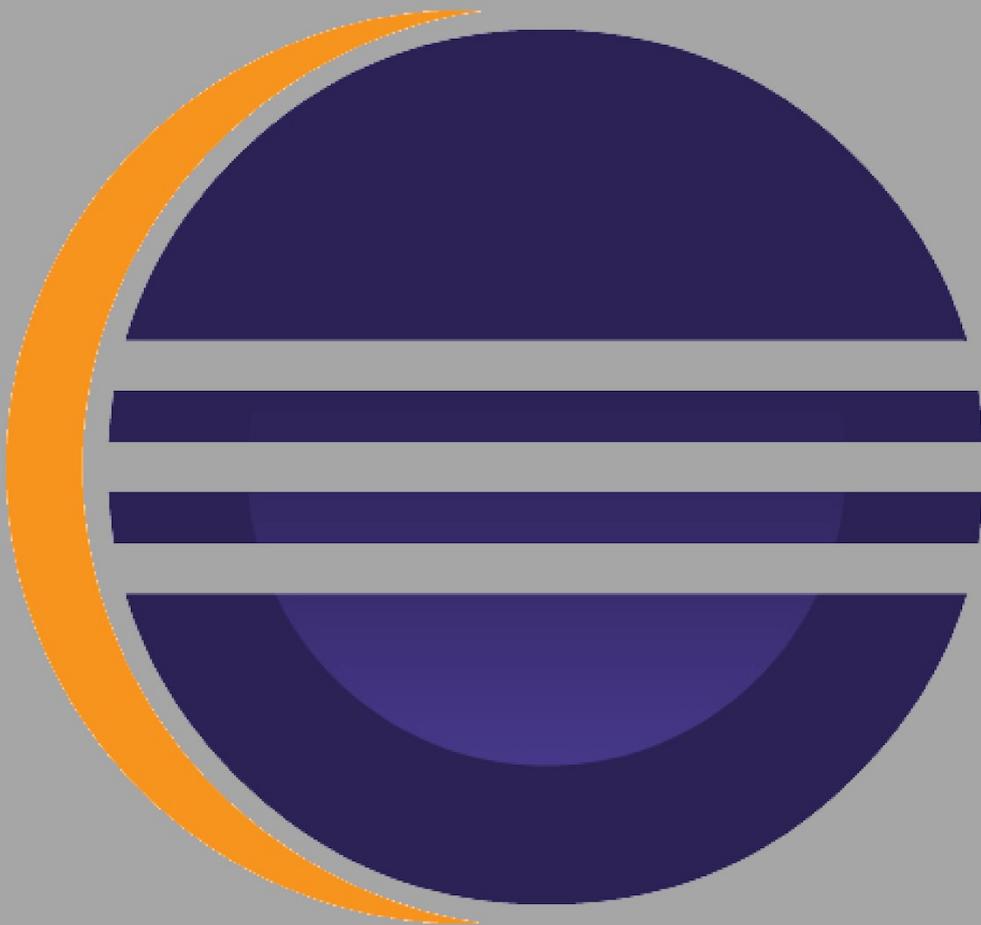


Teoria

Programmazione distribuita



"The ide less blasphemed by programmers"

CARMINE D'ANGELO
EMANUELE VITALE

I SISTEMI DISTRIBUITI

Un **sistema distribuito** consiste di un insieme di macchine, ognuna gestita in maniera autonoma, connesse attraverso una rete. Ogni nodo del sistema distribuito esegue un insieme di componenti che comunicano e coordinano il proprio lavoro attraverso uno strato software detto **middleware**, in maniera che l'utente percepisce il sistema come un'unica entità integrata. I sistemi distribuiti rispondono a motivazioni sia di tipo economico che di natura tecnologica.

- **Motivazioni economiche:** i sistemi distribuiti 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. Quindi, la necessità di affrontare in tempi brevi l'integrazione dei sistemi di Information Technology di aziende diverse, che si sono fuse insieme, richiede un'infrastruttura versatile e agile, che permetta di poter essere operativi in pochissimo tempo. I sistemi informativi di aziende che vengono separate dalla "casa madre" in un meccanismo di cosiddetto "*downsizing*" devono mantenere un certo livello di integrazione con le aziende del gruppo, in una sorta di federazione di sistemi che complica la gestione, prevedendo tre livelli di accesso al sistema informativo: dall'interno della azienda, dall'interno della federazione di aziende e dall'esterno.
I sistemi distribuiti rispondono anche alla esigenza del mercato specifico dell'Information Technology, dove il tempo necessario per poter arrivare al prodotto finale, dalla ideazione, progettazione e realizzazione (il cosiddetto *time to market*) deve essere reso quanto più breve possibile, sia per il ricambio tecnologico continuo, ma anche perché le richieste dei consumatori variano significativamente in poco tempo. Sistemi che devono offrire funzionalità complesse sono tipicamente assemblati utilizzando componenti preesistenti (i cosiddetti prodotti *off the shelf*).
- **Motivazioni Tecnologiche:** lo sviluppo dell'informatica è sempre stato condotto dal rapidissimo sviluppo delle tecnologie hardware. Le capacità di calcolo, comunicazione, memorizzazione che vengono oggigiorno offerte verranno rapidamente superate nel giro di pochissimi anni. Diverse "leggi" empiriche elaborate negli anni si sono provate fedeli nel prevedere la velocità di evoluzione. Ad esempio, ben conosciuta è la Legge di Moore che afferma che la densità dei transistor nei processori si raddoppia ogni 24 mesi. In pratica, questa legge afferma che la potenza di calcolo raddoppia ogni 24 mesi e quindi guadagna un ordine di grandezza (quindi 10 volte) in poco più di 5 anni. Esistono altre leggi empiriche riguardo lo sviluppo delle reti: la Legge di Gilder afferma che la capacità di trasmissione si raddoppia ogni anno, così come il traffico di Internet, mentre per quanto riguarda la capacità di memorizzazione, la Legge di Shugard suggerisce che ogni 4 anni aumenta di un fattore 10.

Notevoli sono le problematiche da affrontare per lo sviluppo di sistemi distribuiti. Innanzitutto, le difficoltà sorgono della complessità dell'ambiente in cui i sistemi distribuiti si trovano ad operare: ambienti dove `e necessario trattare con i malfunzionamenti di parte delle componenti (riconoscimento e recovery), così come `e necessario trattare con i problemi di latenza, quelli generati dall'accesso concorrente alle risorse e quelli derivanti dalla necessità di ottimizzare il carico di lavoro tra i diversi nodi. Tutte queste problematiche vanno affrontate all'interno di un panorama tecnologico in continua evoluzione (cambiamenti ed upgrade di tecnologie hardware e software) e di uno scenario economico in altrettanto rapida mutazione, che richiede la scalabilità dei servizi offerti. Affianco a queste problematiche ci sono quelle dovute alla difficoltà della realizzazione di sistemi eterogenei e complessi, che, spesso, devono trattare con problemi di portabilità, di mancanza di sistemi evoluti di debugging e di tecniche di progettazione software che si basano su approcci non orientati ad oggetti, che non favoriscono la riusabilità e l'estendibilità dei sistemi. La riusabilità e l'integrabilità di soluzioni diverse rappresenta un obiettivo importante.

NOTA:

Esistono altre “leggi” che fanno riferimento al valore aggiunto di una rete di N utenti:

- **Legge di Sarnoff:** afferma che il valore di una rete di broadcasting è direttamente proporzionale al numero degli utenti. Più persone sono collegate, maggiore è il valore della rete. Ha crescita lineare in N. Con N e M utenti attivi sarebbe $N + M$.
- **Legge di Metcalfe:** indica che il valore di un sistema di comunicazione cresce con il quadrato del numero di persone collegate. Ha crescita quadratica in N (N^2). Con N e M utenti attivi avremo $N^2 + M^2 + 2NM$.
- **Legge di Reed:** l'utilità delle grandi reti, formate da reti di reti (con particolare riferimento alle reti di relazione sociale) cresce esponenzialmente con la dimensione della rete. Con N e M utenti attivi avremo $2^N * 2^M - (2^N + 2^M)$.

Un modello di riferimento: Open Distributed Processing

Allo scopo di facilitare lo sviluppo dei sistemi distribuiti, è importante la condivisione di un modello comune in maniera da essere indipendente dalla specifica implementazione tecnologica. Un *modello di riferimento* serve a questo scopo, potendo essere utilizzato anche come terreno comune per la comunicazione durante le fasi iniziali di progettazione. Il modello RM-ODP si basa ed integra il modello tradizionale di rete proposto da ISO/OSI su sette layer, aggiungendo ai 6 già presenti un 7° chiamato application. Questo modello ha come obiettivo quello di gestire i problemi di comunicazione in un sistema rispetto ai problemi di connessione. La standardizzazione che viene presentata da RM-ODP è motivata dalla diffusione di applicazioni e sistemi distribuiti che devono assicurare uso di tecnologie e modelli ampiamente condivisi ed aperti, flessibilità, integrazione e interoperabilità.

Le caratteristiche di un sistema distribuito

Possiamo identificare le caratteristiche di un sistema distribuito attraverso una serie di *parole chiave*:

- **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 che permettono di gestire in maniera più “semplice” la sincronizzazione.
- **Assenza di uno stato globale:** non esiste una maniera per poter determinare lo stato globale del sistema, in quanto la distanza e la eterogeneità del sistema non permette di definire con certezza lo stato in cui si trova ciascun nodo.
- **Malfunzionamenti parziali:** ogni componente di un sistema distribuito può smettere di funzionare correttamente, in maniera indipendente dalle altre componenti e questo fallimento non deve inficiare le funzionalità che sono localizzate altrove nel sistema distribuito.
- **Eterogeneità:** un sistema distribuito per sua stessa definizione è eterogeneo per tecnologia sia hardware che software.
- **Autonomia:** un sistema distribuito non ha un singolo punto dal quale esso può essere controllato, coordinato e gestito.
- **Evoluzione:** i sistemi distribuiti devono assecondare la evoluzione dell’ambiente all’interno del quale vengono realizzati e forniscono le loro funzionalità. Questo significa che un sistema distribuito può cambiare anche in maniera sostanziale durante la sua vita, sia perché cambia l’ambiente sia perché cambia la tecnologia utilizzata. La flessibilità di un sistema distribuito deve assicurare che la migrazione verso ambienti diversi, tecnologie differenti e applicazioni nuove può essere assecondata con successo e senza costi eccessivi.

- **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.

Requisiti di non funzionalità

La realizzazione di un sistema distribuito non è agevole e comporta la necessità di considerare vari aspetti generali e globali della architettura, standardizzati in maniera tale che possano servire da specifica per i vari fornitori di piattaforme hardware e software allo scopo di fornire strumenti adeguati per rendere più agevole la progettazione, implementazione e manutenzione di un sistema distribuito. Questi aspetti specifici di un sistema distribuito non fanno parte dei requisiti funzionali del sistema. Hanno invece a che fare con i cosiddetti requisiti non funzionali, che indicano essenzialmente la qualità del sistema. Questi requisiti non funzionali specificano che la progettazione deve puntare a realizzare sistemi distribuiti che hanno le seguenti caratteristiche:

- **aperti:** in modo da supportare la portabilità di esecuzione e di interoperabilità attraverso interfacce e servizi ben documentati ed aderenti a standard noti e riconosciuti; questo aspetto risulta importante per poter far evolvere il sistema
- **integriti:** così da incorporare al proprio interno sistemi e risorse differenti senza dover utilizzare strumenti ad-hoc. Questo permette di trattare in maniera efficiente (economica) con il problema della eterogeneità hardware, software e di applicazioni.
- **Flessibili:** per poter evolvere e fare evolvere i sistemi distribuiti in maniera da integrare sistemi legacy al proprio interno. Un sistema distribuito dovrebbe anche essere in grado di riconfigurarsi dinamicamente.
- **Modulari:** in modo da permettere ad ogni componente di poter essere autonoma ma con un grado di interdipendenza verso il resto del sistema, quindi si possono aggiungere componenti senza creare problemi.
- **Supportino la federazione di sistemi:** in modo da unire diversi sistemi, 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:** per poter fornire i servizi con vincoli di tempo, di disponibilità e di affidabilità, anche in presenza di malfunzionamenti parziali, che in un sistema distribuito devono sempre essere considerati possibili ed inevitabili. La **tolleranza ai malfunzionamenti** è una delle principali richieste di qualità del servizio di un sistema distribuito in quanto deve raggirare i malfunzionamenti utilizzando (dinamicamente) componenti alternative per fornire funzionalità che alcune componenti non sono in grado temporaneamente di fornire.
- **Scalabili:** perché qualsiasi sistema distribuito accessibile da Internet può essere soggetto a picchi di carico non prevedibili e deve essere in grado di gestirli, e anche un'eventuale evoluzione che può portare ad un aumento di utenti che accedono al servizio.
- **Sicuri:** così che utenti non autorizzati non possano accedere a dati sensibili.
- **Trasparenza:** mascherando i dettagli e le differenze dell' architettura sottostante che assicura la distribuzione dei servizi sulle componenti del sistema. Questa caratteristica risulta centrale per poter permettere l' agevole progettazione ed implementazione: il progettista/programmatore deve avere un certo grado di indipendenza dai dettagli della distribuzione dell' architettura.

La trasparenza di un sistema distribuito

la definizione della *trasparenza* di un sistema è la seguente: i dettagli del sistema che offre le funzionalità operative sono nascosti agli utenti. Questo permette al progettista/sviluppatore di lavorare in un ambiente che non fornisce informazioni specifiche sull'architettura del sistema, il progettista/sviluppatore ignora, ad esempio, l'eterogeneità delle componenti, e richiede al sistema di fornire un certo tipo ed un certo livello di trasparenza su una certa parte della sua applicazione, attraverso una richiesta al sistema. La trasparenza serve anche a permettere un alto riuso delle applicazioni sviluppate che, proprio perché sviluppate nella totale trasparenza dei dettagli sottostanti, possono essere riutilizzate in contesti diversi e su sistemi diversi.

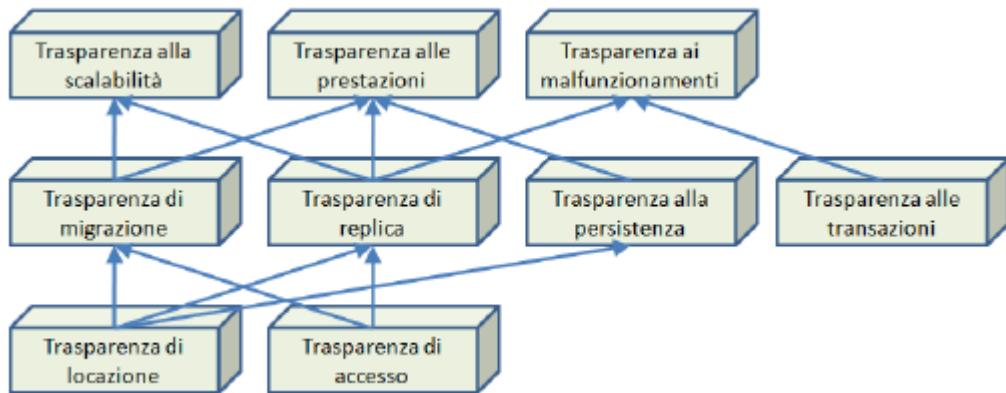
Abbiamo diversi tipi di trasparenza:

- **Trasparenza di accesso:** nasconde le differenze nella rappresentazione dei dati e nel meccanismo di invocazione per permettere l'interoperabilità tra oggetti. Questo significa anche che gli oggetti devono essere accessibili attraverso la stessa interfaccia, sia che siano acceduti da locale sia che siano acceduti da remoto. In questa maniera, un oggetto può essere facilmente spostato a run-time da un nodo ad un altro.
- **Trasparenza di locazione:** non permette di utilizzare informazioni circa la localizzazione nel sistema di una particolare componente, che viene identificata ed utilizzata in maniera indipendente dalla sua posizione. Questo tipo di distribuzione fornisce una vista logica del sistema di *naming*, in modo da disaccoppiare il nome da una posizione all'interno della rete.
- **Trasparenza di migrazione:** il compito di questo tipo di trasparenza è quello di nascondere la possibilità che il sistema faccia migrare un oggetto da un nodo ad un altro, continuando ad essere raggiungibile ed utilizzabile da altri oggetti. Questo viene utilizzato per ottimizzare le prestazioni del sistema (bilanciando il carico tra i nodi oppure riducendo la latenza per accedere ad una componente) o anche per anticipare malfunzionamenti o riconfigurazioni del sistema. La trasparenza di migrazione dipende dalla trasparenza di accesso (che permette di accedere ad un oggetto, anche se locale, solo attraverso la propria interfaccia che viene usata da remoto) e dalla trasparenza di locazione (che nasconde la locazione fisica di un oggetto, permettendone l'accesso attraverso un sistema di naming logico fornito dal sistema).
- **Trasparenza di replica:** Con questo tipo di trasparenza, il sistema maschera il fatto che una singola componente viene replicata da un certo numero di copie (dette *repliche*) che vengono posizionate su altri nodi del sistema, e che offrono esattamente lo stesso tipo di servizio della componente originale. Anche questo tipo di trasparenza dipende da quella di accesso e di locazione.
- **Trasparenza alle transazioni:** un sistema distribuito è implicitamente concorrente, in quanto la esistenza di diverse risorse accessibili da diversi nodi rende la concorrenza la regola e non l'eccezione. La 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. Sia l'utente che il progettista e lo sviluppatore sono ignari delle attività che vengono svolte per assicurare la atomicità delle operazioni e possono semplicemente ritenersi gli unici utenti all'interno del sistema.
- La **trasparenza ai malfunzionamenti** nasconde ad un oggetto il malfunzionamento di oggetti con i quali sta inter-operando. Questo tipo di trasparenza si poggia, ovviamente, sulla trasparenza di replica, in quanto quest'ultima fornisce la possibilità di poter ripetere trasparentemente le operazioni che si erano iniziate su una replica di un oggetto, potendo rieseguirle su una altra replica. Ma si basa anche sulla trasparenza alle transazioni, in quanto operazioni complesse eseguite come una transazione, se interrotte a causa di un malfunzionamento non vengono confermate (*commit*) e quindi non alterano lo stato della risorsa e possono essere ripetute su una replica.

- **Trasparenza alla persistenza:** questo tipo di trasparenza scherma l'utente dalle operazioni che compie il sistema per rendere persistente un oggetto. Infatti, per ottimizzare le prestazioni del sistema, gli oggetti di utilizzo raro non vengono mantenuti attivi ma vengono de-attivati, e memorizzati all'interno della memoria secondaria e riattivati poi successivamente quando c'è ne è una richiesta. La trasparenza alla persistenza si basa, ovviamente, sulla trasparenza di locazione, in quanto l'accesso indipendente dalla posizione fisica dell'oggetto permette una riattivazione dell'oggetto anche su nodi diversi da quelli su cui era stato de-attivato.
- **Trasparenza alla scalabilità:** un sistema viene detto scalabile quando è in grado di poter servire carichi di lavoro via via crescenti senza dover modificare la propria architettura e la propria organizzazione. La trasparenza alla scalabilità assicura che il progettista/sviluppatore non deve curarsi dei dettagli di come il proprio servizio scalerà al crescere delle richieste, ma sarà il sistema che provvederà, attraverso il meccanismo di replica e di migrazione, a fare in modo che le nuove risorse aggiunte al sistema vengano utilizzate per fare fronte al carico crescente.
- **Trasparenza alle prestazioni:** il sistema distribuito che assicura questo tipo di trasparenza rende il progettista/sviluppatore ignaro dei meccanismi che vengono utilizzati per ottimizzare le prestazioni del sistema, durante la fornitura di servizi. In particolare, il sistema può provvedere ad implementare politiche di bilanciamento del carico, spostando componenti da nodi carichi di lavoro verso nodi che hanno maggiori disponibilità di calcolo a disposizione, oppure politiche di minimizzazione della latenza, avvicinando (repliche di) componenti su nodi più vicini agli utenti che li usano più frequentemente, oppure politiche di ottimizzazione delle risorse di memoria, che prevedono la inattivazione di oggetti che non vengono usati frequentemente e che possono essere re-attivati se necessario. Per questo motivo, la trasparenza alle prestazioni si appoggia sulla trasparenza alla migrazione, alla replica ed alla persistenza.

In conclusione, la trasparenza offerta da un sistema viene definita durante la fase di progettazione, la scelta di quali tipi di trasparenza utilizzare richiede un'attenta valutazione per via dei costi.

Di seguito uno schema di come appaiono i 3 livelli di trasparenza:



MIDDLEWARE

Middleware ad oggetti distribuiti

I sistemi distribuiti basati su oggetti distribuiti sono uno degli strumenti più utilizzati per assicurare estendibilità, affidabilità e scalabilità. Gli oggetti distribuiti si trovano alla confluenza di due aree della tecnologia software: i sistemi distribuiti che puntano a realizzare un unico sistema integrato basato sulle risorse offerte da diversi calcolatori messi in rete, e lo sviluppo e la programmazione orientata agli oggetti, che si focalizzano sulle modalità per ridurre la complessità dei sistemi software.

Gli oggetti distribuiti hanno come obiettivo quello di realizzare servizi distribuiti riutilizzabili, in maniera tale che siano efficienti, flessibili, sicuri e robusti. Il tutto basato su un'architettura che utilizza come risorse dei nodi eterogenei, sia per l'hardware che per il software. Questa integrazione viene realizzata attraverso il middleware ad oggetti distribuiti, che risiede tra le applicazioni e lo strato sistema operativo. La comunicazione attraverso i nodi potrebbe avvenire attraverso le primitive di comunicazione di rete che vengono offerte dal S.O. di ogni singolo nodo, risulta però essere troppo complesso per la programmazione di applicazioni, in quanto i programmatore dovrebbero prendersi cura di tutti i dettagli di basso livello, come quello di trasformare le strutture dati del livello applicazione in stream di byte oppure datagram che possono essere trasmessi su rete. Lo scopo del middleware è quello di rendere semplici questi compiti e di fornire delle astrazioni appropriate per i programmatore, che ben si integrino con i tradizionali strumenti che essi utilizzano per realizzare l'applicazione.

Il middleware ad oggetti distribuiti può essere suddiviso in tre strati:

- **Middleware di infrastruttura:** si occupa delle comunicazioni tra i sistemi operativi diversi e della gestione della concorrenza per evitare lo sforzo di utilizzare meccanismi non portabili.
- **Middleware di distribuzione:** basa i suoi servizi sul middleware d'infrastruttura per automatizzare operazioni comuni per la comunicazione. Tra i compiti più importanti abbiamo:
 - richiedere un servizio ad un altro nodo potendo inviare parametri (marshalling), i parametri sono inviati tra piattaforme hardware/software diversi;
 - Utilizzare lo stesso canale di comunicazione (socket, ecc.) per diverse richieste, oppure utilizzare una sola macro-richiesta che include diverse richieste;
 - Modificare la semantica delle operazioni di invocazione oltre quella tradizionale di unicast, come ad esempio la multicast (invocazione su diversi oggetti in contemporanea) oppure l'attivazione di oggetti in risposta ad invocazione di servizi;
 - Riconoscimento e gestione dei malfunzionamenti di rete, attuando strategie per permettere che l'applicazione possa effettuare il recovery di uno strato semanticamente corretto dell'applicazione.
- **Middleware per servizi comuni di supporto:** un layer che serve a fornire i servizi comuni a tutte le applicazioni distribuite. Sono riutilizzabili in tutti i contesti, come la persistenza degli oggetti, la sicurezza e la gestione delle transazioni.

L'obiettivo di questi tre livelli è quello di assicurare che il programmatore di applicazione concentri i propri sforzi sullo sviluppo della logica di business dell'applicazione e che non si debba interessare direttamente dei dettagli di comunicazione a livello di rete. Forniscono anche astrazioni utili per il programmatore, ben integrate nell'ambiente di sviluppo dell'applicazione. Infine, proprio perché il middleware ad oggetti astrae la parte di distribuzione e di servizio delle applicazioni distribuite, lo sviluppo avviene ad alto livello, permettendo di poter utilizzare e riutilizzare framework e soluzioni, appoggiandosi a metodologie evolute di ingegneria del software per rendere maggiormente proficua, efficiente e efficace la soluzione proposta. Il middleware nei sistemi distribuiti può essere esplicito (invocato esplicitamente dalle operazioni) oppure implicito (se ne usufruisce senza invocarlo direttamente, le richieste sono codificate in un file di metadati di descrizione).

Evoluzione del middleware

RPC

La computazione distribuita basata su oggetti distribuiti si basa sull'astrazione fornita da RPC (Remote Procedure Call), un modello presentato negli anni 80 che permetteva ad una procedura presente su una macchina di invocarne un'altra presente su una macchina diversa. RPC fu la prima tecnologia a dover risolvere problemi significativi e complessi di eterogeneità e di concorrenza, i dati venivano trasmessi su stream di byte tramite socket. RPC imponeva la sincronia dell'invocazione, cioè bloccava il client finché il server non avesse risposto all'invocazione remota della procedura. Le operazioni di sincronizzazione, marshalling, data representation e comunicazione tra client e server venivano implementate attraverso gli stub sia lato client che server, e interfacciavano il processo lato chiamante con il processo che offriva la procedura.

Da RPC al Middleware ad Oggetti Distribuiti

Negli anni 90 si iniziò a passare ad un modello che unisse la tecnologia software di programmazione ad oggetti con quella dei sistemi distribuiti. Questo cambiamento è stato motivato dall'evoluzione e dallo sviluppo di sistemi distribuiti molto complessi, che al crescere delle risorse hardware e di comunicazione, diventavano sempre più ampi, complessi, eterogenei e difficili da gestire me da manutenere. Questo portò a sistemi distribuiti che per la prima volta fossero scalabili, tolleranti ai malfunzionamenti, estendibili e di agevole gestione.

Middleware ad oggetti distribuiti nel modello a componenti

Le tecnologie basate sugli oggetti distribuiti hanno rappresentato un passo avanti per la progettazione e realizzazione di sistemi affidabili e scalabili. Il loro obiettivo di assicurare funzionalità e l'efficienza di un sistema ad oggetti distribuiti in un ambiente eterogeneo è stato ottenuto con tecniche diverse che hanno consentito lo sviluppo e diffusione dei sistemi distribuiti. Le tecnologie sviluppate hanno influenzato lo sviluppo delle tecnologie basate sul modello a componenti distribuite (Enterprise Computing).

Una componente distribuita è un blocco riutilizzabile di software che può essere combinato in un sistema distribuito per realizzare funzionalità, queste ultime sono esposte tramite un'interfaccia. Quello che differenzia queste componenti da altri moduli software (oggetti, ecc.) è che possono essere combinati sotto forma di eseguibili binari, piuttosto che sotto forma di azioni da compiere sul codice sorgente. Il modello a oggetti distribuiti si basa sul middleware implicito, così i servizi sono forniti in maniera completamente trasparente allo sviluppatore realizzando una maggiore interoperabilità tra produttori di software diversi.

Il middleware ad oggetti distribuiti offre il supporto per le invocazioni di operazioni tra layer diversi di architetture software, che ha portato alla scalabilità, affidabilità e all'ampia diffusione di applicazioni web e sistemi distribuiti che abbiamo oggi.

Infatti il compito importante e fondamentale che viene svolto dai sistemi a oggetti distribuiti è quello di assicurare interoperabilità tra componenti distribuite, permettendo così di poter creare sistemi che offrono scalabilità, tolleranza ai malfunzionamenti, estendibilità e facilità di gestione.

Esempi di soluzioni basate su oggetti distribuiti

- **CORBA:** fu presentato nel 1991, permetteva ad oggetti distribuiti scritti in diversi linguaggi e quindi eterogenei, di comunicare e collaborare per realizzare un'applicazione distribuita. Tutta la sua architettura si basava sull'invocazione di un servizio (metodo) su un oggetto distribuito, i servizi erano scritti in un linguaggio specifico per CORBA, mentre l'implementazione degli oggetti potevano essere in C, C++, Java, ecc. I problemi che ha avuto furono dovuti alla sua complessità e alla mancanza di interoperabilità.
- **Java RMI:** Java Remote Method Invocation fu proposto da Sun, era interno all'ambiente di Java ed era usato per realizzare applicazioni distribuite basate su oggetti. Eredita molte caratteristiche da Java che l'hanno portato ad essere molto utilizzato per la realizzazione di

applicazione distribuiti, e successivamente è diventato uno strumento base di comunicazione per Java Enterprise Edition.

- **.NET:** Microsoft .NET Framework è la soluzione di Microsoft per realizzare applicazioni ed include un'ampia libreria di soluzioni ed un ambiente di esecuzione chiamato CLR (Common Language Runtime) in modo tale che le applicazioni possano essere scritte in uno dei linguaggi supportati da Microsoft(C#, Visual basic). .Net Remoting è una tecnologia per realizzare comunicazione tra processi basati su oggetti distribuiti, fu presentata in .NET Framework 2.0 ed infine inclusa in WFC (Windows Communication Foundation) all'interno di .NET Framework 3.0.
- **Java Enterprise:** in Java Enterprise risulta cruciale il ruolo di Java Remote Method Invocation per la comunicazione. Infatti la buona pratica di progettazione suggerisce di separare il layer di presentazione da quello business, il primo infatti si incarica di assemblare il contenuto che viene fornito all'interfaccia utente di un'applicazione, mentre il secondo si occupa di gestire la logica dell'applicazione e di interfacciarsi con i sistemi per la gestione dei dati e applicazioni legacy. Java Remote Method Invocation permette di poter accedere da macchine diverse ai servizi forniti da una componente distribuita (Enterprise Java Bean).

THREAD

Prologo ai thread

Con lo sviluppo tecnologico si cerca sempre di più di aumentare la velocità dei processori, ma questo comporta che i nuovi software che saranno creati cercheranno di prendere sempre più risorse.

L'obiettivo comune tra tutti i produttori è:

- ridurre lo spessore del package
- incrementare le prestazioni
- migliorare le caratteristiche termiche e le capacità di connessione (verso sensori o attuatori) dei chip.

I produttori cercano di attuare queste soluzioni perché i transistor non possono essere miniaturizzati all'infinito. Infatti più piccoli diventano i transistori più entra in gioco la termodinamica, che disturba la crescita secondo la legge di Moore del numero di transistor. Per risolvere questo problema si passa dall'avere un singolo processore ad averne di più con ognuno una cache e che condividono tutta la stessa memoria (memoria condivisa).



Fino a non molto tempo fa, i miglioramenti della tecnologia portavano ad un miglioramento delle prestazioni software, infatti CPU con clock maggiore eseguivano il codice con più velocità; adesso il miglioramento è dato dall'avere più transistor ma organizzati in core multipli, che per essere utilizzati hanno bisogno di software in grado di sfruttare il parallelismo tra le applicazioni.

L'accesso in memoria con un singolo core è facile, un singolo thread accede alla memoria (strutturata in oggetti) e compie le sue operazioni.

L'accesso in memoria con i multi core è molto complicato, si possono avere diversi problemi:

- **legati alla sincronia:** infatti i diversi core potrebbero essere asincroni tra di loro

- **ritardi impercettibili:** Cache misses (breve), Page faults (lungo), Context-switch dallo scheduler (molto lungo).

Nota:

Page fault: Il **page fault** è un'eccezione di tipo trap, generata quando un processo cerca di accedere ad una pagina che è presente nel suo spazio di indirizzamento virtuale, ma che non è presente nella memoria fisica, poiché mai stata caricata in essa o perché, precedentemente, spostata su disco di archiviazione.

Context-switch: la **commutazione di contesto** (in inglese **context switch**) è una particolare operazione del sistema operativo che cambia il processo correntemente in esecuzione su una CPU. Questo avviene all'occorrenza di una qualsiasi interruzione dovuta allo scheduler, ma anche a interruzioni dovute a errori di altri processi o segnali; viene effettuato per salvare tutte le informazioni necessarie al riavvio successivo del processo.

Programmazione distribuita e concorrente

La programmazione distribuita implica la conoscenza (di base) della programmazione concorrente, che coinvolge diversi processi che vengono eseguiti insieme.

Abbiamo tre tipi di programmazione concorrente:

- programmazione concorrente eseguita su calcolatori diversi
- processi concorrenti sulla stessa macchina (multitasking): processo padre che genera processi figli per fork()
- programmazione concorrente nello stesso processo: “processi lightweight” all’interno del processo: thread.

Multitasking e multithread

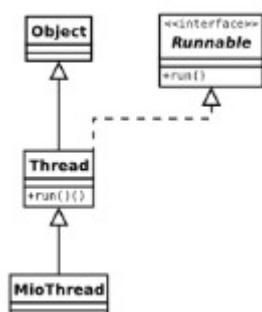
- I S.O. multitasking creano l’illusione (per l’utente) di una macchina completamente dedicata, ma durante l’interazione dell’utente con il proprio programma, il S.O. ha il tempo di servire altri utenti.
- Il multithread è l’estensione del multitask riferito ad un singolo programma, è in grado di eseguire più thread “contemporaneamente”.
- Thread: anche detti processi “light-weight”, a differenza dei processi hanno a disposizione e condividono gli stessi dati (trovandosi all’interno dello stesso processo).
- Meccanismo di comunicazione attraverso una memoria condivisa: uno strumento efficace per costruire programmi che necessitano di svolgere in “parallelo” più compiti, ma è fonte di problemi.

Thread in java

Un processo è un ambiente di esecuzione con uno spazio di memoria privato. La cooperazione tra processi avviene attraverso l’InterProcess Communication come pipe e socket. I thread (lightweight process) esistono all’interno di un processo, condividendo tra loro memoria e file aperti. In java ogni applicazione ha almeno un thread utente (main thread), più alcuni thread di sistema che gestiscono la memoria e i segnali. Il main thread può creare e far partire diversi altri thread.

I thread in java sono oggetti, istanze quindi di una classe Thread. L’evoluzione di java ha portato a due modalità di gestione de thread:

1. istanziare un oggetto thread ogni volta che serve un task asincrono (creazione e gestione a cura del programmatore).
2. Astrarre la gestione, passando un task ad un executor.



Come istanziare un thread in java

1. Estendere la classe `java.lang.Thread`
2. Riscrivere (ridefinire, override) il metodo `run()` nella sottoclasse di `Thread`
3. Creare un'istanza di questa classe derivata
4. Richiamare il metodo `start()` su questa istanza.

```
public class HelloThread extends Thread {  
  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new HelloThread()).start();  
    }  
}
```

Ma estendendo la classe `thread` non potremo più estendere nessun'altra classe e questa è un'importante limitazione, per questo si preferisce usare un'interfaccia:

1. Creare una classe che implementa `java.lang.Runnable` (la stessa implementata anche da `Thread`)
2. Implementare il metodo `run()` in questa classe.
3. Creare un'istanza di `Thread` passandogli un'istanza di questa classe.
4. Richiamare il metodo `start()` sull'istanza di `Thread`.

```
public class HelloRunnable implements Runnable {  
  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new Thread(new HelloRunnable())).start();  
    }  
}
```

Metodi `run()` e `start()`

run:

- intestazione: `public void run()`
- cosa fa: il metodo `run` contiene il codice che sarà eseguito dal thread

start:

- intestazione: `public void start()`
- cosa fa: fa in modo che il thread inizi l'esecuzione; la Java Virtual Machine chiama il metodo `run` del thread su cui si è usato `start()`. Il risultato sarà che due thread vengono eseguiti contemporaneamente: il main thread (che ritorna dalla chiamata al metodo `start`) e l'altro thread (che esegue il suo metodo `run`). Un thread non può essere riavviato una volta completata l'esecuzione.

Alcuni metodi utili:

1) `sleep()`

```
public class SleepMessages {  
    public static void main(String args[])  
        throws InterruptedException {  
  
        String importantInfo[] = {  
            "Mares eat oats",  
            "Does eat oats",  
            "Little lambs eat ivy",  
            "A kid will eat ivy too"  
        };  
  
        for (int i=0; i<importantInfo.length; i++) {  
  
            Thread.sleep(4000);  
  
            System.out.println(importantInfo[i]);  
        }  
    }  
}
```

- Stampa 4 stringhe con un ritardo
- array stringhe
- per ogni stringa
- sospendo il thread per 4 secondi
- stampo
- eccezione lanciata dal thread corrente se interrotto mentre in `sleep()`

- Intestazione: public static void sleep(long millis) throws InterruptedException
- Cosa fa: fa in modo che il thread attualmente in esecuzione si interrompa (interrompendo temporaneamente l'esecuzione) per il numero specificato di millisecondi, in base alla precisione e all'accuratezza dei timer di sistema e degli scheduler. Il thread non perde la proprietà di alcun monitor.

2) Gli interrupt

Un interrupt è un'indicazione che un thread dovrebbe fermare quello che sta facendo e fare qualcos'altro. Un thread invia un interrupt invocando **Interrupt** sul thrad object che deve essere interrotto. Il programma diderà poi cosa fare per rispondere ad un interrupt.

```
// ...
for (int i = 0; i < importantInfo.length; i++) {
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        return;
    }
    System.out.println(importantInfo[i]);
}
// ...
```

- Nella sleep ()
- si intercetta la eccezione
- in caso si esce
- altrimenti (a fine sleep ()) si stampa

Un thread che non invoca un metodo che lancia l'eccezione InterruptedException, può controllare se è stato interrotto utilizzando Thread.interrupted().

```
//
...
if(Thread.interrupted())
{
    throw new InterruptedException();
}
```

Se è stato ricevuto un interrupt si lancia un eccezione gestita in una catch()

3) join()

Avvolte è necessario che un thread attenda il completamento di un altro thread, se *t* è l'oggetto il cui thread è in esecuzione allora avremo *t.join()*, che mette il thread corrente in pausa finché il thread *t* non termina. Si può specificare anche un periodo di attesa come parametro , il tempo però dipenderà dal S.O. quindi non sarà preciso. Il metodo join() risponde ad un interrupt lanciando InterruptedException.

Esempio:

```
public class SimpleThreads {
    static void threadMessage(String msg) {
        String tn = Thread.currentThread().getName();
        System.out.format("%s: %s%n", tn, msg);
    }

    private static class MessageLoop implements Runnable {
        public void run() {
            String impinf[] = {
                "Mares eat oats",
                "Does eat oats",
                "Little lambs eat ivy",
                "A kid will eat ivy too"
            };
            try {
                for (int i = 0; i < impinf.length; i++) {
                    Thread.sleep(4000);
                    threadMessage(impinf[i]);
                } // end for
            } catch (InterruptedException e) {
                threadMessage("I wasn't done!");
            } // end catch
        } // end run()
    } // end class MessageLoop
    // ...
}
```

- mostra un messaggio con il nome del thread
- stampa formattata
- interfaccia
- metodo che viene eseguito allo start del thread
- 4 stringhe
- blocco try ... catch
- pausa di 4 secondi
- poi stampa
- se interrotta

```

// ...
public static void main(String args[])
    throws InterruptedException {
long patience = 1000 * 60 * 60;

if (args.length > 0) {
    try {
        patience = Long.parseLong(args[0]) * 1000;
    } catch (NumberFormatException e) {
        System.err.println("Argument must be an
                           integer.");
        System.exit(1);
    }
}

threadMessage("Starting MessageLoop thread");
long startTime = System.currentTimeMillis();
Thread t = new Thread(new MessageLoop());
t.start() ;<br>
//...

```

- Lancia eccezione
- Ritardo in millisecondi (default 1 ora)
- Se c'è un argomento, è in secondi
- Controllo formato
- Prende il tempo di inizio
- Crea un oggetto Thread da MessageLoop
e lo fa partire

```

// ...
threadMessage("Waiting for MessageLoop to
               finish");

while (t.isAlive()) {
    threadMessage("Still waiting...");

    t.join(1000);

    if (((System.currentTimeMillis() - startTime) >
          patience)
        && t.isAlive()) {

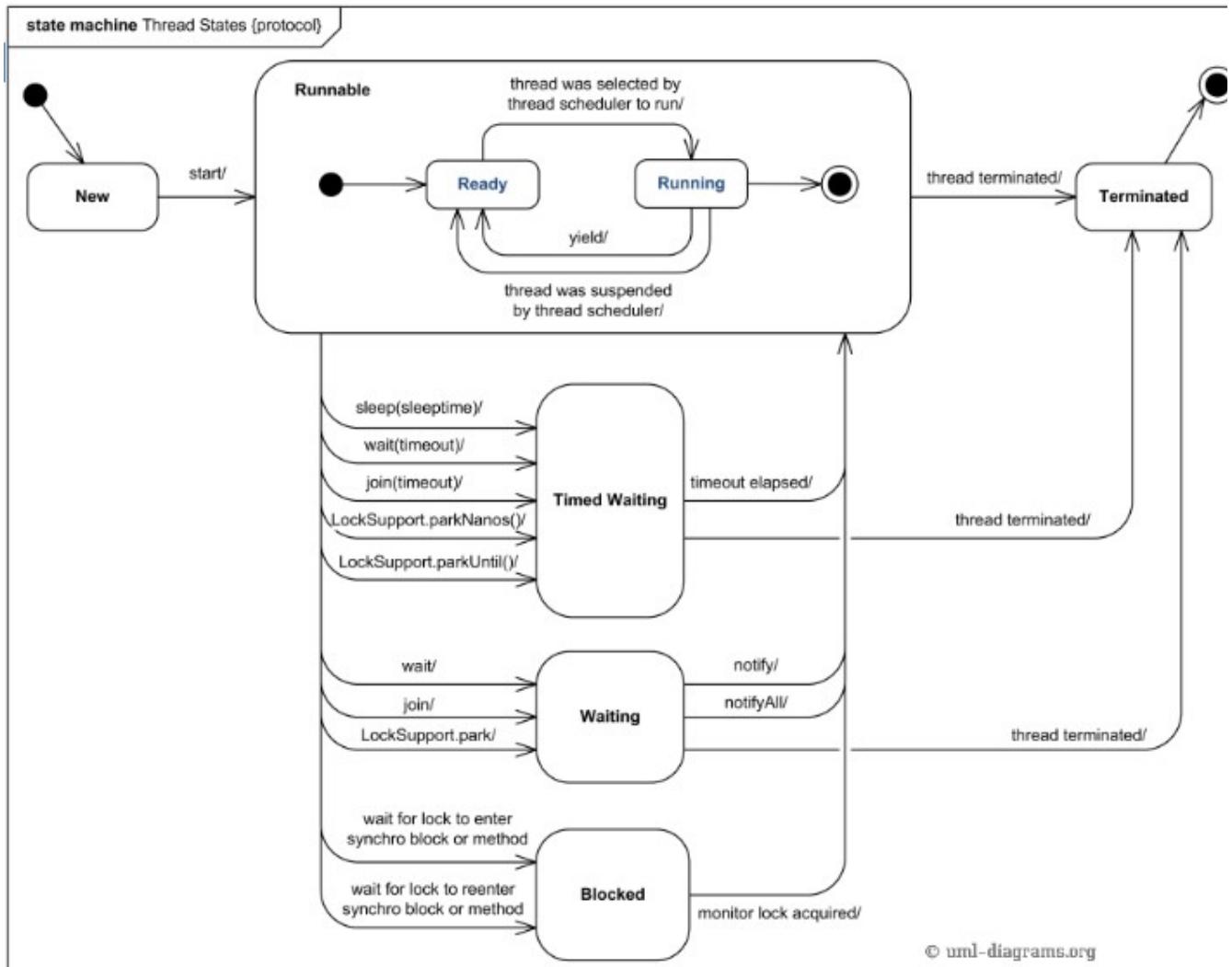
        threadMessage("Tired of waiting!");
        t.interrupt();

        t.join();
    }
}
threadMessage("Finally!");<br>
}

```

- Mentre MessageLoop è in vita ...
- ... aspetta 1 secondo al più
- Se la pazienza è scaduta ...
- ... e il thread è ancora vivo
- Lo si chiude e se ne attende la "fine"
- e si esce!

Ciclo di vita dei threads



Il thread quando viene creato si trova nello stato di **new**, all'invocazione di start parte il metodo run del Thread e passa da **new** a **Runnable**. Quindi sarà in stato **ready** nel sistema, quando viene selezionato passa da **ready** a **running**, qui ha due possibili sviluppi il thread termina e passa in stato **terminated** oppure il thread è sospeso dallo scheduler e torna in stato **ready**. Ci sono altre fasi nella vita dei threads e sono 3:

- **time waiting:** il thread passa in stato di attesa per un determinato periodo di tempo, può tornare in stato runnable o terminare. I metodi che lo portano in questo stato sono :
 - sleep()
 - wait(timeout)
 - join(timeout)
 - LockSupport.parkNanos()
 - LockSupport.parkUntil()
- **waiting:** il thread passa in stato di attesa, finché non viene sbloccato da un metodo, al suo sblocco può passare in runnable o terminated.
I metodi che lo portano in stato di attesa sono:
 - wait()
 - join()
 - LockSupport.park()

I metodi che lo fanno uscire dallo stato wait sono:

- notify()
- notifyAll()

- **blocked:** il thread entra in stato di blocco, ci entra in due casi:

- attende per accedere ad un blocco di istruzioni o un metodo sincronizzato
- attende per riaccedere ad un blocco di istruzioni o un metodo sincronizzato

Si sblocca nel momento in cui può accedere al blocco sincronizzato, passa in stato runnable.

Comunicazione tra thread

I thread comunicano principalmente condividendo accesso a campi (tipi primitivi) e campi che contengono riferimenti a oggetti. È una comunicazione molto efficiente (rispetto ad utilizzare la rete). Si possono verificare due tipi di errore: interferenza di thread, inconsistenza della memoria. Per risolvere questi problemi è necessaria la sincronizzazione, che a sua volta genera problemi di contesa, infatti quando più thread cercano di accedere alla stessa risorsa simultaneamente si ha il deadlock o il livelock.

Interferenza: Quando il risultato di un'operazione dipende dall'ordine di esecuzione di diversi thread, infatti nell'esecuzione di due thread non sappiamo quali operazioni vengono eseguite prima. Es:

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

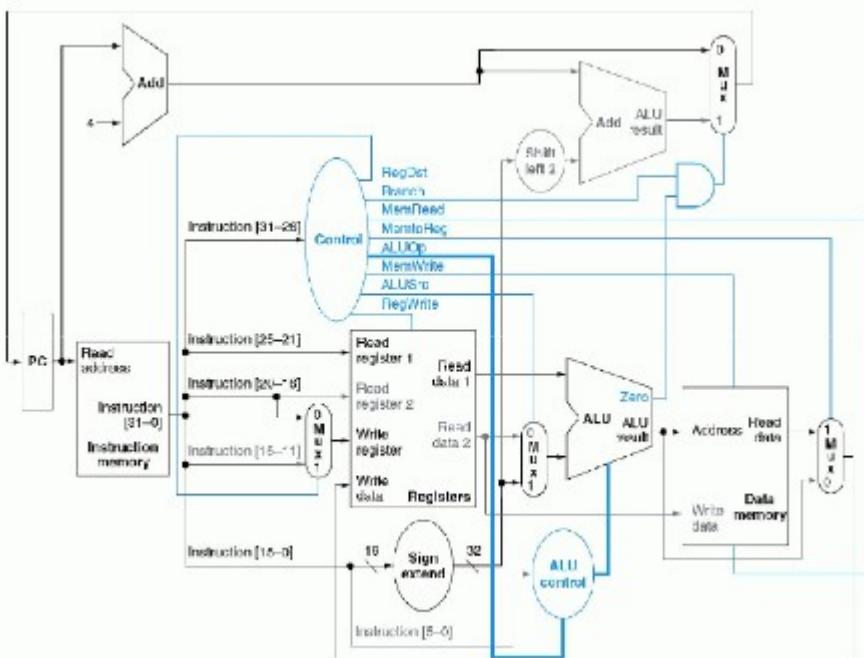
    public void decrement() {
        c--;
    }

    public int value() { ←
        return c;
    }
}
```

- variabile privata
- metodo di incremento
- metodo di decremento

metodo di accesso

L'operazione `c++` è eseguita in tre passi: fetch operando dalla locazione di memoria di `c` nel registro, incremento del registro, memorizzazione in `c`.



Quindi nell'esempio riportato sopra se avviamo due thread potremmo avere molti possibili casi, in cui le operazione vengano eseguite in modo casuale e quindi non è sempre detto che il risultato sia corretto.

Questo è dovuto ad un problema di race condition.

Race condition: quando il risultato di un'operazione dipende dall'ordine di esecuzione di diversi thread.

Gli errori dovuti ad una race condition sono tipicamente transienti e difficile da riprodurre (debugging difficile!), oltre alla transienza e irriplicabilità, l'uso del debugger può alterare l'ordine di esecuzione dei task.

Curiosità

Questi tipi di bug sono chiamati Heisenbug e richiamano il principio di indeterminazione di Heisenberg in fisica quantistica. Questo principio dice che non è possibile misurare simultaneamente la posizione ed il momento di una particella.

Inconsistenza della memoria: quando thread diversi hanno visione diverse dei dati, la soluzione è la **happens-before** che ci garantisce che la memoria scritta da un thread è visibile anche agli altri thread.

Es:

Inizializziamo un contatore a 0 condiviso fra i due thread. Supponiamo che il thread A faccia `contatore++` e che subito dopo B esegua `System.out.println(contatore)`. Il valore stampato da B potrebbe essere 0 perché non abbiamo certezza che la modifica di A sia subito visibile a B.

```
class Foo {  
    int bar = 0;  
    public static void main(String args[]) {  
        (new Foo()).unsafeCall();  
    }  
  
    void unsafeCall () {  
        final Foo thisObj = this;  
  
        Runnable r = new Runnable () {  
            public void run () {  
                thisObj.bar = 1;  
            }  
        };  
  
        Thread t = new Thread(r);  
        t.start();  
  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
  
        System.out.println( "bar = " + bar );  
    }  
}
```

- Dichiarazione variabile
- Nel main si chiama un metodo
- Definizione metodo
- Variabile final
- Un thread che mette 1 in bar
- Lancio del thread
- Si attende 1 secondo...
- ... e si stampa: può essere 0 o 1!

Relazioni happens-before

Una maniera per stabilire questo tipo di relazioni è la sincronizzazione. Due operazioni che abbiamo descritto introducono questo tipo di relazione:

- `Thread.start()`: gli effetti del codice che ha condotto alla creazione sono visibili al nuovo thread.
- `Thread.join()`: quando la terminazione di un thread A causa il return della `join()` di B, tutte le istruzioni di A sono in happens-before con le istruzioni di B che seguono la `join`.

Un'altra maniera è rendere la variabile *volatile*: cioè una scrittura a un campo volatile happens-before ogni successiva lettura della variabile (da parte di qualsiasi thread).

SINCRONIZZAZIONE

La legge di Amdahl

La legge di Amdahl dice:

Lo speedup che si ottiene eseguendo il programma X su n processori, dove p è la parte di X che si può parallelizzare è:
$$S = \frac{1}{1 - p + \frac{p}{n}}$$
.

Infatti Lo speedup S di un programma X è il rapporto tra il tempo impiegato da un processore per eseguire X rispetto al tempo impiegato da n processori per eseguire X . Sia p la parte del programma X che è possibile parallelizzare con n processori la parte parallela prende tempo p/n mentre la parte sequenziale prende tempo $(1-p)$.

La legge di Amdahl ci dice che la parte sequenziale del programma rallenta significativamente per qualsiasi speedup che possiamo pensare di ottenere. Quindi per velocizzare un programma bisogna investire sul rendere parallela la parte predominante in un programma rispetto a quella sequenziale.

Sincronizzazione tra thread

I metodi sincronizzati (synchronized) sono un costrutto del linguaggio Java, che permette di risolvere semplicemente gli errori di concorrenza al costo di inefficienza. Per rendere un metodo sincronizzato, basta aggiungere synchronized alla sua dichiarazione:

```
public class SynchronizedCounter {  
    private int c = 0;  
    public synchronized void increment() {  
        c++;  
    }  
    public synchronized void decrement() {  
        c--;  
    }  
    public synchronized int value() {  
        return c;  
    }  
}
```

Cosa comporta l'uso di metodi sincronizzati?

Non è possibile che due esecuzioni dello stesso metodo sullo stesso oggetto siano interfogliate.

Quando un thread esegue un metodo sincronizzato per un oggetto, gli altri thread che invocano metodi sincronizzati dello stesso oggetto sono sospesi fino a quando il primo thread non ha finito.

Quando un thread esce da un metodo sincronizzato, allora si stabilisce una relazione happens-before con tutte le successive invocazioni dello stesso metodo sullo stesso oggetto, i cambi allo stato, effettuati dal thread appena uscito sono visibili a tutti i thread. I costruttori non possono essere sincronizzati (solo il thread che crea dovrebbe avere accesso all'oggetto in costruzione).

Leaking (ESCAPE) del riferimento

Quando si costruisce un oggetto che sarà condiviso, non si deve far "scappare" il riferimento prima che questo si riferisca ad un oggetto completamente costruito.

Ad esempio, supponiamo di:

avere una List chiamata instances che mantiene tutte le istanze di un oggetto di una classe che nel costruttore si inserisce instances.add(this).

Altri thread potrebbero usare instances e accedere all'oggetto prima che sia completamente costruito.

Lock intrinseci

Un lock intrinseco (o monitor lock) è una entità associata ad ogni oggetti. Un lock intrinseco garantisce sia accesso esclusivo sia accesso consistente (relazione happens-before). Un thread deve:

- acquisire il lock di un oggetto
- rilasciarlo quando ha terminato

Quando il lock che possedeva viene rilasciato, viene stabilita la relazione happens-before. Quando un thread segue un metodo sincronizzato di un oggetto ne acquisisce il lock, e lo rilascia al termine (anche se c'è una eccezione).

Istruzioni sincronizzate

Un altro modo di creare codice sincronizzato è specificando di quale oggetto si usa il lock:

```
public void addName(String name) {  
    synchronized(this) {  
        lastName = name;  
        nameCount++;  
    }  
    nameList.add(name);  
}
```

In questa maniera, si sincronizzano gli accessi a `lastName` e `nameCount` solo durante la modifica, ma poi si provvede in maniera concorrente all'inserimento in lista `nameList.add(name)`.

Sincronizzazione a grana fine

Le istruzioni sincronizzate sono utili per migliorare la concorrenza a grana fine. Supponiamo che una classe MsLunch abbia due campi `c1` e `c2` che non sono mai usati insieme. Tutti gli update a questi campi devono essere sincronizzati, ma non c'è necessità di impedire che un aggiornamento di `c1` venga interfogliato con un aggiornamento di `c2`.

```
public class MsLunch {  
    private long c1 = 0;  
    private long c2 = 0;  
    private Object lock1 = new Object();  
    private Object lock2 = new Object();  
  
    public void inc1() {  
        // molto codice  
        synchronized(lock1) {  
            c1++;  
        }  
        // molto codice  
    }  
  
    public void inc2() {  
        // molto codice  
        synchronized(lock2) {  
            c2++;  
        }  
        // molto codice  
    }  
}
```

- Due variabili
- Dichiarazione di due lock
- Accesso a `c1` con il lock1
- Accesso a `c2` con il lock2
- Con `synchronized` sul metodo si sequenzializza tutto (Amdahl!)
- Con `synchronized` su `this` non sarebbero indipendenti!

Sincronizzazione di metodi statici

I metodi statici si riferiscono alla classe, un metodo statico synchronized previene l'esecuzione interfoliata di tutti gli altri metodi statici. In pratica, si acquisisce il lock dell'oggetto *ClassName.class*.

```
public static void foo() {  
    synchronized (ClassName.class){  
        //Body  
    }  
}
```

Nota: i 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 quell'istanza.

Azioni atomiche

Azioni che non sono interrompibili: o si completano nella loro interezza oppure per niente. Si possono specificare azioni atomiche in Java per:

- read e write su variabili di riferimento e su tipi primitivi (a parte long e double)
- read e write su tutte le variabili volatile

Write a variabili volatile stabiliscono una relazione happens-before con le letture successive. Tipi di dato definiti in *java.util.concurrent.atomic*

Esempio:

```
import java.util.concurrent.atomic.AtomicInteger;  
  
class AtomicCounter {  
    private AtomicInteger c = new AtomicInteger(0);  
  
    public void increment() {  
        c.incrementAndGet();  
    }  
  
    public void decrement() {  
        c.decrementAndGet();  
    }  
  
    public int value() {  
        return c.get();  
    }  
}
```

- Package
- Classe
- Variabile istanza
- Metodo non sincronizzato
- Uso di metodi atomici
- Uso di metodi atomici
- Lettura

Deadlock

Quando due thread sono bloccati ognuno in attesa dell'altro. In questa maniera il nostro programma concorrente si blocca e non c'è maniera di sbloccarlo. Infatti ogni thread è in attesa di una risorsa mantenuta da un altro thread e non esiste modo di progredire.

Starvation

Describe la situazione in cui un thread non è in grado di ottenere accesso regolare alle risorse condivise e non riesce ad avanzare. Risorse rese non disponibili per lunghi periodi da thread "avidì" (greedy).

Livelock

Il thread risponde in base all'azione di un altro thread, il quale risponde all'azione del primo. Sono bloccati in maniera diversa rispetto al deadlock.

I due thread non sono bloccati (non è un deadlock!) ma sono occupati a rispondere alle azioni dell'altro. Anche se sono in esecuzione, non c'è progresso!

Che situazioni si possono verificare con i thread?

I due thread che cercano di accedere a due metodi, a volte allo stesso metodo, contemporaneamente. I metodi possono essere di istanza oppure statici. I metodi possono essere sincronizzati oppure non sincronizzati. Un metodo può essere invocato da uno solo dei thread oppure da entrambi i thread.

Per ogni combinazione si dovrebbe verificare il comportamento, e capirlo completamente.

Il Singleton

È noto nel design pattern della programmazione ad oggetti, restringe l'istanziazione da parte di una classe ad 1 oggetto. È usato per la memorizzazione di stato (esempio: Printer, File, Resource Manager).

Il singleton sfrutta la Lazy allocation cioè l'allocazione avviene solo quando utilizzato per la prima volta.



```
class Singleton
{
    private static Singleton instance; // variabile di classe che mantiene l'istanza
    private Singleton() { // costruttore privato
        // ...
    }
    public static Singleton getInstance() // metodo di classe che restituisce un Singleton
    {
        if (instance == null) { // se non è stato ancora creata un'istanza
            instance = new Singleton(); // allora la crea
        }
        return instance; // restituisci la nuova istanza a disposizione
    }
}
```

Nel caso ci siano più thread il loro interleaving può causare errori, quindi più singleton.

Un'altra soluzione

```
class Singleton
{
    private static Singleton instance; // variabile di classe che mantiene l'istanza
    private Singleton() { // costruttore privato
        // ...
    }
    public static synchronized Singleton getInstance() // metodo sincronizzato
    {
        if (instance == null) { // se non è stato ancora creata un'istanza
            instance = new Singleton(); // allora la crea
        }
        return instance; // restituisci la nuova istanza a disposizione
    }
}
```

La sincronizzazione serve solo la prima volta La sincronizzazione del metodo potrebbe diminuire le performance di un fattore 10. L'overhead di ottenere e rilasciare il lock ogni volta che il metodo viene chiamato sembra non necessario.

Un'altra soluzione

```
class Singleton
{
    private static Singleton instance; // variabile di classe che mantiene l'istanza
    private Singleton() {           // costruttore privato
        // ...
    }
    public static Singleton getInstance() // metodo sincronizzato
    {
        if (instance == null) {      // se non è stato ancora creata un'istanza
            synchronized (Singleton.class) // in maniera sincronizzata
            {
                instance = new Singleton(); // instanzia il singleton
            }
        }
        return instance;           // restituisci la nuova istanza a disposizione
    }
}
```

Non funziona, se due thread A e B arrivano ad eseguire insieme l'if abbiamo che A entra nella sezione critica, crea l'istanza, esce dalla sezione critica e restituisce il primo singleton. A questo punto B entra e crea un nuovo singleton sovrascrivendo il primo.

Double-checked locking

```
public static Singleton getInstance()
{
    if (instance == null)           // se non esiste l'istanza
    {
        synchronized(Singleton.class) si entra in CS
        {
            if (instance == null) // si controlla che nel waiting non sia cambiata la situazione
                instance = new Singleton(); // se è il caso crea un nuovo Singleton
        }
    }
    return instance;
}
```

Problema: nonostante sia pubblicato e ampiamente utilizzato, non è garantito che funzioni. Perché anche se la chiamata al costruttore sembra essere prima dell'assegnazione di instance, questo non è detto che accada. E quindi un Singleton non inizializzato potrebbe essere assegnato a instance e se un altro thread controlla il valore, lo potrà poi usare in maniera scorretta.

Soluzioni

La variabile *instance* resa volatile funziona (java 5 in poi). Altrimenti si possono usare le classi statiche con l'idioma “Initialization-on-demand holder”

```
public class Something {
    private Something() {}

    private static class LazyHolder {
        private static final Something INSTANCE = new Something();
    }

    public static Something getInstance() {
        return LazyHolder.INSTANCE;
    }
}
```

LazyHolder è inizializzata dalla JVM solo quando serve (alla prima `getInstance()`). Essendo un inizializzatore statico, viene eseguito una sola volta (al caricamento) e stabilisce una relazione happens-before su tutte le altre operazioni sulla classe.

SOCKET

Programmazione con i socket

Java fornisce le API per la programmazione di rete all'interno del package `java.net` dove sono presenti classi per trattare indirizzi IPv4 e IPv6, un esempio è la classe `InetAddress`.

La comunicazione tra programmi su Internet avviene tramite il protocollo TCP/IP, vengono usati *socket* che permettono di ricevere e trasmettere dati. Questo permette al programmatore di vedere la comunicazione come un semplice stream di dati. Il protocollo TCP (Transmission Control Protocol) offre una connessione affidabile, mentre l'UDP (User Datagram Protocol) non fornisce una connessione tra due punti ma permette semplicemente di inviare pacchetti dati (datagram).

La scelta di quale protocollo utilizzare dipende essenzialmente da:

- dalla natura dell'applicazione , cioè se richiede lo scambio di flussi di dati oppure di messaggi di dimensione limitata;
- dalla quantità di overhead che si vuole pagare: le comunicazioni TCP richiedono maggiori risorse di calcolo per offrire l'affidabilità della comunicazione;
- dalla criticità dell'affidabilità della comunicazione: se, insomma, perdere un pacchetto nella comunicazione rappresenta un problema oppure può essere supportato dall'applicazione.

La trasmissione tramite socket avviene assegnando una specifica porta che serve ad identificare tra tutti i pacchetti che arrivano quali sono quelli destinati ad una determinata applicazione.

I socket TCP

I socket TCP sono gli endpoint di una comunicazione bidirezionale sulla rete che unisce due programmi. Ad ogni socket viene assegnato un numero di porta che indica quale applicazione è incaricata di trattare i dati, quindi il socket sarà univocamente identificato dall'indirizzo IP e dal numero di porta.

Nell'uso dei socket abbiamo due computer coinvolti, chiamati client e server:

- **Server:** è in esecuzione ed attende che qualche client richieda la connessione.
- **Client:** il programma conosce l'indirizzo della macchina su cui è in esecuzione il server ed il suo numero di porta. Il client deve anche comunicare al server il numero di port locale sul quale riceverà i dati.

Il procedimento di connessione prevede che il server debba *accettare* la connessione e che assegni un nuovo socket per la comunicazione bidirezionale tra client e server. In questo modo il server può sempre tornare ad accettare connessioni da altri client, il server può lanciare anche dei thread per ogni socket stabilito in modo da permettere la gestione concorrente delle comunicazioni con tutti i client.

Queste funzionalità sono offerte dal package `java.net` che offre due classi per i socket: `ServerSocket` e `Socket`. La classe `ServerSocket` implementa un socket di connessione che attende richieste da parte di client; quando ne riceve una, assegna un socket alla connessione bidirezionale restituendo l'oggetto socket che viene utilizzato per la connessione tra client e server.

Gli stream

La comunicazione tra client e server avviene attraverso la scrittura e la lettura di stream associati con il socket e che permettono una facile interazione per poter trasmettere istanze di classi Java attraverso un meccanismo di *serializzazione*. Gli stream I/O sono utili per trattare una sequenza di dati che può essere "diretta a"/"proveniente da" diverse entità (es: file, memoria, socket, ecc.).

Gli stream sono presenti nel package `java.io`. Una sua classe è `InputStream` (la sua controparte è `OutputStream`). La sottoclasse più importante è `ObjectInputStream` che fornisce il meccanismo di deserializzazione quando riceve un oggetto serializzato da `ObjectOutputStream`. Gli oggetti che possono essere trasmessi su questo tipo di stream devono implementare l'interfaccia `Serializable` o `Externalizable`. I tipi primitivi possono essere letti tramite `readByte()` o `readFloat()`.

Gli stream vengono creati utilizzando il meccanismo di wrapping, cioè ogni classe via via più specializzata prende come argomento per il costruttore un'istanza delle classi più alte nella gerarchia. Es: `ObjectInputStream inStream = new ObjectInputStream(Socket.getInputStream());`; A questo scopo, tra le classi che derivano da `Reader` esiste la classe `InputStreamReader` che rappresenta la connessione tra gli stream binari e quelli di testo. In effetti un oggetto di `InputStreamReader` legge bytes dall'InputStream che gli è stato passato (come parametro al costruttore) e li decodifica in caratteri utilizzando la codifica del sistema in suo.

Un'altra classe utile è la classe `BufferedReader` che fornisce una bufferizzazione di uno stream di inout di testo allo scopo di aumentare l'efficienza. Un esempio consiste nell'applicarla allo stream di input di testo `InputStreamReader` ottenuto convertendo lo stream binario offerto dalla classe `System` mediante `System.in`. Es: `BufferedReader bin = new BufferedReader(new InputStreamReader(System.in));`; La sequenza di istruzioni classicamente utilizzata per accedere agli stream di un socket lato server è:

```
ServerSocket serverSocket = new ServerSocket(9000);
socket = serverSocket.accept();
System.out.println("Accettata una connessione... attendo comandi");
ObjectInputStream inStream = new ObjectInputStream(socket.getInputStream());
ObjectOutputStream outStream = new ObjectOutputStream(socket.getOutputStream());
```

Da un oggetto locale...

Nell'esempio `ImpiegatoServer` e `Client`, la classe `ImpiegatoServer` funge da fornitore di servizio cioè server mentre la classe `Clien`t ricopre il ruolo di fruitore di servizio cioè client. È importante distinguere quando client e server sono usati nell'ambito di architetture distribuite e quando si descrive invece il ruolo di oggetti distribuiti; in quest'ultimo caso, infatti ci si riferisce ad una singola chiamata di metodo: l'oggetto server rappresenta l'oggetto che riceve l'invocazione che viene effettuata dall'oggetto client. Il ruolo può invertirsi qualora il "server" diventi client per un'invocazione di un metodo su un altro oggetto remoto. Il client in questo caso non farà altro che instanziare un oggetto della classe `ImpiegatoServer` ed usarne I metodi.

```
1 // Client dell'esempio ImpiegatoLocale
2 public class Client []
3 {
4     public static void main(String[] args)
5     {
6         // Si instanzia un impiegato
7         ImpiegatoServer imp = new ImpiegatoServer("Mario Rossi", "01721", 30000);
8
9         System.out.println("Nome: " + imp.getNome());
10        System.out.println("ID: " + imp.getID());
11        System.out.println("Stipendio: " + imp.getStipendio());
12
13        System.out.println("Aumentiamo lo stipendio di 1000 euro");
14        System.out.println("Ora lo stipendio è di " + imp.aumentaStipendio(1000) + " euro");
15    }
16 }
17 }
```

... all'oggetto remoto

Per rendere distribuito un semplice programma adottiamo il principio dell'astrazione, introducendo uno strato di software che viene utilizzato per nascondere al programmatore la maggior parte del lavoro necessario per permettere l'invocazione remota di metodi: **stub** e **skeleton**.

Lo stub è un oggetto che si trova sul client che rappresenta l'oggetto server in locale verso il client: presenta ed espone gli stessi metodi che vengono esposti sul server. Il suo compito principale è quello di comunicare con lo **skeleton** che si trova sul lato server. Ogni chiamata del client verso I metodi remoti dello stub genera una comunicazione tra lo stub e lo skeleton; quest'ultimo si occupa di effettuare l'invocazione del metodo richiesto sull'oggetto server, ricevere il valore restituito dal

metodo e comunicarlo allo stub che lo restituisce verso il client. Ogni comunicazione fra stub e skeleton avviene attraverso un protocollo comune che deve prevedere come si indica il metodo da eseguire e come si inviano I parametri ed il valore restituito. L'oggetto client conosce I metodi disponibili sul server perchè sia lo stub che lo skeleton implementano un'interfaccia comune detta **interfaccia remota** dove sono definiti I metodi che devono essere invocati in remoto.

Il server e l'interfaccia remota

```
public interface Impiegato {
    public String getNome() throws Throwable;
    public String getID() throws Throwable;
    public int getStipendio() throws Throwable;
    public int aumentaStipendio (int diQuanto) throws Throwable;
}
```

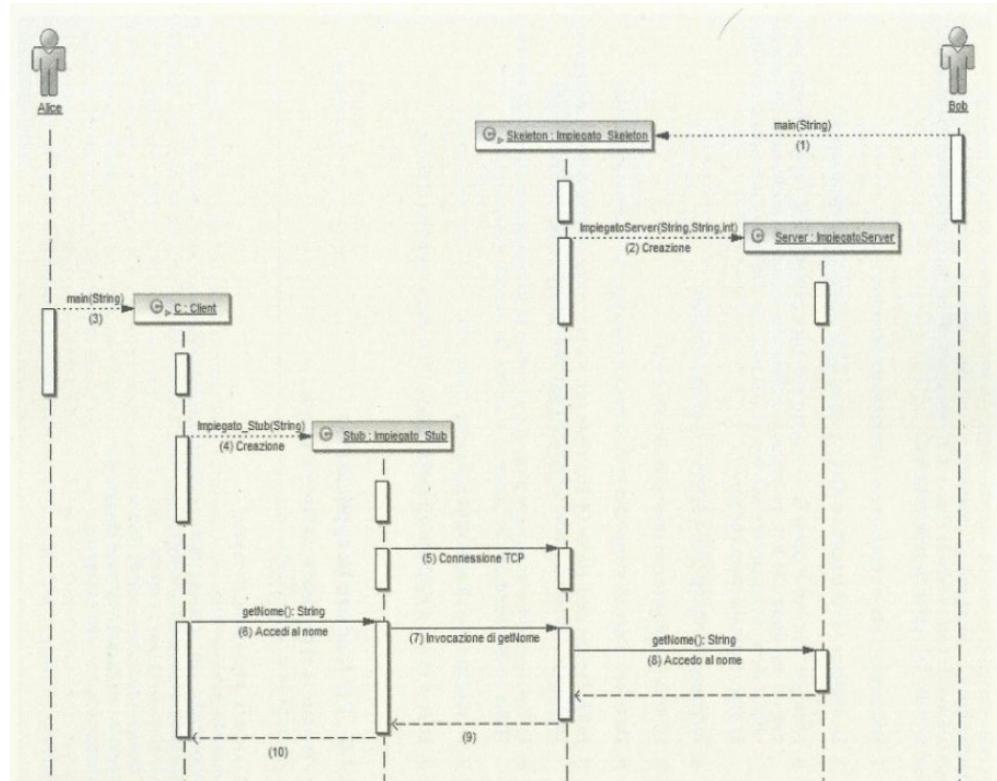
È importante notare che ogni metodo, essendo remoto, viene dichiarato tale da poter lanciare eccezioni, chiaramente dovute all'utilizzo della rete durante l'effettiva chiamata.
(Per il resto del codice consultare il libro da pagina 38 a 42).

Esempio della sequenza delle invocazioni del codice *impiegatoServer*

1. Emanuele lancia il server, eseguendo il programma Skeleton.
2. Lo Skeleton viene lanciato e istanzia l'oggetto ImpiegatoServer.
3. Carmine può lanciare il client Client. Va notato che Carmine può lanciare in ogni momento, ma dopo che

Emanuele ha lanciato lo Skeleton, in quanto le chiamate su oggetti remoti sono inherentemente *sincroni*, cioè si suppone che siano contemporaneamente in esecuzione sia cliente che server.

4. All'interno di Client viene istanziato lo Stub.
5. Lo Stub si connette con un socket allo Skeleton.
6. Da Client viene fatta l'invocazione a *getNome()* di Impiegato_Stub.
7. In Impiegato_Stub viene passata l'invocazione del metodo all'oggetto Impiegato_Skeleton.
8. In impiegato_Skeleton viene ricevuta sul socket la stringa inviata e viene effettuata la chiamata al server del metodo *getNome()* ...



9. ... risultano che viene inviato sul socket.
10. Lo stub riceve il risultato sul socket e viene restituito al client.

Indirizzamento dell'oggetto remoto

Il problema che ci si pone ora è come si può reperire il riferimento all'oggetto remoto da parte del/dei client?. La soluzione più semplice è quella di prevedere che ci sia un servizio disponibile, e la cui locazione è conosciuta, che permetta di poter reperire l'indirizzo dell'oggetto di cui sappiamo solamente il nome, cioè un identificativo. Questo serve a realizzare la trasparenza di locazione, ma solo parzialmente, in quanto il servizio di naming è localizzato infatti si deve sapere dove si trova per utilizzarlo.

(Per il codice consultare il libro da pagina 45 a 48).

```
public class RecordRegistro implements Serializable {
```

```
    private static final long serialVersionUID = -4147133786465982122L;
    private String nome;
    private String indirizzo;

    public RecordRegistro(String n, String i){
        nome = n;
        indirizzo = i;
    }

    public String getNome() {return nome;}

    public String getIndirizzo() {return indirizzo;}
}
```

```
public class Impiegato_Stub implements Impiegato{
```

```
    Socket socket;
    ObjectOutputStream out;
    ObjectInputStream in;

    // Costruttore
    public Impiegato_Stub(String host) throws Throwable {
        socket = new Socket(host, 9000);
        out = new ObjectOutputStream(socket.getOutputStream());
        in = new ObjectInputStream(socket.getInputStream());
    }
```

```
    /** Viene inviata allo skeleton una stringa con il nome del metodo da invocare */
    // Metodo remoto di accesso al Nome
    public String getNome() throws Throwable {
        out.writeObject("getNome");
        out.flush();
        return (String) in.readObject();
    }
```

```
    // Metodo remoto di accesso all'ID
    public String getID() throws Throwable {
        out.writeObject("getID");
        out.flush();
        return (String) in.readObject();
    }
```

```
    // Metodo remoto di accesso allo stipendio
    public int getStipendio() throws Throwable {
        out.writeObject("getStipendio");
        out.flush();
        return in.readInt();
    }
```

```

/** prima di essere aumentato lo stipendio lo skeleton attenderà il parametro e lo passerà poi al metodo del server*/
// Metodo remoto di aumento dello stipendio
public int aumentaStipendio(int diQuanto) throws Throwable {
    out.writeObject("aumentaStipendio");
    out.writeInt(diQuanto);
    out.flush();
    return in.readInt();
}

// Metodo (locale) per la chiusura del socket con lo skeleton
public void close() {
    try {
        socket.close();
    } catch (IOException e){
        System.out.println("Chiusura socket non effettuata con successo!");
    }
}
public class Impiegato_Skeleton extends Thread{

    /** Il suo compito principale risulta nel ricevere le richieste
     * d'invocazione remote che vengono inviate inviate dallo stub
     * e a provvedere ad effettuarle su un oggetto ImpiegatoServer
     * che ha provveduto ad istanziare
     */
    static Logger logger = Logger.getLogger("global");
    public ImpiegatoServer mioServer;

    // Costruttore
    public Impiegato_Skeleton(ImpiegatoServer Server)
    {
        mioServer = Server;
    }

    public static void main (String args[]) {
        // Istanziazione oggetto Server
        ImpiegatoServer impiegato = new ImpiegatoServer("Carmine D'Angelo",
            "231096",50000);
        // Istanziazione skeleton e sua esecuzione
        Impiegato_Skeleton skel = new Impiegato_Skeleton(impiegato);
        skel.start();
        // Registrazione dell'oggetto
        InetAddress addr = null;
        String ipAddrStr = "";
        try { // Trovo l'indirizzo IP locale
            addr = InetAddress.getLocalHost();
            byte[] ipAddr = addr.getAddress();
            // Convertiamo l'indirizzo
            for (int i = 0; i < ipAddr.length; i++) {
                if (i > 0) ipAddrStr += ".";
                ipAddrStr += ipAddr[i]&0xFF;// unsigned bytes
            }
        } catch (UnknownHostException e) {
            logger.severe("Non conosco localhost?? "+ e.getMessage());
            e.printStackTrace();
        }
        logger.info("Registro l'oggetto all'indirizzo " + ipAddrStr);
        RecordRegistro r = new RecordRegistro("D'Angelo", ipAddrStr);
        Socket socket;
        try {
            socket = new Socket(args[0], 7000);
            ObjectOutputStream sock_out = new ObjectOutputStream(socket.getOutputStream());
            sock_out.writeObject(r);
            sock_out.flush();
            socket.close();
        } catch (UnknownHostException e) {
            logger.severe("Host non conosciuto "+ e.getMessage());
            e.printStackTrace();
        } catch (IOException e) {

```

```

        logger.severe("Problemi sul socket per la registrazione "+ e.getMessage());
        e.printStackTrace();
    }

}

// Attività dello skeleton
public void run() {
    Socket socket = null;
    String metodo;
    int parametro;
    System.out.println("Attendo connessioni...");
    try {
        // Creazione ed accept sul socket
        ServerSocket serverSocket = new ServerSocket(9000);
        socket = serverSocket.accept();
        System.out.println("Accettata una connessione... attendo comandi");

        ObjectOutputStream outStream = new ObjectOutputStream(socket.getOutputStream());
        ObjectInputStream inStream = new ObjectInputStream(socket.getInputStream());
        while(true) {
            // Lettura del nome del metodo da eseguire
            metodo = (String) inStream.readObject();
            if(metodo.equals("getNome")) {
                outStream.writeObject(mioServer.getNome());
                outStream.flush();
            }
            else if(metodo.equals("getID")) {
                outStream.writeObject(mioServer.getID());
                outStream.flush();
            }
            else if(metodo.equals("getStipendio")) {
                outStream.writeInt(mioServer.getStipendio());
                outStream.flush();
            }
            else if(metodo.equals("aumentaStipendio")) {
                parametro = inStream.readInt();
                outStream.writeInt(mioServer.aumentaStipendio(parametro));
                outStream.flush();
            }
            else break;
        } // Fine while
    } catch (EOFException t) {
        System.out.println("Terminata la connessione");
    }
    catch (Throwable t){
        t.printStackTrace();
        System.out.println("Skeleton "+t.getMessage());
    }
    finally { // Chiusura del socket e terminazione del programma
        try {
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(0);
        }
    }
} // Fine run()
}

public class ImpiegatoServer implements Impiegato{
    //Server dell'applicazione ImpiegatoRemoto

    // Variabili d'istanza
    private String nome;
    private String ID;
    private int stipendio;

    // Costruttore
    public ImpiegatoServer (String n, String i, int s){
        nome = n;

```

```

        ID = i;
        stipendio = s;
    }

    // Metodi di accesso
    public String getNome() {return nome;}

    public String getID() {return ID;}

    public int getStipendio() {return stipendio;}

    // Cetodi specifici
    public int aumentaStipendio (int diQuanto)
    {
        if(diQuanto > 0)
            stipendio += diQuanto;
        return stipendio;
    }
}

public interface Impiegato {

    public String getNome() throws Throwable;
    public String getID() throws Throwable;
    public int getStipendio() throws Throwable;
    public int aumentaStipendio (int diQuanto) throws Throwable;
}

public class Client {
    // Client dell'esempio ImpiegatoRemoto

    static Logger logger = Logger.getLogger("global");

    public static void main(String[] args)
    {
        try {
            // Cerco l'host su cui è registrato il servizio Impiegato
            RecordRegistro r = new RecordRegistro("D'Angelo",null);
            Socket socket = new Socket(args[0],7000);// host su cui connettersi
            ObjectOutputStream sock_out = new ObjectOutputStream(socket.getOutputStream());
            sock_out.writeObject(r);
            sock_out.flush();
            // Aspetto la risposta
            ObjectInputStream sock_in = new ObjectInputStream(socket.getInputStream());
            RecordRegistro result = (RecordRegistro) sock_in.readObject();
            sock_in.close();
            // Se viene ottenuto un risultato, allora...
            if(result != null) {
                // ... uso l'indirizzo trovato
                Impiegato imp = new Impiegato_Stub(result.getIndirizzo()); // host

                System.out.println("Nome: " + imp.getNome());
                System.out.println("ID: " + imp.getID());
                System.out.println("Stipendio: " + imp.getStipendio());

                System.out.println("Aumentiamo lo stipendio di 1000 euro");
                System.out.println("Ora lo stipendio è di " + imp.aumentaStipendio(1000) + " euro");

                // Per poter usare close è necessario fare un casting, perchè imp implementa
                // un interfaccia remota e non conosce nient'altro dell'oggetto.
                ((Impiegato_Stub) imp).close();
            }
            else // Altrimenti non esiste un oggetto remoto con quel nome
                System.out.println("Non esiste un oggetto remoto con nome D'Angelo");
        } catch (Throwable t) {
            logger.severe("Lanciata Throwabe: " + t.getMessage());
            t.printStackTrace();
        }
    }
}

```

Funzionamento di getOutputStream()

Per un socket i deve sempre aprire prima lo stream di output, questo perché nel caso in cui si apre per prima lo stream di input l'applicazione resta bloccata aspettando di leggere qualcosa nell'header che non ci sarà, perché gli output stream non hanno ancora scritto.

Java RMI (Remote method invocation)

Java RMI è una libreria di Java che permette lo sviluppo di applicazioni distribuite, fornendo la possibilità di effettuare comunicazione remota tra i programmi scritti in Java. Il suo ruolo all'interno della Java Platform è quello di integration library situata al disopra delle librerie standard di Java. Le applicazioni RMI seguono un'architettura client-server, dove il server crea gli oggetti remoti e attende che gli oggetti client ne utilizzino i servizi.

Gli obiettivi della programmazione Java RMI

Java RMI stabilisce obiettivi che si basandosi sulle esperienze precedenti ci assicura un'efficace implementazione del modello ad' oggetti distribuiti. Infatti Java RMI punta ad assicurare la semplicità e integrazione dell'implementazione. Infatti è importante che il sistema sia semplice per un'agevolazione di utilizzo ed implementazione. Questo ne permette la diffusione ma impedisce anche utilizzi errati e scorretti delle potenzialità del sistema possano creare problemi ai sistemi realizzati.

Invocazione trasparente di metodi remoti

Java RMI deve poter offrire al programmatore un meccanismo semplice per l'invocazione di metodi che sono offerti da un oggetto remoto.

Integrazione in Java

Un'importante caratteristica è quella di fare in modo che il modello distribuito si integri in maniera naturale all'interno del linguaggio Java. Questo permette di offrire un ambiente familiare allo sviluppatore, che può usare tutto quello che viene utilizzato per l'uso degli oggetti locali. Java RMI fornisce anche un garbage collector distribuito in modo da preservare la modalità di gestione della memoria di Java che solleva il programmatore dal doversi occupare esplicitamente dell'allocazione e deallocazione della memoria. Infatti per le applicazioni distribuite visto che i server sono in esecuzione continua anche il più piccolo memory leak (un programma continua ad allocare memoria per un oggetto e non la dealloca mai) può portare in pochi giorni ad un server ad esaurire tutta la memoria.

Non trasparenza della natura locale/remota di un oggetto

Il fatto che un oggetto sia remoto oppure locale deve essere chiaro ed evidente, sia in fase di progettazione che in fase di implementazione.

Rendere minima la complessità di client e server

Si deve assicurare la minima complessità all'applicazione distribuita basata su Java RMI, compatibilmente con gli altri obiettivi. Il livello di complicazione che viene introdotto da un oggetto distribuito per l'oggetto client e per l'oggetto server deve essere limitato.

Preservare la sicurezza fornita da Java

Il modello ad' oggetti distribuito non deve alterare il livello di sicurezza che viene offerto da Java. Infatti la sicurezza e la robustezza del linguaggio sono al centro dell'attenzione dei progettisti fin dall'inizio visto la natura distribuita del linguaggio, che prevede l'esecuzione di programmi scaricati dalla rete. La sicurezza del linguaggio Java consiste nell'eseguire un'applicazione all'interno di una sandbox, un ambiente dedicato, ristretto e controllato, all'interno del quale le operazioni che il programma può eseguire non risultano pericolose.

Modalità d'invocazione

Java RMI fornisce diversi tipi di invocazione:

- **Unicast**: da un cliente verso un server
- **Multicast**: da un cliente verso diversi server replicati,

Inoltre deve essere possibile che l'oggetto server sia attivato solo al momento dell'invocazione e che i riferimenti ad oggetti persistenti siano persistenti.

Livelli di trasporto multipli

Java RMI deve essere aperto verso future espansioni che prevedano che il protocollo di trasporto che viene utilizzato possa essere modificato.

Il modello a oggetti distribuiti di Java RMI

Un oggetto remoto è un oggetto i cui metodi possono essere acceduti da un altro spazio di indirizzamento, e potenzialmente da un'altra macchina. La descrizione dei servizi offerti da remoto è contenuta all'interno di un'interfaccia remota che dichiara i metodi remoti. L'oggetto client di oggetti remoti server utilizza esclusivamente l'interfaccia remota dell'oggetto, non la sua implementazione. Questo garantisce che le funzionalità remote risultino astratte verso il client e disaccoppia le due implementazioni, permettendo ad esempio evoluzioni dell'oggetto server senza che il client debba essere modificato.

La struttura delle classi di Java RMI

Java RMI è contenuto in 5 package: *java.rmi* e *java.rmi.server* che contengono il meccanismo delle invocazioni remote, *java.rmi.activation* per gli oggetti attivabili, *java.rmi.dgc* per la Distributed Grabage Collection e *java.rmi.registry* per il servizio di localizzazione.

Interfacce ed eccezioni remote

Prima di definire un oggetto remoto si deve definire per lui un'interfaccia remota. Un'interfaccia remota per Java RMI deve estendere *java.rmi.Remote* che è un'interfaccia marker. Ogni metodo descritto deve soddisfare entrambe le seguenti condizioni:

- Un metodo remoto deve dichiarare esplicitamente l'eccezione *java.rmi.RemoteException*. In questa maniera si gestiscono l'eccezioni remote in maniera esplicita, questa eccezione deriva da *IOException* di Java.
- I parametri remoti di un metodo remoto devono essere dichiarati tramite la propria interfaccia remota, non utilizzando la classe dell'implementazione remota. Questo permetterà di poter passare riferimenti remoti sia come parametri che come valori restituiti.

Il meccanismo dell'interfaccia remota aggiunge un livello ulteriore di accessibilità ai modificatori di accesso dei metodi, infatti i metodi remoti di un'interfaccia remota sono più accessibili dei metodi *public* che risultano accessibili ma solamente da invocazioni all'interno della stessa macchina virtuale.

Implementazioni remote

Per realizzare l'implementazione remota deriva da un'interfaccia remota si può procedere in due modi:

- Il primo prevede che la classe che contiene l'implementazione dell'oggetto derivi esplicitamente da *java.rmi.server*.
- La seconda modalità permette che la classe derivi il comportamento da qualche altra classe (non remota) e che si debba quindi occupare esplicitamente di esportare l'oggetto e di implementare la semantica di alcune operazioni di *Object* per oggetti remoti che sono ridefiniti in *java.rmi.server*.

Il meccanismo di invocazione remota

Riferimenti remoti

Nel modello distribuito, gli oggetti client interagiscono con un oggetto stub che espone localmente le stesse interfacce remote definite dall'oggetto remoto. Lo stub rappresenta l'interfaccia remota dell'oggetto remoto in locale, sulla macchina dove l'oggetto client è in esecuzione. Il client può accedere al tipo di un oggetto remoto controllando quale interfaccia remota implementa, attraverso instanceof. Il sistema fornisce anche un meccanismo per il caricamento dinamico dello stub per rendere dinamicamente disponibile lo stub a run-time agli oggetti client.

Localizzazione e invocazione di oggetti remoti

Per poter invocare un metodo remoto, il client deve aver a disposizione un suo riferimento remoto. Questo può essere reperito in due modi: ottenuto come risultato di altre invocazioni (locali o remote) di metodi; oppure attraverso un servizio di directory. Il secondo metodo fornisce un semplice meccanismo di name server nella classe java.rmi.Naming, che permette di gestire riferimenti a oggetti remoti accessibili specificando un ID (Stringa). Tale classe fornisce metodi per ricercare (lookup()), registrare (bind(), unbind(), rebind()), ed elencare (list()) gli identificati registrati, accedendo al servizio secondo uno standard URL. L'invocazione di un metodo remoto è uguale a quella locale. I metodi remoti implementano l'eccezione *RemoteException*.

Passaggio di parametri

Un metodo remoto può dichiarare solo parametri o valori restituiti che siano serializzabili, vale a dire che implementano l'interfaccia Serializable. Un oggetto locale passato come parametro o restituito come valore da un'invocazione remota viene passato per copia, vale a dire che il contenuto dell'oggetto viene copiato prima di essere serializzato dal meccanismo di serializzazione di Java, quindi la semantica per il passaggio di parametri locali è diversa da quella di Java che passa il riferimento di un oggetto.

Il meccanismo implementato da Java RMI assicura la cosiddetta integrità referenziale: quando vengono passati più riferimenti allo stesso oggetto nella stessa invocazione, allora viene garantito che anche sulla macchina remota alla quale sono stati passati i riferimenti punteranno allo stesso oggetto. Quando si passa un oggetto remoto come parametro o lo si ottiene come valore restituito viene passato il suo stub e non l'implementazione.

La differenza tra il modello a oggetti locale e quello remoto

Una prima modifica applicata alla classe Object da java.rmi.server.RemoteObject è la ridefinizione dei metodi equals(), hashCode(), toString() nel seguente modo:

- Il metodo X.hashCode() viene ridefinito in modo tale che restituisca lo stesso codice per due stub diversi di oggetti remoti che si riferiscono allo stesso oggetto remoto, quindi questi ultimi possono essere usati come chiavi nelle tabelle hash.
- Il metodo X.equals() restituisce un booleano che è vero se il riferimento passato è uguale ad X, il confronto viene fatto sugli stub e quindi l'uguaglianza è effettuata sui riferimenti e non sul contesto. Se l'oggetto passato Y non è un oggetto remoto allora si verifica l'uguaglianza tra uno stub locale e un oggetto remoto (di cui si usa lo stub).
- Il metodo X.toString() deve restituire informazioni circa su quale macchina si trova l'oggetto remoto, il nome della classe e un codice hash.

Queste modifiche al comportamento dei metodi non vengono effettuate qualora si usi il metodo dell'implementazione locale tramite la classe di implementazione locale (exportObject()) e quindi è responsabilità del programmatore prendersi cura del corretto comportamento dell'oggetto remoto. I client degli oggetti remoti interagiscono soltanto con l'interfaccia remota.

Il *passaggio dei parametri* è possibile in modo trasparente infatti i riferimenti remoti possono essere passati (o restituiti) a/di un metodo come parametri (o restituiti come valori).

Per la *gestione dei tipi* si può effettuare un semplice casting di un oggetto remoto ad una qualsiasi interfaccia remota che implementa.

La differenza nelle *invocazioni di metodo* sta nel fatto che le invocazioni remoti forzano il programmatore a dover gestire esplicitamente i fallimenti di invocazioni di metodi remoti. Questo è dovuto al fatto che l'invocazione di un metodo remoto è molto più complessa e soggetta a malfunzionamenti che sono difficili da trattare.

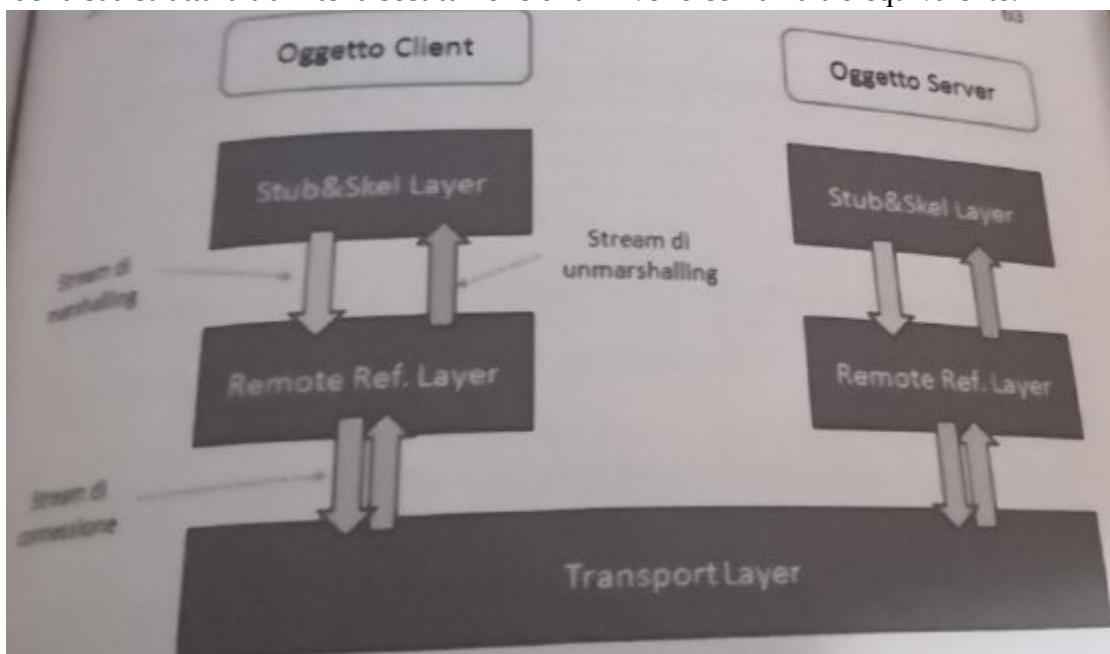
L'architettura di java RMI

I tre layer dell'architettura

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

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

L'applicazione dell'utente si trova in cima a questi livelli e interagisce con il livello stub/skeleton, tutta l'architettura comunica con un protocollo bene definito, in questo modo si può evolvere mantenendo la sua struttura tramite la sostituzione di un livello con un altro equivalente.



Riepilogo passi Java RMI

- Un client che invoca un metodo su un oggetto server remoto, fa uso di uno stub per portare a termine la sua richiesta in quanto il riferimento remoto si riferisce ad un riferimento dello stub del client presente in locale.
- Lo stub implementa l'interfaccia remota dell'oggetto remoto e inoltra la richiesta all'oggetto server attraverso il remote reference layer.
- Il remote reference layer del client si occupa di gestire la semantica delle invocazione remota lato client.
- Il livello di trasporto si occupa di stabilire la connessione con la macchina remota e della successiva gestione della connessione, curando il dispatching delle invocazioni verso gli oggetti remoti. Si inoltra la richiesta al livello di reference del server.
- Il livello di reference del server si occupa di inoltrare la richiesta allo skeleton e di curare la semantica dell'invocazione lato server.

Stub/Skeleton layer

Si occupa di essere l'interfacci tra l'applicazione ed il resto del sistema. L'interfaccia verso il basso consiste nel fornire uno stream di marshall di oggetti Java che vengono passati al remote reference layer. Gli oggetti che vengono passati al remote reference layer vengono passati per copia, lo stub è incaricato di:

- iniziare la connessione con la macchina virtuale remota, chiamando il remote reference layer;
- effettuare il marshalling verso uno stream di marshall, fornito dal layer di remote reference;
- attendere il risultato dell'invocazione;
- effettuare l'unmarshalling dei valori restituiti;
- restituire il valore verso l'oggetto client che ha richiesto l'invocazione.

Lo skeleton è incaricato di effettuare il dispatching verso l'oggetto remoto, quando uno skeleton riceve un'invocazione in entrata si occupa di:

- effettuare l'unmarshalling dal remote reference layer (lato server) dei parametri per l'invocazione;
- invocare il metodo sull'implementazione che si trova nella sua JVM;
- effettuare il marshalling del valore restituito verso chi ha invocato il metodo.

Stub e Skeleton vengono creati dall'RMI rmic, un tool fornito con il JDK che a partire da una classe compilata che rappresenta l'implementazione di un oggetto remoto genera i file stub e skeleton.

Remote Reference Layer

Questo layer si occupa di interfacciare il livello di trasporto con quello di stub/skeleton fornendo e supportando la semantica dell'operazione d'invocazione di un metodo. Abbiamo diverse modalità di invocazioni:

- Invocazioni *unicast*: un singolo client fa un'invocazione ad un singolo server.
- Invocazioni *multicast*: vale a dire un singolo client fa un'invocazione ad una "batteria" di server replicanti, in maniera da poter garantire la ridondanza, se uno di questi è in esecuzione allora risponderà.
- Invocazioni *di oggetti attivabili*: le invocazioni potrebbero essere effettuate ad un oggetto remoto che è persistente, vale a dire viene attivato se arrivano delle invocazioni.
- Invocazioni *di riconnessione*: le invocazioni potrebbero tentare connessioni alternative se l'oggetto remoto originariamente contattato non risponde all'invocazione.

Il protocollo d'invocazione utilizza le due componenti client e server del remote reference layer. Il lato client ha informazione circa il server dell'invocazione e comunica attraverso il livello di trasporto verso il lato server dello stesso layer. Questo layer espone verso l'alto un riferimento ad un oggetto che implementa l'interfaccia `java.rmi.server.RemoteServer` che espone un metodo `invoke()` per effettuare l'inoltro dell'invocazione, che viene chiamato dallo stub. Interagisce verso il basso con il transport layer, utilizzando l'astrazione di una connessione orientata ai flussi, questa connessione potrebbe utilizzare protocolli connectionless senza alterare la modalità con cui il remote layer comunica i dati.

Transport Layer

Il livello di trasporto ha il compito di:

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

Il protocollo utilizzato da RMI è chiamato Java Remote Method Protocol, un protocollo proprietario della Sun.

Caricamento dinamico delle classi

Java Remote Method Invocation permette il passaggio di oggetti come parametri, valori restituiti o eccezioni, attraverso la serializzazione. Questo permette di mantenere il sistema dei tipi che offre i sistemi. Però questo si scontra con il fatto che Java carica le classi al tempo di esecuzione, e può succedere che un oggetto remoto può trovarsi nella situazione in cui non conosce esattamente come è strutturata la classe di cui l'oggetto passato è istanza. Java RMI risolve questo problema attraverso il caricamento dinamico delle classi. Infatti quando si fa il marshalling degli oggetti per la trasmissione essi vengono anche annotati con il codebase cioè un URL (Uniform Resource Locator) di un server WWW da dove è possibile trovare la definizione della classe (.class). Quando viene effettuato l'unmarshalling dell'oggetto, il classloader cerca di risolvere il nome della classe nel suo contesto, poi in caso non sia possibile, viene acceduta la definizione della classe per poter ricreare l'oggetto all'altro capo della comunicazione. Questo meccanismo di caricamento dinamico permette anche il caricamento dinamico degli stub, con lo stesso meccanismo di annotazione che viene effettuato nel marshalling. Questo avviene quando si passa un riferimento remoto che consiste nell'inviare lo stub.

Sicurezza in Java

Il linguaggio Java è considerato sicuro per vari motivi, prima cosa è fortemente tipizzato, offre la gestione automatica della memoria attraverso il garbage collection, rende impossibile effettuare accessi illegali alla memoria e infine l'accesso alla memoria reale non viene determinato a tempo di compilazione ma a tempo di esecuzione, quindi è impossibile conoscere in anticipo l'indirizzo di memoria dove verranno memorizzati gli oggetti.

- Il classloader si occupa di caricare la classe a tempo di esecuzione anche da locazioni remote. Il suo compito è quello di caricare la classe in un name space separato rispetto a quello locale. Infatti nel momento in cui si fa riferimento ad una classe questa viene prima cercata tra le classi del sistema locale e solo successivamente nel namespace dove classe viene riferita, in questo modo si evita la sovrascrittura di una classe di sistema.
- Il bytecode verifica che una classe non sia volontariamente ostile. Controlla quindi che essa sia conforme alle specifiche del linguaggio, che non siano presenti stack underflow/overflow, e che non ci siano violazioni alle regole specificate dai modificatori di accesso.
- Il security Manager si occupa di definire i confini della sandbox, viene interpellato dalla macchina virtuale per ciascun operazione potenzialmente pericolosa e fornisce le autorizzazioni sulla base della politica che ha stabilito l'utente lanciando la macchina virtuale. Il security manager è caricato a tempo di esecuzione e non può essere esteso, sovrascritto o rimpiazzato.

Il meccanismo di marshalling usato da Java RMI

Fare il marshalling in Java significa effettuare una serializzazione modificando la semantica dei riferimenti remoti e aggiungendo informazioni all'oggetto. Il meccanismo di marshalling di Java RMI si basa sulla specializzazione del meccanismo tradizionale di serializzazione effettuata da ObjectOutputStream. Infatti questa classe offre a possibilità di poter modificare il comportamento tramite quale gli oggetti vengono scritti come stream di byte. Questo meccanismo avviene modificando tre metodi della classe ObjectOutputStream:

- il metodo replaceObject(): che può definire un metodo alternativo per serializzare un oggetto sullo stream.
- Il metodo enableReplaceObject(): che restituisce un booleano e stabilisce se la istanza deve oppure no specializzare il meccanismo di serializzazione, usando il metodo replaceObject().
- Il metodo annotateClass(): che permette di inserire informazioni addizionali sulla classe, viene usato per specificare il codebase e permettere quindi il caricamento dinamico.

L'operazione più complessa è quella di `replaceObject()`, questo metodo deve infatti essere richiamato ogni qualvolta viene invocato `writeObject()`, prima che questo provveda alla serializzazione dando la possibilità di sostituire l'oggetto da serializzare. Il metodo `replaceOnject()` fa le seguenti operazioni:

- Se l'oggetto da serializzare è un'istanza di `java.rmi.Remote` (riferimento ad un oggetto remoto), e l'oggetto risulta esportato a runtime di RMI, allora usando `java.rmi.server.RemoteObject.toStub()` viene restituito il suo stub, se non è esportato restituisce l'oggetto remoto stesso.
- Se l'oggetto da serializzare non è istanza di `java.rmi.Remote` allora viene restituito `writeObject()`.

Tramite questo meccanismo viene risolta un'apparente incongruità nelle invocazioni: quando viene passato un riferimento remoto come parametro, questo viene copiato sullo stream, ma non abbiamo il vincolo di dichiarare un riferimento remoto come serializzazione. Infatti esso viene sostituito dal suo stub che è un'istanza di `java.rmi.server.RemoteStub`, che è una classe che implementa l'interfaccia `Serializable`. Questo meccanismo spiega anche la modalità con la quale viene assicurata l'integrità referenziale: poiché i parametri di una stessa invocazione remota utilizzano lo stesso stream di output, parametri che si riferiscono allo stesso oggetto nella stessa invocazione verranno serializzati nel flusso come facenti riferimento allo stesso oggetto, e verranno deserializzati nella stessa maniera all'altro capo dello stream.

Processo di creazione di un programma Java RMI

Il processo per la creazione di un programma si suddivide in due sottoprocessi: uno che procede verso lo sviluppo e esecuzione del server, l'altro che si occupa dello sviluppo del client. Alcuni passi dipendono dai precedenti altri no, infatti il client è solo in minima parte influenzato dall'sviluppo del server.

Definizione dell'interfaccia remota

Il primo passo è la definizione di quale servizi sono offerti dal nostro server, specificandoli all'interno di una interfaccia. L'implementazione sarà al carico del server totalmente nascosta al client. Per specificare un'interfaccia basta semplicemente specificarle come in Java. Le caratteristiche che deve avere in più sono:

- derivare dalle interfacce `mark-up Remote`;
- tutti i suoi metodi devono lanciare l'eccezione `java.rmi.RemoteException`.

Implementazione del server

Un oggetto remoto in Java deve essere istanza di una classe che:

- implementi una o più interfacce remote
- derivi da `java.rmi.UnicastRemoteObject`.

Il fatto che il server deriva da `UnicastRemoteObject` implica che il costruttore del nostro server debba esplicitamente essere scritto in quanto è necessario che il costruttore della sottoclasse lanci esplicitamente l'eccezione `RemoteException` che viene lanciata dal costruttore della superclasse.

Compilazione del server

In Eclipse i valori di default forniti dal wizard di creazione di un progetto Java se accettati mettono i sorgenti della classe in una directory `src` e i file bytecode in una directory `bin` che non è visibile, per vederla è necessario usare la perspective Resource.

Compilazione con lo stub compiler rmic

Avendo specificato interfaccia remota e server si possono generare automaticamente i file che sono necessari (stub e skeleton). Per questo RMI fornisce uno stub compiler chiamato `rmic`, che eseguito un file `.class` del nostro server genera lo stub e lo skeleton.

Servizio di naming: rmiregistry

A questo punto è necessario che il nostro oggetto remoto possa essere accessibile dai client che ne vogliono invocare i servizi. Viene quindi proposto un servizio di naming chiamato rmiregistry, quest'ultimo deve essere lanciato prima di eseguire il server perché tra le prime operazioni il server andrà a registrarsi presso questo registro con un etichetta. Il servizio di naming deve essere lanciato nella directory in cui si trova il .class dello stub.

Esecuzione del server

Prima di eseguire il server dobbiamo seguire una politica di sicurezza per la macchina virtuale, per poter permettere l'esecuzione di operazioni potenzialmente pericolose. Noi adotteremo una politica estremamente liberale:

```
grant {  
    permission java.security.AllPermission;  
};
```

Dobbiamo quindi fornire su linea di comando alla macchina virtuale l'indicazione del file di policy da eseguire tramite il comando `java -Djava.security.policy=policyall`.

Registrazione del server sul servizio di naming

Appena lanciato il server esso deve registrarsi sul servizio di naming. Vengono usati metodi del package `java.rmi.Naming` che prevedono semplici modalità di registrazione, richiesta e deregistrazione. Non è possibile avere un servizio di naming “esterno”.

Implementazione del client

L'implementazione del client non richiede molte modifiche rispetto ad una eventuale applicazione scritta in locale. Si deve essenzialmente risolvere il problema della localizzazione dell'oggetto remoto, effettuata tramite i servizi di `java.rmi.Naming` che permettono di ottenere un riferimento remoto ad un oggetto server. Si deve gestire l'eccezione `RemoteException` che viene lasciata da tutti i metodi remoti. L'implementazione dell'invocazione remota risulta essere a carico dello stub.

Compilazione ed esecuzione del client

Quando si compila il client si deve prestare al fatto che deve essere presente lo stub del servizio nella directory dove il client viene compilato.

L'adapter

Una classe adapter implementa un'interfaccia conosciuta dai suoi Client e fornisce accesso ad una classe (chiamata adaptee) non conosciuta da essi. Questo rappresenta uno dei pattern strutturali più importanti. Attraverso di esso una classe Adapter fornisce un servizio per una classe Adptee, che non deve essere modificata per fornire i servizi direttamente al Client, implementando l'interfaccia appropriata.

Teoria

Java EE 7



"The most blasphemed IDE by all
programmers and not"

CARMINE D'ANGELO
EMANUELE VITALE

Introduzione a Java EE 7

Java EE Application Model

Il modello Java EE si basa su Java e sull'utilizzo della sua macchina virtuale. Java EE è progettato per supportare le applicazioni che implementano servizi aziendali per clienti, dipendenti, fornitori, partner e altri che fanno richieste o contributi all'impresa. Per controllare e gestire meglio queste applicazioni, le funzioni aziendali per supportare questi vari utenti sono condotte nel livello intermedio. Il livello intermedio rappresenta un ambiente strettamente controllato dal dipartimento di information technology di un'azienda. Quest'ultimo viene generalmente eseguito su hardware server dedicato e ha accesso ai servizi completi dell'azienda. Il modello di applicazione Java EE definisce un'architettura per l'implementazione di servizi come applicazioni multitier che forniscono scalabilità, accessibilità e gestibilità necessarie per le applicazioni di livello enterprise. Questo modello separa il lavoro necessario per implementare un servizio multitier nelle seguenti parti:

- La logica business e di presentazione che deve essere implementata dallo sviluppatore
- I servizi di sistema standard forniti dalla piattaforma Java EE

Lo sviluppatore può contare sulla piattaforma per fornire soluzioni per i problemi a livello di sistema di sviluppo di un servizio multitier.

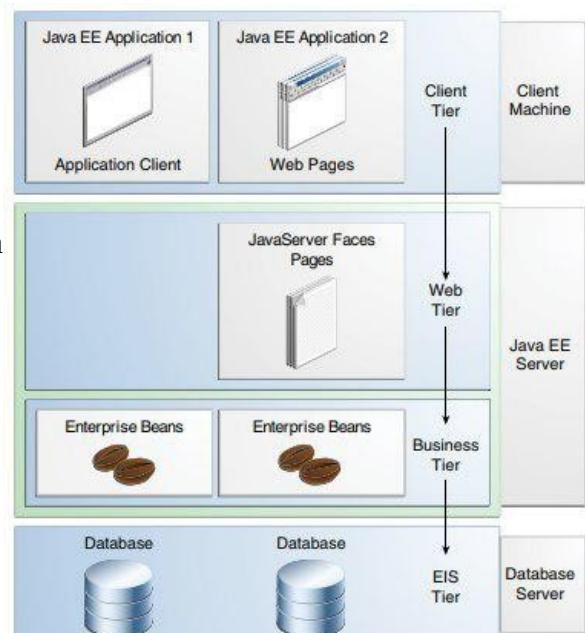
Distributed Multitiered Applications

La piattaforma Java EE utilizza un modello di applicazione multitier distribuito per applicazioni aziendali. La logica dell'applicazione è suddivisa in componenti che sono suddivisi in base alla funzione che svolgono, i componenti dell'applicazione che compongono un'applicazione Java EE sono installati su vari computer.

La figura seguente mostra due applicazioni EE Java multitier suddivise nei livelli descritti nell'elenco seguente. Le parti dell'applicazione Java EE mostrate nella figura sono presentate in Java EE Components.

- I componenti del livello client vengono eseguiti sul computer client.
- I componenti del livello Web vengono eseguiti sul server Java EE.
- I componenti del livello business vengono eseguiti sul server Java EE.
- Enterprise Information System (EIS): il software più avanzato viene eseguito sul server EIS.

Sebbene un'applicazione Java EE possa essere costituita da tutti i livelli mostrati nella figura, in Java EE le applicazioni multi-tier sono generalmente considerate applicazioni a tre livelli perché sono distribuite su tre posizioni: macchine client, la macchina server Java EE e il database o le macchine legacy sul back-end. Le applicazioni a tre livelli che vengono eseguite in questo modo estendono il modello client-server standard a due livelli inserendo un server applicazioni con multithread tra l'applicazione client e lo storage back-end.



Security

Sebbene altri modelli di applicazioni aziendali richiedano misure di sicurezza specifiche della piattaforma in ogni applicazione, l'ambiente di sicurezza Java EE consente di definire i vincoli di sicurezza al momento dell'implementazione. La piattaforma Java EE fornisce regole di controllo degli accessi dichiarative standard definite dallo sviluppatore e interpretate quando l'applicazione viene distribuita sul server. Java EE fornisce inoltre meccanismi di accesso standard in modo che gli sviluppatori di applicazioni non debbano implementare questi meccanismi nelle loro applicazioni. La stessa applicazione funziona in una varietà di ambienti di sicurezza senza modificare il codice sorgente.

Java EE Components

Le applicazioni Java EE sono costituite da componenti. Un componente Java EE è un'unità software funzionale autonoma che viene assemblata in un'applicazione Java EE con le relative classi e file e che comunica con altri componenti.

La specifica Java EE definisce i seguenti componenti Java EE:

- I client e le applet dell'applicazione sono componenti eseguiti sul client.
- I componenti della tecnologia Java Servlet, JavaServer Faces e JSP (JavaServer Pages) sono componenti Web eseguiti sul server.

I componenti EJB (bean enterprise) sono componenti aziendali che vengono eseguiti sul server.

I componenti Java EE sono scritti nel linguaggio di programmazione Java e sono compilati nello stesso linguaggio. Le differenze tra i componenti Java EE e le classi Java "standard" sono che i componenti che sono assemblati in un'applicazione Java EE, sono verificati per essere ben formati e conformi alle specifiche Java EE e vengono distribuiti alla produzione, dove sono eseguiti e gestiti dal server Java EE.

Java EE Clients

Un client Java EE è solitamente un client Web o un'applicazione.

Web Clients

Un client Web è costituito da due parti:

- Pagine Web dinamiche contenenti vari tipi di linguaggio di markup (HTML, XML e così via), generati da componenti Web in esecuzione nel livello Web.
- Un browser Web che esegue il rendering delle pagine ricevute dal server.

Un client web è talvolta chiamato thin client. Solitamente i thin client non eseguono query sui database, eseguono regole aziendali complesse o si connettono a applicazioni legacy. Quando si utilizza un thin client, tali operazioni vengono scaricate sui bean enterprise in esecuzione sul server Java EE, dove possono sfruttare la sicurezza, la velocità, i servizi e l'affidabilità delle tecnologie Java EE sul lato server.

Application Clients

Un client applicativo viene eseguito su un computer client e fornisce agli utenti un modo per gestire attività che richiedono un'interfaccia utente più ricca di quella che può essere fornita da un linguaggio di markup. Un client applicativo ha in genere un'interfaccia utente grafica (GUI) creata dall'API Swing o dall'API AWT (Abstract Window Toolkit), ma un'interfaccia della riga di comando è certamente possibile.

I client dell'applicazione accedono direttamente ai bean enterprise in esecuzione nel livello business. Tuttavia, se i requisiti dell'applicazione lo giustificano, un client applicativo può aprire una connessione HTTP per stabilire la comunicazione con un servlet in esecuzione nel livello Web. I client applicativi scritti in linguaggi diversi da Java possono interagire con i server Java EE, consentendo alla piattaforma Java EE di interagire con sistemi legacy, client e linguaggi diversi da Java.

Applets

Una pagina Web ricevuta dal livello Web può includere un'applet. Scritto nel linguaggio di programmazione Java, un'applet è una piccola applicazione client eseguita nella Java virtual machine installata nel browser web. Tuttavia, i sistemi client avranno probabilmente bisogno del plug-in Java e probabilmente di un file di politica di sicurezza affinché l'applet possa essere eseguito correttamente nel browser web.

I componenti Web sono l'API preferita per la creazione di un programma client Web poiché non sono necessari plug-in o file di criteri di sicurezza sui sistemi client. Inoltre, consentono una progettazione di applicazioni più pulita e più modulare perché forniscono un modo per separare la programmazione delle applicazioni dalla progettazione di pagine web.

The JavaBeans Component Architecture

I livelli server e client potrebbero anche includere componenti basati su JavaBeans architettura di componenti (componenti JavaBeans) per gestire il flusso di dati tra i seguenti:

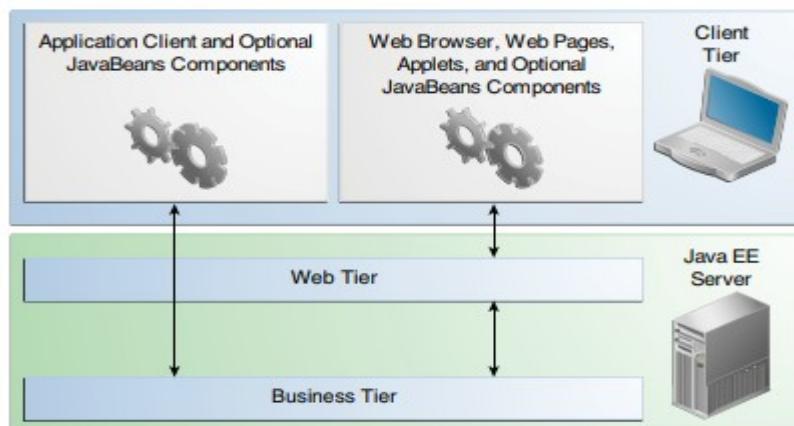
- Un client applicativo o applet e componenti in esecuzione sul server Java EE
- Componenti del server e un database

I componenti JavaBeans non sono considerati componenti Java EE dalla specifica di Java EE. I componenti JavaBeans hanno proprietà e hanno *get* e *set* come metodi per l'accesso alle proprietà. I componenti JavaBeans utilizzati in questo modo sono in genere semplici nella progettazione e nell'implementazione, ma devono essere conformi alle convenzioni di denominazione e progettazione illustrate nell'architettura dei JavaBeans.

Java EE Server Communications

La figura mostra i vari elementi che possono costituire il livello client. Il cliente comunica con il livello business in esecuzione sul server Java EE direttamente o, come nel caso di un client in esecuzione in un browser, passando attraverso pagine Web o servlet in esecuzione nel livello web.

Figure 1–2 Server Communication



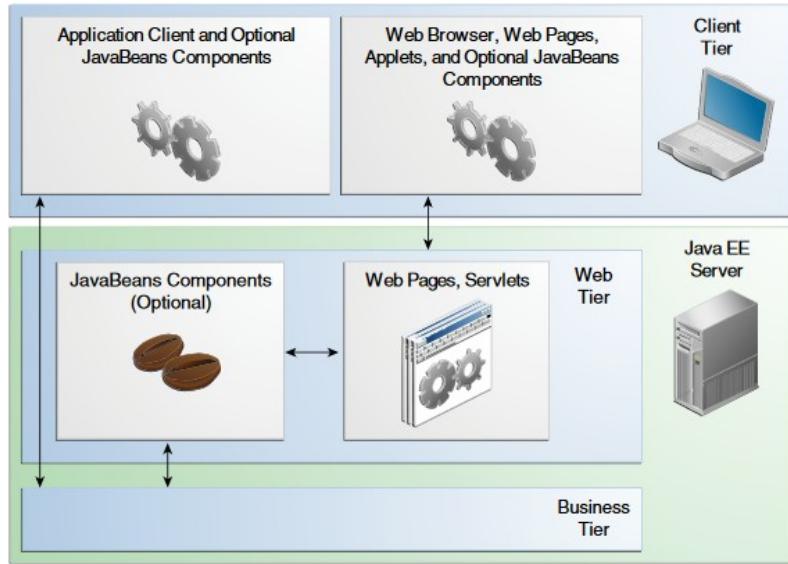
Web components

I componenti Web Java EE sono servlet o pagine Web create utilizzando la tecnologia JavaServer Faces e/o tecnologia JSP (pagine JSP). Le servlet sono classi di linguaggio di programmazione Java che elaborano dinamicamente le richieste e costruiscono le risposte. Le pagine JSP sono documenti basati su testo che vengono eseguiti come servlet ma consentono un approccio più naturale alla creazione di contenuto statico.

La tecnologia JavaServer Faces si basa su servlet e tecnologia JSP e fornisce una struttura di componenti dell'interfaccia utente per le applicazioni Web.

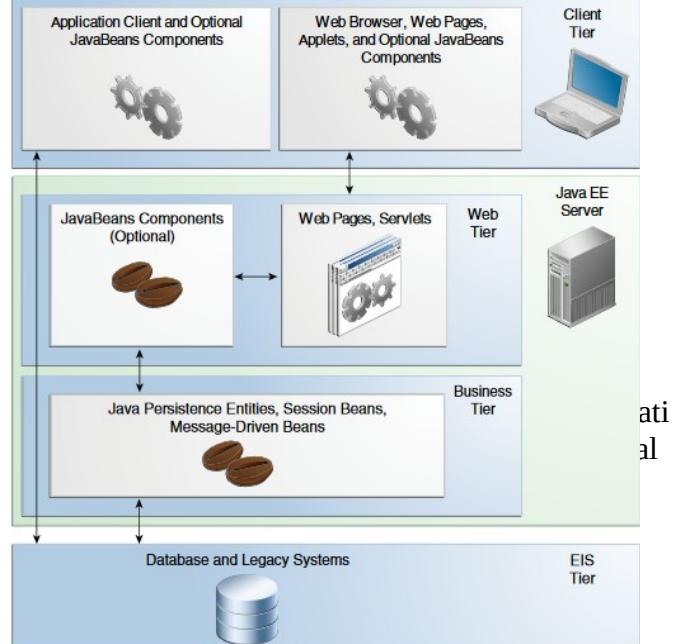
Le pagine e le applet HTML statiche sono raggruppate con componenti Web durante l'assemblaggio dell'applicazione, ma non sono considerate componenti Web dalla specifica Java EE. Le classi di utilità lato server possono anche essere raggruppate con componenti Web e, come le pagine HTML, non sono considerate componenti web.

Come illustrato nella figura, il livello Web, come il livello client, potrebbe includere un componente JavaBeans per gestire l'input dell'utente e inviare tale input ai bean enterprise in esecuzione nel livello aziendale per l'elaborazione.



Business component

Il Business code, è la logica che risolve o soddisfa le esigenze di un particolare dominio aziendale, come banche, vendita al dettaglio o finanza, è gestito da bean enterprise in esecuzione nel livello aziendale o nel livello Web. La figura mostra in che modo un bean enterprise riceve i dati dai programmi client, li elabora (se necessario) e li invia al livello del sistema informativo aziendale per l'archiviazione. Un bean enterprise recupera dalla memoria, li elabora (se necessario) e li programma client.



Enterprise information System tier

Il livello del sistema di informazioni aziendali gestisce il software EIS e include sistemi di infrastruttura aziendale, come la pianificazione delle risorse aziendali (ERP), l'elaborazione delle transazioni mainframe, i sistemi di database e altri sistemi di informazioni legacy. Ad esempio, i componenti dell'applicazione Java EE potrebbero richiedere l'accesso ai sistemi informativi aziendali per la connettività del database.

Java EE Containers

Normalmente, le applicazioni multitier thin client sono difficili da scrivere perché coinvolgono molte linee di codice complesso per gestire la gestione di transazioni e stati, multithreading, pool di risorse e altri dettagli complessi di basso livello. L'architettura Java EE basata su componenti indipendenti dalla piattaforma rende le applicazioni facili da scrivere perché la logica aziendale è organizzata in componenti riutilizzabili. Inoltre, il server Java EE fornisce servizi sottostanti sotto forma di contenitore per ogni tipo di componente.

Container services

I contenitori sono l'interfaccia tra un componente e la funzionalità specifica della piattaforma di basso livello che supporta il componente. Prima che possa essere eseguito, un componente Web, bean enterprise o client applicazione deve essere assemblato in un modulo Java EE e distribuito nel relativo contenitore. Il processo di assemblaggio implica la specifica delle impostazioni del contenitore per ciascun componente nell'applicazione Java EE e per l'applicazione Java EE stessa. Le impostazioni del contenitore personalizzano il supporto sottostante fornito dal server Java EE, inclusi servizi quali sicurezza, gestione delle transazioni, ricerche API JNDI (Naming Java e Directory Interface) e connettività remota. Ecco alcuni dei punti salienti:

- Il modello di sicurezza Java EE consente di configurare un componente Web o un bean enterprise in modo che le risorse di sistema siano accessibili solo agli utenti autorizzati.
- Il modello di transazione Java EE consente di specificare le relazioni tra i metodi che costituiscono una singola transazione in modo che tutti i metodi in una transazione vengano considerati come una singola unità.
- I servizi di ricerca JNDI forniscono un'interfaccia unificata a più servizi di denominazione e directory nell'azienda in modo che i componenti dell'applicazione possano accedere a questi servizi.
- Il modello di connettività remota Java EE gestisce le comunicazioni di basso livello tra client e bean enterprise. Dopo aver creato un bean enterprise, un client richiama i metodi su di esso come se si trovasse nella stessa macchina virtuale.

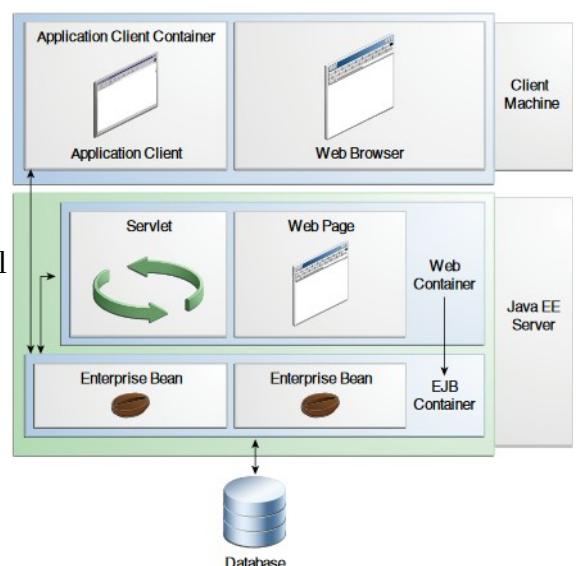
Poiché l'architettura Java EE fornisce servizi configurabili, i componenti all'interno della stessa applicazione possono comportarsi diversamente in base alla posizione in cui vengono distribuiti. Ad esempio, un bean enterprise può avere impostazioni di sicurezza che gli consentono un certo livello di accesso ai dati del database in un ambiente di produzione e un altro livello di accesso al database in un altro ambiente di produzione. Il contenitore gestisce anche servizi non configurabili, quali i cicli di vita dei bean enterprise e dei servlet, il pooling delle risorse di connessione al database, la persistenza dei dati e l'accesso alle API della piattaforma Java EE (vedere API Java EE 7).

Containers types

Il processo di distribuzione installa i componenti dell'applicazione Java EE nei contenitori Java EE, come illustrato nella figura.

Il server e i contenitori sono i seguenti:

- **Server Java EE:** la parte runtime di un prodotto Java EE. Un server Java EE fornisce contenitori EJB e web.
- **Contenitore EJB:** gestisce l'esecuzione dei bean enterprise per le applicazioni Java EE. I bean enterprise e il loro contenitore vengono eseguiti sul server Java EE.
- **Contenitore Web:** gestisce l'esecuzione di pagine Web, servlet e alcuni componenti EJB per applicazioni Java EE. I componenti Web e il loro contenitore vengono eseguiti sul server Java EE.



- **Contenitore del client dell'applicazione:** gestisce l'esecuzione dei componenti del client dell'applicazione. I client dell'applicazione e il loro contenitore vengono eseguiti sul client.
- **Contenitore di applet:** gestisce l'esecuzione di applet. È costituito da un browser Web e un plug-in Java in esecuzione sul client insieme.

Java EE Application Assembly and Deployment

Un'applicazione Java EE è pacchettizzata in una o più unità standard per la distribuzione su qualsiasi sistema compatibile con la piattaforma Java EE. Ogni unità contiene:

- Un componente o componenti funzionali, come un bean enterprise, una pagina Web, un servlet o un'applet.
- Un descrittore di distribuzione opzionale che ne descrive il contenuto.

Una volta che è stata prodotta un'unità Java EE, è pronta per essere distribuita. La distribuzione implica in genere l'utilizzo di uno strumento di distribuzione della piattaforma per specificare informazioni specifiche sulla posizione, come un elenco di utenti locali che possono accedervi e il nome del database locale. Una volta distribuito su una piattaforma locale, l'applicazione è pronta per essere eseguita.

Java EE 7 at a Glance

Le imprese oggi vivono in un mondo competitivo a livello globale. Hanno bisogno di applicazioni per soddisfare le loro esigenze di business, fanno affari 24 ore su 24, 7 giorni su 7 e in diversi paesi. Il tutto riducendo i loro costi, riducendo i tempi di risposta dei loro servizi.

Le applicazioni aziendali devono affrontare cambiamenti e complessità ed essere robuste. Proprio per questo è stata creata Java Enterprise Edition (Java EE).

La prima versione di Java EE (originariamente nota come J2EE) si concentrava sulle preoccupazioni che le aziende dovevano affrontare nel 1999: componenti distribuiti. Da allora, le applicazioni software hanno dovuto adattarsi a nuove soluzioni tecniche come SOAP o servizi web RESTful. La piattaforma Java EE si è evoluta per rispondere a queste esigenze tecniche fornendo vari modi di lavorare attraverso specifiche standard. Nel corso degli anni, Java EE è cambiato ed è diventato più ricco, più semplice, più facile da usare, più portatile e più integrato.

Architecture

Java EE è un insieme di specifiche implementate da diversi contenitori. I contenitori sono ambienti di runtime Java EE che forniscono determinati servizi ai componenti che ospitano come la gestione del ciclo di vita, l'iniezione della dipendenza, la concorrenza e così via. Questi componenti utilizzano contratti ben definiti per comunicare con l'infrastruttura Java EE e con gli altri componenti. Devono essere confezionati in un modo standard (seguendo una struttura di directory definita che può essere compressa in file di archivio) prima di essere distribuiti. Java EE è un superset della piattaforma Java SE, il che significa che le API Java SE possono essere utilizzate da qualsiasi componente Java EE.

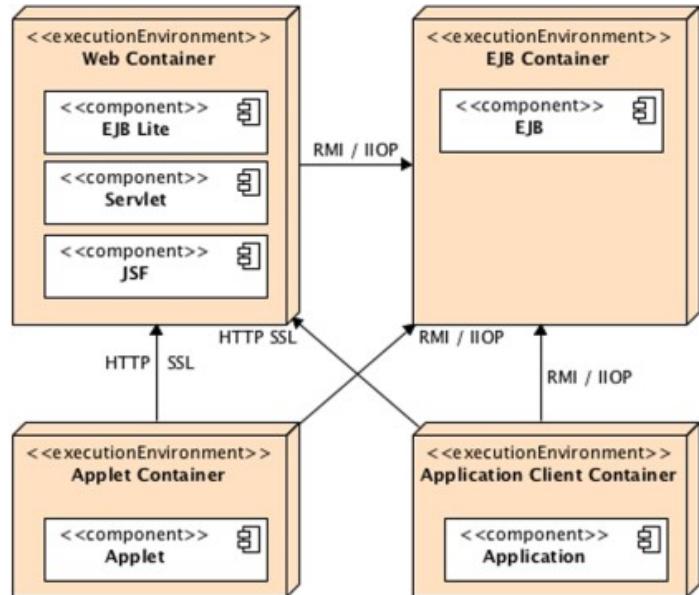


Figure 1-1. Standard Java EE containers

Components

L'ambiente di runtime Java EE definisce quattro tipi di componenti che un'implementazione deve supportare:

- Le **applet** sono applicazioni GUI (interfaccia utente grafica) eseguite in un browser web. Usano la ricca API Swing per fornire potenti interfacce utente.
- Le **applicazioni** sono programmi che vengono eseguiti su un client. In genere sono GUI o programmi di elaborazione batch che hanno accesso a tutte le funzionalità del livello intermedio Java EE.
- Le **applicazioni Web** vengono eseguite in un contenitore Web e rispondono alle richieste HTTP dei client Web. Le servlet supportano anche gli endpoint del servizio Web SOAP e RESTful. Le applicazioni Web possono anche contenere EJB Lite.
- Le **applicazioni aziendali** (costituite da Enterprise Java Beans, Java Message Service, Java Transaction API, chiamate asincrone, servizio timer, RMI / IIOP) vengono eseguite in un contenitore EJB. Gli EJB sono componenti gestiti dal contenitore per l'elaborazione della logica di business transazionale. Possono essere accessibili localmente e in remoto tramite RMI (o HTTP per servizi Web SOAP e RESTful).

Containers

L'infrastruttura Java EE è suddivisa in domini logici denominati contenitori. Ogni contenitore ha un ruolo specifico, supporta un set di API e offre servizi ai componenti (sicurezza, accesso al database, gestione delle transazioni, directory di denominazione, iniezione di risorse). I contenitori nascondono la complessità tecnica e migliorano la portabilità. A seconda del tipo di applicazione che si desidera creare, è necessario comprendere le capacità e i vincoli di ciascun contenitore per poterne utilizzare uno o più. Java EE ha quattro contenitori diversi:

- **I contenitori di applet** sono forniti dalla maggior parte dei browser Web per l'esecuzione di componenti dell'applet. Il contenitore di applet utilizza un modello di sicurezza sandbox in cui il codice eseguito nella "sandbox" non è autorizzato a "giocare fuori dalla sandbox". Ciò significa che il contenitore impedisce a qualsiasi codice scaricato nel computer locale di accedere a risorse di sistema locali, come processi o File.
- **Il contenitore del client dell'applicazione** (ACC) include un set di classi Java, librerie e altri file necessari per portare l'iniezione, la gestione della sicurezza e il servizio di denominazione alle applicazioni Java SE (swing, elaborazione batch o solo una classe con un metodo main ()). L'ACC comunica con il contenitore EJB utilizzando RMI-IIOP e il contenitore Web con HTTP (ad es. Per i servizi Web).
- **Il contenitore web** fornisce i servizi sottostanti per la gestione e l'esecuzione di web-components (servlet, EJB Lite, JSP, filtri, listener, pagine JSF e servizi Web). È responsabile della creazione di istanze, inizializzazione e invocazione di servlet e supporto dei protocolli HTTP e HTTPS. È il contenitore utilizzato per alimentare pagine Web nei browser dei clienti.
- **Il contenitore EJB** è responsabile della gestione dell'esecuzione dei bean enterprise (bean di sessione e bean basati sui messaggi) contenenti il livello della logica business dell'applicazione Java EE. Crea nuove istanze di bean, gestisce il loro ciclo di vita e fornisce servizi quali transazioni, sicurezza, concorrenza, distribuzione, servizio di denominazione o la possibilità di essere richiamati in modo asincrono.

Services

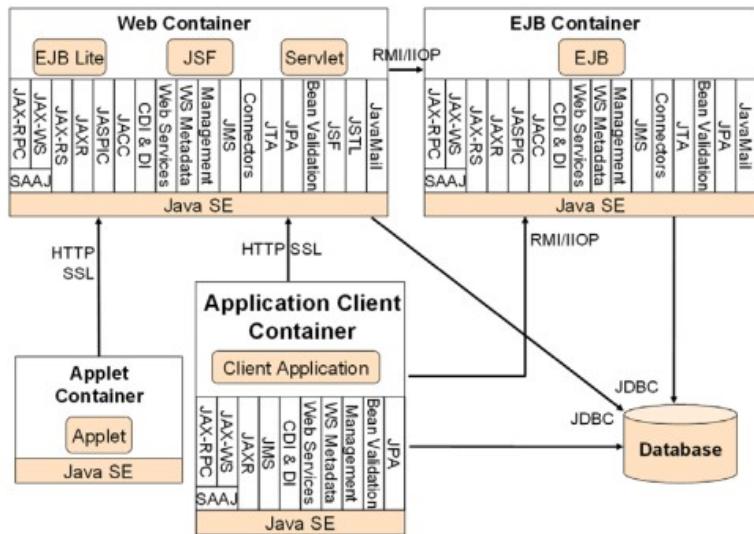
I contenitori forniscono servizi sottostanti ai componenti distribuiti. Come sviluppatore, i contenitori consentono di concentrarsi sull'implementazione della logica business piuttosto che sulla risoluzione dei problemi tecnici affrontati nelle applicazioni aziendali. Java EE offre i seguenti servizi:

- **Java Transaction API:** questo servizio offre un'API di demarcazione delle transazioni utilizzata dal contenitore e dall'applicazione. Fornisce inoltre un'interfaccia tra il gestore delle transazioni e un gestore delle risorse a livello di Service Provider Interface (SPI).
- **Java Persistence API:** API standard per il mapping relazionale agli oggetti (ORM). Con il suo Java Persistence Query Language (JPQL), puoi interrogare oggetti memorizzati nel database sottostante.
- **Convalida:** l'utilizzo del Bean Validation fornisce la dichiarazione dei vincoli a livello di classe e metodo e le funzioni di convalida.
- **Java Message Service:** Ciò consente ai componenti di comunicare in modo asincrono tramite i messaggi. Supporta la messaggistica affidabile point-to-point (P2P) e il modello publish-subscribe (pub-sub).
- **Java Naming and Directory Interface:** questa API, inclusa in Java SE, viene utilizzata per accedere ai sistemi di denominazione e directory. L'applicazione la usa per associare (legare) nomi a oggetti e quindi per trovare questi oggetti (ricerca) in una directory. È possibile cercare origini dati, fabbriche JMS, EJB e altre risorse. Viene utilizzato in modo più trasparente attraverso l'iniezione.
- **JavaMail:** molte applicazioni richiedono la possibilità di inviare e-mail, che possono essere implementate tramite l'uso dell'API JavaMail.
- **JavaBeans Activation Framework:** l'API JAF, inclusa in Java SE, fornisce un framework per la gestione dei dati in diversi tipi MIME. È usato da JavaMail.
- **XML processing:** La maggior parte dei componenti Java EE può essere implementata con descrittori di distribuzione XML opzionali e le applicazioni spesso devono manipolare i documenti XML. L'API Java per l'elaborazione XML (JAXP) fornisce supporto per l'analisi dei documenti con le API SAX e DOM, nonché per XSLT. L'API Streaming per XML (StAX) fornisce un'API pull-parsing per XML.
- **JSON processing:** Novità in Java EE 7 l'API Java per JSON Processing (JSON-P) consente alle applicazioni di analizzare, generare, trasformare e interrogare JSON.
- **Java EE Connector Architecture:** I connettori consentono di accedere a EIS da un componente Java EE. Questi potrebbero essere database, mainframe o programmi di pianificazione delle risorse aziendali (ERP).
- **Security services:** Il servizio JAAS (Java Authentication and Authorization Service) consente ai servizi di autenticare e applicare i controlli di accesso agli utenti. Il Contratto per i fornitori di servizi di autorizzazione di Java (JACC) definisce un contratto tra un server di applicazioni Java EE e un fornitore di servizi di autorizzazione, consentendo ai fornitori di servizi di autorizzazione personalizzati di essere collegati a qualsiasi prodotto Java EE. L'interfaccia del fornitore di servizi di autenticazione Java per contenitori (JASPIC) definisce un'interfaccia standard mediante la quale i moduli di autenticazione possono essere integrati con i contenitori in modo che questi moduli possano stabilire le identità di autenticazione utilizzate dai contenitori.
- **Web Services:** Java EE fornisce supporto per i servizi Web SOAP e RESTful. L'API Java per i servizi Web XML (JAX-WS), che sostituisce l'API Java per RPC basata su XML (JAX-RPC), fornisce il supporto per i servizi Web che utilizzano il protocollo SOAP / HTTP. L'API Java per RESTful Web Services (JAX-RS) fornisce supporto per i servizi Web utilizzando lo stile REST.
- **Iniezione delle dipendenze:** poiché Java EE 5, alcune risorse (origini dati, fabbriche JMS, unità di persistenza, EJB ...) possono essere iniettate nei componenti gestiti. Java EE 7 va oltre utilizzando CDI e le specifiche DI (Dependency Injection per Java).

- Gestione:** Java EE definisce le API per la gestione di contenitori e server mediante un bean enterprise di gestione speciale. L'API JMX (Java Management Extensions) viene utilizzata anche per fornire supporto gestionale.
 - Distribuzione:** Java EE Deployment Specification definisce un contratto tra gli strumenti di distribuzione e i prodotti Java EE per standardizzare la distribuzione delle applicazioni.

The diagram illustrates the Java EE architecture components and their interactions:

 - Application Client Container:** The central component at the bottom.
 - Web Container:** To the left of the Application Client Container, containing EJB Lite, JSF, and Servlet.
 - EJB Container:** To the right of the Application Client Container, containing EJB.
 - Java SE:** The foundation layer at the bottom, containing JavaMail, JSTL, JSF, Bean Validation, JPA, JTA, Connectors, JMS, Management, WS Metadata, Web Services, CD & DI, CACC, JAX-RS, JAX-WS, JAX-RPC, SAAJ, and SAAJ.
 - Interactions:**
 - JSF and Servlet interact with JavaMail via RMI/IIOP.
 - JavaMail interacts with Java SE via RMI/IIOP.
 - Java SE interacts with Application Client Container via HTTP/SSL.
 - Application Client Container interacts with Java SE via HTTP/SSL.
 - Application Client Container interacts with EJB Container via RMI/IIOP.
 - EJB Container interacts with Java SE via RMI/IIOP.



Network Protocols

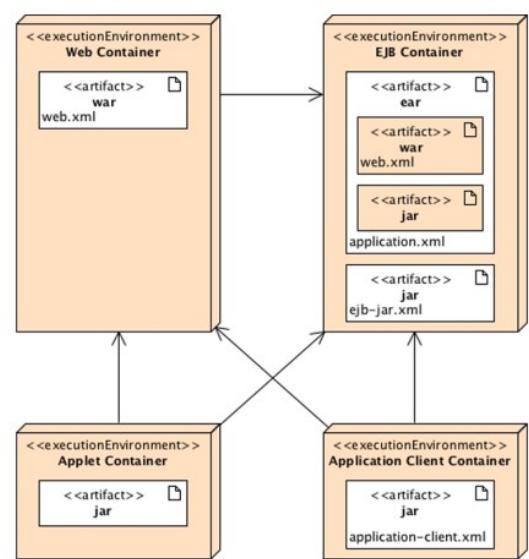
I componenti distribuiti in contenitori possono essere richiamati tramite protocolli diversi. Ad esempio, un servlet distribuito in un contenitore Web può essere chiamato con HTTP così come un servizio Web con un endpoint EJB distribuito in un contenitore EJB. Ecco l'elenco dei protocolli supportati da Java EE:

- **HTTP:** HTTP è il protocollo Web ed è onnipresente nelle moderne applicazioni. L'API lato client è definita dal pacchetto `java.net` in Java SE. L'API HTTP lato server è definita dalle interfacce `servlet`, `JSP` e `JSF`, nonché dai servizi Web `SOAP` e `RESTful`. **HTTPS** è una combinazione di protocollo HTTP e Secure Sockets Layer (SSL).
 - **RMI-IIOP:** Remote Method Invocation (RMI) consente di richiamare oggetti remoti indipendentemente dal protocollo sottostante. Il protocollo RMI nativo di Java SE è Java Remote Method Protocol (JRMP). RMI-IIOP è un'estensione di RMI utilizzata per l'integrazione con CORBA. Il linguaggio di descrizione dell'interfaccia Java (IDL) consente ai componenti dell'applicazione Java EE di richiamare oggetti CORBA esterni utilizzando il protocollo IIOP. Gli oggetti CORBA possono essere scritti in molte lingue (Ada, C, C++, Cobol, ecc.) Così come in Java.

Packaging

Per essere distribuiti in un contenitore, i componenti devono prima essere impacchettati in un archivio standard formattato. Java SE definisce i file Java Archive (jar), che vengono utilizzati per aggregare molti file (classi Java, descrittori di distribuzione, risorse o librerie esterne) in un file compresso (basato sul formato ZIP). Java EE definisce diversi tipi di moduli che hanno il proprio formato di pacchettizzazione basato su questo comune formato jar.

- **Un modulo client** dell'applicazione contiene classi Java e altri file di risorse contenuti in un file jar. Questo file jar può essere eseguito in un ambiente Java SE o in un contenitore client dell'applicazione. Come ogni altro formato di archivio, il file jar contiene una directory META-INF opzionale per le meta-informationi che



descrivono l'archivio. Il file META-INF / MANIFEST.MF viene utilizzato per definire i dati relativi all'estensione e ai pacchetti. Se distribuito in un ACC, il descrittore di deployment può essere posizionato opzionalmente su META-INF / application-client.xml.

- **Un modulo EJB** contiene uno o più session e/o bean message-driven (MDB) racchiusi in un file jar (spesso chiamato file jar EJB). Contiene un descrittore di distribuzione META-INF / ejb-jar.xml facoltativo e può essere distribuito solo in un contenitore EJB.
- **Un modulo di applicazione Web** contiene servlet, JSP, pagine JSF e servizi Web, nonché qualsiasi altro file relativo al Web. A partire da Java EE 6, un modulo di applicazione Web può contenere anche bean EJB Lite. Tutti questi artefatti sono racchiusi in un file jar con estensione **.war** (comunemente noto come war file o archivio Web). Il descrittore di distribuzione Web facoltativo è definito nel file WEB-INF / web.xml. Se il war contiene bean EJB Lite, è possibile impostare un descrittore di distribuzione opzionale su WEB-INF / ejb-jar.xml. I file Java.class vengono posizionati nella directory WEB-INF / classes e nei file jar dipendenti nella directory WEB-INF / lib.
- **Un modulo business** può contenere zero o più moduli di applicazioni Web, zero o più moduli EJB e altre librerie comuni o esterne. Tutto questo è impacchettato in un archivio business (un file jar con estensione **.ear**) in modo che la distribuzione di questi vari moduli avvenga simultaneamente e in modo coerente. Il descrittore di distribuzione del modulo Enterprise facoltativo è definito nel file META-INF / application.xml. La directory lib speciale viene utilizzata per condividere le librerie comuni tra i moduli.

Annotations and Deployment Descriptors

Nel paradigma della programmazione, ci sono due approcci: programmazione imperativa e programmazione dichiarativa.

- **La programmazione imperativa** specifica l'algoritmo per raggiungere un obiettivo (cosa deve essere fatto).
- **La programmazione dichiarativa** specifica come raggiungere questo obiettivo (come deve essere fatto).

In Java EE, la programmazione dichiarativa viene eseguita utilizzando i metadati, ovvero annotazioni o/e descrittori di distribuzione. I componenti vengono eseguiti in un contenitore e questo contenitore fornisce al componente un insieme di servizi. I metadati vengono utilizzati per dichiarare e personalizzare questi servizi e associano informazioni aggiuntive insieme a classi, interfacce, costruttori, metodi, campi o parametri Java. Da Java EE 5, le annotazioni si sono moltiplicate nella piattaforma business. Decorano il tuo codice (classi, interfacce, campi, metodi Java) con le informazioni sui metadati. Il codice seguente mostra un POJO (Plain Old Java Object) che dichiara un certo comportamento usando annotazioni sulla classe e su un attributo (più su EJB, contesto di persistenza e annotazioni nei prossimi capitoli).

Listing 1-1. An EJB with Annotations

```
@Stateless  
@Remote(ItemRemote.class)  
@Local(ItemLocal.class)  
@LocalBean  
public class ItemEJB implements ItemLocal, ItemRemote {  
  
    @PersistenceContext(unitName = "chapter01PU")  
    private EntityManager em;  
    public Book findBookById(Long id) {  
        return em.find(Book.class, id);  
    }  
}
```

L'altro modo di dichiarare i metadati è usando i descrittori di implementazione. Un descrittore di implementazione (DD) fa riferimento a un file di configurazione XML distribuito con il

componente nel contenitore. Il codice seguente mostra un descrittore di implementazione EJB. Come la maggior parte dei descrittori di distribuzione di Java EE 7, definisce lo spazio dei nomi <http://xmlns.jcp.org/xml/ns/javaee> e contiene un attributo di versione con la versione della specifica.

Listing 1-2. An EJB Deployment Descriptor

```
<ejb-jar xmlns="http://xmlns.jcp.org/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                               http://xmlns.jcp.org/xml/ns/javaee/ejb-jar_3_2.xsd"
          version="3.2">

<enterprise-beans>
    <session>
        <ejb-name>ItemEJB</ejb-name>
        <remote>org.agoncal.book.javaee7.ItemRemote</remote>
        <local>org.agoncal.book.javaee7.ItemLocal</local>
        <local-bean/>
        <ejb-class>org.agoncal.book.javaee7.ItemEJB</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
    </session>
</enterprise-beans>
</ejb-jar>
```

I descrittori di implementazione devono essere impacchettati con i componenti nella directory speciale META-INF o WEB-INF da prendere in considerazione. La Tabella 1-1 mostra l'elenco dei descrittori di distribuzione Java EE e le relative specifiche (ulteriori informazioni su ciò nei prossimi capitoli).

File	Specification	Paths
application.xml	Java EE	META-INF
application-client.xml	Java EE	META-INF
beans.xml	CDI	META-INF or WEB-INF
ra.xml	JCA	META-INF
ejb-jar.xml	EJB	META-INF or WEB-INF
faces-config.xml	JSF	WEB-INF
persistence.xml	JPA	META-INF
validation.xml	Bean Validation	META-INF or WEB-INF
web.xml	Servlet	WEB-INF
web-fragment.xml	Servlet	WEB-INF
webservices.xml	SOAP Web Services	META-INF or WEB-INF

Poiché Java EE 5 la maggior parte dei descrittori di implementazione è facoltativa, è possibile utilizzare invece le annotazioni. Il più grande vantaggio delle annotazioni è che riducono significativamente la quantità di codice che uno sviluppatore deve scrivere e, utilizzandole è possibile evitare i descrittori di implementazione. D'altra parte, i descrittori di implementazione sono file XML esterni che possono essere modificati senza richiedere modifiche al codice sorgente e alla ricompilazione. Se si utilizzano entrambi, i metadati vengono sovrascritti dal descrittore di implementazione (ad esempio, XML ha la precedenza sulle annotazioni) quando viene distribuita l'applicazione o il componente.

Standards

Java EE è basato su standard. Ciò significa che Java EE passa attraverso il processo di standardizzazione del JCP ed è descritto in una specifica. In effetti, Java EE è chiamato una specifica ombrello perché raggruppa una serie di altre specifiche (o richieste di specifica Java). Java EE fornisce standard aperti implementati da diversi framework commerciali (WebLogic, Websphere, MQSeries, ecc.) O open source (GlassFish, JBoss, Hibernate, Open JPA, Jersey, ecc.) Per transazioni andling, sicurezza, componenti stateful, oggetti persistenza e così via. Oggi, più che mai nella storia di Java EE, la tua applicazione può essere implementata in qualsiasi server applicativo compatibile con pochissime modifiche.

JCP

JCP è un'organizzazione aperta, creata nel 1998 da Sun Microsystems, che è coinvolta nella definizione delle future versioni e funzionalità della piattaforma Java. Quando viene identificata la necessità di standardizzare un componente o un'API esistente, l'iniziatore (a.k.a. lead) specifica un JSR e forma un gruppo di esperti. Questo gruppo, composto da rappresentanti di aziende, organizzazioni, università o privati, è responsabile dello sviluppo del JSR e deve fornire:

- Una o più specifiche che spiegano i dettagli e definiscono i fondamenti della JSR (Java Specification Request),
- Un'implementazione di riferimento (RI), che è un'attuazione effettiva della specifica,
- Compatibility Test Kit (a.k.a Technology Compatibility Kit, o TCK), che è una serie di test che ogni implementazione deve superare prima di richiedere la conformità alle specifiche.

Una volta approvato dal comitato esecutivo (CE), la specifica viene rilasciata alla comunità per l'implementazione.

Portable

Dalla sua creazione, l'obiettivo di Java EE era di consentire lo sviluppo di un'applicazione e la sua distribuzione su qualsiasi server applicativo senza modificare il codice o i file di configurazione. Non è mai stato così facile come sembrava. Le specifiche non coprono tutti i dettagli e le implementazioni finiscono per fornire soluzioni non portabili. Questo problema, è stato risolto in Java EE specificando una sintassi per i nomi JNDI. Oggi, la piattaforma ha introdotto più proprietà di configurazione portatili che mai, aumentando così la portabilità. Nonostante abbiano deprecato alcune API (sfoltimento), le applicazioni Java EE mantengono la loro compatibilità con le versioni precedenti, consentendo di migrare l'applicazione a versioni più recenti di un server applicazioni senza troppi problemi.

Programming Model

La maggior parte delle specifiche Java EE 7 utilizza lo stesso modello di programmazione. Di solito è un POJO con alcuni metadati (annotazioni o XML) distribuiti in un contenitore. La maggior parte delle volte il POJO non implementa nemmeno un'interfaccia o estende una superclasse. Grazie ai metadati, il contenitore sa quali servizi applicare a questo componente distribuito. In Java EE 7, servlet, bean di backing JSF, EJB, entità, servizi Web SOAP e REST sono classi annotate con descrittori di distribuzione XML facoltativi. Il codice seguente mostra un bean di supporto JSF che risulta essere una classe Java con una singola annotazione CDI.

Listing 1-3. A JSF Backing Bean

@Named

```
public class BookController {  
    @Inject  
    private BookEJB bookEJB;  
  
    private Book book = new Book();  
    private List<Book> bookList = new ArrayList<Book>();
```

```

public String doCreateBook() {
    book = bookEJB.createBook(book);
    bookList = bookEJB.findBooks();
    return "listBooks.xhtml";
}
// Getters, setters
}

```

Anche gli EJB seguono lo stesso modello. Come mostrato di seguito se è necessario accedere a un EJB localmente, è sufficiente una semplice classe annotata senza interfaccia. Gli EJB possono anche essere distribuiti direttamente in un file war senza essere precedentemente impacchettati in un file jar. Ciò rende EJB il componente transazionale più semplice che può essere utilizzato da semplici applicazioni Web a quelle aziendali complesse.

Listing 1-4. A Stateless EJB

```

@Stateless
public class BookEJB {
    @Inject
    private EntityManager em;

    public Book findBookById(Long id) {
        return em.find(Book.class, id);
    }

    public Book createBook(Book book) {
        em.persist(book);
        return book;
    }
}

```

I servizi web restful si sono fatti strada nelle applicazioni moderne. Java EE 7 risponde alle esigenze delle imprese migliorando le specifiche JAX-RS. Come mostrato di seguito un servizio web RESTful è una classe Java annotata che risponde alle azioni HTTP .

Listing 1-5. A RESTful Web Service

```

@Path("books")
public class BookResource {
    @Inject
    private EntityManager em;

    @GET
    @Produces({"application/xml", "application/json"})
    public List<Book> getAllBooks() {
        Query query = em.createNamedQuery("findAllBooks");
        List<Book> books = query.getResultList();
        return books;
    }
}

```

Java EE Specifications Overview

Java EE è una specifica ombrello che raggruppa e integra le versioni precedenti. Oggi, un application server deve implementare 31 specifiche per essere compatibile con Java EE 7 e uno sviluppatore deve conoscere migliaia di API per sfruttare al massimo il contenitore. Anche se ci sono molte specifiche e API da sapere, Java EE 7 si concentra sul portare semplicità alla piattaforma introducendo un semplice modello di programmazione basato su POJO, un profilo Web e un infarinatura di alcune tecnologie obsolete.

Java EE 7 Specifications

Table 1-2. Java Enterprise Edition Specification

Specification	Version	JSR	URL
Java EE	7.0	342	http://jcp.org/en/jsr/detail?id=342
Web Profile	7.0	342	http://jcp.org/en/jsr/detail?id=342
Managed Beans	1.0	316	http://jcp.org/en/jsr/detail?id=316

La specifica Java EE 7 è definita dal JSR 342 e contiene 31 altre specifiche. Un server di applicazioni che mira ad essere conforme a Java EE 7 deve implementare tutte queste specifiche. Nel dominio del servizio Web (Tabella 1-3) non è stato apportato alcun miglioramento al servizio Web SOAP poiché nessuna specifica è stata aggiornata. I servizi web REST sono stati ampiamente utilizzati recentemente nelle principali applicazioni web. JAX-RS 2.0 ha seguito un importante aggiornamento con l'introduzione dell'API client ad esempio. La nuova specifica JSON-P (JSON Processing) è l'equivalente di JAXP (Java API per XML Processing) ma per JSON invece di XML.

Table 1-3. Web Services Specifications

Specification	Version	JSR	URL
JAX-WS	2.2a	224	http://jcp.org/en/jsr/detail?id=224
JAXB	2.2	222	http://jcp.org/en/jsr/detail?id=222
Web Services	1.3	109	http://jcp.org/en/jsr/detail?id=109
Web Services Metadata	2.1	181	http://jcp.org/en/jsr/detail?id=181
JAX-RS	2.0	339	http://jcp.org/en/jsr/detail?id=339
JSON-P	1.0	353	http://jcp.org/en/jsr/detail?id=353

Nelle specifiche Web (Tabella 1-4) non è stata apportata alcuna modifica a JSP o JSTL in quanto queste specifiche non sono state aggiornate. Expression Language è stato estratto da JSP e ora si evolve nel suo JSR (341). Servlet e JSF (Capitoli 10 e 11) sono stati entrambi aggiornati e il nuovissimo WebSocket 1.0 è stato introdotto in Java EE 7.

Table 1-4. Web Specifications

Specification	Version	JSR	URL
JSF	2.2	344	http://jcp.org/en/jsr/detail?id=344
JSP	2.3	245	http://jcp.org/en/jsr/detail?id=245
Debugging Support for Other Languages	1.0	45	http://jcp.org/en/jsr/detail?id=45
JSTL (JavaServer Pages Standard Tag Library)	1.2	52	http://jcp.org/en/jsr/detail?id=52
Servlet	3.1	340	http://jcp.org/en/jsr/detail?id=340
WebSocket	1.0	356	http://jcp.org/en/jsr/detail?id=356
Expression Language	3.0	341	http://jcp.org/en/jsr/detail?id=341

Nel dominio business (Tabella 1-5) ci sono due aggiornamenti principali: JMS 2.0 (capitolo 13) e JTA 1.2 (capitolo 9), che non sono stati aggiornati da più di un decennio. D'altro canto, gli EJB (Capitoli 7 e 8), le specifiche JPA (Capitoli 4, 5 e 6) e Interceptor (Capitolo 2) si sono evoluti con aggiornamenti minori.

Java EE 7 include diverse altre specifiche (Tabella 1-6) come la nuovissima elaborazione Batch (JSR 352) e Concurrency Utilities per Java EE (JSR 236). Alcuni aggiornamenti importanti sono Bean Validation 1.1 (capitolo 3), CDI 1.1 (capitolo 2) e JMS 2.0 (capitolo 13)

Table 1-5. Enterprise Specifications

Specification	Version	JSR	URL
EJB	3.2	345	http://jcp.org/en/jsr/detail?id=345
Interceptors	1.2	318	http://jcp.org/en/jsr/detail?id=318
JavaMail	1.5	919	http://jcp.org/en/jsr/detail?id=919
JCA	1.7	322	http://jcp.org/en/jsr/detail?id=322
JMS	2.0	343	http://jcp.org/en/jsr/detail?id=343
JPA	2.1	338	http://jcp.org/en/jsr/detail?id=338
JTA	1.2	907	http://jcp.org/en/jsr/detail?id=907

Table I-6. Management, Security, and Other Specifications

Specification	Version	JSR	URL
JACC	1.4	115	http://jcp.org/en/jsr/detail?id=115
Bean Validation	1.1	349	http://jcp.org/en/jsr/detail?id=349
Contexts and Dependency Injection	1.1	346	http://jcp.org/en/jsr/detail?id=346
Dependency Injection for Java	1.0	330	http://jcp.org/en/jsr/detail?id=330
Batch	1.0	352	http://jcp.org/en/jsr/detail?id=352
Concurrency Utilities for Java EE	1.0	236	http://jcp.org/en/jsr/detail?id=236
Java EE Management	1.1	77	http://jcp.org/en/jsr/detail?id=77
Java Authentication Service Provider Interface for Containers	1.0	196	http://jcp.org/en/jsr/detail?id=196

Java EE 7 non è composto solo da queste 31 specifiche in quanto dipende fortemente da Java SE 7.

La Tabella 1-7 mostra alcune specifiche che appartengono a Java SE ma influenzano Java EE.

Table I-7. Related Enterprise Technologies in Java SE 7

Specification	Version	JSR	URL
Common Annotations	1.2	250	http://jcp.org/en/jsr/detail?id=250
JDBC	4.1	221	http://jcp.org/en/jsr/detail?id=221
JNDI	1.2		
JAXP	1.3	206	http://jcp.org/en/jsr/detail?id=206
StAX	1.0	173	http://jcp.org/en/jsr/detail?id=173
JAAS	1.0		
JMX	1.2	3	http://jcp.org/en/jsr/detail?id=3
JAXB	2.2	222	http://jcp.org/en/jsr/detail?id=222
JAF	1.1	925	http://jcp.org/en/jsr/detail?id=925
SAAJ	1.3		http://java.net/projects/saaj

Java EE 7 definisce un profilo singolo chiamato Web Profile. Il suo scopo è consentire agli eveloper di creare applicazioni web con l'insieme appropriato di tecnologie. Web Profile 7.0 è specificato in un JSR separato ed è, per ora, l'unico profilo della piattaforma Java EE 7. Altri potrebbero essere creati in futuro (si potrebbe pensare ad un profilo minimo o ad un profilo del portale). La Tabella 1-8 elenca le specifiche incluse nel profilo Web.

Table I-8. Web Profile 7.0 Specifications

Specification	Version	JSR	URL
JSF	2.2	344	http://jcp.org/en/jsr/detail?id=344
JSP	2.3	245	http://jcp.org/en/jsr/detail?id=245
JSTL	1.2	52	http://jcp.org/en/jsr/detail?id=52
Servlet	3.1	340	http://jcp.org/en/jsr/detail?id=340
WebSocket	1.0	356	http://jcp.org/en/jsr/detail?id=356
Expression Language	3.0	341	http://jcp.org/en/jsr/detail?id=341
EJB Lite	3.2	345	http://jcp.org/en/jsr/detail?id=345
JPA	2.1	338	http://jcp.org/en/jsr/detail?id=338
JTA	1.2	907	http://jcp.org/en/jsr/detail?id=907
Bean Validation	1.1	349	http://jcp.org/en/jsr/detail?id=349

(continued)

Table I-8. (continued)

Specification	Version	JSR	URL
Managed Beans	1.0	316	http://jcp.org/en/jsr/detail?id=316
Interceptors	1.2	318	http://jcp.org/en/jsr/detail?id=318
Contexts and Dependency Injection	1.1	346	http://jcp.org/en/jsr/detail?id=346
Dependency Injection for Java	1.0	330	http://jcp.org/en/jsr/detail?id=330
Debugging Support for Other Languages	1.0	45	http://jcp.org/en/jsr/detail?id=45
JAX-RS	2.0	339	http://jcp.org/en/jsr/detail?id=339
JSON-P	1.0	353	http://jcp.org/en/jsr/detail?id=353

Context and Dependency Injection (CDI)

Java SE ha dei java beans , mentre Java EE ha degli Enterprise Java Bean. I POJO sono solo delle classi java che vengono eseguite in una jvm, mentre i java beans sono solo POJO che seguono alcuni pattern e sono seguiti anche loto nella jvm, tutte le altre componenti di Java EE seguono alcuni modelli e sono eseguiti all'interno di un container che fornisce dei servizi. Questo ci lascia con bean gestiti e bean normali. I bean gestiti sono degli oggetti gestiti da un container che supportano solo un piccolo insieme di servizi base: risorse di Injection, gestione del ciclo di vita e interception. Forniscono una fondamenta comune per i diversi tipi di componenti esistenti nella piattaforma Java EE. Gli oggetti CDI sono costruiti su questo modello base chiamato Bean managed. I Bean hanno un ciclo di vita migliorato per oggetti stateful; essendo legati a contesti ben definiti portano ad un approccio tipicamente favorevole all'injection, interception e decoration; sono specializzati con annotazioni di tipo qualificatore. Infatti, con pochissime eccezioni, potenzialmente ogni classe Java che ha un costruttore predefinito ed è in esecuzione all'interno di un contenitore è un bean. Quindi JavaBeans ed Enterprise JavaBeans possono naturalmente trarre vantaggio da questi servizi CDI.

Dipendecy Injection

Dipendecy Injection (DI) è un modello di progettazione che disaccoppia componenti dipendenti, il contenitore inietta quegli oggetti dipendenti per te.

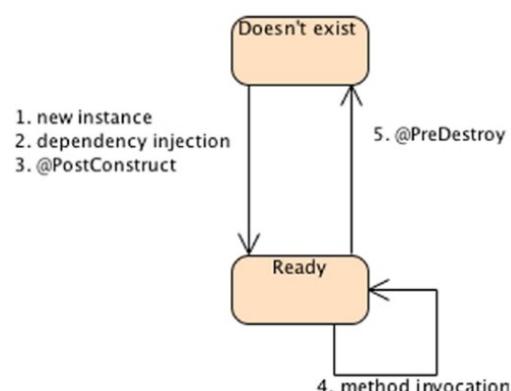
Java EE 5 ha introdotto l'injection di risorse per gli sviluppatori. Permetteva agli sviluppatori di iniettare risorse del contenitore come EJB, gestori di entità, origini dati, fabbriche JMS e destinazioni in un insieme di componenti definiti (Servlet, bean di supporto JSF ed EJB). A tale scopo, Java EE 5 ha introdotto un nuovo set di annotazioni (@Resource, @PersistenceContext, @PersistenceUnit, @EJB e @WebServiceRef).

Questo primo passo in Java EE 5 non è stato sufficiente, quindi Java EE 6 ha creato due nuove specifiche per portare la vera DI alla piattaforma: Dependency Injection (JSR 330) e Contexts and Dependency Injection (JSR 299). Oggi, in Java EE 7, DI va anche oltre legando insieme le specifiche.

Ciclo di vita

Il ciclo di vita di un POJO è piuttosto semplice: come sviluppatore Java puoi creare un'istanza di una classe usando la nuova parola chiave e attendere che Garbage Collector si occupi di liberare memoria. Ma se si desidera eseguire un bean CDI all'interno di un contenitore è necessario iniettare il bean e il contenitore fa il resto, ovvero il contenitore è il responsabile della gestione del ciclo di vita del bean: crea l'istanza; si sbarazza di esso.

La Figura mostra il ciclo di vita di un bean gestito (e quindi un bean CDI). Quando si inietta un bean, il contenitore (EJB, Web o contenitore CDI) è il responsabile della creazione dell'istanza (utilizzando la nuova parola chiave). Quindi risolve le dipendenze e richiama qualsiasi metodo annotato con @PostConstruct prima della prima chiamata al metodo business sul bean. Quindi, la notifica di callback @PreDestroy segnala che l'istanza è in fase di rimozione dal contenitore.



Scopes and Context

I bean CDI possono essere stateful e contextual, il che significa che vivono in un ambito ben definito (CDI viene fornito con ambiti predefiniti: richiesta, sessione, applicazione e ambiti conversazione). Durante questa vita, i riferimenti iniettati ai bean sono anche consapevoli del contesto, cioè l'intera catena delle dipendenze del bean è contextual. Il contenitore gestisce automaticamente tutti i bean all'interno dell'ambito e, al termine della sessione, li distrugge automaticamente.

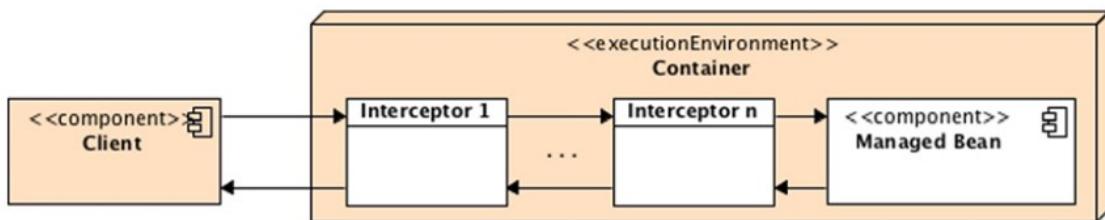
A differenza dei componenti stateless (ad esempio, bean di sessione stateless) o singlettons (ad es. Servlet o singleton), diversi client di un bean stateful vedono lo stesso bean in stati diversi. Quando il bean ha lo stato (sessione, applicazione e conversazione) i client (ad esempio, altri bean) in esecuzione nello stesso contesto vedranno la stessa istanza del bean. Ma i client in un contesto diverso potrebbero vedere un'istanza diversa (a seconda della relazione tra i contesti). In tutti i casi, il cliente non controlla il ciclo di vita dell'istanza creando e distruggendolo esplicitamente; il contenitore lo fa secondo lo scope.

Interceptor

Gli interceptor sono usati per intromettersi sulle invocazioni dei metodi di business. Questi potrebbero essere problemi tecnici (registrare la voce e l'uscita da ciascun metodo, registrare la durata di un richiamo del metodo, memorizzare statistiche sull'uso del metodo, ecc.) O preoccupazioni aziendali (eseguire controlli aggiuntivi se un cliente acquista più di \$ 10.000 di articoli, inviare un ordine di ricarica quando il livello di inventario è troppo basso, ecc.). Queste preoccupazioni possono essere applicate automaticamente tramite AOP all'intera applicazione o a un sottoinsieme di esso.

Managed Beans supporta la funzionalità di tipo AOP fornendo la capacità di intercettare l'invocazione del metodo tramite degli intercettori. Gli intercettori vengono automaticamente attivati dal contenitore quando viene richiamato un metodo Managed Bean.

Come mostrato nella Figura , gli intercettori possono essere concatenati e chiamati prima e / o dopo l'esecuzione di un metodo.



Loose Coupling and Strong Typing

Gli interceptor sono un modo molto efficace per disaccoppiare i problemi tecnici dalla logica aziendale. Con l'injection, un bean non è a conoscenza dell'implementazione concreta di qualsiasi bean con cui interagisce. I bean possono utilizzare le notifiche degli eventi per disaccoppiare i produttori di eventi dai consumatori di eventi o dai decoratori per disaccoppiare le preoccupazioni aziendali.

E tutte queste strutture sono consegnate in modo tipicamente sicuro. CDI non fa mai affidamento su identificatori basati su String per determinare il modo in cui gli oggetti si adattano. Invece, CDI utilizza annotazioni fortemente tipizzate (ad es. Qualificatori, stereotipi e binding di interceptor) per collegare insieme i bean. L'utilizzo dei descrittori XML è ridotto al minimo solo per l'uso d' informazioni veramente specifiche dell'implementazione.

Deployment descriptor

Quasi tutte le specifiche Java EE hanno un deployment descriptor XML opzionale. Solitamente descrive come deve essere configurato un componente, un modulo o un'applicazione (come un'applicazione Web o un'applicazione aziendale). Con CDI, il descrittore di deployment si chiama beans.xml ed è obbligatorio. Può essere usato per configurare determinate funzionalità (intercettori, decoratori, alternative, ecc.), Ma è essenziale per abilitare il CDI. Questo perché CDI ha bisogno di identificare i bean nel percorso della classe (questo è chiamato bean discovery), è così che CDI trasforma POJO in bean CDI.

Se la tua applicazione web contiene diversi file jar e vuoi avere il CDI abilitato sull'intera applicazione, ogni jar avrà bisogno del suo bean.xml per attivare CDI e discovery dei bean per ogni jar.

Come scrivere un Bean CDI

Un CDI Bean può essere qualsiasi tipo di classe che contenga la logica aziendale. Può essere chiamato direttamente dal codice Java tramite injection, oppure può essere richiamato tramite EL da una pagina JSF. Come si può vedere nel Listato 2-1, un bean è un POJO che non eredita o estende da nulla, può inserire riferimenti ad altri bean (@Inject), ha il suo ciclo di vita gestito dal contenitore (@PostConstruct), e può ottenere l'intercettazione del suo metodo tramite(@Transactional , un binding di intercettatore che si vedrà successivamente).

Listing 2-1. A BookService Bean Using Injection, Life-Cycle Management, and Interception

```
public class BookService {  
  
    @Inject  
    private NumberGenerator numberGenerator;  
    @Inject  
    private EntityManager em;  
  
    private Date instantiationDate;  
  
    @PostConstruct  
    private void initDate() {  
        instantiationDate = new Date();  
    }  
  
    @Transactional  
    public Book createBook(String title, Float price, String description) {  
        Book book = new Book(title, price, description);  
        book.setIsbn(numberGenerator.generateNumber());  
        book.setInstanciationDate(instantiationDate);  
        em.persist(book);  
        return book;  
    }  
}
```

Anatomia di un Bean CDI

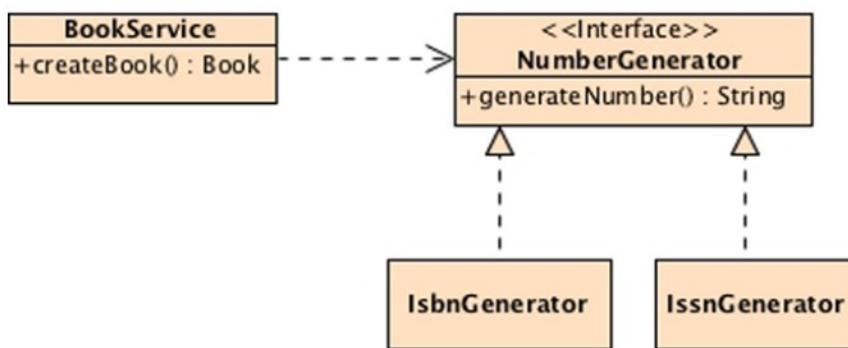
- Secondo la specifica CDI, il contenitore tratta qualsiasi classe che soddisfi le seguenti condizioni come un bean CDI:
 - Non è una classe interna non statica,
 - È una classe concreta, o è annotata @Decorator
 - Ha un costruttore predefinito senza parametri, o dichiara un costruttore annotato @Inject.
 - Un bean può avere un ambito opzionale, un nome EL facoltativo, un insieme di collegamenti di intercettatore e una gestione facoltativa del ciclo di vita.

Dependency Injection

Esempio: una classe Book rappresenta una copia di "H2G2", un Cliente rappresenta te e un PurchaseOrder ti indica che stai acquistando questo libro. Questi oggetti dipendono l'uno dall'altro: un libro può essere letto da un cliente e un ordine di acquisto si riferisce a diversi libri. Questa dipendenza è un valore del design orientato agli oggetti.

Ad esempio, il processo di creazione di un libro (BookService) può essere ridotto all'istanza di un oggetto Book, generando un numero univoco utilizzando un altro servizio (NumberGenerator) e persistendo il libro in un database. Il servizio NumberGenerator può generare un numero ISBN composto da 13 cifre o un formato precedente denominato ISSN con 8 cifre. Il BookService finirebbe quindi a dipendere da un IsbnGenerator o da un IssnGenerator in base a qualche condizione o ambiente.

La Figura mostra un diagramma di classe dell'interfaccia NumberGenerator con un metodo (String generateNumber ()) ed è implementato da IsbnGenerator e IssnGenerator. Il BookService dipende dall'interfaccia per generare un numero di libro.



Dipendenza di un BookService POJO tramite la parola chiave new:

```
public class BookService {  
  
    private NumberGenerator numberGenerator;  
  
    public BookService(NumberGenerator numberGenerator) {  
        this.numberGenerator = numberGenerator;  
    }  
  
    public Book createBook(String title, Float price, String description) {  
        Book book = new Book(title, price, description);  
        book.setIsbn(numberGenerator.generateNumber());  
        return book;  
    }  
}
```

Così ora una classe esterna potrebbe utilizzare il BookService con l'implementazione di cui ha bisogno:

```
BookService bookService = new BookService(new IsbnGenerator());  
BookService bookService = new BookService(new IssnGenerator());
```

@Inject

Poiché Java EE è un ambiente gestito, non è necessario costruire manualmente le dipendenze, ma è possibile lasciarlo fare al contenitore che inserisce un riferimento per noi. In breve, l'iniezione delle dipendenze CDI è la capacità di iniettare i bean in altri in modo errato, il che significa che non c'è XML ma annotazioni.

L'iniezione già esisteva in Java EE 5 con le annotazioni **@Resource**, **@PersistentUnit** o **@EJB**, ad esempio.

Con il CDI puoi iniettare praticamente ovunque grazie all'annotazione `@Inject`. Si noti che in Java EE 7 è ancora possibile utilizzare gli altri meccanismi di iniezione (`@Resource ...`). Il Listato 2-4 mostra come iniettare un riferimento di `NumberGenerator` nel `BookService` usando il CDI `@Inject`.

Listing 2-4. BookService Using `@Inject` to Get a Reference of `NumberGenerator`

```
public class BookService {  
  
    @Inject  
    private NumberGenerator numberGenerator;  
  
    public Book createBook(String title, Float price, String description) {  
        Book book = new Book(title, price, description);  
        book.setIsbn(numberGenerator.generateNumber());  
        return book;  
    }  
}
```

Come si può vedere nel Listato 2-4, una semplice annotazione `@Inject` sulla proprietà informa il contenitore che deve iniettare un riferimento di un'implementazione `NumberGenerator` nella proprietà `numberGenerator`. Questo è chiamato punto di iniezione (il punto in cui si trova l'annotazione `@Inject`). Il Listato 2-5 mostra l'implementazione di `IsbnGenerator`. Come puoi vedere non ci sono annotazioni speciali e la classe implementa l'interfaccia `NumberGenerator`.

Listing 2-5. The `IsbnGenerator` Bean

```
public class IsbnGenerator implements NumberGenerator {  
  
    public String generateNumber() {  
        return "13-84356-" + Math.abs(new Random().nextInt());  
    }  
}
```

Injection point

L'annotazione `@Inject` definisce un punto di iniezione che viene iniettato durante l'istanziazione del bean. L'iniezione può avvenire tramite tre diversi meccanismi: proprietà, setter o costruttore.

- Proprietà: `@Inject`

```
private NumberGenerator numberGenerator;
```

- Costruttore: `@Inject`

```
public BookService (NumberGenerator numberGenerator) {  
    this.numberGenerator = numberGenerator;
```

```
}
```

- Setter: `@Inject`

```
public void setNumberGenerator(NumberGenerator numberGenerator) {  
    this.numberGenerator = numberGenerator;  
}
```

Default injection

Se c'è una sola implementazione di numberGenerator possiamo specificare la sua iniezione tramite l'annotazione @Inject @default. In caso che si scriva soltanto @inject e sia presente sempre solo un'implementazione di numberGenerator il comportamento è lo stesso.

Listing 2-6. The IsbnGenerator Bean with the @Default Qualifier

@Default

```
public class IsbnGenerator implements NumberGenerator {  
  
    public String generateNumber() {  
        return "13-84356-" + Math.abs(new Random().nextInt());  
    }  
}
```

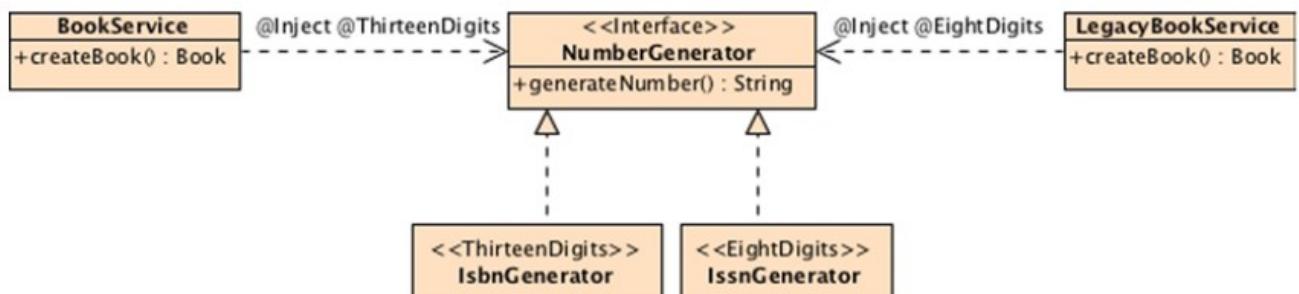
Qualificatori

Al momento dell'inizializzazione del sistema, il contenitore deve convalidare che esista esattamente un bean che soddisfi ciascun punto di iniezione.

Ciò significa che se non è disponibile alcuna implementazione di NumberGenerator, il contenitore ti informerà di una dipendenza insoddisfatta e non distribuirà l'applicazione. Se c'è solo un'implementazione, l'iniezione funzionerà usando il qualificatore @Default, se fossero disponibili più di un'implementazione predefinita, il contenitore informerà l'utente di una dipendenza ambigua e non distribuirà l'applicazione. Questo perché l'algoritmo di risoluzione typesafe ha esito negativo quando il contenitore non è in grado di identificare esattamente un bean da iniettare. Quindi, come fa un componente a scegliere quale implementazione (IsbnGenerator o IssnGenerator) deve essere iniettata?

La maggior parte dei framework fa molto affidamento sulla configurazione XML esterna per dichiarare e iniettare i bean. CDI utilizza qualificatori, che fondamentalmente sono annotazioni Java che portano ad un'iniezione typesafe e disambiguano un tipo senza dover ricorrere a nomi basati su String.

Diciamo che abbiamo un'applicazione con un BookService che crea libri con un numero ISBN di 13 cifre e un LegacyBookService che crea libri con un numero ISSN a 8 cifre. Come si può vedere nella Figura, entrambi i servizi iniettano un riferimento della stessa interfaccia NumberGenerator. I servizi distinguono tra le due implementazioni utilizzando qualificatori.



Un qualificatore è un'annotazione definita dall'utente, a sua volta annotata con @javax.inject.Qualifier. Ad esempio, potremmo introdurre i qualificatori per rappresentare i generatori di numeri a 13 e 8 cifre mostrati nel Listato 2-7 e nel Listato 2-8.

Listing 2-7. The ThirteenDigits Qualifier

```
@Qualifier  
@Retention(RUNTIME)  
@Target({FIELD, TYPE, METHOD})  
public @interface ThirteenDigits { }
```

Listing 2-8. The EightDigits Qualifier

```
@Qualifier  
@Retention(RUNTIME)  
@Target({FIELD, TYPE, METHOD})  
public @interface EightDigits { }
```

Una volta definiti i qualificatori necessari, questi devono essere applicati all'implementazione appropriata. Come puoi vedere sia nel Listato 2-9 che nel Listato 2-10, il qualificatore `@ThirteenDigits` viene applicato al bean `IsbnGenerator` e `@EightDigits` a `IssnGenerator`.

Listing 2-9. The `IsbnGenerator` Bean with the `@ThirteenDigits` Qualifier

```
@ThirteenDigits
public class IsbnGenerator implements NumberGenerator {

    public String generateNumber() {
        return "13-84356-" + Math.abs(new Random().nextInt());
    }
}
```

Listing 2-10. The `IssnGenerator` Bean with the `@EightDigits` Qualifier

```
@EightDigits
public class IssnGenerator implements NumberGenerator {

    public String generateNumber() {
        return "8-" + Math.abs(new Random().nextInt());
    }
}
```

Questi qualificatori vengono quindi applicati ai punti di iniezione per distinguere quale implementazione è richiesta dal cliente. Nel Listato 2-11 il `BookService` definisce esplicitamente l'implementazione a 13 cifre iniettando un riferimento del generatore di numeri `@ThirteenDigits` e nel Listato 2-12 il `LegacyBookService` inietta l'implementazione a 8 cifre.

Listing 2-11. `BookService` Using the `@ThirteenDigits` NumberGenerator Implementation

```
public class BookService {

    @Inject @ThirteenDigits
    private NumberGenerator numberGenerator;

    public Book createBook(String title, Float price, String description) {
        Book book = new Book(title, price, description);
        book.setIsbn(numberGenerator.generateNumber());
        return book;
    }
}
```

OPPURE

Listing 2-12. `LegacyBookService` Using the `@EightDigits` NumberGenerator Implementation

```
public class LegacyBookService {

    @Inject @EightDigits
    private NumberGenerator numberGenerator;

    public Book createBook(String title, Float price, String description) {
        Book book = new Book(title, price, description);
        book.setIsbn(numberGenerator.generateNumber());
        return book;
    }
}
```

Qualifiers with Members

Ogni volta che devi scegliere tra le implementazioni, crei un qualificatore (ad esempio, un'annotazione). Quindi, se hai bisogno di un generatore a due cifre e un generatore di numeri a dieci cifre, creerai annotazioni aggiuntive (ad es. @TwoDigits, @EightDigits, @TenDigits, @ThirteenDigits). Immagina che i numeri generati possano essere sia dispari che pari, per finire con un numero elevato di annotazioni: @TwoOddDigits, @TwoEvenDigits, @EightOddDigits, ecc. Un modo per evitare la moltiplicazione delle annotazioni è utilizzare i membri. Nel nostro esempio potremmo sostituire tutti questi qualificatori utilizzando il qualificatore singolo @NumberOfDigits con un'enumerazione come valore e un booleano per la parità (vedere il Listato 2-13).

Listing 2-13. The @NumberOfDigits with a Digits Enum and a Parity Boolean

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface NumberofDigits {
    Digits value();
    boolean odd();
}

public enum Digits {
    TWO,
    EIGHT,
    TEN,
    THIRTEEN
}
```

Il modo in cui useresti questo qualificatore con i membri non cambia da quello che hai visto finora. Il punto di iniezione qualificherà l'implementazione necessaria impostando i membri di annotazione come segue:

```
@Inject @NumberofDigits(value = Digits.THIRTEEN, odd = false)
private NumberGenerator numberGenerator;
```

And the concerned implementation will do the same.

```
@NumberofDigits(value = Digits.THIRTEEN, odd = false)
public class IsbnEvenGenerator implements NumberGenerator {...}
```

Qualificatori multipli

Un altro modo per qualificare un bean e un punto di iniezione è specificare più qualificatori. Quindi, invece di avere più qualificatori per la parità (@TwoOddDigits, @TwoEvenDigits ...) o avere un qualificatore con membri (@NumberOfDigits), avremmo potuto utilizzare due diversi set di qualificatori: un set per la parità (@Odd e @Even) e un altro per il numero di cifre. Ecco come potresti qualificare un generatore di 13 cifre pari.

```
@ThirteenDigits @Even
public class IsbnEvenGenerator implements NumberGenerator {...}
```

The injection point would use the same syntax.

```
@Inject @ThirteenDigits @Even
private NumberGenerator numberGenerator;
```

Alternative

I qualificatori ti consentono di scegliere tra più implementazioni di un'interfaccia in fase di sviluppo. Ma a volte si desidera iniettare un'implementazione in base a uno scenario di distribuzione specifico. Ad esempio, potresti voler utilizzare un generatore di numeri fittizi in un ambiente di test. Le alternative sono i bean annotati con il qualificatore speciale javax.enterprise.inject.Alternative. Per impostazione predefinita le alternative sono disabilitate e devono essere abilitate nel descrittore beans.xml per renderle disponibili per l'istanziazione e l'iniezione. Il Listato 2-14 mostra un'alternativa del generatore di numeri fittizi.

Listing 2-14. A Default Mock Generator Alternative

```
@Alternative
public class MockGenerator implements NumberGenerator {

    public String generateNumber() {
        return "MOCK";
    }
}
```

Listing 2-15. The beans.xml Deployment Descriptor Enabling an Alternative

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee →
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       version="1.1" bean-discovery-mode="all">

    <alternatives>
        <class>org.agoncal.book.javaee7.chapter02.MockGenerator</class>
    </alternatives>
</beans>
```

Producers

È stato mostrato come iniettare bean CDI in altri bean CDI. Ma puoi anche iniettare le primitive (ad esempio, int, long, float.), I tipi di array e qualsiasi POJO che non è abilitato CDI, grazie ai producers.

Per impostazione predefinita, non è possibile inserire classi come java.util.Date o java.lang.String. Questo perché tutte queste classi sono impacchettate nel file rt.jar (le classi dell'ambiente runtime Java) e questo archivio non contiene un descrittore di distribuzione beans.xml. Se un archivio non ha un bean.xml sotto la directory META-INF, CDI non attiverà la scoperta dei bean e POJO non potranno essere trattati come bean e, quindi, essere iniettabili. L'unico modo per essere in grado di iniettare POJO è usare i campi producers o metodi producers come mostrato nel Listato 2-16.

Listing 2-16. Producer Fields and Methods

```
public class NumberProducer {

    @Produces @ThirteenDigits
    private String prefix13digits = "13-";

    @Produces @ThirteenDigits
    private int editorNumber = 84356;

    @Produces @Random
    public double random() {
        return Math.abs(new Random().nextInt());
    }
}
```

Possiamo così utilizzarle in IsbnGenerator:

Listing 2-17. IsbnGenerator Injecting Produced Types

```
@ThirteenDigits
public class IsbnGenerator implements NumberGenerator {

    @Inject @ThirteenDigits
    private String prefix;

    @Inject @ThirteenDigits
    private int editorNumber;

    @Inject @Random
    private double postfix;

    public String generateNumber() {
        return prefix + editorNumber + postfix;
    }
}
```

Disposers

Negli esempi precedenti (Listato 2-17 e Listato 2-18) abbiamo utilizzato i produttori per creare tipi di dati o POJO in modo che potessero essere iniettati. Li abbiamo creati e non abbiamo dovuto distruggerli o chiuderli una volta usati. Ma alcuni metodi di produzione restituiscono oggetti che richiedono una distruzione esplicita come una connessione JDBC (Java Database Connectivity), una sessione JMS o un gestore di entità. Per la creazione, CDI utilizza i produttori, e per la distruzione, i disposers. Il Listato 2-19 mostra una classe di utility che crea e chiude una connessione JDBC. CreateConnection utilizza un driver JDBC Derby, crea una connessione con un URL specifico, gestisce le eccezioni e restituisce una connessione JDBC aperta. Questo metodo è annotato con @Produces. D'altra parte, il metodo closeConnection termina la connessione JDBC. È annotato con @Dispose.

Listing 2-19. JDBC Connection Producer and Disposer

```
public class JDBCConnectionProducer {

    @Produces
    private Connection createConnection() {
        Connection conn = null;
        try {
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance();
            conn = DriverManager.getConnection("jdbc:derby:memory:chapter02DB", "APP", "APP");

        } catch (InstantiationException | IllegalAccessException | ClassNotFoundException) {
            e.printStackTrace();
        }
        return conn;
    }

    private void closeConnection(@Disposes Connection conn) throws SQLException {
        conn.close();
    }
}

@ApplicationScoped
public class DerbyPingService {

    @Inject
    private Connection conn;

    public void ping() throws SQLException {
        conn.createStatement().executeQuery("SELECT 1 FROM SYSIBM.SYSDUMMY1");
    }
}
```

Scopes

CDI parla di Dependency Injection ma anche di Contesto (la "C" in CDI). Ogni oggetto gestito da CDI ha uno scopo ben definito e un ciclo di vita che è legato a un contesto specifico. In Java, l'ambito di un POJO è piuttosto semplice: si crea un'istanza di una classe usando la nuova parola chiave e ci si basa sulla garbage collection per liberarsene e liberare memoria. Con CDI, un bean è associato a un contesto e rimane in tale contesto fino a quando il bean non viene distrutto dal contenitore. Non c'è modo di rimuovere manualmente un bean da un contesto.

Mentre il livello web ha ambiti ben definiti (applicazione, sessione, richiesta), non c'era nulla di simile per il livello di servizio (per i bean di sessione stateless e stateful). Questo perché quando i bean di sessione o POJO vengono utilizzati all'interno delle applicazioni Web, non sono a conoscenza dei contesti delle applicazioni Web. CDI ha unito i livelli web e di servizio associandoli a scopi significativi. CDI definisce i seguenti ambiti incorporati e fornisce anche punti di estensione in modo da poter creare il proprio:

- ApplicationScoped (@ApplicationScoped): si estende per l'intera durata di un'applicazione. Il bean viene creato una sola volta per tutta la durata dell'applicazione e viene scartato quando l'applicazione viene chiusa. Questo ambito è utile per le classi di utilità o helper o per gli oggetti che memorizzano i dati condivisi dall'intera applicazione (ma è necessario prestare attenzione ai problemi di concorrenza quando è necessario accedere ai dati da più thread).
- SessionScoped(@SessionScoped): comprende diverse richieste HTTP o più invocazioni di metodi 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. Questo ambito è per gli oggetti necessari durante la sessione, come le preferenze dell'utente o le credenziali di accesso.
- RequestScoped(@RequestScoped): corrisponde a una singola richiesta HTTP o a una chiamata di un metodo. Il bean viene creato per la durata dell'invocazione del metodo e viene scartato al termine del metodo. Viene utilizzato per le classi di servizio o i bean di backup JSF necessari solo per la durata di una richiesta HTTP.
- ConversationScoped(@ConversationScoped): comprende tra più chiamate all'interno dei limiti 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 lo stesso del client. Un bean dipendente viene creato ogni volta che viene iniettato e il riferimento viene rimosso quando viene rimosso il target di iniezione. Questo è l'ambito predefinito per CDI.

Conversation

L'ambito della conversation è leggermente diverso dall'applicazione, dalla sessione o dall'ambito della richiesta. Mantiene lo stato associato a un utente, include più richieste ed è demarcato a livello di codice dall'applicazione. Un bean @ConversationScoped può essere utilizzato per un processo di lunga durata in cui vi è un inizio e una fine definiti come la navigazione attraverso una procedura guidata o l'acquisto di articoli e il controllo di un negozio online.

La richiesta di oggetti con ambito ha una durata molto breve che di solito dura per una singola richiesta (richiesta HTTP o richiamo del metodo). Esistono oggetti di livello presentazione che possono essere utilizzati su più di una pagina ma non su tutta la sessione. Per questo, CDI ha uno scope conversazione speciale (@ConversationScoped).

A differenza degli oggetti con ambito sessione scaduti automaticamente dal contenitore, gli oggetti con ambito conversazione hanno un ciclo di vita ben definito che inizia e termina esplicitamente a livello di codice utilizzando l'API javax.enterprise.context.Conversation. Ad esempio, pensa a un'applicazione web per la creazione guidata cliente. Il wizard è composto da tre passaggi.

Nella prima fase, il cliente inserisce le informazioni di accesso (ad es. Nome utente e password).

Nella seconda fase, il cliente inserisce i dettagli dell'account come il nome, il cognome, l'indirizzo e l'indirizzo e-mail. Il passaggio finale della procedura guidata conferma tutte le informazioni raccolte

e crea l'account. Il Listato 2-21 mostra il bean con scope della conversazione che implementa la procedura guidata del creatore del cliente.

Listing 2-21. A Wizard to Create a Customer Using a Conversation

@ConversationScoped

```
public class CustomerCreatorWizard implements Serializable {

    private Login login;
    private Account account;

    @Inject
    private CustomerService customerService;

    @Inject
    private Conversation conversation;

    public void saveLogin() {
        conversation.begin();

        login = new Login();
        // Sets login properties
    }

    public void saveAccount() {
        account = new Account();
        // Sets account properties
    }

    public void createCustomer() {
        Customer customer = new Customer();
        customer.setLogin(login);
        customer.setAccount(account);
        customerService.createCustomer(customer);

        conversation.end();
    }
}
```

Il CustomerCreatorWizard nel Listato 2-21 è annotato con `@ConversationScoped`. Quindi, inietta un `CustomerService`, per creare il Cliente, ma, cosa più importante, inietta una `Conversazione`. Questa interfaccia consente il controllo programmatico sul ciclo di vita dell'ambito della conversazione. Si noti che quando viene richiamato il metodo `saveLogin`, viene avviata la conversazione (`conversation.begin()`). La conversazione è ora iniziata e viene utilizzata per la durata della procedura guidata. Una volta richiamato l'ultimo passo della procedura guidata, viene richiamato il metodo `createCustomer` e la conversazione termina (`conversation.end()`). La Tabella 2-3 offre una panoramica dell'API di conversazione.

Table 2-3. Conversation API

Method	Description
<code>void begin()</code>	Marks the current transient conversation long-running
<code>void begin(String id)</code>	Marks the current transient conversation long-running, with a specified identifier
<code>void end()</code>	Marks the current long-running conversation transient
<code>String getId()</code>	Gets the identifier of the current long-running conversation
<code>long getTimeout()</code>	Gets the timeout of the current conversation
<code>void setTimeout(long millis)</code>	Sets the timeout of the current conversation
<code>boolean isTransient()</code>	Determines if the conversation is marked transient or long-running

Interceptor

Gli intercettori ti consentono di aggiungere preoccupazioni trasversali ai tuoi bean. Come mostrato nella Figura 2-2, quando un client richiama un metodo su un bean gestito (e quindi un bean CDI, un EJB, un servizio web RESTful ...), il contenitore è in grado di intercettare la chiamata e elaborare la business logic prima che il metodo del bean è invocato. Gli intercettori rientrano in quattro tipi:

- Constructor-level interceptors: Interceptor associato a un costruttore della classe target (**@AroundConstruct**).
- Method-level interceptors: Interceptor associato a uno specifico metodo di business (**@AroundInvoke**).
- Timeout method interceptors: Interceptor che si interpone sui metodi di timeout con **@AroundTimeout** (utilizzato solo con il servizio timer EJB).
- Life-cycle callback interceptors: Interceptor che si interpone sulle callback dell'evento del ciclo di vita dell'istanza di destinazione (**@PostConstruct** e **@PreDestroy**).

Target class interceptor

Esistono diversi modi per definire l'intercettazione. Il più semplice consiste nell'aggiungere interceptor (a livello di metodo, timeout o intercettatori del ciclo di vita) al bean stesso come mostrato nel Listato 2-23. CustomerService annota logMethod () con @AroundInvoke. logMethod () è usato per registrare un messaggio quando un metodo viene inserito e chiuso.

Listing 2-23. A CustomerService Using Around-Invoke Interceptor

```
@Transactional
public class CustomerService {

    @Inject
    private EntityManager em;
    @Inject
    private Logger logger;

    public void createCustomer(Customer customer) {
        em.persist(customer);
    }

    public Customer findCustomerById(Long id) {
        return em.find(Customer.class, id);
    }

    @AroundInvoke
    private Object logMethod(InvocationContext ic) throws Exception {
        logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
        try {
            return ic.proceed();
        } finally {
            logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
        }
    }
}
```

La base di un metodo AroundInvoke è:

`@AroundInvoke`

`Object <METHOD>(InvocationContext ic) throws Exception;`

Il metodo non deve essere `ne stati` e `ne final`, il metodo deve avere

`javax.interceptor.InvocationContext` come tipo di variabile e restituire un oggetto, il metodo deve lanciare un'eccezione `Exception`.

Table 2-4. Definition of the InvocationContext Interface

Method	Description
getContextData	Allows values to be passed between interceptor methods in the same InvocationContext instance using a Map.
getConstructor	Returns the constructor of the target class for which the interceptor was invoked.
getMethod	Returns the method of the bean class for which the interceptor was invoked.
getParameters	Returns the parameters that will be used to invoke the business method.
getTarget	Returns the bean instance that the intercepted method belongs to.
getTimer	Returns the timer associated with a @Timeout method.
proceed	Causes the invocation of the next interceptor method in the chain. It returns the result of the next method invoked. If a method is of type void, proceed returns null.
setParameters	Modifies the value of the parameters used for the target class method invocation. The types and the number of parameters must match the bean's method signature, or IllegalArgumentException is thrown.

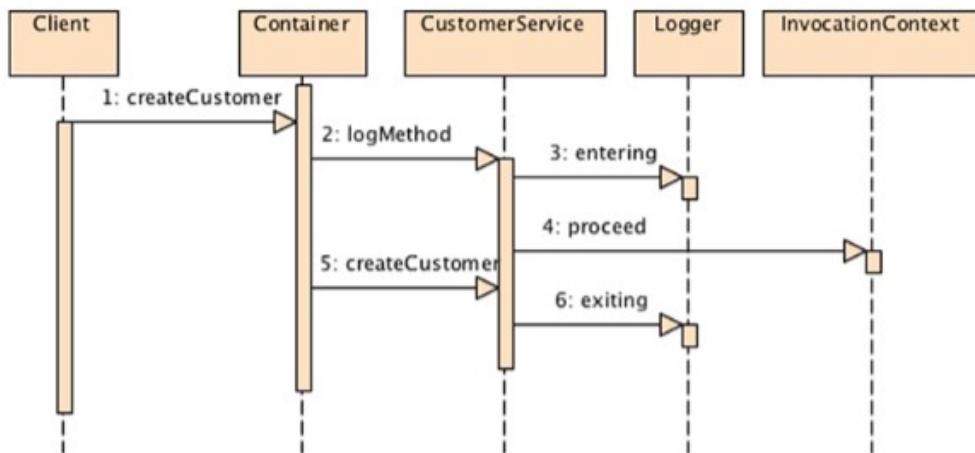


Figure 2-6. A call to a business method being intercepted

Class interceptor

Il listato 2-23 definisce un intercettore che è disponibile solo per CustomerService. Ma la maggior parte delle volte si desidera isolare i problemi trasversali in una classe separata e dire al contenitore di intercettare le chiamate su diversi bean. La registrazione è un tipico esempio di una situazione in cui si desidera che tutti i metodi di tutti i bean effettuino il log in entrata e in uscita dai messaggi. Per specificare un intercettore di classe, è necessario sviluppare una classe separata e indicare al contenitore di applicarlo su un metodo specifico di bean o bean.

Per condividere del codice tra più bean, prendiamo i metodi logMethod () dal Listato 2-23 e isolateli in una classe separata come mostrato nel Listato 2-24. Si noti il metodo init () che è annotato con @AroundConstruct e verrà invocato solo quando viene chiamato il costruttore del bean.

Listing 2-24. An Interceptor Class with Around-Invoke and Around-Construct

```

public class LoggingInterceptor {

    @Inject
    private Logger logger;

    @AroundConstruct
    private void init(InvocationContext ic) throws Exception {
        logger.fine("Entering constructor");
        try {
            ic.proceed();
        } finally {
            logger.fine("Exiting constructor");
        }
    }
}

```

```

@AroundInvoke
public Object logMethod(InvocationContext ic) throws Exception {
    logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
    try {
        return ic.proceed();
    } finally {
        logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
    }
}

```

Listing 2-25. CustomerService Uses an Interceptor on One Method

```

@Transactional
public class CustomerService {

    @Inject
    private EntityManager em;

    @Interceptors(LoggingInterceptor.class)
    public void createCustomer(Customer customer) {
        em.persist(customer);
    }

    public Customer findCustomerById(Long id) {
        return em.find(Customer.class, id);
    }
}

```

In questo caso l'interceptor vale soltanto per createCustomer.

Per farlo valere per tutta la classe basta scrivere:

```

@Transactional
@Interceptors(LoggingInterceptor.class)
public class CustomerService {
    public void createCustomer(Customer customer) {...}
    public Customer findCustomerById(Long id) {...}
}

```

Mentre se si vuole escludere qualche metodo basta scrivere:

```

@Transactional
@Interceptors(LoggingInterceptor.class)
public class CustomerService {
    public void createCustomer(Customer customer) {...}
    public Customer findCustomerById(Long id) {...}
    @ExcludeClassInterceptors
    public Customer updateCustomer(Customer customer) { ... }
}

```

Ciclo di vita dell'interceptor

Con un'annotazione di callback, è possibile informare il contenitore per richiamare un metodo in una determinata fase del ciclo di vita (@PostConstruct e @PreDestroy). Ad esempio, se si desidera registrare una voce ogni volta che viene creata un'istanza di bean, è sufficiente aggiungere un'annotazione @PostConstruct su un metodo del bean e aggiungere alcuni meccanismi di

registrazione. Ma cosa succede se hai bisogno di catturare eventi del ciclo di vita attraverso molti tipi di bean? Gli intercettori del ciclo di vita consentono di isolare un codice in una classe e di richiamarlo quando viene attivato un evento del ciclo di vita.

Il Listato 2-26 mostra la classe ProfileInterceptor con due metodi: logMethod () , utilizzato per postconstruction (@PostConstruct) e profile () , utilizzato per l'intercettazione del metodo (@AroundInvoke).

Listing 2-26. An Interceptor with Both Life-Cycle and Around-Invoke

```
public class ProfileInterceptor {

    @Inject
    private Logger logger;

    @PostConstruct
    public void logMethod(InvocationContext ic) throws Exception {
        logger.fine(ic.getTarget().toString());
        try {
            ic.proceed();
        } finally {
            logger.fine(ic.getTarget().toString());
        }
    }

    @AroundInvoke
    public Object profile(InvocationContext ic) throws Exception {
        long initTime = System.currentTimeMillis();
        try {
            return ic.proceed();
        } finally {
            long diffTime = System.currentTimeMillis() - initTime;
            logger.fine(ic.getMethod() + " took " + diffTime + " millis");
        }
    }
}
```

L'inserimento di più interception avviene nel seguente modo:

Listing 2-28. CustomerService Chaining Serveral Interceptors

```
@Stateless
@Interceptors({I1.class, I2.class})
public class CustomerService {
    public void createCustomer(Customer customer) {...}
    @Interceptors({I3.class, I4.class})
    public Customer findCustomerById(Long id) {...}
    public void removeCustomer(Customer customer) {...}
    @ExcludeClassInterceptors
    public Customer updateCustomer(Customer customer) {...}
}
```

Interceptor Binding

Gli intercettori sono definiti nelle loro specifiche (JSR 318) e possono essere utilizzati in qualsiasi bean gestito (EJB, servlet, servizi Web RESTful ...). Ma la specifica CDI lo ha esteso aggiungendo il binding di interceptor, il che significa che il binding dell'interceptor può essere usato solo se CDI è abilitato.

Se si guarda il Listato 2-25 come esempio, è possibile vedere il modo in cui funzionano gli interceptor; è necessario specificare l'implementazione dell'intercettore direttamente sull'implementazione del bean (ad esempio, @Interceptors (LoggingInterceptor.class)). Questo è tipicamente sicuro, ma non strettamente accoppiato. CDI fornisce l'intercettazione degli intercettori che introduce un livello indiretto e un accoppiamento libero. Un tipo di collegamento interceptor è

un'annotazione definita dall'utente che è a sua volta annotata @InterceptorBinding che associa la classe interceptor al bean senza alcuna dipendenza diretta tra le due classi.

Il Listato 2-29 mostra un binding intercettore chiamato Loggable. Come puoi vedere, questo codice è molto simile a un qualificatore. Un legame intercettore è un'annotazione stessa annotata con @InterceptorBinding, che può essere vuota o avere membri (come quelli visti nel Listato 2-13).

Listing 2-29. Loggable Interceptor Binding

```
@InterceptorBinding  
@Target({METHOD, TYPE})  
@Retention(RUNTIME)  
public @interface Loggable { }
```

Once you have an interceptor binding you need to attach it to the interceptor itself. This is done by annotating the interceptor with both @Interceptor and the interceptor binding (@Loggable in Listing 2-30).

Listing 2-30. Loggable Interceptor

```
@Interceptor  
@Loggable  
public class LoggingInterceptor {  
  
    @Inject  
    private Logger logger;  
  
    @AroundInvoke  
    public Object logMethod(InvocationContext ic) throws Exception {  
        logger.entering(ic.getTarget().toString(), ic.getMethod().getName());  
        try {  
            return ic.proceed();  
        } finally {  
            logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());  
        }  
    }  
}
```

Listing 2-31. CustomerService using the Interceptor Binding

```
@Transactional  
@Loggable  
public class CustomerService {  
  
    @Inject  
    private EntityManager em;  
  
    public void createCustomer(Customer customer) {  
        em.persist(customer);  
    }  
  
    public Customer findCustomerById(Long id) {  
        return em.find(Customer.class, id);  
    }  
}
```

La specifica di un interceptor è disabilitata di default, quindi deve essere abilitata nel bean.xml:

Listing 2-32. The beans.xml Deployment Descriptor Enabling an Interceptor

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee        →
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       version="1.1" bean-discovery-mode="all">

    <interceptors>
        <class>org.agoncal.book.javaee7.chapter02.LoggingInterceptor</class>
    </interceptors>
</beans>
```

Prioritizing interceptors binding

È possibile stabilire le priorità di un'interceptor utilizzando l'annotazione `@javax.annotation.Priority` (o l'equivalente XML in `bean.xml`) insieme a un valore di priorità come mostrato nel Listato 2-33. Questo serve per dare un ordine di esecuzione agli interceptor:

Listing 2-33. Loggable Interceptor Binding

```
@Interceptor
@Loggable
@Priority(200)
public class LoggingInterceptor {

    @Inject
    private Logger logger;

    @AroundInvoke
    public Object logMethod(InvocationContext ic) throws Exception {
        logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
        try {
            return ic.proceed();
        } finally {
            logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
        }
    }
}
```

@Priority prende un numero intero che può assumere qualsiasi valore. La regola è che gli intercettori con valori di priorità più piccoli vengono chiamati per primi. Java EE 7 definisce le priorità a livello di piattaforma e quindi è possibile chiamare gli interceptor prima o dopo determinati eventi. Javax.interceptor. L'annotazione interceptor definisce il seguente insieme di costanti:

- PLATFORM_BEFORE = 0: inizio dell'intervallo per i primi intercettori definito dalla piattaforma Java EE.
 - LIBRARY_BEFORE = 1000: inizio dell'intervallo per i primi intercettori definito dalle librerie di estensione.
 - APPLICAZIONE = 2000: Inizio intervallo per intercettatori definiti dalle applicazioni.
 - LIBRARY_AFTER = 3000: Inizio dell'intervallo per gli intercettori ritardati definiti dalle librerie di estensione.
 - PLATFORM_AFTER = 4000: inizio dell'intervallo per gli intercettori ritardati definito dalla piattaforma Java EE.

Quindi, se vuoi che il tuo intercettore venga eseguito prima di qualsiasi intercettatore di applicazioni, ma dopo qualsiasi intercettore di piattaforma in anticipo, puoi scrivere quanto segue:

```
Decorators @Interceptor  
           @Loggable  
           @Priority(Interceptor.Priority.LIBRARY_BEFORE + 10)  
           public class LoggingInterceptor {...}
```

Gli intercettori eseguono compiti trasversali e sono perfetti per risolvere problemi tecnici come la gestione delle transazioni, la sicurezza o la registrazione. Gli intercettatori non sono appropriati per separare le preoccupazioni legate al business. Questo possono farlo però i decorators.

I decorators sono un modello di design comune della “Gang of Four”. L’idea è di prendere una classe e avvolgere un’altra classe intorno ad essa (ad esempio, decorarla). In questo modo, quando chiami un classe decorata, passi sempre attraverso il decoratore circostante prima di raggiungere la classe target. I decoratori hanno lo scopo di aggiungere ulteriore logica a un metodo di business. Non sono in grado di risolvere problemi tecnici. Intercettatori e decoratori, sebbene simili in molti modi, sono complementari.

Prendiamo l’esempio di un generatore di numeri ISSN. ISSN è un numero di 8 cifre che è stato sostituito dal codice ISBN (numero di 13 cifre). Invece di avere due generatori di numeri separati (come quello nel Listato 2-9 e nel Listato 2-10) puoi decorare il generatore ISSN per aggiungere un algoritmo in più che trasforma un numero di 8 cifre in un numero di 13 cifre.

Il Listato 2-34 implementa un tale algoritmo come decoratore. La classe

FromEightToThirteenDigitsDecorator è annotata con javax.decorator.Decorator, implementa le interfacce di business (il NumberGenerator) e sostituisce il metodo generateNumber (un decoratore può essere dichiarato come una classe astratta in modo che non debba implementare tutti i metodi commerciali delle interfacce se ce ne sono molti). Il metodo generateNumber () richiama il bean di destinazione per generare un ISSN, aggiunge alcune logiche di business per trasformare tale numero e restituisce un numero ISBN.

Listing 2-34. Decorator Transforming an 8-Digit Number to 13

```
@Decorator
public class FromEightToThirteenDigitsDecorator implements NumberGenerator {

    @Inject @Delegate
    private NumberGenerator numberGenerator;

    public String generateNumber() {
        String issn = numberGenerator.generateNumber();
        String isbn = "13-84356" + issn.substring(1);
        return isbn;
    }
}
```

I decoratori come le alternative e gli interception devono essere definiti nel file beans.xml:

```
<decorators>
    <class>org.agoncal.book.javaee7.chapter02.FromEightToThirteenDigitsDecorator</
class>
</decorators>
```

Eventi

Gli eventi consentendo ai bean di interagire senza alcuna dipendenza dal tempo di compilazione. Un bean può definire un evento, un altro bean può attivare l’evento e un altro bean può gestire l’evento. I bean possono essere in pacchetti separati e persino in livelli separati dell’applicazione. Questo schema di base segue il modello di osservatore/osservabile della “Gang of Four”. I produttori di eventi attivano gli eventi utilizzando l’interfaccia javax.enterprise.event.Event. Un produttore solleva gli eventi chiamando il metodo fire(), passa l’oggetto evento e non dipende dall’osservatore. Nel Listato 2-36 il BookService attiva un evento (bookAddedEvent) ogni volta che viene creato un libro. Il code bookAddedEvent.fire (book) attiva l’evento e notifica a tutti i metodi osservatori che osservano questo particolare evento. Il contenuto di questo evento è l’oggetto del libro stesso che verrà portato dal produttore al consumatore.

Listing 2-36. The BookService Fires an Event Each Time a Book Is Created

```
public class BookService {  
  
    @Inject  
    private NumberGenerator numberGenerator;  
  
    @Inject  
    private Event<Book> bookAddedEvent;  
  
    public Book createBook(String title, Float price, String description) {  
        Book book = new Book(title, price, description);  
        book.setIsbn(numberGenerator.generateNumber());  
        bookAddedEvent.fire(book);  
        return book;  
    }  
}
```

L'evento è così mandato ad un osservatore, un osservatore può essere un bean con uno o più metodi osservatori. Ogni parametro del metodo dell'osservatore è annotato con `@Observes`.

Listing 2-37. The InventoryService Observes the Book Event

```
public class InventoryService {  
  
    @Inject  
    private Logger logger;  
    List<Book> inventory = new ArrayList<>();  
  
    public void addBook(@Observes Book book) {  
        logger.info("Adding book " + book.getTitle() + " to inventory");  
        inventory.add(book);  
    }  
}
```

In presenza di più eventi si possono aggiungere delle proprie annotazioni, il codice precedente per esempio con l'aggiunta di un nuovo evento cambierà nel seguente modo:

Listing 2-38. The BookService Firing Several Events

```
public class BookService {  
  
    @Inject  
    private NumberGenerator numberGenerator;  
  
    @Inject @Added  
    private Event<Book> bookAddedEvent;  
  
    @Inject @Removed  
    private Event<Book> bookRemovedEvent;  
  
    public Book createBook(String title, Float price, String description) {  
        Book book = new Book(title, price, description);  
        book.setIsbn(numberGenerator.generateNumber());  
        bookAddedEvent.fire(book);  
        return book;  
    }  
  
    public void deleteBook(Book book) {  
        bookRemovedEvent.fire(book);  
    }  
}
```

Listing 2-39. The InventoryService Observing Several Events

```
public class InventoryService {  
  
    @Inject  
    private Logger logger;  
    List<Book> inventory = new ArrayList<>();  
  
    public void addBook(@Observes @Added Book book) {  
        logger.info("Adding book " + book.getTitle() + " to inventory");  
        inventory.add(book);  
    }  
  
    public void removeBook(@Observes @Removed Book book) {  
        logger.info("Removing book " + book.getTitle() + " from inventory");  
        inventory.remove(book);  
    }  
}
```

Java Persistence API

Le entità sono oggetti che vivono poco tempo in memoria e in modo persistente. Java persistence ci permette di marcare una classe con l'annotazione `@Entity`, che fa sì che venga creata nel database una tabella sul codice scritto in java. Caratteristiche di un'entità:

- La classe dell'entità deve contenere l'annotazione `@javax.persistence.Entity`.
- L'annotazione `@javax.persistence.Id` è denominata per definire una semplice chiave primaria.
- La classe entità ha un costruttore vuoto che deve essere `public()` o `protected()`. Non può contenere la parola chiave `final`.
- Se l'entità deve essere passata come oggetto remoto deve implementare l'interfaccia `Serializable`.

Listing 4-1. Simple Example of a Book Entity

```
@Entity
public class Book {

    @Id @GeneratedValue
    private Long id;
    private String title;
    private Float price;
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;

    public Book() {
    }

    // Getters, setters
}
```

Object relation-mapping

Il principio di ORM è delegare a strumenti o framework esterni (nel nostro caso, JPA) il compito di creare una corrispondenza tra oggetti e tabelle. Il mondo di classi, oggetti e attributi può quindi essere mappato su database relazionali costituiti da tabelle contenenti righe e colonne. La mappatura offre una vista orientata agli oggetti.

I metadati consentono al provider di persistenza di riconoscere un'entità e interpretare la mappatura. I metadati possono essere scritti in due diversi formati.

- Annotazioni: il codice dell'entità viene annotato direttamente con tutti i tipi di annotazioni descritte nel pacchetto `javax.persistence`.
- Descrittori XML: invece di (o in aggiunta a) annotazioni, è possibile utilizzare descrittori XML.

Il mapping è definito in un file XML esterno che verrà distribuito con le entità. Questo può essere molto utile quando la configurazione del database cambia a seconda dell'ambiente. Per accedere agli elementi del database si usano i metodi di accesso(get) sull'oggetto restituito dal database.

Entità query

La parte centrale dell'API responsabile per l'orchestrazione delle entità è `javax.persistence.EntityManager`. Il suo ruolo è quello di gestire le entità, leggere e scrivere su un determinato database e consentire operazioni CRUD (creazione, lettura, aggiornamento ed eliminazione) semplici su entità e query complesse tramite JPQL. In termini tecnici, il gestore di entità è solo un'interfaccia la cui implementazione viene eseguita dal provider di persistenza (ad es. EclipseLink). Il seguente snippet di codice mostra come ottenere un gestore di entità e mantenere un'entità di libro:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("chapter04PU");
EntityManager em = emf.createEntityManager();
em.persist(book);
```

Le query che si usano per interrogare la tabella vanno scritte nella stessa classe dove si è definita l'entità, inserendo una nuova annotazione chiamata @NamedQuery():

```
@Entity
@NamedQuery(name = "findBookH2G2", →
            query = "SELECT b FROM Book b WHERE b.title = 'H2G2' ")
public class Book {

    @Id @GeneratedValue
    private Long id;
    private String title;
    private Float price;
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;

    // Constructors, getters, setters
}
```

Un esempio di main per il nostro esercizio è il seguente:

```
public class Main {

    public static void main(String[] args) {

        // 1-Creates an instance of book
        Book book = new Book("H2G2", "The Hitchhiker's Guide to the Galaxy", 12.5F, ←
                            "1-84023-742-2", 354, false);

        // 2-Obtains an entity manager and a transaction
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("chapter04PU");
        EntityManager em = emf.createEntityManager();

        // 3-Persists the book to the database
        EntityTransaction tx = em.getTransaction();
        tx.begin();
        em.persist(book);
        tx.commit();

        // 4-Executes the named query
        book = em.createNamedQuery("findBookH2G2", Book.class).getSingleResult();

        // 5-Closes the entity manager and the factory
        em.close();
        emf.close();
    }
}
```

Tutto questo genererà un file chiamato persistence.xml, in cui ci sarà il nome del nostro database, un username e password per accedergli.

```
<persistence-unit name="chapter04PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>org.agoncal.book.javaee7.chapter04.Book</class>
    <properties>
        <property name="javax.persistence.schema-generation-action" value="drop-and-create"/>
        <property name="javax.persistence.schema-generation-target" value="database"/>
        <property name="javax.persistence.jdbc.driver" →
                    value="org.apache.derby.jdbc.ClientDriver"/>
        <property name="javax.persistence.jdbc.url" →
                    value="jdbc:derby://localhost:1527/chapter04DB;create=true"/>
        <property name="javax.persistence.jdbc.user" value="APP"/>
        <property name="javax.persistence.jdbc.password" value="APP"/>
    </properties>
</persistence-unit>
</persistence>
```

Elementary mapping

In java EE 7 esiste un annotazione che permette di cambiare il nome ad una tabella: `@Table`, un esempio del suo uso è: `@Table(name = "t_table")`; inserita subito dopo `@Entity`.

Per identificare una chiave primaria si usa `@Id` mentre per indicare che un valore è autogenerato `@GeneratedValue`, i valori che può assumere sono i seguenti:

1. Primitive Java types: byte, int, short, long, char
2. Wrapper classes of primitive Java types: Byte, Integer, Short, Long, Character
3. Arrays of primitive or wrapper types: int[], Integer[], etc.
4. Strings, numbers, and dates: java.lang.String, java.math.BigInteger, java.util.Date, java.sql.Date

Si possono anche creare delle classi per usare id con più valori, queste sono scritte con l'annotazione `@Embeddable`, usa la stessa notazione dei JavaEnterprise Bean:

```
Embeddable
public class NewsId {
    private String title;
    private String language;
    // Constructors, getters, setters, equals, and hashCode
}
```

Questa classe sarà poi usata nella nostra classe entità con l'annotazione `@EmbeddedId`:

```
@Entity
public class News {
    @EmbeddedId
    private NewsId id;
    private String content;
    // Constructors, getters, setters
}
```

Nel main avremo poi:

```
NewsId pk = new NewsId("Richard Wright has died on September 2008", "EN")
```

```
News news = em.find(News.class, pk);
```

Un'altro metodo per dichiarare una chiave composta è tramite l'annotazione `@IdClass`. È un approccio diverso in cui ogni attributo sulla classe della chiave primaria deve anche essere dichiarato nella classe dell' entità e annotato con `@Id`.

```
public class NewsId {
    private String title;
    private String language;
    // Constructors, getters, setters, equals, and hashCode
}
<---- new class ---->
@Entity
@IdClass(NewsId.class)
public class News {
    @Id private String title;
    @Id private String language;
    private String content;
    // Constructors, getters, setters
}
```

@Column

Utilizzando l'annotazione `@Column` possiamo modificare i valori di una colonna:

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Column {
    String name() default "";
    boolean unique() default false;
    boolean nullable() default true;
```

```

        boolean insertable() default true;
        boolean updatable() default true;
        String columnDefinition() default "";
        String table() default "";
        int length() default 255;
        int precision() default 0; // decimal precision
        int scale() default 0;
        // decimal scale
    }

```

La sintassi è @Column(nome attributo dell'interfaccia = valore, ...)

@Temporal

Per memorizzare un attributo di tipo tempo si utilizza:

```

@Temporal(TemporalType.DATE)
private Date dateOfBirth;
@Temporal(TemporalType.TIME)
private Date hour;
@Temporal(TemporalType.TIMESTAMP)
private Date creationDate;

```

L'annotazione @Transient si usa per non mappare un attributo o tabella nel database.

Relazioni

Diversi tipi di associazioni possono esistere tra le classi.

Prima di tutto, un'associazione ha una direzione. Può essere unidirezionale (vale a dire, un oggetto può navigare verso un altro) o bidirezionale (vale a dire, un oggetto può navigare verso un altro e viceversa). In Java, si utilizza la sintassi punto (.). Per spostarsi tra gli oggetti. Ad esempio, quando si scrive customer.getAddress().GetCountry(), si passa dall'oggetto Cliente a Indirizzo e quindi all'attributo Country.

In UML (Unified Modeling Language), per rappresentare un'associazione unidirezionale tra due classi, si utilizza una freccia per indicare l'orientamento.



Per indicare un'associazione bidirezionale tra due classi si usa:



Mentre per indicarne la molteplicità:



La cardinalità delle associazioni possono essere le seguenti:

Cardinality	Direction
One-to-one	Unidirectional
One-to-one	Bidirectional
One-to-many	Unidirectional
Many-to-one/one-to-many	Bidirectional
Many-to-one	Unidirectional
Many-to-many	Unidirectional
Many-to-many	Bidirectional

Chiavi esterne in Java EE 7

Prendiamo le seguenti classi:

```
@Entity  
public class Customer {  
  
    @Id @GeneratedValue  
    private Long id;  
    private String firstName;  
    private String lastName;  
    private String email;  
    private String phoneNumber;  
    private Address address;  
  
    // Constructors, getters, setters  
}
```

```
@Entity  
public class Address {  
  
    @Id @GeneratedValue  
    private Long id;  
    private String street1;  
    private String street2;  
    private String city;  
    private String state;  
    private String zipcode;  
    private String country;  
  
    // Constructors, getters, setters  
}
```

Tramite l'uso di alcune notazioni possiamo creare riferimenti esterni:`@OneToOne`, `@OneToMany`, `@ManyToMany`.

Nell'esempio precedente abbiamo:

Nella classe Customer:

```
@OneToOne(name = "address_fk");  
private Address address;
```

Nella classe Address

```
@OneToOne(mappedBy = "address")  
private Customer customer;
```

Entity Manager

L'entity manager è un elemento centrale di JPA. Gestisce lo stato e il ciclo di vita delle entità, nonché l'interrogazione delle entità all'interno di un contesto di persistenza. È responsabile della creazione e rimozione di istanze di entità persistenti e ricerca di entità tramite la loro chiave primaria, può utilizzare le query JPQL per recuperare le entità seguendo determinati criteri.

Il punto di forza di JPA è che le entità possono essere utilizzate come oggetti regolari da diversi livelli di un'applicazione e essere gestite dal gestore di entità quando è necessario caricare o inserire dati nel database.

```
public interface EntityManager {  
  
    // Factory to create an entity manager, close it and check if it's open  
    EntityManagerFactory getEntityManagerFactory();  
    void close();  
    boolean isOpen();  
  
    // Returns an entity transaction  
    EntityTransaction getTransaction();  
  
    // Persists, merges and removes an entity to/from the database  
    void persist(Object entity);  
    <T> T merge(T entity);  
    void remove(Object entity);  
  
    // Finds an entity based on its primary key (with different lock mechanisms)  
    <T> T find(Class<T> entityClass, Object primaryKey);  
    <T> T find(Class<T> entityClass, Object primaryKey, LockModeType lockMode);  
    <T> T getReference(Class<T> entityClass, Object primaryKey);  
  
    // Locks an entity with the specified lock mode type (optimistic, pessimistic...)  
    void lock(Object entity, LockModeType lockMode);  
  
    // Synchronizes the persistence context to the underlying database  
    void flush();  
    void setFlushMode(FlushModeType flushMode);  
    FlushModeType getFlushMode();  
  
    // Refreshes the state of the entity from the database, overwriting any changes made  
    void refresh(Object entity);  
    void refresh(Object entity, LockModeType lockMode);  
  
    // Clears the persistence context and checks if it contains an entity  
    void clear();  
    void detach(Object entity);  
    boolean contains(Object entity);  
  
    // Sets and gets an entity manager property or hint  
    void setProperty(String propertyName, Object value);  
    Map<String, Object> getProperties();  
  
    // Creates an instance of Query or TypedQuery for executing a JPQL statement  
    Query createQuery(String qlString);  
    <T> TypedQuery<T> createQuery(CriteriaQuery<T> criteriaQuery);  
    <T> TypedQuery<T> createQuery(String qlString, Class<T> resultClass);
```

```

// Creates an instance of Query or TypedQuery for executing a named query
Query createNamedQuery(String name);
<T> TypedQuery<T> createNamedQuery(String name, Class<T> resultClass);

// Creates an instance of Query for executing a native SQL query
Query createNativeQuery(String sqlString);
Query createNativeQuery(String sqlString, Class resultClass);
Query createNativeQuery(String sqlString, String resultSetMapping);

// Creates a StoredProcedureQuery for executing a stored procedure in the database
StoredProcedureQuery createStoredProcedureQuery(String procedureName);
StoredProcedureQuery createNamedStoredProcedureQuery(String name);

// Metamodel and criteria builder for criteria queries (select, update and delete)
CriteriaBuilder getCriteriaBuilder();
Metamodel getMetamodel();
Query createQuery(CriteriaUpdate updateQuery);
Query createQuery(CriteriaDelete deleteQuery);

// Indicates that a JTA transaction is active and joins the persistence context to it
void joinTransaction();
boolean isJoinedToTransaction();

// Return the underlying provider object for the EntityManager
<T> T unwrap(Class<T> cls);
Object getDelegate();

// Returns an entity graph
<T> EntityGraph<T> createEntityGraph(Class<T> rootType);
EntityGraph<?> createEntityGraph(String graphName);
<T> EntityGraph<T> getEntityGraph(String graphName);
<T> List<EntityGraph<? super T>> getEntityGraphs(Class<T> entityClass);
}

```

Un entity manager e una transazione si ottengono nel seguente modo:

```

// Obtains an entity manager and a transaction
EntityManagerFactory emf = Persistence.createEntityManagerFactory("chapter06PU");
EntityManager em = emf.createEntityManager();

// Persists the book to the database
EntityTransaction tx = em.getTransaction();
tx.begin();
em.persist(book);
tx.commit();

```

La creazione di un entity manager dall'applicazione è abbastanza semplice utilizzando una factory, ma ciò che differenzia l'applicazione gestita dal contenitore gestito è il modo in cui viene acquisita la factory. Un ambiente gestito dal contenitore si verifica quando l'applicazione si evolve in un servlet o in un contenitore EJB. In un ambiente Java EE, il modo più comune per acquisire un entity manager è dall'annotazione `@PersistenceContext` o dalla ricerca JNDI. Il componente in esecuzione in un contenitore (Servlet, EJB, servizio Web, ecc.). Non ha bisogno di creare o chiudere l'entity

manager, in quanto il suo ciclo di vita è gestito dal contenitore. Il Listato 6-3 mostra il codice di un bean di sessione stateless in cui iniettiamo un riferimento dell'unità di persistenza chapter06PU.

Listing 6-3. A Stateless EJB Injected with a Reference of an Entity Manager

```
@Stateless
public class BookEJB {

    @PersistenceContext(unitName = "chapter06PU")
    private EntityManager em;

    public void createBook() {

        // Creates an instance of book
        Book book = new Book("H2G2", "The Hitchhiker's Guide to the Galaxy", 12.5F,
                            "1-84023-742-2", 354, false);

        // Persists the book to the database
        em.persist(book);
    }
}
```

Table 6-2. EntityManager Interface Methods to Manipulate Entities

Method	Description
void persist(Object entity)	Makes an instance managed and persistent
<T> T find(Class<T> entityClass, Object primaryKey)	Searches for an entity of the specified class and primary key
<T> T getReference(Class<T> entityClass, Object primaryKey)	Gets an instance, whose state may be lazily fetched
void remove(Object entity)	Removes the entity instance from the persistence context and from the underlying database
Method	Description
<T> T merge(T entity)	Merges the state of the given entity into the current persistence context
void refresh(Object entity)	Refreshes the state of the instance from the database, overwriting changes made to the entity, if any
void flush()	Synchronizes the persistence context to the underlying database
void clear()	Clears the persistence context, causing all managed entities to become detached
void detach(Object entity)	Removes the given entity from the persistence context, causing a managed entity to become detached
boolean contains(Object entity)	Checks whether the instance is a managed entity instance belonging to the current persistence context

Per persistere un elemento sul database si usa il metodo persist() sull'entity manager: em.persist(oggetto) e si conferma il tutto con commit(), l'ordine in cui si effettuano le persist è influente finché non si usa commit(), perché finché la transizione non è confermata i dati restano in memoria e non nel database.

Possiamo ricercare un entità tramite il metodo find():

```
Customer customer = em.find(Customer.class, 1234L)
if (customer!= null) {
    // Process the object
}
```

Possiamo cercare un riferimento dell'entità tramite getReference():

```
try {
    Customer customer = em.getReference(Customer.class, 1234L)
    // Process the object
} catch(EntityNotFoundException ex) {
    // Entity not found
}
```

Per rimuovere un entità si usa il metodo remove(): em.remove(oggetto). Questo può portare ad elementi orfani nel database. Questo può essere risolto aggiungendo nell'annotazione `@OneToOne(cascade = {CascadeType.PERSIST, CascadeType.REMOVE})`.

L'uso del metodo refresh è utilizzato per riprendere il valore originale dal database:

```
Customer customer = em.find(Customer.class, 1234L)
assertEquals(customer.getFirstName(), "Antony");

customer.setFirstName("William");

em.refresh(customer);
assertEquals(customer.getFirstName(), "Antony");
```

Il metodo EntityManager.contains () restituisce un booleano e consente di verificare se una particolare istanze dell'entità è attualmente gestita dal gestore dell'entità all'interno del contesto di persistenza corrente.

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");

tx.begin();
em.persist(customer);
tx.commit();

assertTrue(em.contains(customer));

tx.begin();
em.remove(customer);
tx.commit();

assertFalse(em.contains(customer));
```

Il metodo clear () è semplice: svuota il contesto di persistenza, causando il distacco di tutte le entità gestite. Il metodo detach (Object entity) rimuove l'entità data dal contesto di persistenza.

Il metdo merge() riassegna un oggetto alla persistenza:

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");

tx.begin();
em.persist(customer);
tx.commit();

em.clear();

// Sets a new value to a detached entity
customer.setFirstName("William");

tx.begin();
em.merge(customer);
tx.commit();
```

Se un entità è già gestita dall'entity manager ogni modifica effettuata sull'oggetto si trasmetterà automaticamente nel database:

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");

tx.begin();
em.persist(customer);

customer.setFirstName("Williman");

tx.commit();
```

L'uso di cascade prevede diversi eventi `@OneToOne(cascade = {CascadeType.Events})`:

Table 6-3. Possible Events to Be Cascaded

Cascade Type	Description
PERSIST	Cascades persist operations to the target of the association
REMOVE	Cascades remove operations to the target of the association
MERGE	Cascades merge operations to the target of the association
REFRESH	Cascades refresh operations to the target of the association
DETACH	Cascades detach operations to the target of the association
ALL	Declares that all the previous operations should be cascaded

JPQL

Potrebbe essere necessario recuperare un'entità in base a criteri diversi dall'ID (per nome, ISBN, ecc.) O recuperare un insieme di entità in base a criteri diversi (ad esempio, tutti i clienti residenti negli Stati Uniti). Questa possibilità è inherente ai database relazionali e JPA ha un linguaggio che consente questa interazione: JPQL.

JPQL viene utilizzato per definire ricerche su entità persistenti indipendenti dal database sottostante. JPQL è un linguaggio di query che affonda le sue radici nella sintassi di SQL.

Ma la differenza principale è che in SQL i risultati ottenuti sono sotto forma di righe e colonne (tabelle), mentre JPQL utilizza un'entità o una raccolta di entità. La sintassi JPQL è orientata agli oggetti e quindi più facilmente comprensibile dagli sviluppatori la cui esperienza è limitata ai linguaggi orientati agli oggetti. Gli sviluppatori gestiscono il loro modello di dominio di entità, non una struttura di tabella, utilizzando la notazione a punti (ad esempio, myClass.myAttribute). JPQL utilizza il meccanismo di mappatura per trasformare una query JPQL in un linguaggio comprensibile da un database SQL. La query viene eseguita sul database sottostante con chiamate SQL e JDBC, quindi le istanze delle entità hanno i loro attributi impostati e restituiti all'applicazione. Lo schema di una query JPQL è il seguente:

```
SELECT <select clause>
FROM <from clause>
[WHERE <where clause>]
[ORDER BY <order by clause>]
[GROUP BY <group by clause>]
[HAVING <having clause>]
```

In JPQL esistono diverse tipi di query, e sono le seguenti:

- Query dinamiche: questa è la forma più semplice di query, costituita da nient'altro che una stringa di query JPQL specificata dinamicamente in fase di runtime.
- Query con nome (naming): le query denominate sono statiche e non modificabili.
- Criteria API: JPA 2.0 ha introdotto il concetto di API di query orientata agli oggetti.
- Query native: questo tipo di query è utile per eseguire un'istruzione SQL nativa anziché un'istruzione JPQL.
- Query stored procedure: JPA 2.1 introduce una nuova API per chiamare procedure stored.

Table 6-4. EntityManager Methods for Creating Queries

Method	Description
Query createQuery(String jpqlString)	Creates an instance of Query for executing a JPQL statement for dynamic queries
Query createNamedQuery(String name)	Creates an instance of Query for executing a named query (in JPQL or in native SQL)
Query createNativeQuery(String sqlString)	Creates an instance of Query for executing a native SQL statement
Query createNativeQuery(String sqlString, Class resultClass)	Native query passing the class of the expected results
Query createNativeQuery(String sqlString, String resultSetMapping)	Native query passing a result set mapping
<T> TypedQuery<T> createQuery(CriteriaQuery<T> criteriaQuery)	Creates an instance of TypedQuery for executing a criteria query
<T> TypedQuery<T> createQuery(String jpqlString, Class<T> resultClass)	Typed query passing the class of the expected results
<T> TypedQuery<T> createNamedQuery(String name, Class<T> resultClass)	Typed query passing the class of the expected results
StoredProcedureQuery create.StoredProcedureQuery(String procedureName)	Creates a StoredProcedureQuery for executing a stored procedure in the database
1. StoredProcedureQuery create.StoredProcedureQuery(String procedureName, Class... resultClasses)	Stored procedure query passing classes to which the result sets are to be mapped
2. StoredProcedureQuery create.StoredProcedureQuery(String procedureName, String... resultSetMappings)	Stored procedure query passing the result sets mapping
StoredProcedureQuery create.Named.StoredProcedureQuery(String name)	Creates a query for a named stored procedure

L'interfaccia Query contiene i seguenti metodi:

```
public interface Query {  
  
    // Executes a query and returns a result  
    List getResultList();  
    Object getSingleResult();  
    int executeUpdate();  
  
    // Sets parameters to the query  
    Query setParameter(String name, Object value);  
    Query setParameter(String name, Date value, TemporalType temporalType);  
    Query setParameter(String name, Calendar value, TemporalType temporalType);  
    Query setParameter(int position, Object value);  
    Query setParameter(int position, Date value, TemporalType temporalType);  
    Query setParameter(int position, Calendar value, TemporalType temporalType);  
    <T> Query setParameter(Parameter<T> param, T value);  
    Query setParameter(Parameter<Date> param, Date value, TemporalType temporalType);  
    Query setParameter(Parameter<Calendar> param, Calendar value, TemporalType temporalType);  
  
    // Gets parameters from the query  
    Set<Parameter<?>> getParameters();  
    Parameter<?> getParameter(String name);  
    Parameter<?> getParameter(int position);  
    <T> Parameter<T> getParameter(String name, Class<T> type);  
    <T> Parameter<T> getParameter(int position, Class<T> type);  
    boolean isBound(Parameter<?> param);  
    <T> T getParameterValue(Parameter<T> param);  
    Object getParameterValue(String name);  
    Object getParameterValue(int position);  
  
    // Constrains the number of results returned by a query  
    Query setMaxResults(int maxResult);  
    int getMaxResults();  
    Query setFirstResult(int startPosition);  
    int getFirstResult();  
  
    // Sets and gets query hints  
    Query setHint(String hintName, Object value);  
    Map<String, Object> getHints();  
  
    // Sets the flush mode type to be used for the query execution  
    Query setFlushMode(FlushModeType flushMode);  
    FlushModeType getFlushMode();  
  
    // Sets the lock mode type to be used for the query execution  
    Query setLockMode(LockModeType lockMode);  
    LockModeType getLockMode();  
  
    // Allows access to the provider-specific API  
    <T> T unwrap(Class<T> cls);  
}
```

I metodi più usati sono:

- `getResultSet()`: che restituisce una lista di risultati.
- `GetSingleResult()`: che restituisce un singolo risultato.

Query dinamiche

Le query dinamiche vengono create tramite il metodo `EntityManager.createQuery()`:

```
Query query = em.createQuery("SELECT c FROM Customer c");
List<Customer> customers = query.getResultList();
```

Visto che Query ritorna una lista di oggetti non tipizzati per ritornare una lista di oggetti di tipo `Customer` possiamo usare:

```
TypedQuery<Customer> query = em.createQuery("SELECT c FROM Customer c", Customer.class);
List<Customer> customers = query.getResultList();
```

Si possono anche concatenare stringhe per creare query più complesse:

```
String jpqlQuery = "SELECT c FROM Customer c";
```

```
if (someCriteria)
    jpqlQuery += " WHERE c.firstName = 'Betty'";
query = em.createQuery(jpqlQuery);
List<Customer> customers = query.getResultList();
```

Si possono anche settare dei parametri all'interno della query:

```
query = em.createQuery("SELECT c FROM Customer c where c.firstName = :fname");
query.setParameter("fname", "Betty");
List<Customer> customers = query.getResultList();
```

Infine con `setMaxResults(numeroMassimo)` possiamo dire quanti risultati vogliamo dalla restituzione della query.

Named query

Le named query sono statiche e non modificabili, si inseriscono nella classe con la notazione `@Entity`:

```
@Entity
@NamedQueries({
    @NamedQuery(name = "findAll", query="select c from Customer c"),
    @NamedQuery(name = "findVincent", query="select c from Customer c where c.firstName = 'Vincent'"),
    @NamedQuery(name = "findWithParam", query="select c from Customer c where c.firstName = :fname")
})
```

L'uso nel main è il seguente:

```
Query query = em.createNamedQuery("findWithParam");
query.setParameter("fname", "Vincent");
query.setMaxResults(3);
List<Customer> customers = query.getResultList();
```

Possono anche essere settati dei parametri e il numero massimo di elementi desiderati:

```
Query query = em.createNamedQuery("findWithParam").setParameter("fname", "Vincent").setMaxResults(3);
```

Criteria API (or Object-Oriented Queries)

JPA 2.0 ha creato una nuova API, denominata Criteria API e definita nel pacchetto `javax.persistence.criteria`. Permette di scrivere qualsiasi query in modo orientato agli oggetti e

sintatticamente corretto. La maggior parte degli errori che uno sviluppatore potrebbe fare scrivere una dichiarazione si trovano in fase di compilazione, non in fase di runtime. L'idea è che tutte le parole chiave JPQL (SELECT, UPDATE, DELETE, WHERE, LIKE, GROUP BY ...) sono definite in questa API. In altre parole, l'API Criteria supporta tutto ciò che JPQL può fare, ma con una sintassi basata su oggetti.

Esempio di Criteria API:

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = builder.createQuery(Customer.class);
Root<Customer> c = criteriaQuery.from(Customer.class);
criteriaQuery.select(c).where(builder.equal(c.get("firstName"), "Vincent"));
Query query = em.createQuery(criteriaQuery).getResultList();
List<Customer> customers = query.getResultList();
```

Per garantire la sicurezza del tipo, JPA 2.1 può generare una classe metamodel statica per ogni entità. La convenzione è che ogni entità X avrà una classe di metadati chiamata X_ (con un carattere di sottolineatura).

```
@Generated("EclipseLink")
@StaticMetamodel(Customer.class)
public class Customer_ {

    public static volatile SingularAttribute<Customer, Long> id;
    public static volatile SingularAttribute<Customer, String> firstName;
    public static volatile SingularAttribute<Customer, String> lastName;
    public static volatile SingularAttribute<Customer, Integer> age;
    public static volatile SingularAttribute<Customer, String> email;
    public static volatile SingularAttribute<Customer, Address> address;
}
```

Nella classe di metadati statici, ogni attributo dell'entità Customer è definito da una sottoclasse di javax.persistence.metamodel.Attribute (CollectionAttribute, ListAttribute, MapAttribute, SetAttribute o SingularAttribute). Ciascuno di questi attributi ha uso generico ed è fortemente tipizzato (ad es. SingularAttribute <Cliente, Intero>, età). Il Listato 6-28 mostra lo stesso identico codice del Listato 6-25 ma rivisitato con la classe statica metamodel (il c.get ("age") è convertito in c.get (Customer_.age)). Un altro vantaggio della sicurezza del tipo è che il metamodello definisce l'attributo età come un numero intero, quindi non è necessario eseguire il cast dell'attributo in un numero intero usando come (Integer.class).

Listing 6-28. A Type-Safe Criteria Query Selecting All the Customers Older Than 40

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = builder.createQuery(Customer.class);
Root<Customer> c = criteriaQuery.from(Customer.class);
criteriaQuery.select(c).where(builder.greaterThan(c.get(Customer_.age), 40));
Query query = em.createQuery(criteriaQuery).getResultList();
List<Customer> customers = query.getResultList();
```

Native Queries

JPQL ha una sintassi molto ricca che ti consente di gestire entità in qualsiasi forma e garantisce la portabilità tra i database. JPA consente di utilizzare funzionalità specifiche di un database utilizzando query native. Le query native eseguono un'istruzione SQL nativa (SELECT, UPDATE o DELETE) come parametro e restituiscono un'istanza Query per l'esecuzione di tale istruzione SQL. Tuttavia, non è previsto che le query native siano trasferibili tra i database. Se il codice non è portatile, perché non utilizzare le chiamate JDBC? La ragione principale per utilizzare le query native JPA piuttosto che le chiamate JDBC è perché il risultato della query verrà automaticamente convertito in entità. Se si desidera recuperare tutte le entità Customer dal database utilizzando SQL, è necessario utilizzare il metodo EntityManager.createNativeQuery() che ha come parametri la query SQL e la classe entità a cui il risultato deve essere associato.

```
Query query = em.createNativeQuery("SELECT * FROM t_customer", Customer.class);
```

```
List<Customer> customers = query.getResultList();
```

Nella classe avremo:

```
@Entity  
@NamedNativeQuery(name = "findAll", query="select * from t_customer")  
@Table(name = "t_customer")  
public class Customer {...}
```

Stored Procedure Queries

Finora tutte le diverse query (JPQL o SQL) hanno lo stesso scopo: inviare una query dall'applicazione al database che la eseguirà e restituirà un risultato. Le stored procedure sono diverse nel senso che sono effettivamente memorizzate nel database stesso ed eseguite all'interno di quest'ultimo.

Una stored procedure è una subroutine disponibile per le applicazioni che accedono a un database relazionale. L'utilizzo tipico potrebbe essere un'elaborazione ampia o complessa che richiede l'esecuzione di numerose istruzioni SQL o un'attività ripetitiva ad alta intensità di dati. Le stored procedure sono solitamente scritte in un linguaggio proprietario vicino a SQL e quindi non facilmente trasportabili tra i fornitori di database. Ma memorizzare il codice all'interno del database anche in modo non proporzionale offre numerosi vantaggi:

- Prestazioni migliori grazie alla precompilazione della stored procedure e al riutilizzo del piano di esecuzione
- Mantenere statistiche sul codice per mantenerlo ottimizzato
- Ridurre la quantità di dati trasmessi su una rete mantenendo il codice sul server
- Modifica del codice in una posizione centrale senza la replica in diversi programmi diversi
- Procedure memorizzate, che possono essere utilizzate da più programmi scritti in linguaggi diversi (non solo Java)
- Nascondere i dati grezzi consentendo solo alle stored procedure di accedere ai dati
- Miglioramento dei controlli di sicurezza concedendo agli utenti il permesso di eseguire una stored procedure indipendentemente dalle autorizzazioni della tabella sottostante.

Listing 6-29. Abstract of a Stored Procedure Archiving Books

```
CREATE PROCEDURE sp_archive_books @archiveDate DATE, @warehouseCode VARCHAR AS  
    UPDATE T_Inventory  
        SET Number_Of_Books_Left = 1  
        WHERE Archive_Date < @archiveDate AND Warehouse_Code = @warehouseCode;  
  
    UPDATE T_Transport  
        SET Warehouse_To_Take_Books_From = @warehouseCode;  
END
```

È usata nel seguente modo:

```
@Entity  
@NamedStoredProcedureQuery(name = "archiveOldBooks", procedureName = "sp_archive_books",  
    parameters = {  
        @StoredProcedureParameter(name = "archiveDate", mode = IN, type = Date.class),  
        @StoredProcedureParameter(name = "warehouse", mode = IN, type = String.class)  
    }  
)
```

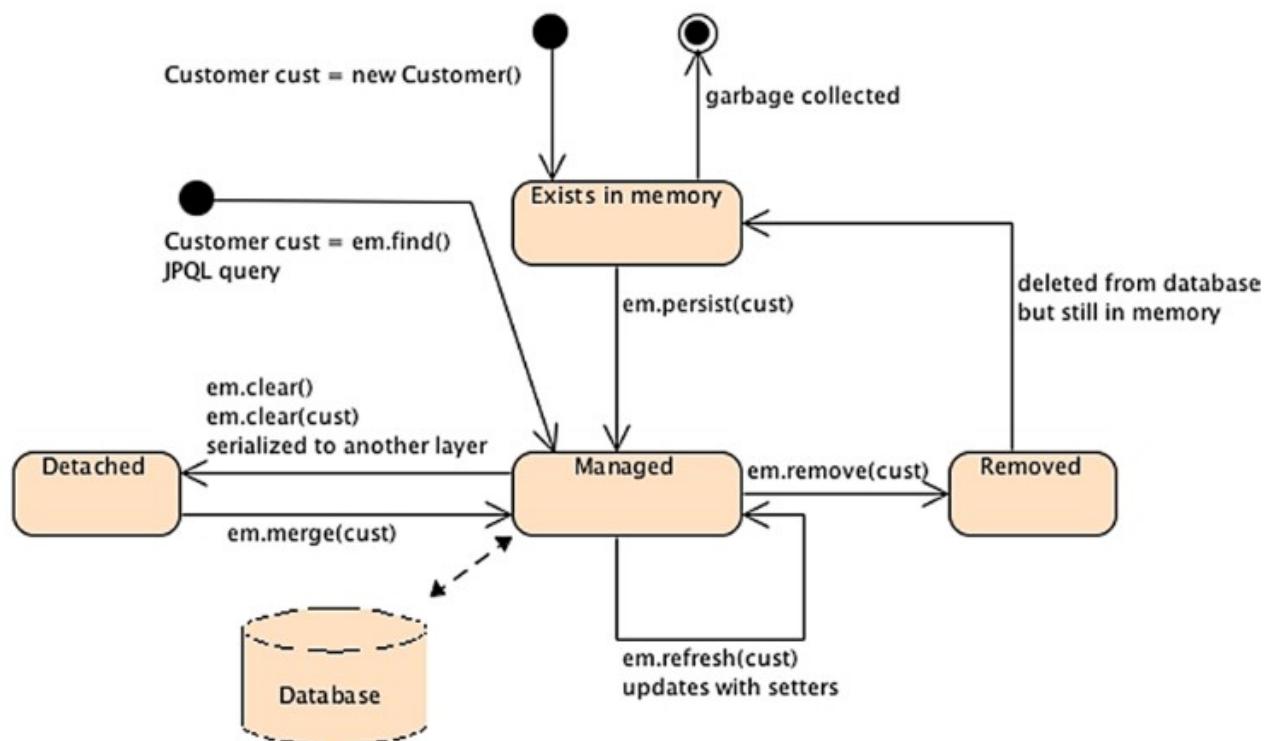
Utilizzo:

```
StoredProceduredQuery query = em.createNamedStoredProcedureQuery("archiveOldBooks");  
query.setParameter("archiveDate", new Date());  
query.setParameter("maxBookArchived", 1000);  
query.execute();
```

```
StoredProceduredQuery query = em.createStoredProcedureQuery("sp_archive_old_books");  
query.registerStoredProcedureParameter("archiveDate", Date.class, ParameterMode.IN);  
query.registerStoredProcedureParameter("maxBookArchived", Integer.class, ParameterMode.IN);  
  
query.setParameter("archiveDate", new Date());  
query.setParameter("maxBookArchived", 1000);  
query.execute();
```

Entity life cycle

Quando un'entità viene istanziata (con il nuovo operatore), viene vista come un normale POJO dalla JVM (cioè, separata) e può essere utilizzata come oggetto regolare dall'applicazione. Quindi, quando l'entità viene mantenuta dal gestore dell'entità, si dice che è gestita. Quando un'entità viene gestita, il gestore dell'entità sincronizzerà automaticamente il valore dei suoi attributi con il database sottostante (ad esempio, se si modifica il valore di un attributo utilizzando un metodo set mentre l'entità è gestita, questo nuovo valore verrà automaticamente sincronizzato con il database).



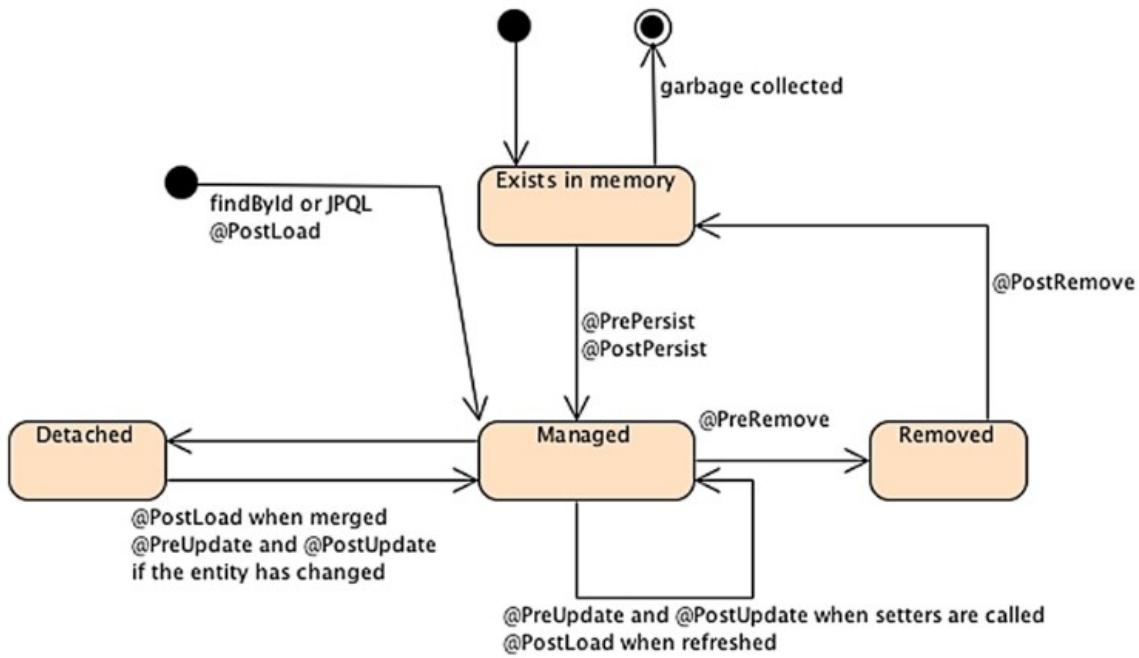
Per creare un'istanza dell'entità Customer, si utilizza l'operatore new. Questo oggetto esiste in memoria, sebbene JPA non ne sappia nulla. Se non si utilizza l'oggetto, esso sarà eliminato e quella sarà la fine del suo ciclo di vita. Quello che si può fare è mantenere un'istanza del Customer con il metodo EntityManager.persist (). In quel momento, l'entità viene gestita e il suo stato è sincronizzato con il database. Durante questo stato gestito, è possibile aggiornare gli attributi utilizzando i metodi setter (ad esempio, customer.setFirstName ()) o aggiornare il contenuto con un metodo EntityManager.refresh (). Tutte queste modifiche verranno sincronizzate tra l'entità e il database. Durante questo stato, se si chiama il metodo EntityManager.contains (customer), verrà restituito true poiché il cliente è contenuto nel contesto di persistenza (ad esempio, gestito). Un altro modo per gestire un'entità è quando viene caricato dal database. Quando si utilizza il metodo EntityManager.find () o si crea una query JPQL per recuperare un elenco di entità, tutte vengono gestite automaticamente e si può iniziare ad aggiornare o rimuovere i relativi attributi. Quando lo stato è gestito, è possibile chiamare il metodo EntityManager.remove () e l'entità viene eliminata dal database e non sarà più gestita. Ma l'oggetto Java continua a vivere nella memoria, e puoi ancora usarlo finché il garbage collector non se ne libera. Ora guardiamo allo stato di detach. Oltre ai metodi clear() e detach() c'è anche un altro modo, più sottile, per staccare un'entità: quando è serializzata. In molti esempi le entità non implementano nulla, ma, se hanno bisogno di attraversare una rete per essere richiamate in remoto o cross layer per essere visualizzate in un livello di presentazione, devono implementare l'interfaccia java.io.Serializable. Questa non è una restrizione JPA ma una restrizione Java. Quando un'entità gestita viene serializzata, attraversa la rete e viene deserializzata, viene vista come un oggetto detach. Per ricollegare un'entità, è necessario chiamare il metodo EntityManager.merge (). Un caso di uso comune è quando si utilizza un'entità in una pagina JSF. Diciamo che un'entità Cliente viene visualizzata in un modulo su una pagina JSF remota per essere aggiornata. Essendo remota, l'entità deve essere serializzata sul lato server prima di essere inviata al livello di presentazione. In quel momento, l'entità viene automaticamente staccata. Una volta visualizzati, se i dati vengono modificati e devono essere aggiornati, il modulo viene inviato e l'entità viene rinviata al server, deserializzata e deve essere unita per essere nuovamente collegata.

I metodi di callback e i listener consentono di aggiungere la propria logica aziendale quando determinati eventi del ciclo di vita si verificano su un'entità o in generale ogni volta che si verifica un evento del ciclo di vita su qualsiasi entità.

Callbacks annotation

Table 6-7. Life-Cycle Callback Annotations

Annotation	Description
@PrePersist	Marks a method to be invoked before EntityManager.persist() is executed.
@PostPersist	Marks a method to be invoked after the entity has been persisted. If the entity autogenerated its primary key (with @GeneratedValue), the value is available in the method.
@PreUpdate	Marks a method to be invoked before a database update operation is performed (calling the entity setters or the EntityManager.merge() method).
@PostUpdate	Marks a method to be invoked after a database update operation is performed.
@PreRemove	Marks a method to be invoked before EntityManager.remove() is executed.
@PostRemove	Marks a method to be invoked after the entity has been removed.
@PostLoad	Marks a method to be invoked after an entity is loaded (with a JPQL query or an EntityManager.find()) or refreshed from the underlying database. There is no @PreLoad annotation, as it doesn't make sense to preload data on an entity that is not built yet.



Prima di inserire un'entità nel database, il gestore dell'entità chiama il metodo annotato con `@PrePersist`. Se l'inserimento non genera un'eccezione, l'entità viene mantenuta, la sua identità viene inizializzata e viene quindi richiamato il metodo annotato con `@PostPersist`. Questo è lo stesso comportamento per gli aggiornamenti (`@PreUpdate`, `@PostUpdate`) e cancella (`@PreRemove`, `@PostRemove`). Un metodo annotato con `@PostLoad` viene chiamato quando un'entità viene caricata dal database (tramite una query `EntityManager.find()` o JPQL). Quando l'entità è scollegata e deve essere unita, il gestore dell'entità deve prima verificare se ci sono differenze con il database.

`(@PostLoad)` e, in tal caso, aggiornare i dati (`@PreUpdate`, `@PostUpdate`). Come appare nel codice? Le entità possono avere non solo attributi, costruttori, getter e setter ma anche la logica aziendale utilizzata per convalidare il loro stato o calcolare alcuni dei loro attributi. Questi possono essere costituiti da normali metodi Java che vengono richiamati da altre classi o annotazioni di callback (anche denominate metodi di callback), come mostrato nel Listato 6-38. Il gestore di entità li richiama automaticamente in base all'evento attivato.

Esempio: →

```

@PrePersist
@PreUpdate
private void validate() {
    if (firstName == null || "".equals(firstName))
        throw new IllegalArgumentException("Invalid first name");
    if (lastName == null || "".equals(lastName))
        throw new IllegalArgumentException("Invalid last name");
}

@PostLoad
@PostPersist
@PostUpdate
public void calculateAge() {
    if (dateOfBirth == null) {
        age = null;
        return;
    }

    Calendar birth = new GregorianCalendar();
    birth.setTime(dateOfBirth);
    Calendar now = new GregorianCalendar();
    now.setTime(new Date());
    int adjust = 0;
    if (now.get(DAY_OF_YEAR) - birth.get(DAY_OF_YEAR) < 0) {
        adjust = -1;
    }
    age = now.get(YEAR) - birth.get(YEAR) + adjust;
}
  
```

Listeners

I metodi di callback in un'entità funzionano bene quando si dispone di una logica aziendale correlata solo a quell'entità. I listener di entità vengono utilizzati per estrarre la logica di business in una classe separata e condividerla tra altre entità. Un listener di entità è solo un POJO su cui è possibile definire uno o più metodi di callback del ciclo di vita. Per registrare un listener, l'entità deve utilizzare l'annotazione `@EntityListeners`.

Usando l'esempio del cliente, estrapoliamo i metodi `calculateAge()` e `validate()` per separare le classi listener, `AgeCalculationListener` e `DataValidationListener`.

Listing 6-39. A Listener Calculating the Customer's Age

```
public class AgeCalculationListener {

    @PostLoad
    @PostPersist
    @PostUpdate
    public void calculateAge(Customer customer) {
        if (customer.getDateOfBirth() == null) {
            customer.setAge(null);
            return;
        }

        Calendar birth = new GregorianCalendar();
        birth.setTime(customer.getDateOfBirth());
        Calendar now = new GregorianCalendar();
        now.setTime(new Date());
        int adjust = 0;      if (now.get(DAY_OF_YEAR) - birth.get(DAY_OF_YEAR) < 0) {
            adjust = -1;
        }
        customer.setAge(now.get(YEAR) - birth.get(YEAR) + adjust);
    }
}
```

Il metodo prende ora un oggetto di tipo `Customer`.

Listing 6-40. A Listener Validating the Customer's Attributes

```
public class DataValidationListener {

    @PrePersist
    @PreUpdate
    private void validate(Customer customer) {
        if (customer.getFirstName() == null || "".equals(customer.getFirstName()))
            throw new IllegalArgumentException("Invalid first name");
        if (customer.getLastName() == null || "".equals(customer.getLastName()))
            throw new IllegalArgumentException("Invalid last name");
    }
}
```

Solo una semplice regola si applica a una classe listener. Il primo è che la classe deve avere un costruttore pubblico senza argomenti.

Quando invochi il metodo di callback su un listener, il metodo deve avere accesso allo stato dell'entità (ad esempio, nome e cognome del cliente, che devono essere convalidati). I metodi devono avere un parametro di un tipo che è compatibile con il tipo di entità, poiché l'entità relativa all'evento viene passata nel callback. Un metodo di callback definito su un'entità ha la seguente firma senza parametro:

- void <METHOD>();
- void <METHOD>(Object anyEntity);
- void <METHOD>(Customer customerOrSubclasses);

Possiamo creare una classe listener Debug applicabile a qualunque entità per verificare il loro stato durante le chiamate di callback:

Listing 6-42. A Debug Listener Usable by Any Entity

```
public class DebugListener {

    @PrePersist
    void prePersist(Object object) {
        System.out.println("prePersist");
    }

    @PreUpdate
    void preUpdate(Object object) {
        System.out.println("preUpdate");
    }

    @PreRemove
    void preRemove(Object object) {
        System.out.println("preRemove");
    }
}
```

Per poter utilizzare questo listener lo rendiamo di default per ogni entità e quindi andremo a modificare il file persistence.xml:

```
<persistence-unit-metadata>
    <persistence-unit-defaults>
        <entity-listeners>
            <entity-listener class="org.agoncal.book.javaee7.chapter06.DebugListener"/>
        </entity-listeners>
    </persistence-unit-defaults>
</persistence-unit-metadata>
```

Per escludere un default listener da un entità basta aggiungere la notazione: @ExcludeDefaultListeners

Enterprise Java Bean

Tipi di EJB

I bean di sessione sono ideali per l'implementazione della business logic, processi e workflow. E poiché le applicazioni aziendali possono essere complesse, la piattaforma Java EE definisce diversi tipi di EJB. Un bean di sessione può avere i seguenti tratti:

- Stateless: il bean di sessione non contiene nessuno stato di conversazione tra i metodi e qualsiasi istanza può essere utilizzata per qualsiasi client. È utilizzato per gestire attività che possono essere concluse con una singola chiamata di metodo.
- Stateful: il bean di sessione contiene lo stato della conversazione, che deve essere mantenuto tra i metodi per un singolo utente. È utile per le attività che devono essere eseguite in più passaggi.
- Singleton: un singolo bean di sessione è condiviso tra i client e supporta l'accesso simultaneo. Il contenitore si assicurerà che esista solo un'istanza per l'intera applicazione.

I tre tipi di bean di sessione hanno tutti le loro caratteristiche specifiche, ovviamente, ma hanno anche molto in comune.

Prima di tutto, hanno lo stesso modello di programmazione. Come vedremo più avanti, un bean di sessione può avere un'interfaccia locale e / o remota, o nessuna interfaccia. I bean di sessione sono componenti gestiti dal contenitore, quindi devono essere impacchettati in un archivio (jar, war o ear file) e distribuiti in un contenitore.

Servizi offerti dal container

Indipendentemente dal fatto che il contenitore sia incorporato o eseguito in un processo separato, fornisce servizi di base comuni a molte applicazioni aziendali come le seguenti:

- Comunicazione client remota: senza scrivere codice complesso, un client EJB (un altro EJB, un'interfaccia utente, un processo batch, ecc.) Può invocare i metodi da remoto tramite protocolli standard.
- Iniezione delle dipendenze: il contenitore può iniettare diverse risorse in un EJB (destinazioni JMS e fabbriche, origini dati, altri EJB, variabili di ambiente, ecc.) Così come qualsiasi POJO grazie al CDI.
- Gestione dello stato: per i bean di sessione con stato, il contenitore gestisce il loro stato in modo trasparente. È possibile mantenere lo stato per un determinato client, come se si stesse sviluppando un'applicazione desktop.
- Pooling: per bean senza stato e MDB, il contenitore crea un pool di istanze che possono essere condivise da più client. Una volta invocato, un EJB ritorna al pool per essere riutilizzato invece di essere distrutto.
- Ciclo di vita dei componenti: il contenitore è responsabile della gestione del ciclo di vita di ciascun componente.
- Messaggistica: il contenitore consente agli MDB di ascoltare le destinazioni e consumare i messaggi senza troppi impianti idraulici JMS.
- Gestione delle transazioni: con la gestione dichiarativa delle transazioni, un EJB può utilizzare annotazioni per informare il contenitore sul criterio di transazione che dovrebbe utilizzare. Il contenitore si occupa del commit o del rollback.
- Sicurezza: il controllo di accesso a livello di classe o metodo può essere specificato sugli EJB per imporre l'autorizzazione dell'utente e del ruolo.
- Supporto per la concorrenza: tranne che per i singleton, dove è necessaria una dichiarazione di concorrenza, tutti gli altri tipi di bean sono di natura thread-safe. È possibile sviluppare applicazioni ad alte prestazioni senza preoccuparsi dei problemi relativi ai thread.
- Interceptor: le preoccupazioni trasversali possono essere messe in intercettori, che saranno invocati automaticamente dal container.
- Richiamo del metodo asincrono: da EJB 3.1 è ora possibile effettuare chiamate asincrone senza coinvolgere la messaggistica.

Esempio di codice di Enterprise Java Bean:

```
@Stateless  
public class BookEJB {  
    @PersistenceContext(unitName = "chapter07PU")  
    private EntityManager em;  
    public Book findBookById(Long id) {  
        return em.find(Book.class, id);  
    }  
    public Book createBook(Book book) {  
        em.persist(book);  
        return book;  
    }  
}
```

Bean class

Una classe bean di sessione è una classe Java standard che implementa la logica aziendale. I requisiti per sviluppare una classe bean di sessione sono i seguenti:

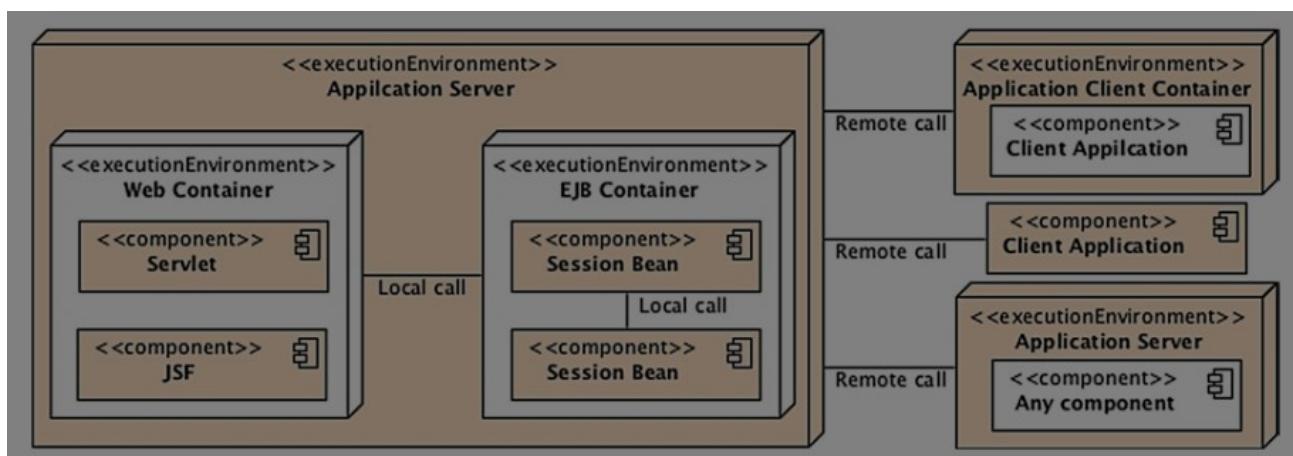
- La classe deve essere annotata con `@Stateless`, `@Stateful`, `@Singleton` o l'equivalente XML in un descrittore di deployment.
- Deve implementare i metodi delle sue interfacce, se presenti.
- La classe deve essere definita come pubblica e non deve essere definitiva o astratta.
- La classe deve avere un costruttore no-arg pubblico che il contenitore utilizzerà per creare istanze.
- La classe non deve definire il metodo `finalize()`.
- I nomi dei metodi commerciali non devono iniziare con `ejb` e non possono essere definitivi o statici.
- L'argomento e il valore di ritorno di un metodo remoto devono essere tipi RMI legali.

Remote, Local, and No-Interface Views

A seconda di dove un client richiama un bean di sessione, la classe bean dovrà implementare interfacce remote o locali o nessuna interfaccia. Se la tua architettura ha client che risiedono all'esterno dell'istanza JVM del contenitore EJB, devono utilizzare un'interfaccia remota.

È possibile utilizzare la chiamata locale quando il bean e il client sono in esecuzione nella stessa JVM. Può trattarsi di un EJB che richiama un altro bean o un componente Web (Servlet, JSF) in esecuzione in un contenitore Web nella stessa JVM.

È inoltre possibile che l'applicazione utilizzi sia le chiamate remote che quelle locali sullo stesso bean di sessione.



Un bean di sessione può implementare diverse interfacce o nessuna. Un'interfaccia aziendale è un'interfaccia Java standard che non estende alcuna interfaccia specifica per EJB. Come qualsiasi altra interfaccia Java, le interfacce aziendali definiscono un elenco di metodi che saranno disponibili per l'applicazione client. Possono usare le seguenti annotazioni:

- `@Remote`: indica un'interfaccia aziendale remota. I parametri del metodo vengono passati per valore e devono essere serializzabili come parte del protocollo RMI.
- `@Local`: indica un'interfaccia aziendale locale. I parametri del metodo vengono passati per riferimento dal client al bean.

Non è possibile contrassegnare la stessa interfaccia con più di un'annotazione. I bean di sessione che hai visto finora in questo capitolo non hanno interfaccia. La vista senza interfaccia è una variante della vista locale che espone localmente tutti i metodi commerciali pubblici della classe bean senza l'uso di un'interfaccia di business separata.

```
@Local
public interface ItemLocal {
    List<Book> findBooks();
    List<CD> findCDs();
}

@Remote
public interface ItemRemote {
    List<Book> findBooks();
    List<CD> findCDs();
    Book createBook(Book book);
    CD createCD(CD cd);
}

@Stateless
public class ItemEJB implements ItemLocal, ItemRemote {
    // ...
}
```

Listing 7-3. A Bean Class Defining a Remote, Local and No Interface

```
public interface ItemLocal {
    List<Book> findBooks();
    List<CD> findCDs();
}

public interface ItemRemote {
    List<Book> findBooks();
    List<CD> findCDs();
    Book createBook(Book book);
    CD createCD(CD cd);
}

@Stateless
@Remote(ItemRemote.class)
@Local(ItemLocal.class)
@LocalBean
public class ItemEJB implements ItemLocal, ItemRemote {
    // ...
}
```

JNDI Name

La specifica Java EE definisce i nomi JNDI portatili con la seguente sintassi:

java:<scope>/[<app-name>]/<module-name>/<bean-name>[!<fully-qualified-interface-name>]

Ogni parte del nome JNDI ha il suo significato:

- <scope> definisce una serie di spazi dei nomi standard che si associano ai vari ambiti di un'applicazione Java EE:
 - global: java: prefisso globale consente a un componente in esecuzione all'esterno di un'applicazione Java EE di accedere a uno spazio dei nomi globale.
 - app: il prefisso java: app consente a un componente in esecuzione all'interno di un'applicazione Java EE di accedere a uno spazio dei nomi specifico dell'applicazione.
 - module: il prefisso java: module consente a un componente in esecuzione all'interno di un'applicazione Java EE di accedere a uno spazio dei nomi specifico del modulo.
 - comp: il prefisso java: comp è uno spazio dei nomi specifico del componente privato e non è accessibile da altri componenti.
- <nome-app> è richiesto solo se il bean di sessione è inserito in un file ear o war. In questo caso, il <nome-app> si imposta automaticamente sul nome del file ear o war (senza estensione .ear o .war).
- <nome-modulo> è il nome del modulo in cui è inserito il bean di sessione. Può essere un modulo EJB in un file jar autonomo o un modulo Web in un file war. <Nome-modulo> imposta automaticamente il nome base dell'archivio senza estensione file.
- <nome-bean> è il nome del bean di sessione.

- <nome-dell'interfaccia completo> è il nome completo di ciascuna interfaccia aziendale definita. Per la vista senza interfaccia, il nome può essere il nome completo della classe del bean.

Listing 7-5. A Stateless Session Bean Implementing Several Interfaces

```
package org.agoncal.book.javaee7;
@Stateless
@Remote(ItemRemote.class)
@Local(ItemLocal.class)
@LocalBean
public class ItemEJB implements ItemLocal, ItemRemote {
    // ...
}
```

Una volta implementato, il contenitore creerà tre nomi JNDI in modo che un componente esterno sia in grado di accedere a ItemEJB utilizzando i seguenti nomi JNDI globali:

```
java:global/cdbookstore/ItemEJB!org.agoncal.book.javaee7.ItemRemote
java:global/cdbookstore/ItemEJB!org.agoncal.book.javaee7.ItemLocal
java:global/cdbookstore/ItemEJB!org.agoncal.book.javaee7.ItemEJB
```

Stateless Bean

Nelle applicazioni Java EE, i bean senza stato sono i componenti di sessione più popolari. Sono semplici, potenti ed efficienti e rispondono al compito comune di eseguire l'elaborazione aziendale senza stato. Cosa significa stateless? Significa che un'attività deve essere completata in una singola chiamata di metodo.

I servizi stateless sono ideali quando è necessario implementare un'attività che può essere conclusa con una singola chiamata di metodo (passando tutti i parametri necessari). I servizi stateless sono indipendenti, sono autonomi e non richiedono informazioni o stati da una richiesta all'altra.

`@Stateless`

```
public class ItemEJB {
    @PersistenceContext(unitName = "chapter07PU")
    private EntityManager em;
    public List<Book> findBooks() {
        TypedQuery<Book> query = em.createNamedQuery(Book.FIND_ALL, Book.class);
        return query.getResultList();
    }
    public List<CD> findCDs() {
        TypedQuery<CD> query = em.createNamedQuery(CD.FIND_ALL, CD.class);
        return query.getResultList();
    }
    public Book createBook(Book book) {
        em.persist(book);
        return book;
    }
    public CD createCD(CD cd) {
        em.persist(cd);
        return cd;
    }
}
```

Listing 7-6. `@Stateless` Annotation API

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface Stateless {
    String name() default "";
    String mappedName() default "";
    String description() default "";
}
```

Stateful Bean

I bean senza stato forniscono metodi di business ai loro clienti ma non mantengono uno stato di conversazione con loro. I bean di sessione con stato, d'altra parte, preservano lo stato della conversazione. Sono utili per le attività che devono essere eseguite in diversi passaggi, ognuno dei quali si basa sullo stato mantenuto in un passaggio precedente. Prendiamo l'esempio di un carrello della spesa in un sito Web di e-commerce.

La correlazione uno-a-uno ha un prezzo perché, come avrete intuito, se avete un milione di clienti, avrete in memoria un milione di bean con stato. Per evitare un ingombro di memoria così grande, il contenitore cancella temporaneamente i bean con stato dalla memoria prima che la successiva richiesta dal client li riporti. Questa tecnica è chiamata passivation e attivazione. Passivation è il processo di rimozione di un'istanza dalla memoria e salvataggio in una posizione persistente (un file su un disco, un database, ecc.). Ti aiuta a liberare memoria e liberare risorse (un database o connessioni JMS, ecc.).

L'attivazione è il processo inverso per ripristinare lo stato e applicarlo a un'istanza. La passivation e l'attivazione vengono eseguite automaticamente dal contenitore; non dovresti preoccuparti di farlo da solo, dato che si tratta di un servizio container. Quello di cui dovresti preoccuparti è liberare qualsiasi risorsa (ad es., Connessione al database, connessione alle factory JMS, ecc.) Prima che il bean sia passivato.

Esempio:

```
@Stateful
@StatefulTimeout(value = 20, unit = TimeUnit.SECONDS)
public class ShoppingCartEJB {
    private List<Item> cartItems = new ArrayList<>();
    public void addItem(Item item) {
        if (!cartItems.contains(item))
            cartItems.add(item);
    }
    public void removeItem(Item item) {
        if (cartItems.contains(item))
            cartItems.remove(item);
    }
    public Integer getNumberOfItems() {
        if (cartItems == null || cartItems.isEmpty())
            return 0;
        return cartItems.size();
    }
    public Float getTotal() {
        if (cartItems == null || cartItems.isEmpty())
            return 0f;
        Float total = 0f;
        for (Item cartItem : cartItems) {
            total += (cartItem.getPrice());
        }
        return total;
    }
    public void empty() {
        cartItems.clear();
    }
    @Remove
    public void checkout() {
        // Do some business logic
        cartItems.clear();
    }
}
```

Singleton

Un bean singleton è un bean di sessione che viene istanziato una volta per applicazione. Implementa il pattern Singleton. Un singleton assicura che esiste un'unica istanza di una classe nell'intera applicazione e fornisce un punto globale per accedervi. Esistono molte situazioni che richiedono gli oggetti Singleton, ovvero dove l'applicazione richiede solo un'istanza di un oggetto: un mouse, un gestore finestre, uno spooler della stampante, un file system e così via.

Un altro caso di uso comune è un sistema di memorizzazione nella cache in cui l'intera applicazione condivide una singola cache (ad esempio una Hashmap) per memorizzare oggetti.

@Singleton

```
public class CacheEJB {  
    private Map<Long, Object> cache = new HashMap<>();  
    public void addToCache(Long id, Object object) {  
        if (!cache.containsKey(id))  
            cache.put(id, object);  
    }  
    public void removeFromCache(Long id) {  
        if (cache.containsKey(id))  
            cache.remove(id);  
    }  
    public Object getFromCache(Long id) {  
        if (cache.containsKey(id))  
            return cache.get(id);  
        else  
            return null;  
    }  
}
```

Container-Managed Concurrency

Con CMC il contenitore è responsabile del controllo dell'accesso simultaneo all'istanza del bean singleton. È quindi possibile utilizzare l'annotazione @Lock per specificare in che modo il contenitore deve gestire la concorrenza quando un client richiama un metodo. L'annotazione può assumere il valore READ (condiviso) o WRITE (esclusivo).

- @Lock (LockType.WRITE): un metodo associato a un blocco esclusivo non consentirà invocazioni simultanee fino al completamento dell'elaborazione del metodo. Ad esempio, se un client C1 richiama un metodo con un blocco esclusivo, il client C2 non sarà in grado di richiamare il metodo fino al termine di C1.
- @Lock (LockType.READ): un metodo associato a un blocco condiviso consentirà un numero qualsiasi di altre chiamate simultanee all'istanza del bean. Ad esempio, due client, C1 e C2, possono accedere simultaneamente a un metodo con un blocco condiviso.

L'annotazione @Lock può essere specificata sulla classe, i metodi o entrambi. Specificare sulla classe significa che si applica a tutti i metodi. Se non si specifica l'attributo di blocco della concorrenza, si presume che sia @Lock (WRITE) per impostazione predefinita.

```
@Singleton  
@Lock(LockType.WRITE)  
@AccessTimeout(value = 20, unit = TimeUnit.SECONDS)  
public class CacheEJB {  
    private Map<Long, Object> cache = new HashMap<>();  
    public void addToCache(Long id, Object object) {  
        if (!cache.containsKey(id))  
            cache.put(id, object);  
    }
```

```

public void removeFromCache(Long id) {
    if (cache.containsKey(id))
        cache.remove(id);
}
@Lock(LockType.READ)
public Object getFromCache(Long id) {
    if (cache.containsKey(id))
        return cache.get(id);
    else
        return null;
}
}

```

Invoking Enterprise Java Beans

Invoking with Injection

Java EE utilizza diverse annotazioni per inserire riferimenti di risorse (@Resource), gestori di entità (@PersistenceContext), servizi Web (@WebServiceRef) e così via. Ma l'annotazione @javax.ejb.EJB è specificamente intesa per l'iniezione di riferimenti di bean di sessione nel codice client. L'iniezione delle dipendenze è possibile solo negli ambienti gestiti come contenitori EJB, contenitori Web e contenitori client-applicazione. Se EJB ha delle interfacce bisogna fare l'injection anche di quelle.

```

@Stateless
public class ItemEJB {...}
// Client code injecting a reference to the EJB
@EJB ItemEJB itemEJB;

```

Invoking with CDI

Un @EJB può essere iniettato come CDI utilizzando l'annotazione @Inject. Può anche essere chiamato con il suo nome JNDI, ma visto che si fa l'injection ad una stringa questo comporta l'uso di @Producers:

```
@Produces @EJB(lookup = "java:global/classes/ItemEJB") ItemRemote itemEJBRMote;
```

Invoking with JNDI

I bean di sessione possono anche essere cercati usando JNDI. JNDI viene principalmente utilizzato per l'accesso remoto quando un client non è gestito dal contenitore e non può utilizzare l'integrazione delle dipendenze. Ma JNDI può anche essere utilizzato dai client locali, anche se l'iniezione della dipendenza risulta in un codice più semplice. L'iniezione viene effettuata al momento dell'implementazione. Se esiste la possibilità che i dati non vengano utilizzati, il bean può evitare il costo dell'iniezione di risorse eseguendo una ricerca JNDI. JNDI è un'alternativa all'iniezione; tramite JNDI, il codice estrae i dati solo se sono necessari, invece di accettare i dati push che potrebbero non essere necessari.

JNDI è un'API per accedere a diversi tipi di servizi di directory, consentendo ai clienti di associare e cercare oggetti tramite un nome. JNDI è definito in Java SE ed è indipendente dall'implementazione sottostante, il che significa che è possibile cercare oggetti in una directory LDAP (Lightweight Directory Access Protocol) o in un Domain Name System (DNS) utilizzando un'API standard.

L'oggetto viene preso usando il metodo lookup().

```
Context ctx = new InitialContext();
ItemRemote itemEJB = (ItemRemote) ctx.lookup("java:global/cdbookstore/ItemEJB!org.agoncal.book.javaee7.ItemRemote");
```

Callbacks and Authorization

Un client non crea un'istanza di un bean di sessione utilizzando il nuovo operatore, ottiene un riferimento a un bean di sessione tramite l'iniezione della dipendenza o tramite la ricerca JNDI. Il contenitore è quello che crea l'istanza e la distrugge. Ciò significa che né il client né il bean sono responsabili per determinare quando viene creata l'istanza del bean, quando vengono immesse le dipendenze o quando l'istanza viene distrutta.

Il contenitore è responsabile della gestione del ciclo di vita del bean.

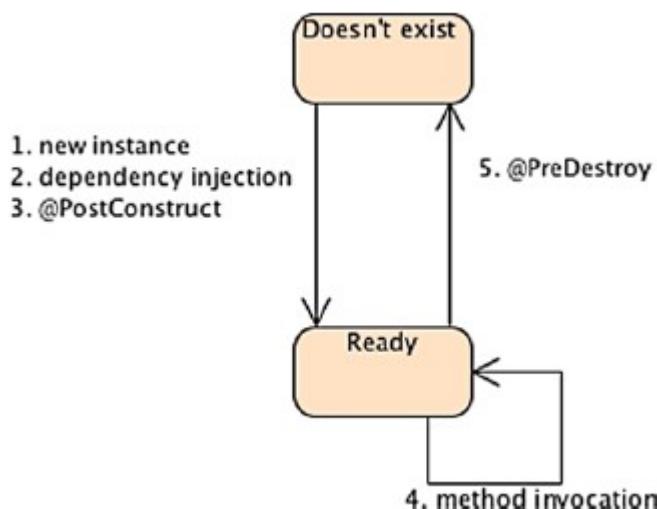
Tutti i bean di sessione hanno due fasi ovvie nel loro ciclo di vita: creazione e distruzione. Inoltre, i bean di sessione stateful passano attraverso le fasi di passivation e attivazione. Analogamente ai metodi di callback utilizzati nelle entità, gli EJB consentono al contenitore, durante determinate fasi della sua vita, di richiamare automaticamente i metodi annotati (@PostConstruct, @PreDestroy, ecc.). Questi metodi possono inizializzare le informazioni di stato sul bean, cercare le risorse usando JNDI o rilasciare connessioni al database.

Stateless and Singleton

I bean stateless e singleton hanno in comune il fatto che non mantengono lo stato di conversazione con un client dedicato. Entrambi i tipi di bean consentono l'accesso da parte di qualsiasi client:

Stateless fa questo per ogni istanza, mentre Singleton fornisce l'accesso concorrente a una singola istanza. Entrambi condividono lo stesso ciclo di vita e sono descritti come segue:

1. Il ciclo di vita di un bean di sessione stateless o singleton inizia quando un client richiede un riferimento al bean (utilizzando l'iniezione di dipendenza o la ricerca JNDI). Nel caso di un singleton, può anche avviarsi quando il container è bootstrapped(self-stated) (usando l'annotazione @Startup). Il contenitore crea una nuova istanza di bean di sessione.
2. Se l'istanza appena creata utilizza l'iniezione delle dipendenze tramite annotazioni (@Inject, @Resource, @EJB, @PersistenceContext, ecc.) o descrittori di distribuzione, il contenitore inietta tutte le risorse necessarie.
3. Se l'istanza ha un metodo annotato con @PostConstruct, il contenitore lo richiama.
4. L'istanza del bean elabora la chiamata invocata dal client e rimane in modalità ready per elaborare le chiamate future. I bean stateless rimangono nella modalità ready fino a quando il contenitore libera uno spazio nella pool. I singleton rimangono nella modalità Ready fino a quando il contenitore non viene spento.
5. Quando il contenitore non ha più bisogno dell'istanza richiama il metodo annotato con @PreDestroy, se presente, e termina la vita dell'istanza del bean.



I bean stateless e singleton condividono lo stesso ciclo di vita, ma ci sono alcune differenze nel modo in cui sono creati e distrutti. Quando si distribuisce un bean di sessione stateless, il contenito-

re crea diverse istanze e le aggiunge in un pool. Quando un client chiama un metodo su un bean di sessione stateless, il contenitore seleziona un'istanza dal pool, delega il richiamo del metodo a quell'istanza e lo restituisce al pool. Quando il contenitore non ha più bisogno dell'istanza (solitamente quando il contenitore vuole ridurre il numero di istanze nel pool), lo distrugge.

Per i bean di sessione singleton, la creazione dipende dal fatto che siano istanziati con (@Startup) o meno, o se dipendono (@DependsOn) da un altro singleton creato. Se la risposta è sì, il contenitore creerà un'istanza al momento della distribuzione. In caso contrario, il contenitore creerà un'istanza quando un client invoca un metodo di business. Poiché i singleton durano per tutta la durata dell'applicazione, l'istanza viene distrutta quando il contenitore si arresta.

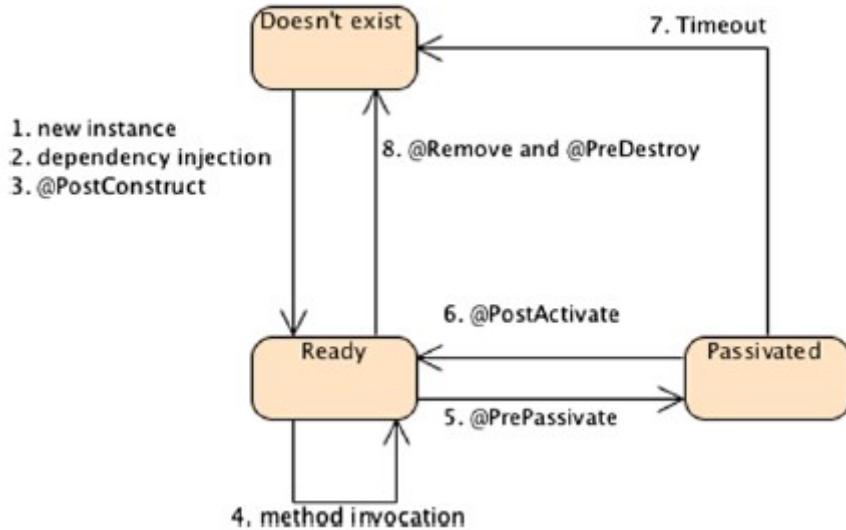
Stateful

I bean di sessione stateful sono programmaticamente non molto diversi dai bean di sessione stateless o singleton: solo la modifica dei metadati è diversa (@Stateful anziché @Stateless o @Singleton). Ma la vera differenza è che i bean stateful mantengono lo stato della conversazione con il loro client e quindi hanno un ciclo di vita leggermente diverso. Il container genera un'istanza e la assegna solo a un client. Quindi, ogni richiesta da quel client viene passata alla stessa istanza.

Se un client non richiama l'istanza del bean per un tempo sufficientemente lungo, il contenitore deve cancellarlo prima che JVM esaurisca la memoria, conservare lo stato dell'istanza in un'archiviazione permanente e quindi riportare l'istanza con il suo stato quando è necessario. Il contenitore impiega la tecnica della passivation e dell'attivazione.

La passivation avviene quando il contenitore serializza l'istanza del bean su un supporto di memorizzazione permanente (file su un disco, database, ecc.) invece di tenerlo in memoria. L'attivazione, che è l'opposto, viene eseguita quando l'istanza del bean è nuovamente necessaria dal client. Il contenitore deserializza il bean dalla memoria permanente e lo riattiva in memoria. Ciò significa che gli attributi del bean devono essere serializzabili (deve essere una primitiva Java o implementare l'interfaccia `java.io.Serializable`). La Figura seguente mostra il ciclo di vita dei bean con stato e la descrive come segue:

1. Il ciclo di vita di un bean stateful inizia quando un client richiede un riferimento al bean (usando l'injection dependency o la ricerca JNDI). Il contenitore crea una nuova istanza di bean di sessione e la archivia in memoria.
2. Se l'istanza appena creata utilizza l'iniezione delle dipendenze attraverso le annotazioni (@Inject, @Resource, @EJB, @PersistenceContext, ecc.) o descrittori di distribuzione, il contenitore inietta tutte le risorse necessarie.
3. Se l'istanza ha un metodo annotato con @PostConstruct, il contenitore lo richiama.
4. Il bean esegue la chiamata richiesta e rimane in memoria, in attesa di richieste client successive.
5. Se il client rimane inattivo per un periodo di tempo, il contenitore richiama il metodo annotato con @PrePassivate, se presente, e passa l'istanza del bean in un'archiviazione permanente.
6. Se il client richiama un bean passivato, il contenitore lo riattiva in memoria e richiama il metodo annotato con @PostActivate, se presente.
7. Se il client non richiama un'istanza di bean passivate per il periodo di timeout della sessione, il contenitore lo distrugge.
8. In alternativa al passaggio 7, se il client chiama un metodo annotato da @Remove, il contenitore richiama quindi il metodo annotato con @PreDestroy, se presente, e termina la vita dell'istanza del bean.



In alcuni casi, un bean stateful contiene risorse aperte come socket di rete o connessioni al database. Poiché un contenitore non può mantenere queste risorse aperte per ciascun bean, è necessario chiudere e riaprire le risorse prima e dopo la passivation utilizzando i metodi di callback. Un'altra possibilità è disattivare il comportamento di attivazione/passivation predefinito o il bean di stato. Si deve stare molto attento a quest'ultima cosa, ma se è ciò di cui si ha veramente bisogno si può annotare il bean stateful con `@Stateful` (`passivationCapable = false`).

Callbacks

Come hai appena visto, ogni tipo di bean di sessione ha il proprio ciclo di vita gestito dal contenitore. Il contenitore ti consente di fornire facoltativamente il tuo codice aziendale quando cambia lo stato del bean. La modifica da uno stato a un altro può essere intercettata dal contenitore per richiamare i metodi annotati da una delle annotazioni elencate nella seguente tabella:

Annotation	Description
<code>@PostConstruct</code>	Marks a method to be invoked immediately after you create a bean instance and the container does dependency injection. This annotation is often used to perform any initialization.
<code>@PreDestroy</code>	Marks a method to be invoked immediately before the container destroys the bean instance. The method annotated with <code>@PreDestroy</code> is often used to release resources that had been previously initialized. With stateful beans, this happens after timeout or when a method annotated with <code>@Remove</code> has been completed.
<code>@PrePassivate</code>	Stateful beans only. Marks a method to be invoked before the container passivates the instance. It usually gives the bean the time to prepare for serialization and to release resources that cannot be serialized (e.g., it closes connections to a database, a message broker, a network socket, etc.).
<code>@PostActivate</code>	Stateful beans only. Marks a method to be invoked immediately after the container reactivates the instance. Gives the bean a chance to reinitialize resources that had been closed during passivation.

Le seguenti regole si applicano a un metodo di callback:

- Il metodo non deve avere parametri e deve restituire void.
- Il metodo non deve generare un'eccezione controllata ma può generare eccezioni di runtime. Lanciare un'eccezione di runtime ripristinerà la transazione se ne esiste una.
- Il metodo può avere accesso pubblico, privato, protetto o a livello di pacchetto, ma non deve essere statico o final.
- Un metodo può essere annotato con più annotazioni (il metodo ini () mostrato in seguito nel Listato 8-2 è annotato con `@PostConstruct` e `@PostActivate`). Tuttavia, su un bean può essere presente solo un'annotazione di un determinato tipo (ad esempio, non è possibile avere due annotazioni `@PostConstruct` nello stesso bean di sessione).

- Un metodo di callback può accedere alle voci dell'ambiente dei bean. Questi callback vengono in genere utilizzati per allocare e/o rilasciare le risorse del bean. Ad esempio, il Listato 8-1 mostra il bean singleton CacheEJB usando un'annotazione `@PostConstruct` per inizializzare la sua cache. Subito dopo aver creato la singola istanza di CacheEJB, il contenitore richiama il metodo `initCache()`.

Listing 8-1. A Singleton Initializing Its Cache with the `@PostConstruct` Annotation

```
@Singleton
public class CacheEJB {

    private Map<Long, Object> cache = new HashMap<>();

    @PostConstruct
    private void initCache() {
        cache.put(1L, "First item in the cache");
        cache.put(2L, "Second item in the cache");
    }

    public Object getFromCache(Long id) {
        if (cache.containsKey(id))
            return cache.get(id);
        else
            return null;
    }
}
```

Listing 8-2. A Stateful Bean Initializing and Releasing Resources

```
@Stateful
public class ShoppingCartEJB {

    @Resource(lookup = "java:comp/defaultDataSource")
    private DataSource ds;
    private Connection connection;

    private List<Item> cartItems = new ArrayList<>();

    @PostConstruct
    @PostActivate
    private void init() {
        connection = ds.getConnection();
    }

    @PreDestroy
    @PrePassivate
    private void close() {
        connection.close();
    }

    // ...

    @Remove
    public void checkout() {
        cartItems.clear();
    }
}
```

Authorization

Lo scopo principale del modello di sicurezza EJB è controllare l'accesso al codice business.

Nell'autenticazione Java EE viene gestito dal livello Web (o un'applicazione client); il principale ei suoi ruoli vengono quindi passati al livello EJB e l'EJB controlla se l'utente autenticato è autorizzato ad accedere a un metodo in base al suo ruolo. L'autorizzazione può essere effettuata in modo dichiarativo o programmatico.

Con l'autorizzazione dichiarativa, il controllo degli accessi viene effettuato dal contenitore EJB.

Con l'autorizzazione programmatica, il controllo dell'accesso viene effettuato nel codice utilizzando l'API JAAS.

Declarative Authorization

L'autorizzazione dichiarativa può essere definita nel bean utilizzando annotazioni o nel descrittore di distribuzione XML.

L'autorizzazione dichiarativa implica la dichiarazione di ruoli, l'assegnazione dell'autorizzazione ai metodi (o all'intero bean) o la modifica temporanea di un'identità di sicurezza. Questi controlli sono fatti dalle annotazioni descritte nella Tabella 8-6. Ogni annotazione può essere utilizzata sul bean e/o sul metodo.

Table 8-6. Security Annotations

Annotation	Bean	Method	Description
@PermitAll	X	X	Indicates that the given method (or the entire bean) is accessible by everyone (all roles are permitted).
@DenyAll	X	X	Indicates that no role is permitted to execute the specified method or all methods of the bean (all roles are denied). This can be useful if you want to deny access to a method in a certain environment (e.g., the method launchNuclearWar() should only be allowed in production but not in a test environment).
@RolesAllowed	X	X	Indicates that a list of roles is allowed to execute the given method (or the entire bean).

Listing 8-5. A Stateless Bean Allowing Certain Roles

```
@Stateless  
@RolesAllowed({"user", "employee", "admin"})  
public class ItemEJB {  
  
    @PersistenceContext(unitName = "chapter08PU")  
    private EntityManager em;  
  
    public Book findBookById(Long id) {  
        return em.find(Book.class, id);  
    }  
  
    public Book createBook(Book book) {  
        em.persist(book);  
        return book;  
    }  
  
    @RolesAllowed("admin")  
    public void deleteBook(Book book) {  
        em.remove(em.merge(book));  
    }  
}
```

L'annotazione @DeclareRoles è leggermente diversa in quanto non consente o nega alcun accesso. Dichiara i ruoli per l'intera applicazione. Quando si distribuisce l'EJB nel Listato 8-6, il container dichiarerà automaticamente i ruoli utente, dipendente e amministratore controllando l'annotazione

Listing 8-6. A Stateless Bean Using @PermitAll and @DenyAll

```
—  
@Stateless  
@RolesAllowed({"user", "employee", "admin"})  
public class ItemEJB {  
  
    @PersistenceContext(unitName = "chapter08PU")  
    private EntityManager em;  
  
    @PermitAll  
    public Book findBookById(Long id) {  
        return em.find(Book.class, id);  
    }  
  
    public Book createBook(Book book) {  
        em Listing 8-7. A Stateless Bean Declaring Roles  
        re  
    } @Stateless  
    @DeclareRoles({"HR", "salesDpt"})  
    @Role @RolesAllowed({"user", "employee", "admin"})  
    publ public class ItemEJB {  
        em  
    } @PersistenceContext(unitName = "chapter08PU")  
    private EntityManager em;  
    @Den  
    publ public Book findBookById(Long id) {  
        re  
    }  
    public Book createBook(Book book) {  
        em.persist(book);  
        return book;  
    }  
  
    @RolesAllowed("admin")  
    public void deleteBook(Book book) {  
        em.remove(em.merge(book));  
    }  
}
```

@RolesAllowed. Ma potresti voler dichiarare altri ruoli nel dominio di sicurezza per l'intera applicazione (non solo per un singolo EJB) attraverso l'annotazione @DeclareRoles.

Questa annotazione, che si applica solo a livello di classe, accetta una serie di ruoli e li dichiara nel dominio di sicurezza. In effetti, si dichiarano i ruoli di sicurezza utilizzando una di queste due annotazioni o una combinazione di entrambi.

Se vengono utilizzate entrambe le annotazioni, vengono dichiarate le aggregazioni dei ruoli in @DeclareRoles e @RolesAllowed.

Di solito dichiariamo i ruoli per l'intera applicazione aziendale, quindi in questo caso ha più senso dichiarare ruoli nel descrittore di distribuzione che con l'annotazione @DeclareRoles.

Quando si distribuisce ItemEJB nel Listato 8-7, vengono dichiarati i cinque ruoli HR, salesDpt, user, employee e admin.

Quindi, con l'annotazione @RolesAllowed, a determinati di questi ruoli viene concesso l'accesso a determinati metodi (come spiegato in precedenza).

L'ultima annotazione, @RunAs, è utile se è necessario assegnare temporaneamente un nuovo ruolo al ruolo principale esistente.

Potrebbe essere necessario farlo, ad esempio, se stai invocando un altro EJB all'interno del tuo metodo, ma l'altro EJB richiede un ruolo diverso.

Listing 8-8. A Stateless Bean Running as a Different Role

```
@Stateless
@RolesAllowed({"user", "employee", "admin"})
@RunAs("inventoryDpt")
public class ItemEJB {

    @PersistenceContext(unitName = "chapter08PU")
    private EntityManager em;

    @EJB
    private InventoryEJB inventory;

    public List<Book> findBooks() {
        TypedQuery<Book> query = em.createNamedQuery("findAllBooks", Book.class);
        return query.getResultList();
    }

    public Book createBook(Book book) {
        em.persist(book);
        inventory.addItem(book);
        return book;
    }
}
```

Programmatic Authorization

L'autorizzazione dichiarativa copre la maggior parte dei casi di sicurezza necessari per un'applicazione. Ma a volte hai bisogno di una grana più fine per autorizzare l'accesso (consentendo un blocco di codice anziché l'intero metodo, permettendo o negando l'accesso a un individuo anziché a un ruolo, ecc.). È possibile utilizzare l'autorizzazione programmatica per consentire o bloccare in modo selettivo l'accesso a un ruolo o a un principal. Questo perché hai un accesso diretto all'interfaccia java.security.Principal di JAAS e al contesto EJB per verificare il ruolo del principale nel codice.

L'interfaccia SessionContext definisce i seguenti metodi relativi alla sicurezza:

- isCallerInRole (): questo metodo restituisce un valore booleano e verifica se il chiamante ha un determinato ruolo di sicurezza.
- getCallerPrincipal (): questo metodo restituisce java.security.Principal che identifica il chiamante.

Per mostrare come utilizzare questi metodi, diamo un'occhiata a un esempio. L'ItemEJB nel Listato 8-9 non usa alcuna sicurezza dichiarativa, ma ha ancora bisogno di fare qualche tipo di controllo a livello di codice. Prima di tutto, il bean deve ottenere un riferimento al suo contesto (usando l'annotazione @Resource). Con questo contesto, il metodo deleteBook () può verificare se il chiamante ha o meno un ruolo di amministratore. In caso contrario, lancia una java.lang.SecurityException per notificare all'utente la violazione dell'autorizzazione. Il metodo createBook() esegue una logica business utilizzando i ruoli e il principal. Si noti che il metodo getCallerPrincipal() restituisce un oggetto Principal, che ha un nome. Il metodo controlla se il nome dell'entità è paul, quindi imposta il valore "utente speciale" sull'entità del libro.

Listing 8-9. A Bean Using Programmatic Security

```
@Stateless  
public class ItemEJB {  
  
    @PersistenceContext(unitName = "chapter08PU")  
    private EntityManager em;  
  
    @Resource  
    private SessionContext ctx;  
  
    public void deleteBook(Book book) {  
        if (!ctx.isCallerInRole("admin"))  
            throw new SecurityException("Only admins are allowed");  
  
        em.remove(em.merge(book));  
    }  
  
    public Book createBook(Book book) {  
        if (ctx.isCallerInRole("employee") && !ctx.isCallerInRole("admin")) {  
            book.setCreatedBy("employee only");  
        } else if (ctx.getCallerPrincipal().getName().equals("paul")) {  
            book.setCreatedBy("special user");  
        }  
        em.persist(book);  
        return book;  
    }  
}
```

Transactions

La gestione delle transazioni è una questione importante per le imprese. Consente alle applicazioni di disporre di dati coerenti e di elaborare tali dati in modo affidabile. La gestione delle transazioni è una preoccupazione di basso livello che uno sviluppatore di business non dovrebbe dover codificare autonomamente. Gli EJB forniscono questi servizi in un modo molto semplice: programmaticamente con un alto livello di astrazione o dichiaratamente utilizzando i metadati. La maggior parte del lavoro di un'applicazione aziendale riguarda la gestione dei dati: archiviarli (in genere in un database), recuperarli, elaborarli e così via. Spesso questo viene fatto simultaneamente da diverse applicazioni che tentano di accedere agli stessi dati. Un database ha meccanismi di basso livello per preservare l'accesso simultaneo, e usa le transazioni per assicurare che i dati rimangano in uno stato coerente. I bean fanno uso di questi meccanismi.

Understanding Transactions

I dati sono cruciali per il business e devono essere precisi indipendentemente dalle operazioni eseguite e dal numero di applicazioni che accedono contemporaneamente ai dati. Una transazione viene utilizzata per garantire che i dati vengano mantenuti in uno stato coerente. Rappresenta un gruppo logico di operazioni che devono essere eseguite come una singola unità, anche nota come unità di lavoro.

Queste operazioni possono coinvolgere dati persistenti in uno o più database, l'invio di messaggi a un MOM (middleware orientato ai messaggi) o il richiamo di servizi Web. Le transazioni devono garantire un grado di affidabilità e robustezza e seguire le proprietà ACID.

ACID

L'ACID si riferisce alle quattro proprietà che definiscono una transazione affidabile: Atomicità, Consistenza, Isolamento e Durata.

Property	Description
Atomicità	Una transazione è composta da una o più operazioni raggruppate in un'unità di lavoro. Alla conclusione della transazione, entrambe queste operazioni vengono eseguite correttamente (commit) o nessuna di queste viene eseguita (rollback) se accade qualcosa di inaspettato o irrecuperabile.
Consistenza	Alla conclusione della transazione, i dati vengono lasciati in uno stato coerente.
Isolamento	Lo stato intermedio di una transazione non è visibile alle applicazioni esterne. Durata Una volta eseguita la transazione, le modifiche apportate ai dati sono visibili ad altre applicazioni
Durabilità	Una volta eseguita la transazione, le modifiche apportate ai dati sono visibili ad altre applicazioni

Read Conditions

L'isolamento delle transazioni può essere definito utilizzando diverse condizioni di lettura (lettura sporche, lettura ripetibili e lettura fantasma). Descrivono cosa può accadere quando due o più transazioni operano sugli stessi dati nello stesso momento.

A seconda del livello di isolamento che si ha messo in atto, si può totalmente evitare o consentire l'accesso simultaneo.

- Letture sporche: si verificano quando una transazione legge le modifiche non vincolanti apportate dalla transazione precedente.
- Letture ripetibili: si verificano quando i dati letti sono garantiti per apparire uguali se letti di nuovo durante la stessa transazione.
- Letture fantasma: si verificano quando i nuovi record aggiunti al database sono rilevabili dalle transazioni avviate prima dell'inserto. Le query includeranno i record aggiunti da altre transazioni dopo l'avvio della transazione.

Transaction Isolation Levels

I database utilizzano diverse tecniche di blocco per controllare come le transazioni accedono contemporaneamente ai dati. I meccanismi di bloccaggio influiscono sulla condizione di lettura descritta in precedenza. I livelli di isolamento sono comunemente usati nei database per descrivere come il blocco viene applicato ai dati all'interno di una transazione. I quattro tipi di livelli di isolamento sono:

- Lettura non vincolata (livello di isolamento meno restrittivo): la transazione può leggere dati non salvati. Possono verificarsi letture sporche, non ripetibili e fantasma.
- Lettura impegnata: la transazione non può leggere dati non salvati. Le letture sporche sono prevenute ma non le letture fantasma o non ripetibili.
- Lettura ripetibile: la transazione non può modificare i dati letti da una transazione diversa. Le letture sporche e non ripetibili vengono prevenute ma possono verificarsi letture fantasma.
- Serializable (livello di isolamento più restrittivo): la transazione ha una lettura esclusiva. Le altre transazioni non possono né leggere né scrivere gli stessi dati.

In generale, poiché il livello di isolamento diventa più restrittivo, le prestazioni del sistema diminuiscono poiché alle transazioni viene impedito di accedere agli stessi dati. Tuttavia, il livello di isolamento applica la coerenza dei dati. Si noti che non tutti i RDBMS (Relational Database Management Systems) implementano questi quattro livelli di isolamento.

JTA Local Transactions

Per il corretto funzionamento delle transazioni devono essere presenti diversi componenti e seguire le proprietà ACID. Quando c'è una sola risorsa transazionale, tutto ciò che serve è una transazione JTA locale. Una resource local transactions è una transazione che si ha con una singola risorsa specifica utilizzando la propria API specifica. La Figura 9-1 mostra l'applicazione che interagisce con una risorsa attraverso un gestore delle transazioni e un gestore delle risorse

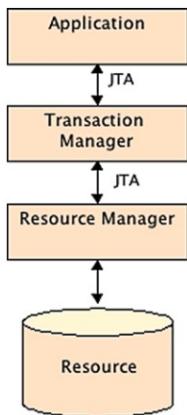


Figure 9-1. A transaction involving one resource

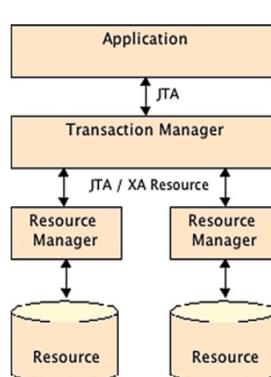
I componenti mostrati nella Figura 9-1 riassumono la maggior parte dell'elaborazione specifica della transazione dall'applicazione.

- Il gestore delle transazioni è il componente principale responsabile della gestione delle operazioni transazionali. Crea le transazioni per conto dell'applicazione, informa il gestore risorse che sta partecipando a una transazione (un'operazione nota come arruolamento) e conduce il commit o il rollback sul gestore risorse.
- Il gestore risorse è responsabile della gestione delle risorse e della registrazione con il gestore delle transazioni. Un esempio di un gestore risorse è un driver per un database relazionale, una risorsa JMS o un connettore Java
- La risorsa è la memoria permanente da cui si legge o scrive (un database, una destinazione messaggio, ecc.).

Non è responsabilità dell'applicazione conservare le proprietà ACID. L'applicazione decide di eseguire il commit o il rollback della transazione e il gestore delle transazioni prepara tutte le risorse per farlo in modo corretto.

Distributed Transactions and XA

Molte applicazioni aziendali utilizzano più di una risorsa per transazione. Tali transazioni a livello aziendale richiedono un coordinamento speciale che coinvolge XA e Java Transaction Service (JTS). La Figura 9-2 mostra un'applicazione che utilizza la demarcazione delle transazioni su più risorse. Ciò significa che, nella



stessa unità di lavoro, l'applicazione database e inviare un messaggio

Figure 9-2. An XA transaction involving two resources

Per avere una transazione affidabile su più risorse, il gestore delle transazioni deve utilizzare un'interfaccia di gestione risorse XA (eXtended Architecture). XA è uno standard specificato da Open Group (www.opengroup.org) per l'elaborazione di transazioni distribuite (DTP) che conserva le proprietà ACID. È supportato da JTA e consente a gestori di risorse eterogenei di diversi fornitori di interoperate attraverso un'interfaccia comune. XA utilizza un commit a due fasi (2pc) per garantire che tutte le risorse eseguano il commit o il rollback di una determinata transazione contemporaneamente.

Nella figura 9-3 durante la fase 1, ogni gestore di risorse riceve una notifica tramite un comando di "preparazione" che sta per essere pubblicato un commit. Ciò consente ai responsabili delle risorse di dichiarare se possono applicare le loro modifiche o meno. Se tutti indicano che sono pronti, la transazione è autorizzata a procedere e a tutti i gestori di risorse viene richiesto di eseguire il commit nella seconda fase.

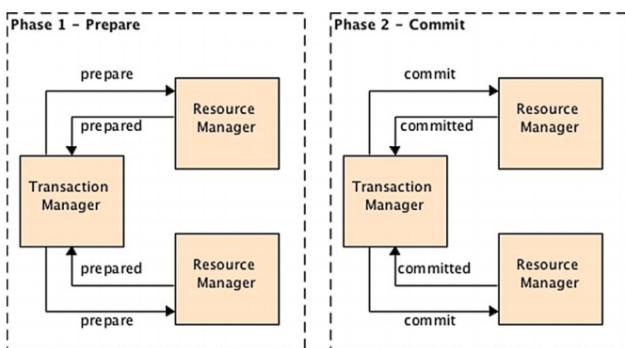


Figure 9-3. Two-phase commit

La maggior parte delle volte le risorse sono distribuite attraverso la rete (vedere la Figura 9-4). Un tale sistema si basa su JTS. JTS implementa la specifica OTS (Object Transaction Service) OMG (Object Management Group), consentendo ai gestori delle transazioni di partecipare alle transazioni distribuite tramite il protocollo Inter-ORB (IIOP) Internet.

Rispetto alla Figura 9-2, in cui esiste un solo gestore transazioni, la Figura 9-4 consente la propagazione delle transazioni distribuite tramite IIOP. Ciò consente di distribuire le transazioni tra diversi computer e diversi database di diversi fornitori. JTS è destinato ai fornitori che forniscono l'infrastruttura del sistema di transazione. Come sviluppatore EJB, non devi preoccuparti di questo; basta usare JTA, che si interfaccia con JTS ad un livello più alto.

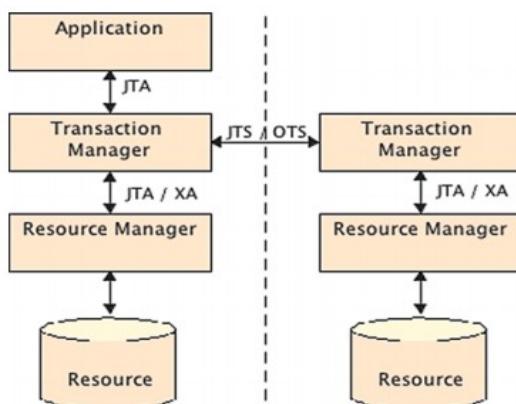


Figure 9-4. A distributed XA transaction

Transaction Specifications Overview

In Java EE 7, EJB e Managed Beans gestiscono le transazioni tramite l'API JTA (Java Transaction API) specificata da JSR 907. JTA definisce un insieme di interfacce per l'applicazione o il contenitore per delimitare i confini delle transazioni e definisce anche le API da gestire il gestore delle transazioni. Il pacchetto javax.transaction definisce queste interfacce.

JTS è una specifica per la creazione di un gestore delle transazioni che supporta le interfacce JTA ad alto livello e la mappatura Java standard delle specifiche di CORBA Object Transaction Service 1.1 a basso livello. JTS fornisce interoperabilità delle transazioni utilizzando il protocollo IIOP standard CORBA per la propagazione delle transazioni tra server.

JTS è destinato ai fornitori che forniscono l'infrastruttura del sistema di transazioni per il middleware aziendale.

Per quanto riguarda la gestione delle transazioni, è improbabile che si desideri utilizzare le API JTS / JTA non elaborate in Java. Al contrario, si delegano le transazioni al contenitore EJB che contiene un gestore transazioni (che utilizza internamente JTS e JTA).

What's New in JTA 1.2?

JTA 1.2 offre supporto per le transazioni gestite dal contenitore indipendenti da EJB e un'annotazione @TransactionScope per l'ambito dei bean CDI.

L'API JTA è composta da classi e interfacce raggruppate in due pacchetti descritti nella Tabella 9-2.

Package	Description
javax.transaction	Contiene le principali API JTA
javax.transaction.xa	Interfacce e classi per realizzare transazioni XA distribuite

Transaction Support in EJBs

Quando sviluppatate la logica di business con EJB, non dovete preoccuparvi della struttura interna dei gestori delle transazioni o dei gestori delle risorse perché JTA astrae la maggior parte della complessità sottostante. Con gli EJB, puoi sviluppare un'applicazione transazionale molto facilmente, lasciando il contenitore per implementare i protocolli di transazione di basso livello, come il commit a due fasi o la propagazione del contesto di transazione. Un contenitore EJB è un gestore transazioni che supporta JTA e JTS per partecipare a transazioni distribuite che coinvolgono altri contenitori EJB e / o altre risorse transazionali. In una tipica applicazione Java EE, i bean di sessione stabiliscono i limiti di una transazione, chiamano le entità per interagire con il database o inviano messaggi JMS in un contesto di transazione.

Dalla sua creazione, il modello EJB è stato progettato per gestire le transazioni. In effetti, le transazioni sono naturali per gli EJB e, per impostazione predefinita, ciascun metodo viene automaticamente incluso in una transazione. Questo comportamento predefinito è noto come una transazione gestita dal contenitore (CMT), poiché le transazioni sono gestite dal contenitore EJB (a.k.a. demarcazione della transazione dichiarativa). È inoltre possibile scegliere di gestire le transazioni manualmente utilizzando transazioni gestite da bean (BMT), denominate anche demarcazione della transazione programmatica. La demarcazione delle transazioni determina dove iniziano e terminano le transazioni.

Container-Managed Transactions

Quando si gestiscono le transazioni in modo dichiarativo, si delegano i criteri di demarcazione al contenitore. Non devi usare esplicitamente JTA nel tuo codice (anche se JTA è usato sotto); è possibile lasciare il contenitore per delimitare i confini delle transazioni iniziando e committendo (commit) automaticamente le transazioni in base ai metadati. Il contenitore EJB fornisce servizi di

gestione delle transazioni a bean di sessione e MDB. In un bean enterprise con una transazione gestita dal contenitore, il contenitore EJB imposta i limiti delle transazioni.

Il Listato 9-1 mostra il codice di un bean di sessione stateless usando CMT. Come puoi vedere, non è stata aggiunta alcuna annotazione aggiuntiva o alcuna interfaccia speciale da implementare. Gli EJB sono per natura transazionali. Con la configurazione per eccezione, vengono applicati tutti i valori predefiniti di gestione delle transazioni (OBBLIGATORIO è l'attributo di transazione predefinito come spiegato più avanti in questa sezione).

Listing 9-1. A Stateless Bean with CMT

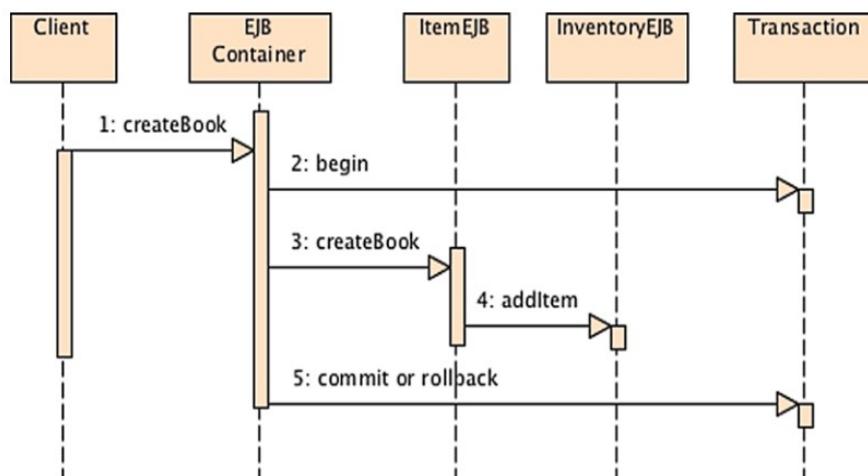
```
@Stateless
public class ItemEJB {

    @PersistenceContext(unitName = "chapter09PU")
    private EntityManager em;
    @Inject
    private InventoryEJB inventory;

    public List<Book> findBooks() {
        TypedQuery<Book> query = em.createNamedQuery(FIND_ALL, Book.class);
        return query.getResultList();
    }

    public Book createBook(Book book) {
        em.persist(book);
        inventory.addItem(book);
        return book;
    }
}
```

Potresti chiedere cosa rende il codice nel listato 9-1 transazionale. La risposta è il contenitore. La Figura 9-5 mostra cosa succede quando un client richiama il metodo `createBook()`. La chiamata client viene intercettata dal contenitore, che verifica immediatamente prima di richiamare il metodo se un contesto di transazione è associato alla chiamata. Per impostazione predefinita, se non è disponibile alcun contesto di transazione, il contenitore inizia una nuova transazione prima di immettere il metodo e quindi richiama il metodo `createBook()`. Una volta che il metodo è terminato, il contenitore esegue automaticamente il commit della transazione o lo ritira (se viene generato un particolare tipo di eccezione, come vedremo più avanti nella sezione "Eccezioni e transazioni").

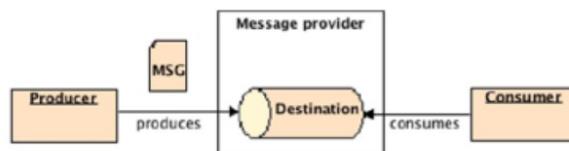


JMS

Understanding messaging

MOM (Message-oriented middleware) è un software (provider) che permette lo scambio di messaggi asincroni fra sistemi eterogenei. Può essere visto come un buffer che produce e consuma messaggi. È intrinsecamente loosely coupled dal momento che i produttori non sanno chi è all'altra estremità del canale di comunicazione ad usare il messaggio. Il produttore e il consumatore non devono essere disponibili contemporaneamente per comunicare.

Quando un messaggio viene inviato, il software che memorizza il messaggio e lo invia è detto Provider (broker). Il sender del messaggio è chiamato Producer e la locazione in cui il messaggio è memorizzato è detta destinazione. La componente che riceve il messaggio è detta Consumer. Ogni componente interessata ad un messaggio in una particolare destinazione può consumarlo.



Java Message Service (JMS)

In Java EE, l'API che gestisce questi concetti è Java Message Service (JMS), un set di interface e classi per:

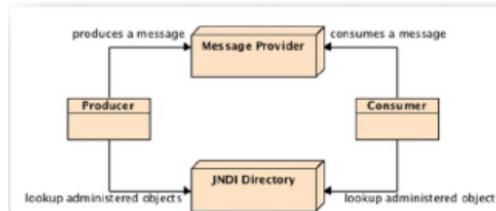
- Connetersi ad un provider
- Creare un messaggio
- Inviare un messaggio
- Ricevere un messaggio

In un EJB container, Message-Driven Beans (MDBs) possono essere usati per ricevere messaggi in container-managed way (auto gestiti).

Architettura di Messaging

Componenti di un'architettura di messaging:

- Un Provider: componente necessaria per instradare messaggi gestisce il buffering e il delivery dei messaggi.
- Clients: una qualunque applicazione Java o una componente che produce o consuma dei messaggi per/da un provider. Il termine “Client” si usa genericamente per producer, sender, publisher, consumer, receiver, subscriber.
- Messages: oggetti che i client inviano/ricevono dal provider.
- Administered objects: oggetti (connection factories e destinazioni) fornite attraverso JNDI lookups o injection.



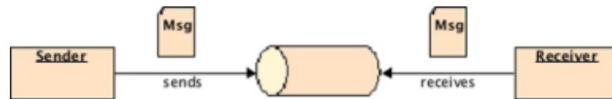
Il Provider permette comunicazione asincrona fornendo una destinazione dove i messaggi possono essere mantenuti finché non vengono instradati verso un client. Esistono due differenti tipi di destination:

- Point-to-point (P2P) model: la destinazione è chiamata coda. Il client inserisce un messaggio in coda, mentre un altro client riceve il messaggio. Una volta fatto acknowledge, il message provider rimuove il messaggio dalla coda.

- Publish-subscribe (pub-sub) model: la destinazione è chiamata topic. Il client pubblica un messaggio con un topic, e tutti i sottoscrittori al topic riceveranno il messaggio.

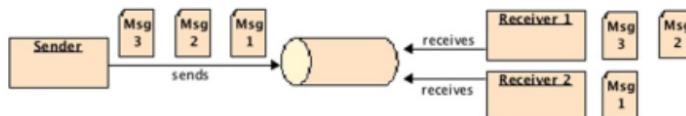
Point-to-Point model

Il messaggio viaggia da un singolo producer verso un singolo consumer. Ogni messaggio viene inviato ad una specifica coda, ed il receiver riceve il messaggio dalla coda. La coda mantiene i messaggi finché non vengono consumati o scadono.



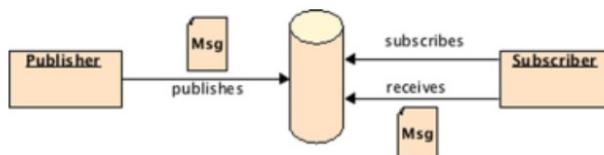
Nel modello P2P, esiste un solo receiver per ogni messaggio. Una coda può avere consumers multipli, ma quando un receiver consuma il messaggio, questo viene tolto dalla coda, e nessun altro receiver potrà consumarlo. Il modello P2P non garantisce che i messaggi siano instradati in un particolare

ordine. Un provider può riceverli in un particolare ordine, o random, o in qualunque altro ordine.



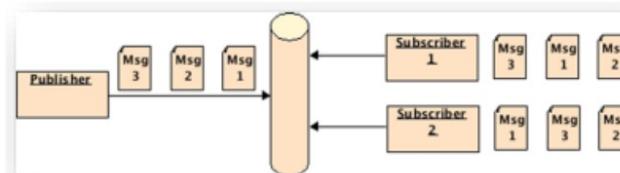
Publish-Subscribe Model

Nel modello pub-sub model, un singolo messaggio è inviato ad un singolo producer per potenzialmente diversi consumers. Il modello è costruito intorno ai concetti di topics, publishers, e subscribers. I Consumers sono chiamati subscribers e hanno necessità di sottoscriversi ad un topic, in più forniscono il meccanismo di subscribing/unsubscribing, che occorre dinamicamente.



Il topic conserva i messaggi fino a quando non vengono distribuiti a tutti i subscribers, esiste una dipendenza temporale fra publisher e subscriber.

I subscribers NON ricevono i messaggi inviati PRIMA della loro sottoscrizione e, se il subscriber è inattivo per un periodo di tempo determinato, esso non riceve messaggi vecchi quando diventa nuovamente attivo. Multipli subscribers possono consumare lo stesso messaggio. È Utile per broadcast-type applications: singolo messaggio recapitato a diversi consumatori.



Administered objects

Oggetti che si configurano amministrativamente, e non programmaticamente. Il provider permette di configurare questi oggetti e li rende disponibili nello spazio dei nomi JNDI. Come JDBC datasources questi oggetti vengono creati solo una volta. I due tipi di oggetti amministrati sono:

Connection factory: usato dai clienti per creare una connessione a una destinazione

- Destinazioni: punti di distribuzione del messaggio che ricevono, mantengono, e distribuiscono messaggi.
- Le destinazioni possono essere code (P2P) o topic (pub-sub).

Message-Driven Beans

I Message-Driven Beans (MDBs) sono message consumer asincroni eseguiti in un EJB container. L'EJB container si occupa dei servizi (transactions, security, concurrency, message acknowledgment, etc.), mentre l'MDB si occupa di consumare messaggi MDBs sono stateless. L'EJB container può avere numerose istanze, eseguite in concorrenza per processare messaggi provenienti da diversi producers. In generale gli MDBs sono in ascolto su una destination (queue o topic) e, quando il messaggio arriva, lo consuma e lo processa. Poiché sono stateless, gli MDBs non mantengono stato attraverso invocazioni separate. Gli MDBs rispondono a messaggi ricevuti dal container laddove gli stateless session beans rispondono a richieste client attraverso un' interfaccia appropriata: local, remote, o no-interface.

Java Messaging Service API

JMS è un insieme di standard Java API che permette alle applicazioni di creare, inviare, ricevere e leggere messaggi in maniera asincrona. Definisce un insieme di interfacce e classi per la comunicazione con altri message providers. JMS è analogo a JDBC:

- JDBC permette la connessione a differenti databases (Derby, MySQL, Oracle, DB2, etc.)
- JMS permette la connessione a diversi providers (OpenMQ, MQSeries, SonicMQ, etc.)

Classic=1.1, Simplified=2.x, Legacy=1.0

Classic API	Simplified API	Legacy API (P2P)	Legacy API (Pub-Sub)
ConnectionFactory	ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	JMSSession	QueueConnection	TopicConnection
Session	JMSSession	QueueSession	TopicSession
Destination	Destination	Queue	Topic
Message	Message	Message	Message
MessageConsumer	JMSConsumer	QueueReceiver	TopicSubscriber
MessageProducer	JMSPublisher	QueueSender	TopicPublisher
JMSException	JMSRuntimeException	JMSException	JMSException

Connection Factory

Le Connection factories sono degli administered objects. L'interfaccia javax.jms.ConnectionFactory incapsula i parametri definiti da un amministratore. Per usare un administered object come una ConnectionFactory, il client deve eseguire una JNDI lookup (o usare injection). Per esempio, nel seguente code fragment si ottiene un JNDI InitialContext object e lo si usa per fare look up di una connectionFactory attraverso il suo JNDI name:

```
Context ctx = new InitialContext();
ConnectionFactory ConnectionFactory = (ConnectionFactory) ctx.lookup("jms/javaee7/ConnectionFactory");
```

Destination

Una destination è un administered object che contiene “provider-specific configuration information” come ad esempio un destination address. Questo meccanismo è nascosto al JMS client attraverso l'uso dell'interfaccia javax.jms.Destination. Come per le connection factory, una JNDI lookup è necessaria per restituire tali oggetti:

```
Context ctx = new InitialContext();
Destination queue = (Destination) ctx.lookup("jms/javaee7/Queue");
```

Simplified API

Le altre interfacce:

- JMSSession: active connection ad un MS provider e single-threaded context per inviare e ricevere messaggi
- JMSPublisher: oggetto creato da un JMSSession per inviare messaggi ad una coda o ad un topic
- JMSConsumer: oggetto creato da un JMSSession per ricevere messaggi inviati ad una coda o ad un topic

Message Producers

Produttori fuori dal container

Un oggetto JMSProducer viene creato da un JMSContext e usato per inviare messaggi

I passi da seguire:

- Ottenere una connection factory ed una coda con JNDI
- Creare un JMSContext usando la factory
- Creare un JMSProducer usando il contesto
- Inviare un messaggio usando il metodo send() del producer

```
public class Producer {  
    public static void main(String[] args)  
    { try{  
        Context jndiContext = new InitialContext();  
  
        ConnectionFactory connectionFactory = (ConnectionFactory) jndiContext.lookup("jms/javaee7/ConnectionFactory");  
        Destination queue = (Destination) jndiContext.lookup("jms/javaee7/Queue");  
  
        try(JMSContext context =  
            connectionFactory.createContext()) {  
            context.createProducer().send(queue,  
                "Text message sent at"+  
                new Date());  
        }  
    } catch(NamingException e) {  
        e.printStackTrace();  
    }  
}
```

La classe Producer produce un Message in una Queue.

Produttore in un container

```
@Stateless  
public class ProducerEJB {  
    @Resource(lookup ="jms/javaee7/ConnectionFactory")  
    private ConnectionFactory connectionFactory;  
  
    @Resource(lookup ="jms/javaee7/Queue")  
    private Queue queue;  
  
    public void sendMessage() {  
        try(JMSContext context =  
            connectionFactory.createContext()) {  
            context.createProducer().send(queue,  
                "Text message sent at "+ new Date());  
        }  
    }  
}
```

ProducerEJB in esecuzione all'interno di un Container usando @Resource.

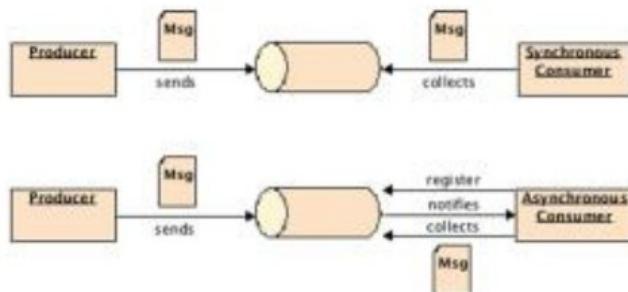
Produttore in un container con CDI

```
public class Producer {  
  
    @Inject  
    @JMSConnectionFactory("jms/javaee7/ConnectionFactory")  
    private JMSContext context;  
  
    @Resource(lookup ="jms/javaee7/Queue")  
    private Queue queue;  
  
    public void sendMessage() {  
        context.createProducer().send(queue,  
            "Text message sent at "+ new Date());  
    }  
}
```

Managed Bean che produce un Message using @Inject.

Message Consumers

- Sincroni: il ricevitore esplicitamente preleva il messaggio dalla destinazione, invocando receive()
- Asincroni: il ricevitore si registra all'arrivo di un messaggio e implementa MessageListener , in modo che all'arrivo del messaggio viene invocato il suo metodo onMessage()



Consumer sincrono

```
public class Consumer {  
    public static void main(String[] args)  
    {  
        try{  
            Context jndiContext = new InitialContext();  
  
            ConnectionFactory connectionFactory = (ConnectionFactory)  
                jndiContext.lookup("jms/javaee7/ConnectionFactory");  
            Destination queue = (Destination)  
                jndiContext.lookup("jms/javaee7/Queue");  
  
            try(JMSContext context =  
                connectionFactory.createContext()) {  
                while(true) {  
                    String message =  
                        context.createConsumer(queue).receiveBody(String.class);  
                }  
            }catch(NamingException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

- › Preleva il contesto JNDI
- › Cerca gli oggetti amministrati: factory per le connessioni ...
- › ... e coda
- › Acquisisce il contesto (con ciclo di vita gestito da try-with-resources)
- › Ciclo infinito (*busy waiting!!!*)
- › Prova a ricevere

La classe Consumer consuma Messages in modo sincrono

Consumer asincrono

```
public class Listener implements MessageListener {  
    public static void main(String[] args) {  
        try{  
  
            Context jndiContext = new InitialContext();  
  
            ConnectionFactory connectionFactory = (ConnectionFactory)  
                jndiContext.lookup("jms/javaee7/ConnectionFactory");  
            Destination queue = (Destination)  
                jndiContext.lookup("jms/javaee7/Queue");  
            try(JMSContext context =  
                connectionFactory.createContext()) {  
                context.createConsumer(queue).setMessageListener(  
                    new Listener());  
            }catch(NamingException e) {  
                e.printStackTrace();  
            }  
  
            public void onMessage(Message message) {  
                System.out.println("Async Message received:"+  
                    message.getBody(String.class));  
            }  
        }  
    }  
}
```

- › Implementa interfaccia per listener
- › Metodo statico
- › Preleva il contesto JNDI
- › Lookup per gli oggetti administered
- › Con la factory di connessioni JMS
- › Crea un oggetto di tipo Listener e lo registra
- › All'arrivo di un messaggio, questa è la callback

Il Consumer è un Message Listener

Messaging

Come assicurare un recapito affidabile

- JMS definisce diversi livelli di affidabilità per assicurare che i messaggi siano instradati correttamente
 - anche se il provider va in crash o è sotto carico elevato
 - o se le destination hanno messaggi che dovrebbero essere expired
- I meccanismi sono i seguenti
 - **Filtering messages:** usando i selector è possibile ricevere solo i messaggi che si desiderano
 - **Setting message time-to-live:** Scegliere il time-to-live (expiration time) in modo da non instradare messaggi se obsoleti
 - **Specifying message persistence:** specificare la persistenza di messaggi (nonostante possibili malfunzionamento del provider)
 - **Controlling acknowledgment:** controllo degli ack a vari livelli
 - **Creating durable subscribers:** assicurare instradamento di messaggi verso un “unavailable” subscriber in un pub-sub model
 - **Setting priorities:** Priorità di messaggi

Filtering dei messaggi

- Si fa in modo che arrivino **solo** i messaggi a cui si è interessati
 - Messaggio mandato in broadcast a diversi client, si definisce un selector in modo che venga consumato solo da consumer interessati
- Nessuno spreco di tempo e banda per ricevere cose non di interesse
- Si può fare selezione su headers o metadati (JMSPriority < 6) o su proprietà custom (orderAmount < 200)
- Il message selector è una stringa che contiene una espressione:

```
context.createConsumer(queue, "JMSPriority < 6").receive();
context.createConsumer(queue, "JMSPriority < 6 AND orderAmount < 200").receive();
context.createConsumer(queue, "orderAmount BETWEEN 1000 AND 2000").receive();
```

- Il messaggio viene creato dal Producer usando metodi per settare proprietà e priorità (nell'header)

```
context.createTextMessage().setIntProperty("orderAmount", 1530);
context.createTextMessage().setJMSPriority(5);
```

Producer

```
context.createTextMessage().setIntProperty("orderAmount", 1530);
context.createTextMessage().setJMSPriority(5);
```

Consumer

```
context.createConsumer(queue, "JMSPriority < 6").receive();
context.createConsumer(queue, "JMSPriority < 6 AND orderAmount < 200").receive();
context.createConsumer(queue, "orderAmount BETWEEN 1000 AND 2000").receive();
```

- Selector expression possono usare:
 - logical operators (NOT, AND, OR)
 - comparison operators (=, >, >=, <, <=, <>)
 - Arithmetic operators (+, -, *, /)
 - expressions ([NOT] BETWEEN, [NOT] IN, [NOT] LIKE, IS [NOT] NULL)

- and so on.

Evitare messaggi obsoleti:

- Un setting del time-to-live può essere di aiuto per evitare che messaggi obsoleti vengano recapitati ai destinatari
 - Si setta il tempo in millisecondi, passato il quale il provider (il broker) rimuove il messaggio
 - Si utilizza il metodo del producer:

```
context.createProducer().setTimeToLive(1000).send(queue, message);
```

Gestire la persistenza

- JMS supporta 2 modalità di message delivery: persistent e nonpersistent
 - Persistent delivery: messaggio salvato sul provider
 - Non-persistent delivery: messaggio non salvato
 - Persistent delivery è il valore di default... che può essere “degradato” per migliorare le prestazioni

```
context.createProducer().setDeliveryMode(DeliveryMode.NON_PERSISTENT).send(queue, message);
```

Controllo degli Acknowledgment

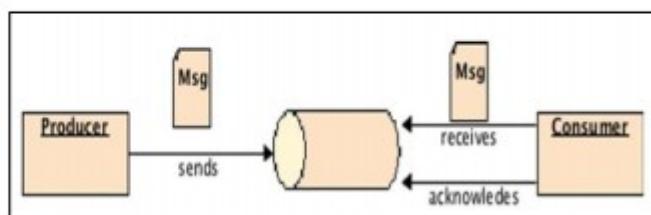
- Si vuole ricevere una verifica del recapito del messaggio al destinatario
- Diverse modalità di acknowledgment:
 - AUTO_ACKNOWLEDGE: la sessione automaticamente fa ack di un messaggio
 - CLIENT_ACKNOWLEDGE: ack esplicito del client
 - chiamando il metodo Message.acknowledge()
- Esempio di Producer che usa l'annotazione @JMSessionMode per settare

```
// Producer
@Inject
@JMSConnectionFactory("jms/connectionFactory")
@JMSessionMode(JMSContext.AUTO_ACKNOWLEDGE)
private JMSContext context;
```

l'acknowledgment mode

```
...
context.createProducer().send(queue, message);
```

```
// Consumer
message.acknowledge();
```



- Esempio di consumer che fa esplicito acknowledges del messaggio chiamando il metodo acknowledge()

Durable Consumer

- Nel modello publish-subscribe un consumer che non è in esecuzione perde i messaggi che vengono postati sul topic
- Con i durable consumer si può controllare che i messaggi vengano mantenuti dal provider fino a quando tutti i consumer li hanno ricevuti

- Con i durable subscribers, un consumer che si riconnette riceve i messaggi che sono arrivati durante la disconnessione
- Creazione attraverso JMSContext con una id specifica “unica”

```
context.createDurableConsumer(topic, "uniqueID").receive();
```

```
context.createDurableConsumer(topic, "javaee7DurableSubscription").receive();
```

- A questo punto il client inizia la connessione e riceve messaggi
- Il nome "**uniqueID**" (nel nostro esempio `javaee7DurableSubscription`) è usato come identificatore della durable subscription
- Ogni durable consumer deve avere un unique ID
 - che corrisponde alla dichiarazione di un'unica connection factory
 - per ogni potenziale durable consumer

Scegliere priorità del messaggio

- Contenute nell'header del messaggio
- Valori da 0 (bassa priorità) a 9 (alta priorità)
- Esempio:

```
context.createProducer().setPriority(2).send(queue, message);
```

- Concatenazione di diversi meccanismi:

```
context.createProducer().setPriority(2)
    .setTimeToLive(1000)
    .setDeliveryMode(DeliveryMode.NON_PERSISTENT)
    .send(queue, message);
```

EJB che scambiano messaggi

- Un Message Driven Bean (MDB) è un consumatore di messaggi, asincrono, invocato dal container quando arriva un messaggio
- Parte delle specifiche di Enterprise JavaBeans: simili a stateless
- Tramite CDI può accedere a altri EJB, JDBC, risorse JMS, entity manager, etc.
- Perché usare un MDB anziché un JMS Client?
 - Il vantaggio di usare un MDB (rispetto ad un JMS Client) è che transazione, multithread, sicurezza, etc., sono gestiti dal container

```
@MessageDriven(mappedName =
    "jms/javaee7/Topic")
public class BillingMDB
    implements MessageListener

    public void onMessage(Message message)
    {
        System.out.println("Received:"
            + message.getBody(String.class));
    }
}
```

> Definisce un MDB
 > Implementa un
 MessageListener e quindi il
 metodo `onMessage()`

Cosa fare quando si riceve
 un messaggio

Come è fatto un MDB

- MDB non è parte del modello EJB Lite: serve una implementazione full EE
- Annotazione con **@javax.ejb.MessageDriven** (o XML equivalente)
- Implementare la interfaccia del listener
- Definita come public, non final o abstract
- Deve esserci un costruttore senza argomenti, per permetterne l'istanziazione automatica da parte del container
- La classe non deve avere il metodo **finalize()**

```
@MessageDriven(mappedName = "jms/javaee7/Topic",
    activationConfig={
        @ActivationConfigProperty(propertyName="acknowledgeMode",
            propertyValue = "Auto-acknowledge"),
        @ActivationConfigProperty(propertyName="messageSelector",
            propertyValue = "orderAmount < 3000")
    })
public class BillingMDB implements MessageListener {
    public void onMessage(Message message) {←
        System.out.println("Message received:"+
            message.getBody(String.class));
    }
}
```

- › Nella definizione del MDB
- › Viene definita la configurazione ...
- › ... con le sue proprietà
- › ... filtro di messaggi
- › **Metodo per gestire i messaggi**

MDB context: MessageDrivenContext

- Questa interfaccia fornisce accesso al runtime context, che il container fornisce per una istanza di un MDB
- Il container passa l'interfaccia MessageDrivenContext all'istanza, che rimane associata per il lifetime dell'MDB
- Permette all'MDB di fare roll back di una transazione, ottenere il caller (user principal), ecc.

MDB as a consumer

- Per natura, gli MDBs sono progettati per funzionare come asynchronous message consumers
- Gli MDBs implementano una message listener interface, che viene “risvegliata” (triggered) dal container quando un messaggio arriva
- Può un MDB essere un synchronous consumer?
 - SI, ma non è raccomandato
 - Synchronous message consumers bloccano le risorse del server (gli EJBs si bloccheranno in un loop senza eseguire nessun lavoro ed il container non sarà in grado di liberarli)
 - Gli MDBs, come gli stateless session beans, vivono in un pool di una certa taglia
 - Quando il container ha bisogno di una istanza la rende dal pool e la usa
 - Se l'istanza va in un loop infinito, il pool si svuoterà e tutte le risorse saranno bloccate in un busy looping
 - L'EJB container può generare nuove istanze di MDB incrementando il pool ma aumentando così il consumo di memoria
 - Per questa ragione, session beans e MDBs non dovrebbero essere usati come synchronous message consumers
- Gli MDBs possono ANCHE diventare message producers
 - Workflow che prevede che essi ricevano messaggi da una destinazione, li processino, e li rinviano ad un'altra destinazione
- Per aggiungere questa capacità bisogna usare le API JMS
- Vediamo un esempio....

```

@MessageDriven(mappedName ="jms/javaee7/Topic",
activationConfig = {
    @ActivationConfigProperty(propertyName="acknowledgeMode",
        propertyValue ="Auto-acknowledge"),
    @ActivationConfigProperty(propertyName="messageSelector",
        propertyValue ="orderAmount BETWEEN 3 AND 7")
})
public class BillingMDB implements MessageListener {
    @Inject
    @JMSConnectionFactory("jms/javaee7/ConnectionFactory")
    @JMSSessionMode(JMSContext.AUTO_ACKNOWLEDGE)
    private JMSContext context;

    @Resource(lookup ="jms/javaee7/Queue")
    private Destination printingQueue;

    public void onMessage(Message message) {
        System.out.println("Message received:"
            + message.getBody(String.class));
        sendPrintingMessage();
    }

    private void sendPrintingMessage() throws JMSException
    { context.createProducer().send(printingQueue,
        "Message has been received and resent");
    }
}

```

- MDB con il topic
- Configurazione
- Auto-ack
- Filtro messaggi
- Implementa listener
- Context iniettato dalla CF
- ...nella variabile
- Destinazione iniettata dal container
- Metodo quando riceve messaggi
- Metodo per inviare

Transazioni... anche per i messaggi

- Transazioni per scambio di messaggi: un certo numero di messaggi vanno recapitati tutti insieme o nessuno
- In quanto EJB, le transazioni possono essere Bean-managed oppure Containermanaged

SOA (Service-Oriented Architecture)

Con **Service-Oriented Architecture (SOA)** si indica generalmente un'architettura software adatta a supportare l'uso di servizi Web per garantire l'interoperabilità tra diversi sistemi così da consentire l'utilizzo delle singole applicazioni come *componenti* del processo di business e soddisfare le richieste degli utenti in modo integrato e trasparente. L'architettura è basata anche sui servizi, i servizi hanno l'obiettivo di incapsulare una ben precisa funzionalità di business (logica applicativa), per renderla

disponibile e accessibile come servizio software da parte di client software sul web – ciascun servizio può essere usato per costruire diverse applicazioni e processi di business. In più permette anche l'integrazione dei processi di business all'interno dell'organizzazione.

I servizi sono di tipo self-contained: cioè deployati su una piattaforma di middleware standard che si occupa di descriverli, pubblicarli, localizzarli ed invocarli.

Abbiamo invece tre tipi di interoperabilità:

- Indipendenti dal contesto: service provider e consumer sono debolmente accoppiati.
- Indipendenti dalle specifiche tecnologiche o sistemi operativi.
- Indipendenti dai linguaggi di programmazione utilizzati.

Nota: Differenza JavaRMI e WebService

Un WebService può essere implementato da qualunque linguaggio, mentre Java RMI solo in linguaggio Java.

Caratteristiche dei servizi

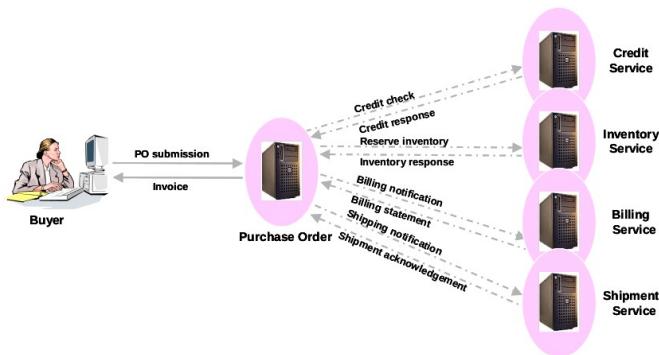
I servizi hanno tre caratteristiche:

- Tecnologia neutra: devono essere invocati attraverso tecnologie standard a minimo comune denominatore disponibili per quasi tutti gli ambienti IT. Ciò implica che i meccanismi di

chiamata (protocolli, descrizioni e meccanismi di scoperta) dovrebbero essere conformi agli standard ampiamente accettati.

- Loosely coupled: non devono richiedere conoscenze o strutture interne o convenzioni (contesto) dal lato del cliente o del servizio.
- Supportare la trasparenza della posizione: i servizi devono avere le loro definizioni e le informazioni sulla posizione memorizzate in un repository come UDDI ed essere accessibili da una varietà di client che possono localizzare e invocare i servizi indipendentemente dalla loro posizione.

I servizi possono essere implementati su una singola macchina o su un gran numero e varietà di dispositivi, e possono essere distribuiti su una rete locale o più ampiamente su varie reti geografiche (incluse reti mobili e ad hoc). I servizi Web costituiscono un'infrastruttura computerizzata composta da molti moduli diversi che cercano di comunicare attraverso la rete per formare virtualmente un singolo sistema logico. I servizi Web sono applicazioni modulari, auto-descrittive e autonome accessibili su Internet. Sono la risposta ai problemi di rigide implementazioni di relazioni predefinite e servizi isolati sparsi su Internet. Un servizio Web è un servizio disponibile tramite una rete come Internet che completa le attività, risolve i problemi o conduce transazioni.



Altre componenti di SOA

Oltre a questi componenti, un SOA può usare altri elementi che forniscono servizi infrastrutturali – ad esempio

- un registry dei servizi
- un server per la composizione e l'orchestrazione di servizi
- un enterprise service bus, per l'invocazione di servizi

Il collegamento tra servizi può avvenire sulla base di diversi tipi di connettori: ad esempio, SOAP, REST oppure l'uso di messaggi asincroni.

Application service provider

L'**application service provider** (ASP) è un modello architettonale per l'erogazione di servizi informatici che prevede una spinta remotizzazione elaborativa ed applicativa. Spesso il termine è usato indicando l'erogazione di servizi informatici in "modalità ASP".

Il modello architettonale prevede che la tecnologia di elaborazione (hardware) e quella applicativa (software) vengano gestite centralmente presso un service provider lasciando all'utente finale la scelta dei tempi e dei modi di fruizione del servizio. Tipicamente, lo strumento software lato cliente che funge da interfaccia con il servizio applicativo è il web browser.

I vantaggi sostanziali di un tale tipo di servizio si ritrovano in un risparmio di costi da parte del cliente (es. manutenzione hardware e software on-site) che dovrà pagare l'utilizzo del servizio che include i costi di licenza dei programmi, di manutenzione dell'hardware ecc.

I dati elaborati dal cliente possono venir memorizzati sull'infrastruttura storage del fornitore oppure localmente. Il termine ASP è stato rimpiazzato in tempi recenti da SaaS (Software as a service).

Software as a service

Software as a service (SaaS), o **Software come servizio**, è un modello di distribuzione del software applicativo dove un produttore di software sviluppa, opera (direttamente o tramite terze parti) e gestisce un'applicazione web che mette a disposizione dei propri clienti via Internet. I clienti utilizzano il software tramite API accessibili via web, spesso come Web service o REST.

Le principali applicazioni attuali di questo modello sono:

- la gestione delle relazioni coi clienti (CRM)
- le Enterprise resource planning (ERP)
- la video conferenza
- la gestione delle risorse umane
- le comunicazioni unificate (Unified Communications)
- il lavoro collaborativo
- le e-mail

Software as a Service (SaaS) può essere visto come il lato *marketing* del Service-Oriented Architecture (SOA). Il termine SaaS è diventato il termine di riferimento, rimpiazzando i precedenti termini Application service provider (ASP) e On-Demand.

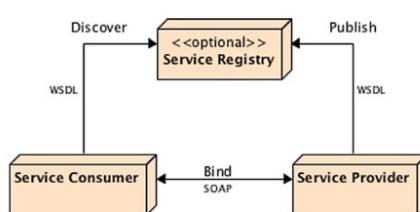
SOAP Web Services

Il termine web services implica "qualcosa" accessibile sul "web" che ti dà un "servizio". Il termine "servizi web" divenne rapidamente una parola d'ordine, assimilato a Service Oriented Architecture (SOA) e oggi i servizi web fanno parte della nostra vita architettonica quotidiana. Le applicazioni di servizi Web possono essere implementate con tecnologie diverse come SOAP, descritte in questo capitolo, o REST.

Si dice che i servizi web SOAP (Simple Object Access Protocol) siano "vagamente accoppiati" perché il client, ovvero il consumatore, di un servizio web non si devono conoscere i dettagli d'implementazione (come il linguaggio usato per svilupparlo, la piattaforma su cui gira). Il consumatore è in grado di invocare un servizio Web SOAP utilizzando un'interfaccia intuitiva che descrive i metodi commerciali disponibili (parametri e valore di ritorno). L'implementazione sottostante può essere eseguita in qualsiasi linguaggio (Visual Basic, C #, C, C ++, Java, ecc.). Un consumatore e un fornitore di servizi sarà ancora in grado di scambiare dati in un modo liberamente accoppiato: utilizzando documenti XML. Un consumatore invia una richiesta a un servizio Web SOAP sotto forma di un documento XML e, facoltativamente, riceve una risposta, anche in XML. I servizi Web SOAP riguardano anche la distribuzione. Il software distribuito è in circolazione da molto tempo, ma, a differenza sistemi distribuiti esistenti, i servizi Web SOAP sono adattati al Web. Il protocollo di rete predefinito è HTTP, un protocollo stateless noto e robusto.

In parole semplici, i servizi Web SOAP costituiscono una sorta di logica aziendale esposta tramite un servizio (ad esempio, il fornitore di servizi) a un cliente (ad esempio, il consumatore del servizio). Tuttavia, a differenza degli oggetti o EJB, i servizi Web SOAP forniscono un'interfaccia liberamente accoppiata utilizzando XML. Gli standard del servizio Web SOAP specificano che l'interfaccia a cui viene inviato un messaggio deve definire il formato della richiesta e della risposta del messaggio e i meccanismi per pubblicare e scoprire le interfacce dei servizi Web (il registro del servizio).

Nella Figura è possibile vedere un'immagine di alto livello dell'interazione di un servizio Web SOAP. Il servizio Web SOAP può facoltativamente registrare la propria interfaccia in un registro (Universal Description Discovery and Integration, o UDDI) in modo che un utente possa scoprirla. Una volta che il consumatore conosce l'interfaccia del servizio e il formato del messaggio, può inviare una richiesta al fornitore di servizi e ricevere una risposta.



I servizi Web SOAP dipendono da diverse tecnologie e protocolli per il trasporto e la trasformazione dei dati da un consumatore a un fornitore di servizi in un modo standard. Quelli che incontrerai più spesso sono i seguenti:

- L'XML (Extensible Markup Language) è il fondamento di base su cui vengono creati e definiti i servizi Web SOAP (SOAP, WSDL e UDDI).
- WSDL (Web Services Description Language) definisce il protocollo, l'interfaccia, i tipi di messaggi e le interazioni tra il consumatore e il fornitore.
- Il protocollo SOAP (Simple Object Access Protocol) è un protocollo di codifica dei messaggi basato su tecnologie XML, che definisce una busta per la comunicazione dei servizi Web.
- I messaggi vengono scambiati utilizzando un protocollo di trasporto. Sebbene Hypertext Transfer Protocol (HTTP) sia il protocollo di trasporto più utilizzato, è possibile utilizzare anche altri come SMTP o JMS.
- Universal Description Discovery, and Integration (UDDI) è un meccanismo di rilevamento e di rilevamento dei servizi opzionale, simile alle Pagine Gialle; può essere utilizzato per archiviare e classificare interfacce di servizi Web SOAP (WSDL).

Con queste tecnologie standard, i servizi Web SOAP offrono un potenziale quasi illimitato. I client possono chiamare un servizio, che può essere associato a qualsiasi programma e adattarsi a qualsiasi tipo di dati e struttura per lo scambio di messaggi tramite XML.

XML

Poiché XML è la perfetta tecnologia di integrazione che risolve il problema dell'indipendenza e dell'interoperabilità dei dati, è il DNA dei servizi Web SOAP. Viene utilizzato non solo come formato del messaggio ma anche come il modo in cui i servizi sono definiti (WSDL) o scambiati (SOAP). Associati con questi documenti XML, gli schemi (XSD) sono usati per convalidare i dati scambiati tra il consumatore e il fornitore. Storicamente, i servizi Web SOAP si sono evoluti dall'idea di base di "RPC (Remote Procedure Call) utilizzando XML".

WSDL

WSDL è il linguaggio di definizione dell'interfaccia (IDL) che definisce le interazioni tra consumatori e servizi Web SOAP. È fondamentale per un servizio Web SOAP poiché descrive il tipo di messaggio, la porta, il protocollo di comunicazione, le operazioni supportate, la posizione e ciò che il consumatore deve aspettarsi in cambio. Definisce il contratto a cui il servizio garantisce che si conformerà. Si può pensare a WSDL come a un'interfaccia Java ma scritta in XML.

Per garantire l'interoperabilità, è necessaria un'interfaccia standard di servizi Web per consentire a un consumatore e a un produttore di condividere e comprendere un messaggio. Questo è il ruolo di WSDL.



Il Listato seguente mostra un esempio di WSDL che rappresenta un servizio Web SOAP di convalida della carta di credito (questo servizio accetta una carta di credito come input e la convalida). Questo WSDL è puro XML e, se si legge attentamente, verranno visualizzate tutte le informazioni necessarie per un utente per individuare un servizio Web (soap: posizione dell'indirizzo), richiamare un

metodo (operazione nome = "convalida") e utilizzare un'appropriata protocollo di trasporto (sapone: trasporto obbligatorio).

```

<input>
  <soap:body use="literal"/>
</input>
<output>
  <soap:body use="literal"/>
</output>
</operation>
</binding>
<service name="CardValidatorService">
  <port name="CardValidatorPort" binding="tns:CardValidatorPortBinding">
    <soap:address location="http://localhost:8080/chapter14/CardValidatorService"/>
  </port>
</service>
</definitions>

```

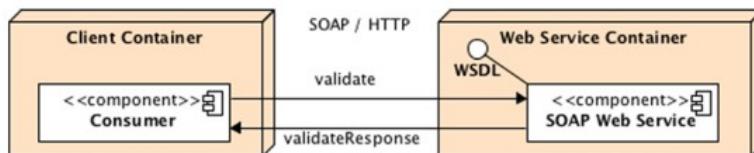
WSDL utilizza l'XML per descrivere cosa fa un servizio, come richiamare le sue operazioni e dove trovarlo. Segue una struttura fissa contenente diverse parti (tipi, messaggi, portType, binding, servizio). La Tabella 14-1 elenca un sottoinsieme degli elementi e degli attributi WSDL. WSDL è un linguaggio molto più ricco di quello elencato

nella Tabella 14-1 ed è definito dal W3C. Se vuoi saperne di più su WSDL, la sua struttura e i suoi tipi di dati, dovresti controllare il relativo sito web W3C.

Element	Description
definitions	Is the root element of the WSDL, and it specifies the global declarations of namespaces that are visible throughout the document
types	Defines the data types to be used in the messages. In this example, it is the XML Schema Definition (CardValidatorService?xsd=1) that describes the parameters passed to the web service request and the response
message	Defines the format of data being transmitted between a web service consumer and the web service itself. Here you have the request (the validate method) and the response (validateResponse)
portType	Specifies the operations of the web service (the validate method). Each operation refers to an input and output message
binding	Describes the concrete protocol (here SOAP) and data formats for the operations and messages defined for a particular port type
service	Contains a collection of <port> elements, where each port is associated with an endpoint (a network address location or URL)
port	Specifies an address for a binding, thus defining a single communication endpoint

SOAP

WSDL descrive un'interfaccia astratta del servizio web mentre SOAP fornisce un'implementazione concreta, definendo i messaggi XML scambiati tra il consumatore e il provider (vedi Figura). SOAP è il protocollo dell'applicazione standard per i servizi Web. Fornisce il meccanismo di comunicazione per connettere servizi Web, scambiando dati XML formattati attraverso un protocollo di rete, comunemente HTTP. Come il WSDL, SOAP fa molto affidamento su XML perché un messaggio SOAP è un documento XML che contiene diversi elementi (una busta, un'intestazione, un corpo, ecc.). Invece di usare HTTP per richiedere una pagina web da un browser, SOAP invia un messaggio XML tramite una richiesta HTTP e riceve una risposta tramite una risposta HTTP.



SOAP è progettato per fornire un protocollo di comunicazione indipendente e astratto in grado di connettere servizi distribuiti. I servizi connessi possono essere creati utilizzando qualsiasi combinazione di hardware e software che supporti un determinato protocollo di trasporto.

Usiamo l'esempio di convalida della carta di credito. Il consumatore chiama il servizio web SOAP per convalidare una carta di credito superando tutti i parametri necessari

Listing 14-3. The SOAP Envelope Sent for the Request

```

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" xmlns:cc="http://chapter14.javaee7.book.agoncal.org/">
  <soap:Header/>
  <soap:Body>
    <cc:validate>
      <argo number="123456789011" expiry_date="10/12" control_number="544" type="Visa"/>
    </cc:validate>
  </soap:Body>
</soap:Envelope>

```

Listing 14-4. The SOAP Envelope Received for the Response

```

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" xmlns:cc="http://chapter14.javaee7.book.agoncal.org/">
  <soap:Body>
    <cc:validateResponse>
      <return>true</return>
    </cc:validateResponse>
  </soap:Body>
</soap:Envelope>

```

(numero di carta di credito, data di scadenza, tipo e numero di controllo) e riceve un booleano che informa il consumatore se la carta è valida o meno. Gli elenchi 14-3 e 14-4, rispettivamente, mostrano la struttura di questi due messaggi SOAP.

Table 14-2. SOAP Elements and Attributes

Element	Description
Envelope	Defines the message and the namespace used in the document. This is a required root element
Header	Contains any optional attributes of the message or application-specific infrastructure such as security information or network routing
Body	Contains the message being exchanged between applications
Fault	Provides information about errors that occur while the message is processed. This element is optional

UDDI

I consumatori e i fornitori che interagiscono tra loro sul Web devono essere in grado di trovare informazioni che consentano loro di interconnettersi. O il consumatore conosce la posizione esatta del servizio che desidera invocare o deve trovarlo. UDDI fornisce un approccio standard per individuare le informazioni su un servizio Web e su come richiamarlo. Il provider di servizi pubblica un WSDL in un registro UDDI disponibile su Internet. Quindi può essere scoperto e scaricato dai potenziali consumatori. È opzionale in quanto è possibile richiamare un servizio Web senza UDDI se si conosce già la posizione del servizio Web.

UDDI è un registro basato su XML di servizi Web, simile a una directory Pagine Gialle, in cui le aziende possono registrare i loro servizi. Questa registrazione include il tipo di attività, la posizione geografica, il sito web, il numero di telefono e così via. Altre aziende possono quindi cercare nel registro e scoprire informazioni su specifici servizi web. Queste informazioni forniscono ulteriori metadati sul servizio, descrivendo il suo comportamento e la posizione effettiva del documento WSDL.

Writing SOAP Web Services

Poiché il documento WSDL è il contratto tra il consumatore e il servizio, può essere utilizzato per generare il codice Java per il consumatore e il servizio. Questo è l'approccio top-down, noto anche come contratto. Questo approccio inizia con il contratto (WSDL) definendo operazioni, messaggi e così via. Quando consumatore e fornitore concordano sul contratto, è quindi possibile implementare le classi Java in base a tale contratto. Metro fornisce alcuni strumenti (wsimport) che generano classi da un WSDL.

Con l'altro approccio, chiamato bottom-up, la classe di implementazione esiste già e tutto ciò che è necessario è creare il WSDL. Ancora una volta, Metro fornisce utility (wsgen) per generare un WSDL dalle classi esistenti. In entrambi i casi, potrebbe essere necessario modificare il codice per adattarlo al WSDL o viceversa. JAX-WS viene in nostro aiuto. Con un modello di sviluppo semplice e alcune annotazioni, è possibile modificare la mappatura da Java a WSDL.

Nonostante tutte queste specifiche, concetti, standard e organizzazioni, scrivere e consumare un servizio web nell'approccio top-down è molto semplice. I servizi Web SOAP seguono il paradigma "facilità di sviluppo" di Java EE 7 e non richiedono la scrittura di alcun WSDL o SOAP. Il servizio Web è solo un POJO annotato che deve essere distribuito in un contenitore di servizi Web. Il Listato 14-5 mostra il codice di un servizio web che convalida una carta di credito.

```
@WebService //definisce la classe come webservice
public class CardValidator {
    public boolean validate(CreditCard creditCard) {
        Character lastDigit = creditCard.getNumber().charAt(
            creditCard.getNumber().length() - 1);
        if (Integer.parseInt(lastDigit.toString()) % 2 != 0) {
            return true;
        } else {
            return false;
        }
    }
}
```

```

}

Visto che il metodo richiama una carta di credito, ci serve anche la sua definizione in un documento XML:
@XmlRootElement // mappa la classe in formato xml
public class CreditCard {
    @XmlAttribute(required = true) // mappa l'attributo come tag xml
    private String number;
    @XmlAttribute(name = "expiry_date", required = true)
    private String expiryDate;
    @XmlAttribute(name = "control_number", required = true)
    private Integer controlNumber;
    @XmlAttribute(required = true)
    private String type;
    // Constructors, getters, setters
}

```

Come la maggior parte dei componenti Java EE 7, i servizi Web SOAP si basano sul paradigma di configurazione per eccezione, che specifica che la configurazione di un componente è un'eccezione. È necessaria solo un'annotazione per trasformare un POJO in un servizio web SOAP cioè

`@WebService`. I requisiti per scrivere un servizio Web sono i seguenti:

- La classe deve essere annotata con `@ javax.jws.WebService` o l'equivalente XML in un descrittore di distribuzione (`webservices.xml`).
- La classe (a.k.a bean di implementazione del servizio) può implementare zero o più interfacce (a.k.a interfaccia endpoint del servizio) che devono essere annotate con `@WebService`.
- La classe deve essere definita come pubblica e non deve essere definitiva o astratta.
- La classe deve avere un costruttore pubblico predefinito.
- La classe non deve definire il metodo `finalize ()`.
- Per trasformare un servizio Web SOAP in un endpoint EJB, la classe deve essere annotata con `@ javax.ejb.Stateless` o `@ javax.ejb.Singleton` (consultare il Capitolo 7).
- Un servizio deve essere un oggetto stateless e non deve salvare lo stato specifico del client attraverso le chiamate di metodo.

Un Web Services può essere anche un EJB aggiungendo l'annotazione `@Stateless`, `@Statefull`.

Web Service

I Web Services, a livello di sistema, sono definiti in termini di XML, cioè operazioni definite da WSDL, scambiando messaggi SOAP. A livello di programmazione Java, le applicazioni sono definite in termini di oggetti, interfacce e metodi. Una traduzione da oggetti Java objects ad operazioni WSDL è richiesta.

JAXB runtime usa le annotazioni per eseguire marshal/unmarshal di una classe da/verso XML. Allo stesso modo, JWS usa le annotazioni per mappare classi Java in WSDL e per definire come fare il marshal di una invocazione di un metodo in una richiesta SOAP l'unmarshal di una risposta SOAP in un'istanza del valore restituito dal metodo.

Annotazioni messe a disposizione

JAX-WS (JSR 224) e WS-Metadata specifications (JSR 181) definiscono due differenti tipi di annotazioni:

- WSDL mapping annotations: annotazioni che appartengono al package `javax.jws` e permettono il mapping WSDL/Java: `@WebMethod`, `@WebResult`, `@WebParam`, e `@OneWay` sono le annotazioni usate per personalizzare i metodi esposti.
- SOAP binding annotations: annotazioni che appartengono al package `javax.jws.soap` e permettono la personalizzazione di SOAP binding: `@SOAPBinding` e `@SOAPMessageHandler`.

@WebService

L'annotazione @javax.jws.WebService marca una classe o una interfaccia come un web service.

```
@WebService  
public class CardValidator {  
    //...  
}
```

Una classe

```
@WebService  
public interface Validator {  
    //...  
}  
@WebService(endpointInterface =  
        "org.agoncal.book.javaee7.chapter14.Validator")  
public class CardValidator implements Validator {  
    //...  
}
```

.... o interfaccia che è un WS

L'annotazione @WebService ha un insieme di attributi (Listing 14-8) che permettono di personalizzare:

- il nome del Web Service nel file WSDL
- la locazione

@WebService(portName ="CreditCardValidator", serviceName ="ValidatorService")

public class CardValidator {

//...

}

Versione originale

```
<service name="CardValidatorService">  
    <port name="CardValidatorPort" binding="tns:CardValidatorPortBinding">  
        <soap:address location="http://localhost:8080/chapter14/CardValidatorService"/>  
    </port>  
</service>  
</definitions>
```

Dopo la modifica

```
<service name="ValidatorService">  
    <port name="CreditCardValidator" binding="tns:CreditCardValidatorBinding">  
        <soap:address location="http://localhost:8080/chapter14/ValidatorService"/>  
    </port>  
</service>
```

@WebMethod

Di default, tutti i metodi di un SOAP web service sono esposti nel file WSDL e le regole di default vengono usate per il mapping. Per applicare personalizzazione si può usare l'annotazione @javax.jws.WebMethod sui metodi.

Nel nostro esempio:

- Due metodi verranno rinominati
- Un metodo verrà escluso dal file WSDL

```
@WebService  
public class CardValidator {  
    @WebMethod(operationName = "ValidateCreditCard")  
    public boolean validate(CreditCard creditCard) {  
        //Business logic  
    }  
  
    @WebMethod(operationName = "ValidateCreditCardNumber")  
    public void validate(String creditCardNumber) {  
        //Business logic  
    }  
  
    @WebMethod(exclude = true)  
    public void validate(Long creditCardNumber) {  
        //Business logic  
    }  
}
```

Listing 14-10. Fragment of WSDL with Renamed Methods

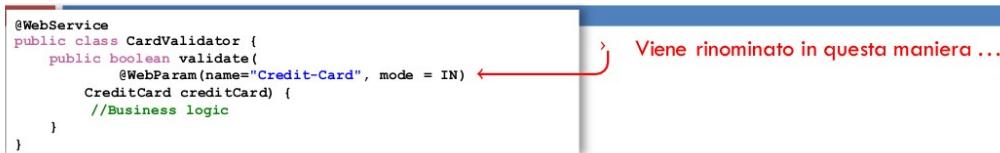
```
<message name="ValidateCreditCard">  
    <part name="parameters" element="tns:ValidateCreditCard"/>  
</message>  
    <message name="ValidateCreditCardResponse">  
        <part name="parameters" element="tns:ValidateCreditCardResponse"/>  
</message>  
    <message name="ValidateCreditCardNumber">  
        <part name="parameters" element="tns:ValidateCreditCardNumber"/>  
</message>  
    <message name="ValidateCreditCardNumberResponse">  
        <part name="parameters" element="tns:ValidateCreditCardNumberResponse"/>  
</message>  
    <portType name="CardValidator">  
        <operation name="ValidateCreditCard">  
            <input message="tns:ValidateCreditCard"/>  
            <output message="tns:ValidateCreditCardResponse"/>  
        </operation>
```

```
@WebService  
public class CardValidator {  
    @WebResult(name = "IsValid") ←  
    public boolean validate(CreditCard creditCard) {  
        //Business logic  
    }  
}
```

@WebParam

L'annotazione @javax.jws.WebParam permette di personalizzare i parametri dei metodi di un web service:

- Nomi dei parametri del file WSDL
- Il namespace
- Tipo



Dopo la modifica:

```
<!-- Default -->
<xss:element name="argo" type="tns:creditCard" minOccurs="0"/>
<!-- Renamed to Credit-Card -->
<xss:element name="Credit-Card" type="tns:creditCard" minOccurs="0"/>
```

SOAP binding

Un binding descrive come il web service è legato al protocollo di messaging (SOAP).

Esistono due differenti tipi di programming styles per SOAP binding definiti in WSDL 1.1:

- RPC e document (messaging), che influenzano il modo in cui il SOAP body content è strutturato. Document: Il messaggio SOAP contiene il documento. Inviato nel <soap:Body> element senza regole di formattazione addizionali. È la scelta di default.
- RPC: Il messaggio SOAP contiene i parametri ed i valori restituiti. Il <soap:Body> contiene un elemento con il nome del metodo remote o procedura da invocare. È presente un elemento per ogni parametro della procedura remota.

Un SOAP binding (Document or RPC) deve scegliere fra due differenti formati di serialization/deserialization:

- Literal: Data serializzati secondo un XML schema
- Encoded: il SOAP encoding specifica come oggetti, strutture, array devono essere serializzati

```
<!-- Document style -->
<message name="validate">
  <part name="parameters" element="tns:validate"/>
</message>
<message name="validateResponse">
  <part name="parameters" element="tns:validateResponse"/>
</message>
...
<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
```

```
<!-- RPC Style -->
<message name="validate">
  <part name="argo" type="tns:creditCard"/>
</message>
<message name="validateResponse">
  <part name="return" type="xsd:boolean"/>
</message>
...
<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
```

Gestione delle eccezioni

In Java, quando si verifica un errore, si lanciano le eccezioni e qualche altra classe dovrà gestirle. Con SOAP non si può procedere nello stesso modo perché Consumer e Service potrebbero essere scritti in linguaggi diversi e separati dalla rete. L'idea è quindi di usare SOAP fault in un SOAP message.

JAX-WS runtime automaticamente converte le eccezioni Java in SOAP fault messages, restituiti al client. Per far registrare delle nostre eccezioni automaticamente si deve usare l'annotazione

@WebFault.

Esempio:

```
@WebFault(name = "CardValidationFault")
public class CardValidatorException extends Exception {
    public CardValidatorRTException() {
```

Listing 14-24. SOAP Fault in a SOAP Envelope

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>org.agoncal.book.javaee7.chapter14.CardValidatorException</faultstring>
      <detail>
        <ns2:CardValidationFault xmlns:ns2="http://chapter14.javaee7.book.agoncal.org/">
          <message>The credit card number is invalid</message>
        </ns2:CardValidationFault>
      </detail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

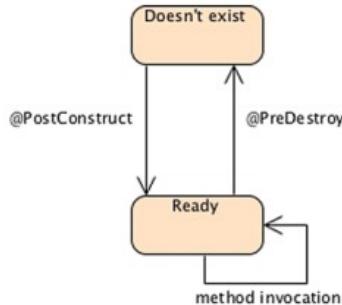
```

        super();
    }
    public CardValidatorRTEException(String message) {
        super(message);
    }
}

```

Ciclo di vita di un WS

I WS SOAP hanno un ciclo di vita simile a quelli dei beans. Metodi con PostConstruct vengono invocati dopo che il WS viene istanziato, mentre metodi con PreDestroy vengono invocati prima di chiuderlo. Differenza con EJB: gli EJB hanno anche gli interceptors (mentre i WS no).



Contesti in un WS

Un SOAP web service ha un environment context e può accedervi iniettando una reference di javax.xml.ws.WebServiceContext con l'annotazione **@Resource**. All'interno di questo contesto, il web service può ottenere informazioni di runtime : l'endpoint implementation class, il message context, security information circa la richiesta appena servita, ecc.

```

@WebService
public class CardValidator {
    @Resource
    private WebServiceContext context;
    public boolean validate(CreditCard creditCard) {
        if(!context.isUserInRole("Admin"))
            throw new SecurityException(
                "Only Admins can validate cards");
        //Business logic
    }
}

```

- › Un WS
- › Definizione classe
- › Iniezione del contesto di WS
- › Uso per validare il ruolo dell'utente
- › ... altrimenti si lancia una eccezione

Method	Description
getMessageContext	Returns the MessageContext for the request being served at the time this method is called. It can be used to access the SOAP message headers, body, and so on.
getUserPrincipal	Returns the Principal that identifies the sender of the request currently being serviced.
isUserInRole	Returns a Boolean indicating whether the authenticated user is included in the specified logical role.
getEndpointReference	Returns the EndpointReference associated with this endpoint.

Deployment descriptor

SOAP web services permette di definire i meta-dati usando le annotazioni oppure via XML. Il deployment descriptor, localizzato sotto la directory WEB-INF, si chiama webservices.xml. Sovrascrive oppure estende le annotazioni.

```

<webservices xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/webservices_1_3.xsd"
  version="1.3">
  <webservice-description>
    <webservice-description-name>
      CardValidatorWS
    </webservice-description-name>
    <port-component>
      <port-component-name>
        CardValidator
      </port-component-name>
      <wsdl-port>OverriddenPort</wsdl-port>
      <service-endpoint-interface>
        org.agonal.book.javaee7.chapter14.Validator
      </service-endpoint-interface>
      <service-impl-bean>
        <servlet-link>
          CardValidatorServlet
        </servlet-link>
      </service-impl-bean>
    </port-component>
  </webservice-description>
</webservices>

```

- › XML schema per la validazione
- › Nome del WS
- › Informazioni sulla porta
- › Interfaccia
- › **Servlet di implementazione**

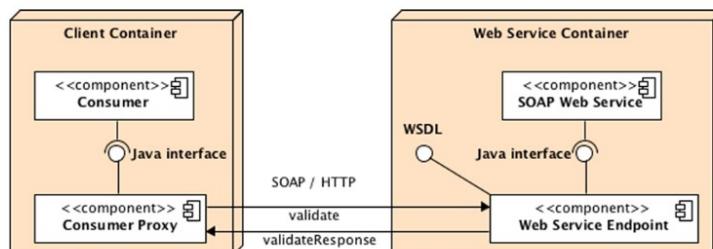
Come invocare un Web Service

Un Web Service si invoca usando il WSDL per descrivere il servizio e alcuni stub (proxy) generati automaticamente dal sistema; si può facilmente usare WS senza usare direttamente HTTP, SOAP, etc.

Una situazione abbastanza simile a quanto viene fatto per RMI. La differenza rispetto ad RMI è che, sul remote host, il Web service può essere scritto in un linguaggio diverso da Java.

WSDL è il contratto standard fra il consumer ed il servizio. Metro fornisce un utility tool WSDL-to-Java (wsimport) per generare interfacce e classi Java a partire da un file WSDL. Questi proxy sono la rappresentazione Java di un web service endpoint (servlet or EJB). Questi proxy instradano le chiamate locali Java al remote web service usando HTTP.

Quando un metodo sul proxy viene invocato converte i parametri del metodo in un messaggio SOAP (la richiesta). Invia la richiesta al web service endpoint per ottenere un risultato, la risposta SOAP viene convertita in una istanza del tipo restituito. Non si ha necessità di conoscere il “lavoro interno” del proxy, né il suo codice.



Poiché JAX-WS disponibile in Java SE, un SOAP web service consumer può essere:

- un qualunque tipo di Java code con una main class in esecuzione nella JVM
- una qualunque Java EE component in esecuzione in un container (Web, EJB o application client container)

Se eseguita in un container, il consumer può ottenere una istanza del proxy attraverso injection oppure programmatically (crendola). Per iniettare un web service, bisogna usare l'annotazione @javax.xml.ws.WebServiceRef annotation o un CDI producer.

Il

```
public class WebServiceConsumer {  
    public static void main(String[] args) {  
        CreditCard creditCard = new CreditCard();  
  
        creditCard.setNumber("12341234");  
        creditCard.setExpiryDate("10/12");  
        creditCard.setType("VISA");  
        creditCard.setControlNumber(1234);  
  
        CardValidator cardValidator =  
            new CardValidatorService().getCardValidatorPort();  
  
        cardValidator.validate(creditCard);  
    }  
}
```

- › Una classe standard
- › Metodo statico
- › Si dichiara una carta di credito
- › La si inizializza
- › Si ottiene un riferimento al proxy, da cui si ottiene un riferimento al service ...
- › ... che si invoca come un metodo "qualsiasi"

consumer usa una istanza di CardValidatorService (generata da WSDL grazie a wsimport) usando la keyword new. Con getCardValidatorPort() invoca il business method localmente. Una chiamata locale viene fatta sul metodo validate() del proxy, che invocherà il remote web service, creando la richiesta SOAP, facendo il marshal dei messaggi credit card messages, ecc. Il proxy trova il target service perchè il default endpoint URL è nel WSDL file, e quindi integrato nella implementazione proxy.

```
public class WebServiceConsumer {  
    @WebServiceRef  
    private static CardValidatorService cardValidatorService;  
  
    public static void main(String[] args) {  
        CreditCard creditCard = new CreditCard();  
  
        creditCard.setNumber("12341234");  
        creditCard.setExpiryDate("10/12");  
        creditCard.setType("VISA");  
        creditCard.setControlNumber(1234);  
  
        CardValidator cardValidator =  
            cardValidatorService.getCardValidatorPort();  
        cardValidator.validate(creditCard);  
    }  
}
```

- › Annotazione per l'iniezione Injection per ottenere un riferimento al SOAP Web Service client proxy
- › Dichiarazione di un proxy, iniettato dal container
- › Oggetto da inviare
- › Set degli attributi
- › Uso del proxy iniettato
- › ... per invocare il WS