Context and Dependency Injection

La prima versione di Java EE ha introdotto il contetto di **inversion of control (IoC),** ciò significa che il container assume il controllo del business code e fornisce servizi tecnici. Prendere il controllo ha significato gestire il ciclo di vita dei component, dependency injection e la configurazione dei components.

Java EE6 ha introdotto **Context and Dependency Injection** che trasforma quasi tutti i componenti Java EE in beans injectable (iniettabili), interceptable (intercettabili) e manageable (gestibili).

CDI è costruito sul concetto di "loose coupling, strong typing", ciò significa che **i bean sono debolmente accoppiati ma viene mantenuta la tipizzazione forte.** Il disaccoppiamento va ancora oltre grazie all'utilizzo di interceptor, decorator ed eventi sull'intera piattaforma.

Introduzione ai Beans

I POJO sono semplicemente classi java che vengono eseguite nella JVM. **I Java Bean sono POJO che seguono determinate convenzioni** (metodi getter e setter, costruttore di default) e vengono eseguiti nella JVM. Tutti gli altri components di Java EE

seguono un determinato pattern (ad esempio Enterprise JavaBean) e sono eseguiti in un container che fornisce dei servizi.
Abbiamo dunque **Managed Beans e Beans.**
I Managed Beans sono oggetti gestiti dal container che supportano solo alcuni servizi di base: come resource injection, gestione del ciclo di vita e interception.
Potenzialmente ogni classe Java che ha un costruttore di default e viene eseguito all'interno di un container è un bean, con pochissime eccezioni.

Dependency Injection
La DI è un design patter che disaccoppia componenti dipendenti e fa parte dell'inversione del controllo.
Invece di un oggetto che fa il lookup ad altri oggetti, il container inietta questi oggetti dipendenti. È il cosiddetto Principio di Hollywood, "Don't call us, we'll call yout".
La DI è stata introdotta in Java EE 5. Ha consentino agli sviluppatori di iniettare risorse del container come EJB, entity manager e data source in un insieme di componenti definiti. A questo capo, Java EE 5, ha introdotto un nuovo set di annotazioni.
Questo primo passo non era sufficiente, quindi Java

EE 6 ha creato due specifiche **Dependecy Injection e Contetexts and Dependency Injection.**

Life-Cycle Management

Se vogliamo eseguire un EJB in un container non possiamo usare new. È necessario iniettare il bean, e il container farà il restio. Il container è responsabile della gestione del ciclo di vita del bean: crea l'istanza e poi se ne sbarazza.

La figura mostra il ciclo di vita di un managed bean (e quindi un bean CDI). Quando invochiamo un bean, il container è responsabile della creazione dell'istanza, quindi risolve le dipendenze e invoca qualsiasi metodo annotato con @PostConstruct prima del primo richiamo del metodo di business del bean. Poi i metodi annotati con @PreDestroy vengono eseguiti prima che l'oggetto sia rimosso dal container.

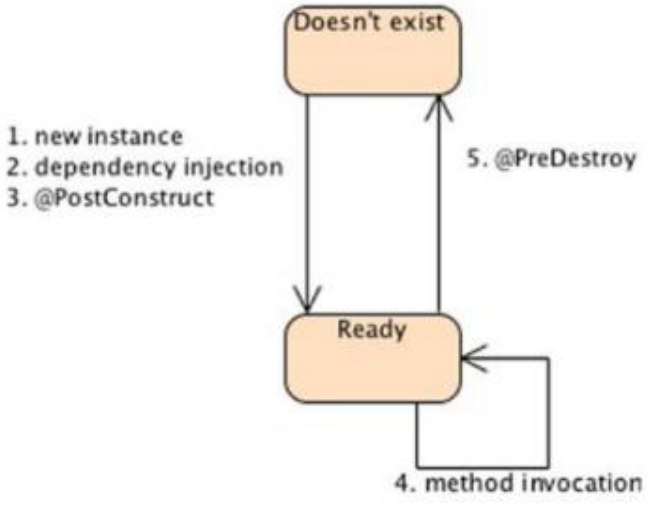
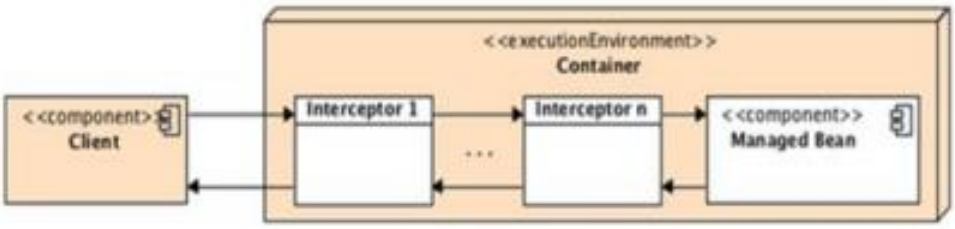


Figure 2-1. Managed Bean life cycle

Interception

Gli interceptor sono utilizzati per interporsi prima dell'esecuzione di un business method. Da questo punto di vista è simile alla programmazione orientata agli aspetti (AOP). L'AOP è un paradigma di programmazione che separa le preoccupazioni trasversali dal business code. **I Managed Beans supportano funzionalità AOP**, fornendo la possibilità di intercettare l'invocazione del metodo tramite Interceptors. Questi vengono automaticamente invocati dal container quando viene richiamato un metodo.

Come mostrato in figura gli Interceptor **possono essere concatenati e chiamati prima e/o dopo l'esecuzione di un metodo.**



Si potrebbe pensare a un container EJB come a una catena di interceptor. **Quando si sviluppa un session bean ci si può concentrare sul business code ma, dietro le quinte, quando un client invoca un metodo**

sull'EJB, il container intercetta l'invocazione e applica diversi servizi. Con gli interceptor, si aggiungono i propri meccanismi in modo trasparente al business code.

Deployment Descriptor

Con CDI, il Deployment Descriptor si chiama **beans.xml** ed è obbligatorio. Può essere utilizzato per configurare alcune funzionalità (Interceptors, Decorators, Alternatives, etc..) ma **è essenziale per abilitare CDI, questo perché il CDI ha bisogno di identificare i Beans nel class path.**
Senza un file beans.xml nel class path, CDI non sarà in grado di utilizzare Injection, Interception, Decoration e così via.

@Inject

Poiché Java EE è un ambiente maanged, non è necessario costruire manualmente le dipendenze, ma è possibile lasciare che sia il container a fare un inject. **Con CDI è possibile iniettare quasi tutto ovunque grazie all'annotazione @Inject.**
Quest'ultima informa il container che ha bisogno di iniettare un riferimento ad un'implementazione. Questo è chiamato **injection point**.
Consideriamo l'esempio in cui abbiamo bisogno di

iniettare un oggetto che estende l'interfaccia NumberGenerator. Supponiamo inoltre che NumberGenerator abbia solo un'implementazione (IsbnGenerator). Il CDI sarà quindi in grado di iniettarlo semplicemente usando @Inject.
Questa è chiamata **default injection.** Ogni volta che un bean o un punto di iniezione non dichiara esplicitamente un qualificatore, il container assume il qualificatore @javax.enterprise.inject.Default.
Infatti il codice:
@Inject
private NumberGenerator numberGenerator;
è identico a:
@Inject **@Default**
private NumberGenerator numberGenerator;
Se invece dobbiamo scegliere tra più implementazioni, servono i qualificatori.

Qualifiers

Al momento dell'inizializzazione del sistema, il container deve convalidare che esiste esattamente un bean che soddisfi ogni injection point. Ciò significa che se non è disponibile nessun implementazione di NumberGenerator, il container ti informerà di unsatisfied depency e non farà il deploy

dell’applicazione. Se esiste una sola implementazione, l’injection funzionerà utilizzando il qualificatore @Default. **Se sono disponibili più implementazioni predefinite, il container vi informerà di una dipendenza ambigua e non farà il deploy.** Questo perché il container non è in grado di identificare esattamente di quale bean fare l’inject. Quindi, come fa un componente a scegliere quale implementazione deve essere iniettata.
CDI utilizza i qualificatori, che sono sostanzialmente annotazioni Java che preservano la typesafe injection e disambiguano un tipo senza dover ricorrere a nomi basati su stringhe.
Un qualificatore rappresenta una semantica associata a un tipo che è soddisfatto da alcune implementazioni di questo genere.

Producers

Abbiamo visto come fare l’inject di Beans CDI in latri Beans CDI. **Grazie ai Producers, è anche possibile fare l’inject delle primitive, i tipi di array, e qualsiasi POJO che non è abilitato CDI.** Con CDI abilitato intendo qualsiasi classe racchiusa in un archivio contente un file beans.xml.

Scopes

Ogni oggetto gestito da CDI ha uno scope ben definito e un ciclo di vita vincolato ad un contesto specifico. **Con CDI, un bean è legato a un contesto e rimane in quel contesto finché non viene distrutto dal container.** CDI definisce i seguenti scope e offre anche extension point in modo da poterne creare altri:

- Application scope (@ApplicationScoped): si estende per l'intera durata di un'applicazione. Questo ambito è utile per le **classi di utilità o di supporto o per gli oggetti che memorizzano dati condivisi dall'intera applicazione.**
- Session scope (@SessionScoped): il bean viene creato per la durata di una sessione HTTP e viene eliminato quando la sessione termina.
- Request scope (@RequestScoped): il bean viene creato per la durata dell'invocazione del metodo e viene eliminato quando il metodo termina.
- Conversation scope (@ConversationScoped): le conversazioni vengono utilizzate in 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. Questo è lo scope di default per CDI.

Interceptors

Gli interceptors **consentono di aggiungere cross-cutting concernes**. Il container è in grado di intercettare la chiamata e elaborare la business logic prima che il metodo del bean sia invocato.
Gli Interceptors rientrano in **quattro tipi:**
- **Constructor-level interceptors:** interceptor a livello di costruttore (@AroundConstruct).
- **Method-level interceptors:** interceptor associati a uno specifico metodo business (@AroundInvoke).
- Timeout method interceptors: interceptors che si interpone sui metodi di timeout @AroundTimeout).
- **Life-cycle callback interceptors:** interceptor che si interpone sul ciclo di vita dell'istanza di destinazione even callback (@PostConstruct e @PreDestroy).

Decorators

I Decorators sono un design pattern della Gang of Four. L'idea è di prendere una classe e avvolgere intorno ad essa un'altra classe. In questo modo quando chiami una classe decorata prima di arrivare alla classe target si passa per il decoratore. I decorators servono ad aggiungere logica ai metodi di

business.
Prendiamo l’esempio di un generatore di numeri ISSN. ISSN è un numero di 8 cifre che è stato sostituito dal codice ISBN. Invece di avere due generatori di numeri separati puoi decorare il generatore ISSN per aggiungere un algoritmo in più che trasforma un numero di 8 cifre in un numero di 13 cifre. Facciamo un esempio:
La classe FromEightToThirteenDigitsDecorator è annotato con @Decorator, implementa le interfacce di business (NumberGenerator) e sostituisce il metodo generateNumber. Il metodo generateNumber() invoca il bean di destinazione per generare un ISNN, aggiunge alcune logiche di business per trasformare tale numero e restituisce un numero ISBN.
Un decorator deve avere un injection point annotato con @Delegate che è dello stesso tipo del bean che decora.

Eventi

Mentre interceptor e decorators generano comportamenti addizionali al momento dell’implementazione o in fase di esecuzione, **gli eventi non dipendono dal tempo di compilazione, e**

possono essere gestiti da bean diversi presenti anche in package separati o persino su strati diversi dell'applicazione. Tutto ciò seguendo uno schema che segue **l'Observer pattern.**
L'event producer fa partire gli event tramite l'interfaccia Event. Un producer provoca gli eventi chiamando fire().
Gli eventi sono fatti partire dagli event producer e sottoscritti dall'osservatore di eventi specifico come parametro annotato con l'**@Observes** e altri qualificatori opzionali, e al metodo Observer arriva una notifica di un event se l'oggetto event combacia.

Java Persistence API

Alcuni oggetti devono essere persistenti, ovvero devono essere memorizzati deliberatamente in forma permanente supporti magnetici, memorie flash e così via. Il principio della **mappatura oggetto-relazionale** (ORM) è quello di riunire il mondo del database e degli oggetti.
Iniziamo a parlare di "entità" e non più di "oggetti". Gli oggetti sono istanze che vivono solo nella memoria, mentre le entità vivono brevemente nella memoria ma in modo persistente in un database.
È possibile mantenere un'entità nel database,

rimuoverla e interrogarla utilizzando un linguaggio di query Java Persistence Query Language (JPQL).
Con i metodi di callback e i listeners, JPA ti consente di collegare alcuni business code agli eventi del ciclo di vita.
Un'entità è un oggetto Plain Old Java (POJO). Ciò significa che un'entità viene dichiarata, istanziata e utilizzata come qualsiasi altra classe Java. Un'entità ha attributi che possono essere manipolati tramite getter e setter.
Per essere mappato in una tavola basta aggiungere delle annotazioni.

Anatomy of an Entity
Una POJO per essere un'enetity deve seguire queste regole:
- La classe entità deve essere annotata con @Entity (o denotata nel descrittore XML come entità).
- **L'annotazione @Id deve essere utilizzata per indicare una semplice chiave primaria.**
- La classe entità deve avare un **costruttore senza parametri,** che deve essere pubblico o protetto.
- La classe entità deve essere una classe top-level. **Un enum o un'interfaccia non possono essere designati come un'entità.**

- La classe entità **non deve essere final. Nessun metodo o variabile di istanza persistente dell'entità devono essere final.**
- Se un'istanza di entità deve essere passata per valore come oggetto detached, la classe entità deve implementare l'interfaccia Serializzabile.

Object-Relational Mapping
JPA mappa gli oggetti in un Database attraverso i Metadata. Essi permettono al persistance provider di riconoscere le entità e interpretare il mapping.
Possono essere scritti in due differenti formati:
- **Annotazioni:** il codice dell'entità è annotata direttamente con tutte le possibili annotazioni descritte nel package javax.persistence.
- **XML descriptors:** insieme alle annotazioni, o al posto delle stesse, si possono usare gli xml descriptors. Il mapping è definito da un file xml esterno che verrà poi distribuito con le entità.
Java EE 5 ha introdotto l'idea di **configurazione per eccezione, in cui il container applica regole di default, a meno che non venga specificato diversamente.**
Ciò significa che, per tutti gli attributi (eccetto l'attributo annotato con @Id), si applicano le seguenti

regole di mappatura predefinite:
- **L’entità è mappata in una tabella dello stesso nome**.
- Gli attributi sono mappati su una colonna che ha lo stesso nome. **Se si desidera modificare questa mappatura di default, si usa @Column**.

Querying Entities

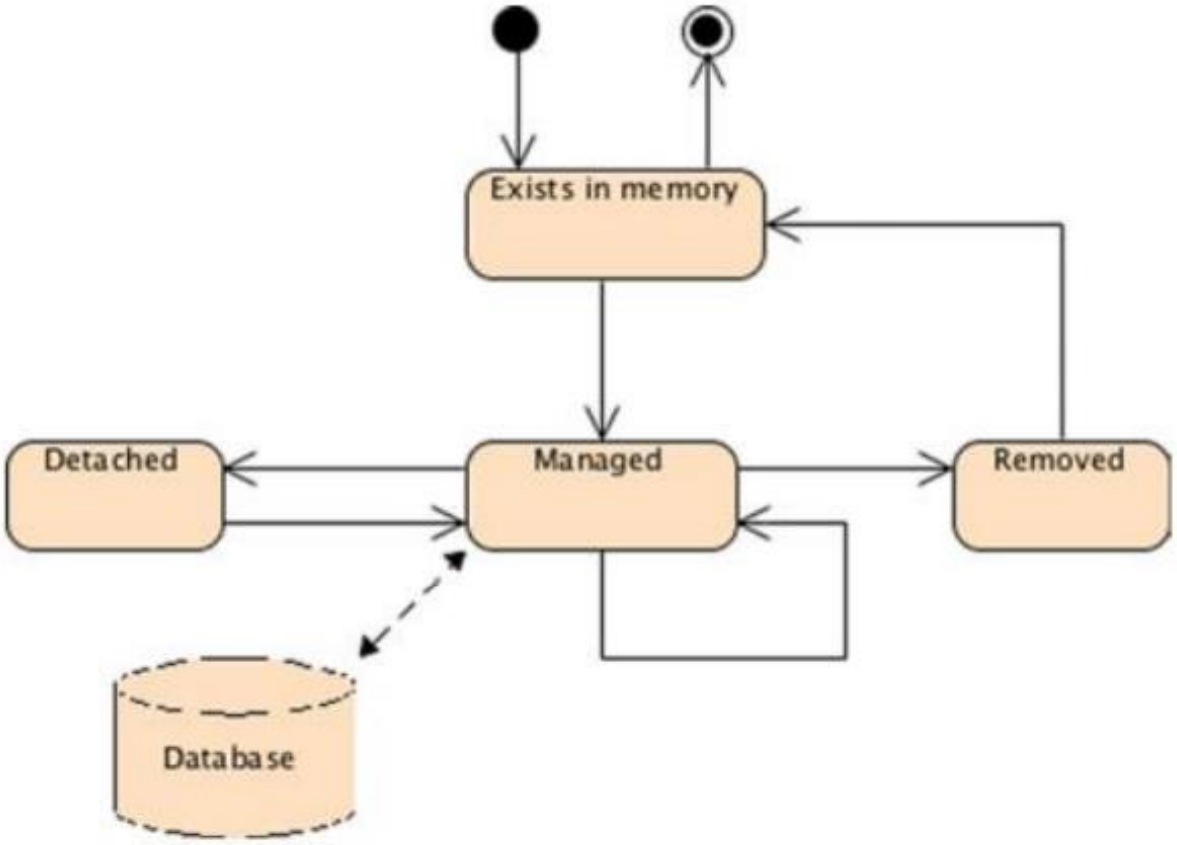
JPA ti permette di mappare le entità in un database e interrogare quest’ultimo usando diversi criteri. **Il pezzo centrare responsabile dell’organizzazione delle entità è l’Entity Manager.** Il suo ruolo è quello di gestire le entità, leggere e scrivere su un determinato database e consentire semplici operazioni CRUD sulle entità (create, read, update e delete) e query complesse che utilizzano JPQL.
L’Entity Manager consente di interrogare le entità utilizzano non il linguaggio SQL, bensì JPQL. Dichiarazioni JPQL manipolano oggetti e attributi, non tabelle e colonne. Un’istruzione JPQL può essere eseguita con query dinamiche o query statiche. Le **query statiche, note anche come NamedQuery,** vengono definite utilizzando le annotazioni @NamedQuery o con i metadati XML.

Persistence Unit

La peristence unit indica all'Entity Manager il tipo di database da utilizzare e i parametri di connessione, che sono definiti nel file **persistence.xml.**

Entity Life Cycle and Callbacks

Le entità sono solo POJO. Quando l'Entity Manager gestisce i POJO, queste hanno un'identità di persistenza e il database sincronizza il loro stato. Quando sono "non gestiti" (cioè sono detached) possono essere utilizzati come qualsiasi altra classe Java. Questo significa che tali entità hanno un ciclo di vita. Quando l'Entity Manager gestisce l'oggetto, esso viene mappato e il suo stato viene sincronizzato, finche il metodo EntityManager.remove() non cancella il dato dal database. Anche se però viene eliminato in questo modo, l'oggetto Java rimane in vita finché non viene rimosso dal garbage collector.



Relationships in Relational Databases

In JPA quando si dispone di un'associazione tra una classe e un'altra, nel database si ottiene un riferimento di tabella. Questo riferimento può essere modellato in due modi diversi: utilizzando una chiave esterna, o utilizzando una tabella di join. In termini di database, una colonna che si riferisce a una chiave di un'altra tabella è una foreign key.

La maggior parte delle entità deve essere in grado di fare riferimento o avere relazioni con altre entità. JPA rende possibile mappare le associazioni in maniera tale che sia possibile mappare un'entità ad un'altra

nel modello relazionale. **Per specificare le cardinalità si possono usare le annotazioni @OneToOne, @OneToMany, @ManyToOne o @ManyToMany.**

Mapping Persistent Object

La sintassi JPQL assomiglia a SQL ma funziona contro oggetti di entità piuttosto che direttamente con le tabelle di database. JPQL non vede la sottostante struttura del database ma piuttosto oggetti e attributi.

Entity Manager

Quando un Entity Manager ottiene un riferimento a un'entità, si dice che l'entità è "managed". Fino a quel momento, l'entità viene considerata come un normale POJO. Quando l'entità è "detached" (non gestita) ritorna ad essere un semplice POJO e può essere quindi utilizzata ad altri livelli.
Per usare un entity manager bisogna ottenerlo. In un application managed environment l'applicazione è responsabile di ottenere esplicitamente un'istanza di entity manager e di gestirne il ciclo di vita. In Java EE il modo più comune di ottenere un entity manager è con l'annotazione @PersistenceContext.

Persistence Context

Un persistence context è un insieme di entità gestite in un certo istante di tempo per una certa transazione.

Quando chiamiamo il metodo persist(), se non esiste già, l'entità viene aggiunta al persistence context. **Il persistence context può essere visto come una sorta di cache, uno spazio in cui l'entity manager mettere le entità prima di fare il flush nel database.**

Serve una persistence unit da cui creare un entity manager. Una persistence unit contiene indicazioni con cui gestire il database e la lista delle entità che possono essere gestite da un entity manager.

La persistence unit è il ponte tra il persistence context e il database.

JPQL

JQP è il linguaggio query utilizzato per ottenere entità tramite criteri diversi dal semplice ID, e ha le sue radici nel linguaggio standard per l'interrogazione del database, l'SQL. La fondamentale differenza fra i due è che in SQL i risultati sono ottenuti sotto forma di righe e colonne (tabelle), mentre **JPQL restituisce un'entità o una collezione di entità, e la sua sintassi è orientata agli oggetti.**

Queries

JPA ha 5 diversi tipi di queries:

- Dynamic queries: specificata dinamicamente a runtime.

- **Named queries:** sono statiche e non modificabili.

- **Criteria API:** introducono il concetto di object-oriented query API.

- **Native queries:** per eseguire una query in **SQL** anziché in JPQL.

- Stored procedure queries: introduce una nuova API per chiamare stored procedure.

Dynamic Queries

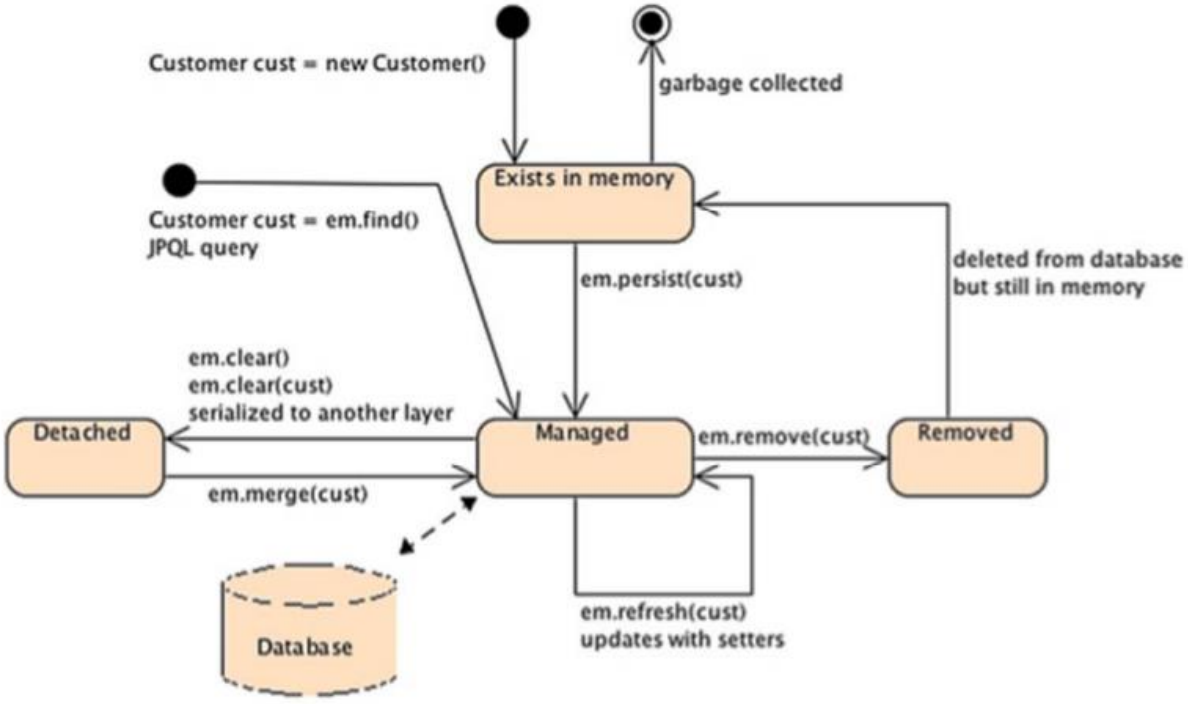
Per creare una query dinamica si usa il metodo EntityManager.createQuery() che prende come parametro una stringa che indica la query.
Questo metodo restituisce un oggetto Query.

Named Queries

Queste sono statiche e non possono essere cambiate, ma sono più efficienti perché il persistence provider può trasformarla in SQL quando parte l'applicazione e non ogni volta che la query viene eseguita.
Le Named Query sono espresse in metadati all'interno di un XML o con @NamedQuery.

Entity Life Cycle

Quando un'entità viene istanziata viene visto solo come un POJO dalla JVM. Quando viene resa persistente dall'Entity Manager, viene detta managed. Quando un'entità è managed l'entity manager 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 sarà sincronizzato automaticamente con il database.

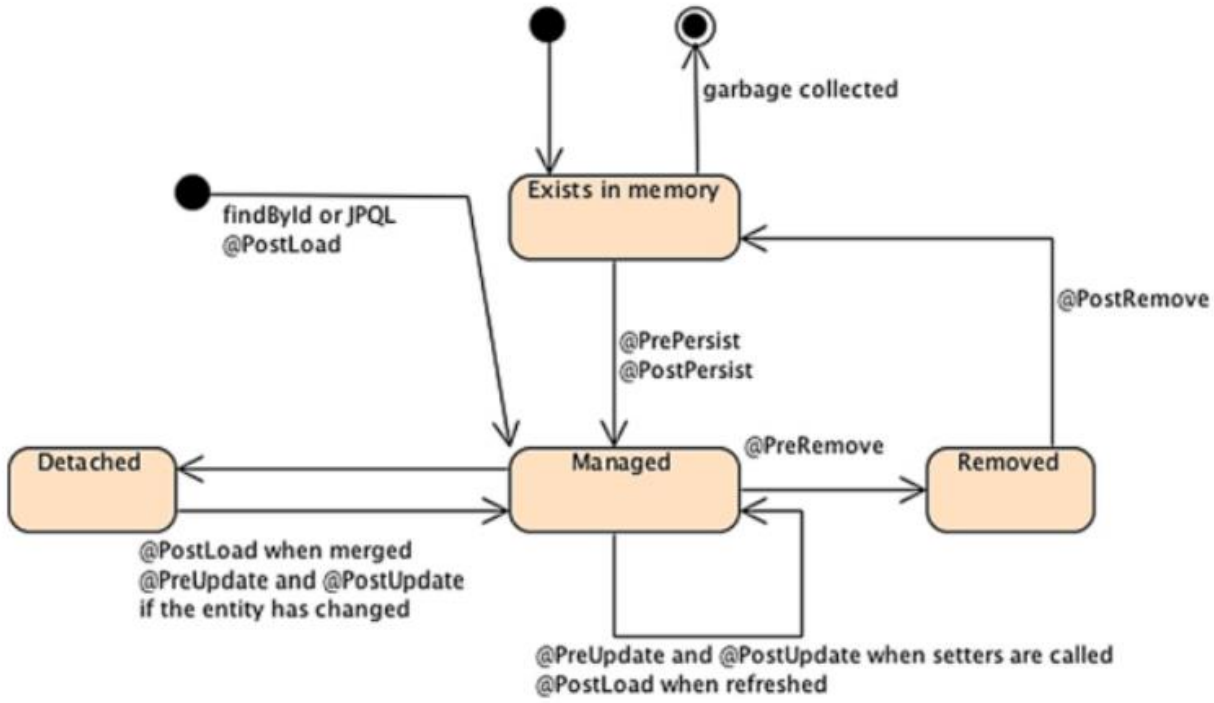


I metodi callback e i listener consentono di aggiungere la propria logica business quando

determinati eventi di ciclo di vita si verificano in un'entità, o, generalmente, ogni volta che un evento di ciclo di vita si verifica in qualsiasi entità.

Callback

Il ciclo di vita di un'entità rientra in 4 categorie: **persisting, updating, removing e loading** che corrispondo alle operazioni **inserting, updating, deleting e selecting.**



Le seguenti regole si applicano ai metodi di callback:

- un metodo può essere public, private o protected ma non static o final.
- può essere annotato con molte annotazioni ma ci può stare solo un'annotazione di un dato tipo per ogni entità (ad esempio, non si possono avere due

annotazioni @PrePersist in un'entità).
- può lanciare unchecked exception ma non checked. Lanciare un'eccezione farà il roll back della transazione se ne esiste una.
- può invocare JNDI, JDBC, JMD and EJBs ma non entity manager o query.
- con l'ereditarietà il metodo della classe padre sarà invocato prima di quello della classe figlia.
- se si usa il cascade anche i metodi di callback saranno chiamati in cascata. Se elimino un Customer e ho il cascade su Address verrà chiamato @PreRemove sia di Customer che di Adress.

Enterprise JavaBeans

Gli EJB sono dei componenti server side che incapsulano business logic e si occupano delle transazioni e della sicurezza, hanno anche uno stack integrato per i messaggi, per fare scheduling…
Agiscono come punto d'entrata per le tecnologie del presentation tier.
Gli EJB sono POJO annotati di cui verrà fatto il deploy in un container. Un EJB container è un ambiente che fornisce servizi come transazioni, gestione della concorrenza, pooling e autorizzazioni di sicurezza.

Types of EJBs

I session bean servono per l'implementazione della logica aziendale, dei processi e del workflow. La piattaforma Java EE definisce diversi tipi di EJB:

- Stateless: il bean non contiene uno stato di conversazione tra i metodi e qualsiasi istanza può essere utilizzata per qualsiasi client. Viene utilizzato per gestire compiti che possono essere conclusi con una sola chiamata di metodo.
- Stateful il bean contiene lo stato di conversazione, che deve essere mantenuto attraverso i metodi per un singolo utente. È utile per i compiti che devono essere eseguiti in diversi passaggi.
- Singleton: un singolo bean di sessione è condiviso tra i client e supporta l'accesso concorrente. Il container assicurerà che esista solo un'istanza per l'intera applicazione.

Bean Class

I requisiti per implementare un session bean sono:

- la classe deve essere annotata con @Stateless, @Stateful, @Singleton o con l'equivalente XML nel deployment descriptor.
- deve implementare i metodi delle interfacce se ce ne sono.

-La classe dev’essere public e non dev’essere final o abstract.
- deve avere un costruttore pubblico senza argomenti.
- non deve avere un metodo finalize().
- i metodi di business non devono essere final o static e non devono iniziare con ejb.
- Gli argomenti e i valori di ritorno devono essere tipi legali per RMI.

Remote, Local and No-Interface Views

A seconda di dove un Client invoca un Session Bean, la classe Bean dovrà implementare un’interfaccia locale, remota o nessuna delle due.
Le interfacce possono essere annotate con:
- @Remote che denota un’interfaccia di business remota. I parametri dei metodi sono passati per valore e devono essere Serializable (RMI).
- @Local che denota un’interfaccia di business locale. I parametri devono essere passati per riferimento dal client al bean.

Stateless Beans

Stateless vuol dire che un task deve essere concluso in una singola invocazione di metodo. Per ogni stateless EJB il container ne tiene un certo numero di

istanze in memoria e le condivide tra i client. Quando un client invoca un metodo su un bean stateless il container prende un'istanza dalla pool e la assegna al client. Quando la richiesta del client finisce l'istanza torna nella pool per essere riutilizzata. Un bean stateless è annotato con @Stateless e poiché vive in un container può usare dei servizi del container come la dependency injection. Per gli stateless session bean il persistence context è transazionale, questo significa che ogni metodo invocato in questo EJB è transazionale.

Stateful Beans

I bean stateful preservano lo stato della conversazione. Sono utili per compiti che devono essere eseguiti in diversi passaggi, ognuno dei quali si basa sullo stato mantenuto in un passaggio precedente. Quando un client invoca un EJB stateful il container ha bisogno di usare la stessa istanza per ogni invocazione. Gli stateful bean non possono essere riusati da altri client. C'è una relazione a uno a uno tra bean e un client.

Per evitare un ingombro di memoria, il container cancella temporaneamente i bean stateful dalla memoria, prima che vengano richiamati dalla

richiesta successiva del Client. Questa tecnica è chiamata attivazione e passivazione dei bean in memoria. Passivare un bean signfiica rimuovere l'istanza dalla memoria e salvarla in memoria persistente. Attivazione è il passaggio inverso.

Singleton

Un bean singleton è un bean istanziato una sola volta per tutta la applicazione e assicura che ci sia una sola istanza di una classe in tutta l'applicazione che può essere acceduta da qualsiasi punto della stessa.

Packaging

EJB ha bisogno di essere inserito in un package prima di essere distribuito in un container runtime. Nello stesso archivio, solitamente si trovano le classi bean enterprise, le interfacce di quest'utile, ogni superclasse o superinterfaccia necessaria, le eccezioni, le classi di aiuto e un deployment descritor opzionale (ejb-jar-xml).

Session Bean Life Cycle

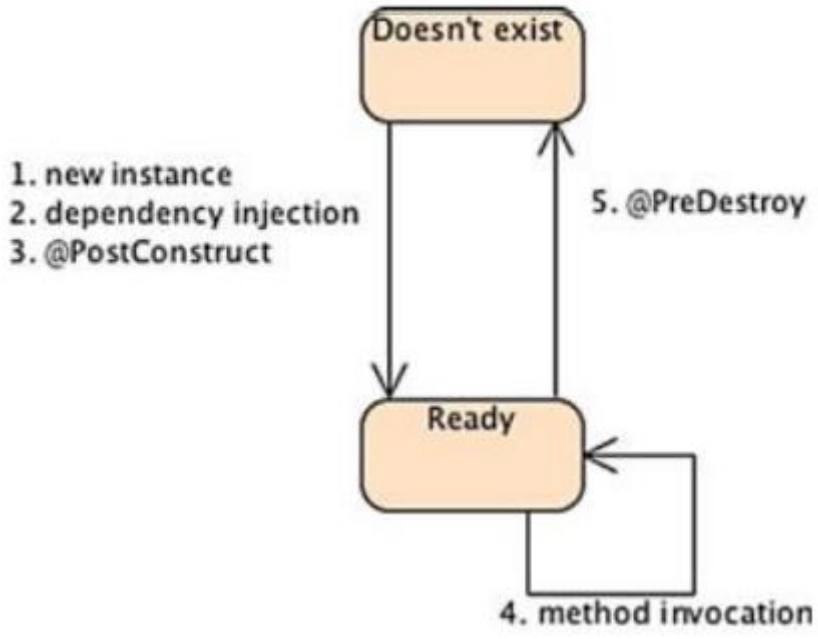
Per ottenere un riferimento al session bean, non usiamo l'operatore new, ma otteniamo un riferimento tramite JNDI o dependency injection. Quindi né il client né il bean è responsabile di

determinare quando un oggetto è creato o distrutto, se ne occupa il container.
Tutti i bean hanno due fasi ovvie: creazione e distruzione. In più, i bean stateful hanno anche attivazione e passivazione.

Statless and Singleton

Stateless e Singleton bean hanno in comune il fatto che non mantengono lo stato di conversazione con un client. Hanno lo stesso ciclo di vita:
1 - Il ciclo di vita di un bean stateless o di un singleton inizia quando il client chiede un riferimento al bean (usando dependency injection o JNDI). Nel caso del singleton può avvenire quando il container viene avviato (usando l'annotazione @Singleton).
2 - Se l'istanza è creata attraverso dependency injection attraverso le annotazioni (@Inject, @Resource, @EJB) o il deployment descriptor, il container inietta tutte le risorse necessarie.
3 - Se l'istanza ha un metodo @PostConstruct il container lo invoca.
4 - Il bean processa la chiamata invocata dal client e resta nello stato ready per processare altre chiamate.
Gli stateless bean restano in ready finche il container non deve liberare spazio nella pool, i singleton finché

il container viene spento.
4 - Il container non ha più bisogno dell'istanza, invoca i metodi @PreDestroy se ci sono e termina la vita dell'istanza.



Gli stateless e i Singleton bean condividono lo stesso ciclo di vita, ma ci sono delle differenze nel modo in cui vengono creati e distrutti.
Quando si deploya un bean di sessione Stateless, il container crea diverse istanze e le aggiunge in un pool. Quando un client chiama un metodo su un bean stateless, il contenitore seleziona un'istanza dal pool, delega l'invocazione del metodo a quell'istanza e poi lo restituisce al pool. Quando il container non ha bisogno più dell'istanza, lo distrugge.

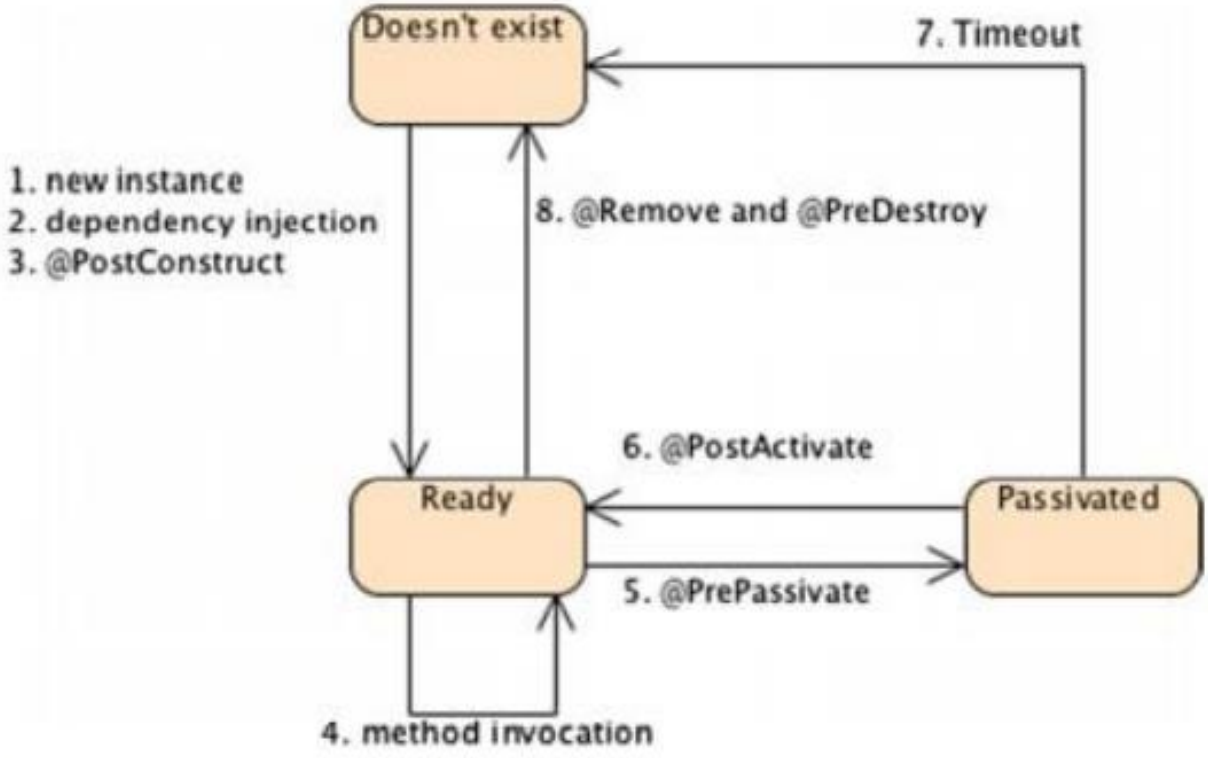
Per i singleton, la creazione dipende dal fatto che siano istanziati con @Startup o meno, o se dipendono (@DependsOn) da un altro singleton. Se la risposta è sì, il container creerà un'istanza al momento della distribuzione. In caso contrario, il container creerà un'istanza quando un client invoca un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

Stateful

I bean Stateful si differenziano dai bean Singleton e Stateless per il mantenimento dello stato di conversazione con il proprio Client e quindi hanno un ciclo di vita diverso. Il container genera un'istanza e la assegna solo a un client. Quindi, ogni richiesta da quel client viene passata alla stessa istanza. Seguendo questo principio se ci sono 1000 utenti simultanei esisteranno 1000 istanza del bean Stateful.
Ricordiamo che c'è una relazione 1 a 1 tra uno stateful bean e un client. Se un Client non richiama l'istanza del bean, il bean viene passivato e riattivato quando ne ha bisogno.
Il ciclo di vita è così descritto:
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 container crea una nuova istanza di bean di sessione e la archivia in memoria.
2 - Se l’istanza appena creata utilizza dependency injection tramite annotazioni o il deployment descriptor, il container 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.
5 - Se il client rimane inattivo per un periodo di tempo, il container 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 container lo riattiva in memoria e richiama il metodo annotato con @PostActivate, se presente.
7 - Se il client non richiama un’istanza di bean passivata per il periodo di timeout della sessione, il container 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.
Spesso uno stateful apre risorse come socket o connesioni al db. Poiché il container non può avere le risorse aperte per ogni bean, bisogna chiudere e riaprire le risorse ogni volta che il bean viene attivato o passivato usando i metodi di callback.



Authorization

Lo scopo principale del modello di sicurezza degli EJB è controllare l'accesso al business code.
L'autenticazione in Java EE è gestita dal web tier.
L'EJB controlla se l'utente autenticato è autorizzato

ad accedere al metodo in base al suo ruolo.
L'autorizzazione può essere fatta sia in modo dichiarativo che programmatico.
Con l'autorizzazione dichiarativa, il controllo degli accessi viene effettuato dal container EJB. Con l'autorizzazione programmatica, il controllo dell'accesso viene effettuato nel codice utilizzando l'API JAAS.

Declarative Authorization (Dichiarativa)
Le autorizzazioni dichiarative possono essere definite nel bean usando le annotazioni o nel deployment descriptor con l'XML.
L'autorizzazione dichiarativa implica la dichiarazione di ruoli, l'assegnazione ai metodi (o all'intero bean) o la modifica temporanea di un'identità di sicurezza.
Esistono varie notazioni, tra cui:
- @PermitAll (bean e method) che indica che è accessibile da tutti.
- @DenyAll (bean e method) che indica che non è accessibile da nessuno.
- @RolesAllowed (bean e method) viene utilizzata per autorizzare un elenco di ruoli per accedere a un metodo. L'annotazione @DeclareRoles (bean) può essere utilizzata per dichiarare altri ruoli.

- @RunAs è utile se è necessario assegnare temporaneamente un nuovo ruolo al principale esistente. Potrebbe essere necessario farlo se stai invocando un altro EJB all'interno del tuo metodo, ma l'altro EJB richiede un ruolo diverso.
L'autorizzazione dichiarativa ti consente di accedere facilmente a una potente politica di autenticazione. Ma cosa succede se è necessario fornire le impostazioni di sicurezza a un individuo o applicare alcune logiche di business basate sul ruolo corrente?

Programmatic Authorization

A volte si ha bisogno di una grana più fine per autorizzare l'accesso (consentendolo ad un blocco di codice anziché l'intero metodo, permettendo o negando l'accesso a un individuo anziché a un ruolo, etc…). Questo è permesso grazie all'interfaccia Principal di JAAS.
L'interfaccia SessionContext definisce i seguenti metodi relativi alla sicurezza:
- isCallerInRole(): questo metodo restituisce un valore boolean e verifica se il chiamante ha un determinato ruolo di sicurezza.
- getCallerPrincipal(): questo metodo restituisce Principal che identifica il chiamante.

```
@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

Le transactions permettono di avere dati consistenti (coerenti) che possono essere poi processati in modo affidabile. La maggior parte del lavoro di un'applicazione enterprise riguarda la gestione dei dati: la memorizzazione, il loro recupero, l'elaborazione, e così via… Spesso questo viene fatto contemporaneamente da diverse applicazioni tentano di accedere agli stessi dati. Un database ha meccanismi di basso livello per preservare l'accesso

simultaneo, quindi vengono utilizzate le transaction per garantire che i dati rimangano in uno stato coerente.

Understanding Transactions

Le transaction rappresentano un Gruppo logico di operazioni che vengono eseguite come una sola unità, chiamata “unit of work”.
Queste operazioni sono eseguite o in sequenza o in parallelo. Ogni operazione deve riuscire affinché la transaction stessa abbia successo (sia committed). Se una delle operazioni fallisce, anche la transaction fallisce (è rolled back).
Le transaction devono essere in grado di fornire affidabilità e robustezza, e seguono le proprietà ACID.

ACID

Con ACID si fa riferimento alle quattro proprietà che definiscono una Transazione: Atomicity, Consistency, Isolation e Durability.
- Atomicity: una transazione è composta da una o più operazioni raggruppate in un’unita di lavoro. Al fine della conclusione della transazione, tutte le operazioni non possono essere eseguite singolarmente. Dunque, o tutte hanno successo (commit) o nessuna di esse (rollback).

- Consistency: in quanto alla fine di una transaction, i dati devono rimanere in uno stato consistente.
- Isolation: gli stati intermedi di una transaction non devono essere visibili ad applicazioni esterne.
- Durability: i cambiamenti effettuati sui dati da un transaction devono essere visibili ad altre applicazioni.

Transaction SupportIN EJBs

Quando si sviluppa business logic con EJB, non bisogna preoccuparsi della struttura interna del Transaction Manager o dei Resourse Manager perché JTA astrae la maggior parte della complessità sottostante.
Con gli EJB si può sviluppare facilmente un'applicazione transactional, lasciando al container il compito di implementare i vari protocolli. Dalla sua creazione, il modello EJB è stato progettato per gestire le transazioni. Infatti, l'EJB di default include ciascun metodo in una transaction. Questo comportamento di default prende il nome di container-managed transaction (CMT).
è possibile scegliere di gestire le transaction usando un bean-managed transaction (BMT).

I transaction demarcation determinano quando le transaction iniziano e finiscono.

Container-Managed Transaction

Quando gestisci le transaction in modo dichiarativo, la politica riguardante la demaracation la si delega al container. Non è necessario usare il JTA nel codice, basta lasciare al container che automaticamente inizia e farà il commit delle transaction basandosi sui metadata.
Il container intercetta la chiamata del client e cerca un transaction context associato con essa. Se nessun context è disponibile, il container inizia una nuova transaction e a quel punto invoca il metodo. Una volta che il metodo è terminato, il container esegue il commit o il roll back.

Bean Managed Transactions

In alcuni casi, la CMT dichiarativa non può fornire la granularità di demarcazione richiesta (ad esempio, un metodo non può generare più di una transazione).
Per risolvere questo problema, gli EJB offrono un modo programmatico per gestire le demarcazioni delle transazioni con BMT.
BMT ci permette di gestire esplicitamente i confini di una transaction (inziio, commit, rollback), usando

JTA, al posto di lasciarlo fare al contaier, come nel CMT.
Per disattivare la demarcazione di default CMT e passare alla modalità BMT, si usa la annotazione @TransactionManagement (o nel file XML).
Con il BTM, l'applicazione richiede la transaction, e l'EJB container crea una transaction fisica e si prende cura di pochi dettagli di basso livello. Per effettuare un BMT si usa l'interfaccia UserTransaction, i cui metodi sono:
- begin: inizia una nuova transaction e la associa al thread corrente.
- commit: effettua il commit della transaction associata al thread corrente.
- rollBack: effettua il rollback della transaction associata al thread corrente.
- setRollbackOnly: marca la transaction corrente per il rollback.
- getStatus: ottiene lo stauts della transaction corrente.
- setTransactionTimeout: modifica il timeout della transaction corrente.

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class InventoryEJB {

  @PersistenceContext(unitName = "chapter09PU")
  private EntityManager em;
  @Resource
  private UserTransaction ut;

  public void oneItemSold(Item item) {
    try {
      ut.begin();

      item.decreaseAvailableStock();
      sendShippingMessage();



      if (inventoryLevel(item) == 0)
        ut.rollback();
      else
        ut.commit();

    } catch (Exception e) {
      ut.rollback();
    }
    sendInventoryAlert();
  }
}
```

Messaging

Ad eccezione delle chiamate asincrone in EJB (grazie all'annotazione @Asynchronous), la maggior parte delle componenti Java EE utilizzano le chiamate sincrone.

Quando parliamo di messaggistica, intendiamo generalmente comunicazioni asincrone tra componenti.

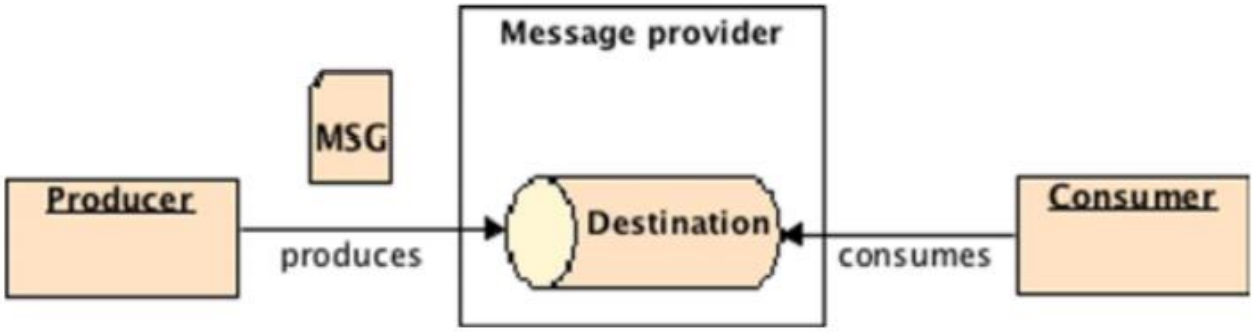
Il middleware orientato ai messaggi (MOM, Message-

Oriented Middleware) è un software che consente lo scambio di messaggi in modo asincrono tra sistemi eterogenei.
Può essere visto come un buffer tra sistemi che producono e consumano messaggi.
I produttori non sanno chi è dall'altro capo del canale di comunicazione a consumare il messaggio.
Produttore e consumatore non devono essere disponibili contemporaneamente per poter comunicare, infatti, non si conoscono nemmeno l'un l'altro, poiché usano un buffer intermedio.
MOM è dunque diverso dalle altre tecnologie, come l'invocazione di metodi remoti (RMI) che richiedono un'applicazione per conoscere la firma dei metodi delle applicazioni remote.
MOM si basa su un modello di interazione asincrona, quindi consente a queste applicazioni di funzionare in modo indipendente e, allo stesso tempo, di far parte di un processo di flusso di lavoro delle informazioni.

Understanding Messaging

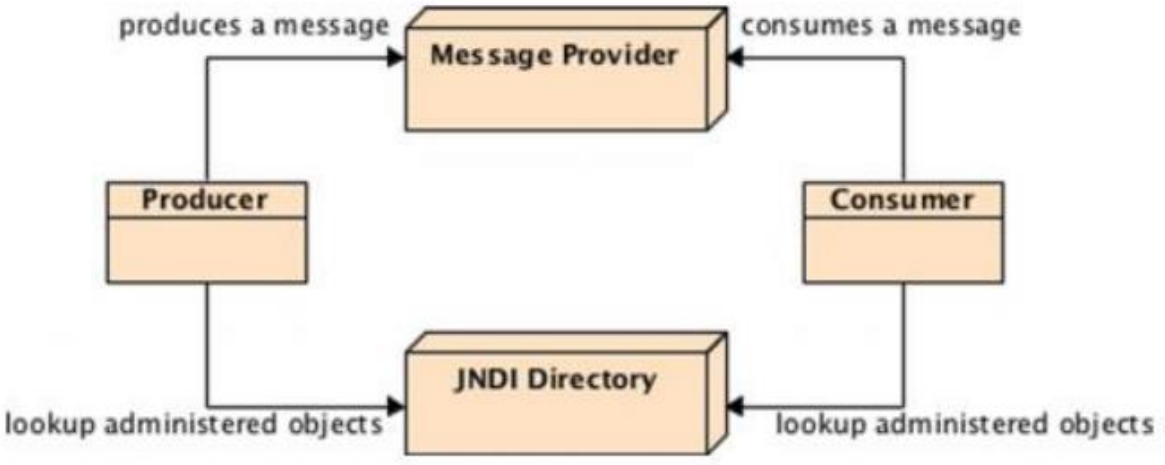
Quando viene inviato un messaggio, il software che memorizza il messaggio e lo invia viene chiamato **Provider** (o talvolta **broker**).
Il mittente del messaggio è chiamato **Producer** e il

percorso in cui è archiviato il messaggio è chiamato **destinazione**.
Il componente che riceve il messaggio è chiamato **consumatore**. Qualsiasi interessato a un messaggio in quella particolare destinazione può consumarlo.



In Java EE, l'API che si occupa di questi concetti è chiamata **Java Message Service (JMS)**. Ha una serie di interfacce e classi che consentono di connettersi a un provider, creare un messaggio, inviarlo e riceverlo.
JMS non trasporta fisicamente messaggi, è solo un'API; lo richiede ad un provider che si occupa della gestione dei messaggi.
Quando si esegue in un EJB container, Message-Driven Beans (MDB) può essere utilizzato per ricevere messaggi in maniera container-managed.
Ad un livello elevato, un'architettura di messaggistica è composta dai seguenti componenti:
- **Provider (fornitore)**: JMS è solo un'API, quindi ha bisogno di un'implementazione sottostante per

instradare i messaggi, cioè il fornitore (a.k.a. un broker di messaggi). Il provider gestisce il buffering e la consegna di messaggi.
- **Client:** un client è una qualsiasi applicazione o componente Java che produce o consuma un messaggio da/verso il provider. “Client” è il termine generico per produttore, mittente, editore, consumatore, destinatario o sottoscrittore.
- **Messaggi:** gli oggetti che i client inviano o ricevono dal provider.
- Oggetti Amministrati: un broker di messaggi deve fornire oggetti amministrati al client (connessioni factories e destinazioni) tramite le ricerche JNDI o l’injection.

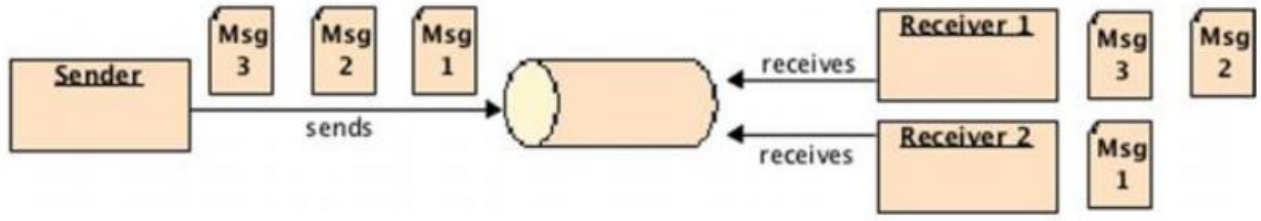


Il provider di messaggistica consente la comunicazione asincrona fornendo una destinazione in cui i messaggi possono essere conservati fino a

quando non possono essere consegnati a un cliente. Esistono due diversi tipi di destinazione, ciascuno da applicare a un modello di messaggistica specifico:
- Modello point-to-point (P2P): in questo modello la destinazione utilizzata per contenere i messaggi viene chiamata **queue**. Quando si utilizza la messaggistica P2P, un cliente mette un messaggio su una coda e un altro client riceve il messaggio. Una volta che il messaggio è stato riconosciuto, il fornitore del messaggio rimuove il messaggio dalla coda.
- **Modello publish-subsribe (pub-sub)**: la destinazione è chiamata **topic.** Quando si utilizza la pubblicazione/sottoscrittura dei messaggi, un client pubblica un messaggio su un topic e tutti gli iscritti a tale topic devono ricevere il messaggio.

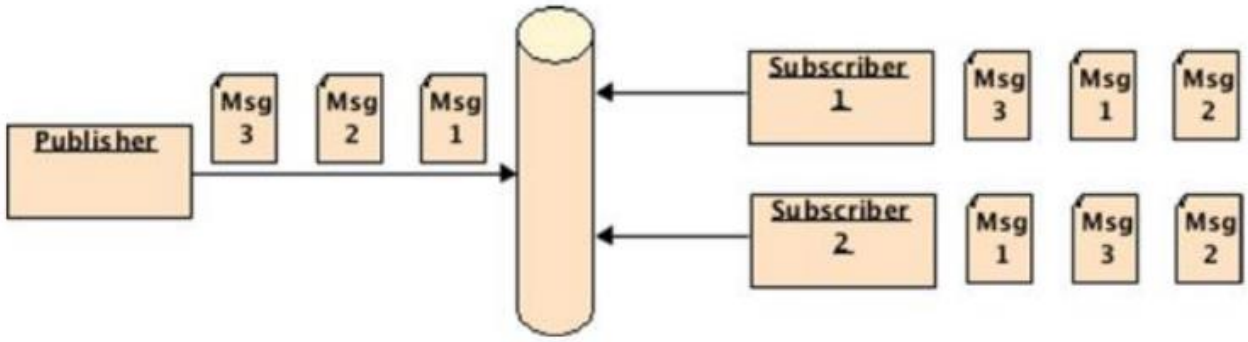
Point-to-Point
In questo modello, un messaggio viaggia da un singolo produttore a un singolo consumatore. Il modello è costruito attorno al concetto di code di messaggi, mittenti e destinatario. Una coda conserva i messaggi inviati dal mittente fino a quando non vengono consumati. Il mittente può produrre messaggi e mandarli in coda quando vuole, e un ricevitore può consumarli quando vuole. Una volta

che il ricevitore viene creato, riceverà tutti i messaggi che sono stati inviati alla coda, perfino quelli inviati prima della sua creazione
Ogni messaggio viene inviata a una coda specifica e il destinatario estrae i messaggi dalla coda. Le code mantengono tutti i messaggi inviati fino a quando non vengono consumati o fino alla loro scadenza. Il modello P2P viene utilizzato se c'è solo un ricevitore per ogni messaggio. Si noti che una coda può avere multipli consumatori, ma una volta che un ricevitore consuma un messaggio, viene tolto dalla coda e nessun altro consumatore può riceverlo.



Publish-Subscribe
Nel modello pub-sub, un singolo messaggio viene inviato da un singolo produttore a diversi potenziali consumatori. Il modello è costruito attorno al concetto di **topic, publishers** e **subscribers.**
I consumatori sono chiamati **subscribers** perché devono prima iscriversi a un topic. Il provider gestisce il meccanismo di sottoscrizione/annullamento

dell’iscrizione a ciò si verifica in modo dinamico.
Il topic conserva i messaggi fino a quando non vengono distribuiti a tutti i sottoscritti. A differenza del modello P2P, c’è la dipendenza temporale tra publishers e subscribers; i subscribers non ricevono i messaggi inviati prima della loro iscrizione e, se sono inattivi per un periodo specificato, non riceveranno i messaggi fin quando non ritornano attivi.
Più subscribers possono consumare lo stesso messaggio.



Administered Objects
Gli adiministered objects sono oggetti configurati amministrativamente, anziché programmaticamente. Il provider consente a questi oggetti di essere configurati e li rende disponibili nello spazio dei nomi JNDI. Come datasource JDBC, gli administered vengono creati una sola volta. Ve ne sono di due tipi:
- Connection factories: utilizzate dai client per creare una connessione a una destinazione.

- Destinations: punti di distribuzione di messaggi che ricevono, trattengono e distribuiscono i messaggi

Message-Driven Beans

Message-Driven Beans (MDB) sono consumatori di messaggi asincroni, eseguiti all'interno di un contenitore EJB. Il contenitore EJB si prende cura di diversi servizi (transazioni, sicurezza, concorrenza, message acknowledgment, etc…) mentre MDB si concentra sul consumo di messaggi. I MDB sono stateless il che significa che il contenitore EJB può avere numerose istanze, in esecuzione contemporaneamente, per elaborare i messaggi in arrivo da vari produttori.
Anche se sembrano bean senza stato, le applicazioni client non possono accedere direttamente ai MDB; l'uno modo per comunicare con un MDB è inviare un messaggio alla destinazione che l'MDB sta ascoltando.
In generale, gli MDB ascoltano una destinazione (queue o topic) e, quando arriva un messaggio, lo consumano e lo elaborano. Possono inoltre delegare la business logic ad altri bean di sessione stateless in modo sicuro e transazionale. Poiché sono stateless, gli MDB non manatengolo lo stato tra invocazioni

separate da un messaggio ricevuto a quello successivo. MDB risponde ai messaggi ricevuti dal container, mentre i bean di sessione stateless rispondo alle richieste del client attraverso un'interfaccia appropriata.

Java Messagin Service API

JMS definisce un comune set di interfacce e classi che permettono ai programmi di comunicare con altri message providers.
Questa API permette una comunicazione asincrona tra clients fornendo una connessione al provider, e una session dove i messaggi possono essere creati e inviati o ricevuti.

Connection Factory

Le Connection Factory sono administered objects che permettono a un'applicazione di connettersi a un provider.
Per usare un administered objects il client ha bisogno di fare un JNDI lookup (o usare l'injection).

Destination

La destination è un administered object configurato specificatamente con informazioni come l'indirizzo di destinazione.

Messages

Per comunicare i client si scambiano messaggi; un produttore invierà messaggi ad una destinazione, e un consumatore li riceverà. I messaggi sono oggetti che incapsulano informazioni e sono divisi in 3 parti:

- Header: contiene informazioni standard per identificare ed instradare il messaggio.
- Proprietà: coppie nome/valore che l'applicazione può settare o leggere. Le proprietà permettono anche alle destinazioni di filtrare i messaggi basandosi su dei valori.
- Un corpo: contiene il messaggio in sé e può avere diversi formati.

Simplified API

JMS 2.0 introduce un'API semplificata che consiste principalmente di tre nuove interfacce: JMSContext, JMSProducer, JMSConsumer.

JMSContext

Combina la funzionalità di due oggetti dell'Api classica 1.1: Connection e Session.

Un JMSContext può essere creato dall'applicazione chiamando o il metodo createContext su ConnectionFactory oppure con @Inject se l'applicazione è in esecuzione in un container.

Quando un’applicazione deve inviare messaggi, utilizza il metodo createProducer per creare un JMSProducer che fornisce metodi per configurare e inviare messaggi. Per riceve messaggi può utilizzare il metodo createConsumer.

Asyncrhnous Delivery

L’esecuzione asincrona è basata sulla gestione di eventi. Un client può registrare un oggetto che implementi l’interfaccia MessageListener. Un listener di messaggi è un oggetto che agisce come un gestore asincrono di eventi. Quando arriva un messaggio, il provider li consegna tramite la chiamata del metodo onMessagge() del listener, che prende come argomento un tipo Message.

Meccanismi di affidabilità

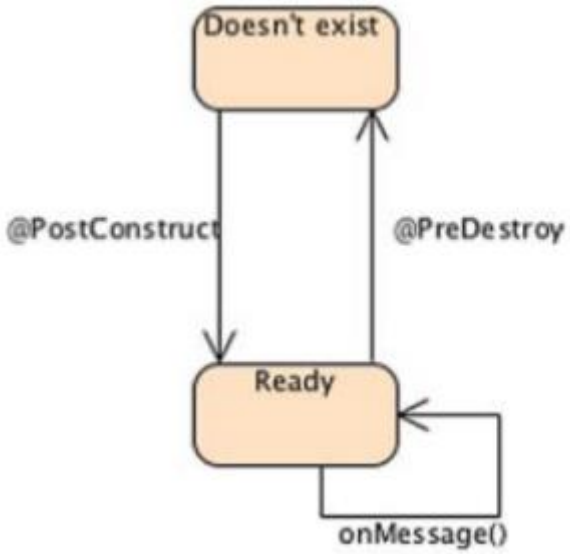
JMS definisce diversi livelli di affidabilità per assicurare che un messaggio sia consegnato. Questi meccanismi sono:

- Filtraggio messaggi: tramite selector è possibile filtrare i messaggi che si vogliono ricevere.
- Time-to-live di messaggi: specificare una scadenza in modo tale che i messaggi obsoleti non vengano consegnati.
- Persistenza dei messaggi: specificare che i messaggi

sono persistenti in caso di fallimenti del provider.
- Controllo dell'acknowledgment: specificare vari livelli di acknowledgment dei messaggi.
- Creazione iscrizioni durature: assicurare che i messaggi siano consegnati a un iscritto disponibile in un modello pub-sub.
- Definire priorità.

Life Cycle and Callback Annotations

Il ciclo di vita di un MDB è identico a quello del bean di sessione stateless: l'MDB esiste ed è pronto a consumare messaggi oppure non esiste.



Web Service

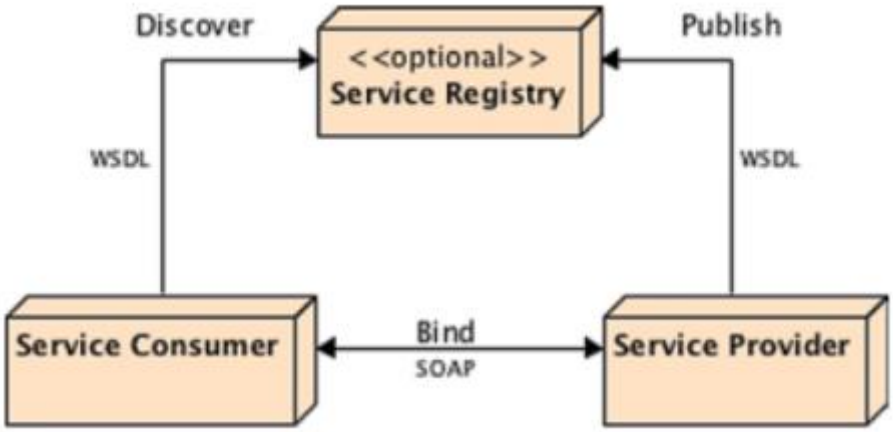
Il termine web service indica "qualcosa" accessibile sul "web" che ti dà un "servizio". Il termine "servizi web" è assimilato alla parola Service Oriented

Architecture (SOA). Le applicazioni dei web service possono essere implementate con tecnologie diverse come SOAP o REST.
I servizi web SOAP (Simple Object Access Protocol) sono “debolmente accoppiati” perché il client, ovvero il consumer, non deve conoscere i dettagli dell’implementazione. Il consumer è in grado di invocare un servizio Web SOAP utilizzando un’interfaccia intuitiva che descrive i metodi business disponibili (parametri e valori di ritorno). L’implementazione sottostante può essere eseguita in qualsiasi linguaggio. Un consumer e un producer di servizi saranno in grado di scambiare dati in un modo debolmente accoppiato: utilizzando documenti XML. Un consumer invia una richiesta a un servizio Web SOAP sotto forma di un documento XML e, facoltativamente, riceve una risposta anche in XML. Il protocollo di rete predefinito è http, un protocollo stateless noto e robusto.

Understanding SOAP Web Services

I servizi Web SOAP costituiscono una sorta di logica aziendale esposta tramite un servizio a un cliente. Tuttavia a differenza degli oggetti o EJB, i servizi Web SOAP forniscono un’interfaccia debolmente

accoppiata utilizzando XML. Gli standard dei SOAP web services specificano che l'interfaccia a cui viene inviato un messaggio dovrebbe definire il formato della richiesta e della risposta del messaggio e i meccanismi per pubblicare e scoprire le interfacce dei Web service (il registro del servizio). Il SOAP web service può facoltativamente registrare la propria interfaccia in un registro (Universal Description Discovery and Integration, o UDDI) in modo che un utente possa scoprirlo.



I SOAP web services dipendono da diverse tecnologie e protocolli per il trasporto e la trasformazione dei dati da un consumatore a un fornitore in modo standard. I più frequenti sono:

- L'XML è la base su cui vengono creati e definiti i SOAP web services.
- 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 di servizi Web.
- I messaggi vengono scambiati utilizzando un protocollo di trasporto. Sebbene http sia il protocollo di trasporto più diffuso è possibile utilizzarne altri come SMTP o JMS.
- Universal Description Discrovery, and Integration (UDDI) è un meccanismo opzionale di scoperta dei servizi, simile alle Pagine Gialle; può essere utilizzato per archiviare e classificare interfacce di servizi Web SOAP (WSDL).

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).

WSDL

WSDL è il linguaggio di definizione dell'interfaccia

(IDL) che definisce le interazioni tra consumatori e SOAP web service.
Per garantire l'interoperabilità è necessaria un'interfaccia standard per consentire a un consumatore e a un produttore di condividere e comprendere un messaggio. Questo è il ruolo di WSDL.

SOAP

WSDL descrive un'interfaccia astratta del web service mentre SOAP fornisce un'implementazione concreta, definendo i messaggi XML scambiati tra il consumatore e il provider.

UDDI

I consumatori e i fornitori che interagiscono tra loro sul Web devono essere in grado di trovare informazioni che consentono 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 Web service e su come richiamarlo.

Writing SOAP Web Services

Per scrivere un servizio web SOAP si può iniziare dal

WSDL o passare direttamente alla codifica di Java. Il documento WSDL può essere utilizzato per generare il codice Java per il consumatore e il servizio. Questo è l'approccio top-down, noto anche contract first. Metro fornisce alcuni strumenti (wsimport) che generano classi da un WSDL. Con l'altro approccio, chiamato bottom-up, la class di implementazione esiste già e tutto ciò che è necessario è creare il WSDL. Metro fornisce utility (wsgen) per generare un WSDL dalle classi esistenti.
L'approccio bottom-up può comportare applicazioni molto inefficiente, poiché i metodi e le classi Java non hanno alcuna relazione con l'ideale granularità dei messaggi che attraversano la rete.

Anatomy of a SOAP Web Service

I servizi web SOAP si bano sul paradigma di configuration-by-exception. È necessaria solo un'annotazione per trasformare un POJO in un servizio web SOAP @WebService.
I requisiti per scrivere un servizio Web sono i seguenti:
- la classe deve essere annotata con @WebService o l'equivalente XML in un deployment descriptor.
- La classe 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 final 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 @Stateless o @Singleton.
- Un servizio deve essere un oggetto stateless e non deve salvare lo stato specifico del client attraverso le chiamate di metodo.
Un bean di sessione stateless o singleton può anche essere utilizzato per implementare un servizio che verrà distribuito in un container EJB (a.k.a. un endpoint EJB).

WSDL mapping

È necessaria una traduzione dagli oggetti Java alle operazioni WSDL. Il runtime JAXB utilizza annotazioni per determinare come eseguire il marshall/unmarshall di una classe in/da XML. Allo stesso modo, JWS utilizza annotazioni per mappare e classi Java in WSDL e determinare come eseguire il

marshalling di una chiamata di metodo a una richiesta SOAP.

@WebMethod

Ci consente di poter rinominare un metodo o di escluderlo dal WSDL.
@WebMethod(operationName = “…”)
@WebMethod(exclude = true)

@WebResult

Questa notazione controlla il nome generato del valore restituito dal messaggio nel WSDL.
Per impostazione predefinita, il nome del valore restituito nel WSDL è impostato a return.
@WebResult(name = “…”)

@WebParam

Simile a @WebResult, ci permette di personalizzare i parametri per i metodi del servizio web.
Di default il nome del parametro è argn con n = posizione del parametro.
@WebParam(name = “…”)

@OneWat

Può essere utilizzata su metodi che non hanno un valore i ritorno come i metodi che restituiscono void.
Questa annotazione può essere vista come

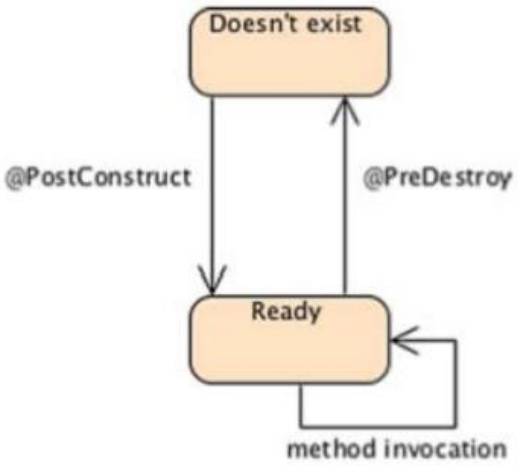
un’interfaccia di markup che informa il contenitore su tale chiamata si può ottimizzare, in quando non vi è alcun ritorno.

Handling Exceptions

In Java quando qualcosa va storto viene generata un’eccezione e qualche altra classe all’interno della JVM deve gestirla. Con i servizi Web questo meccanismo non può funzionare perché il consumer e il producer potrebbero non essere scritti nello stesso linguaggio. Quindi l’idea è di usare un errore SOAP nel messaggio. In runtime JAX-WS converte automaticamente le eccezioni Java in messaggi di errore SOAP restituiti al client.

Life Cycle and Callback

I servizi Web SOAP hanno un ciclo di vita che ricorda i managed bean. È lo stesso ciclo di vita dei componenti che non hanno nessun stato.



WebServiceContext

Un SOAP web service ha un contesto di ambiente e può accedervi inserendo un riferimento di WebServiceContext con l'annotazione di @resource. In questo contesto, il web service può ottenere informazioni di runtime sulla classe di implementazione dell'endpoint, sul contesto del messaggio e sulle informazioni di sicurezza relative a una richiesta che viene servita.

Invoking SOAP Web Services

Con WSDL e alcuni strumenti per generare gli stub client Java (o proxy), è possibile richiamare facilmente un servizio Web senza preoccupare di http o SOAP. Chiamare un web service è simile al chiamare un oggetto distribuito RMI. Come RMI, JAX-WS consente al programmatore di utilizzare una chiamata al metodo locale per richiamare un servizio distribuito. La differenza è che, sull'host remoto, il servizio Web può essere scritto in un altro linguaggio di programmazione. Metro fornisce uno strumento di utilità WSDL-to-java (wsimport) che genera interfacce e classi Java da un WSDL. Questo proxy inoltra quindi la chiamata locale Java al servizio Web remoto utilizzando http. Quando viene richiamato un metodo

su questo proxy, questo converte i parametri del metodo in un messaggio SOAP (la richiesta) e lo invia all'endpoint del web service. Per ottenere il risultato, la risposta SOAP viene riconvertita in un'istanza del tipo restituito.

Anatomy of a SOAP Consumer

Poiché JAX-WS è disponibile in Java SE, un utente del servizio Web può essere qualsiasi tipo di codice Java da un main class in esecuzione sulla JVM a qualsiasi componente Java EE in esecuzione in un container. Se viene eseguito in un container, il consumatore può ottenere un'istanza del proxy tramite l'iniezione o creandola programmaticamente.

Se il tuo consumer è in esecuzione all'esterno di un container, è necessario richiamare programmaticamente il web service. Il consmer utilizza un'istanza di Service che è stata generata dal WSDL grazie a wsimport, utilizzando new. Da questa ottiene la classe proxy con il metodo getPort() per invocare localmente i metodi di business.