

Introduzione a Java EE

JEET 1-Overview

Oggi gli sviluppatori sentono sempre di più il bisogno di sviluppare applicazioni distribuite, transazionali e portabili che sfruttino velocità, sicurezza e affidabilità di tecnologie lato server.

Le applicazioni enterprise spesso interagiscono con altri software enterprise (sistemi legacy) e devono essere progettate, costruite e prodotte con meno soldi, con una maggiore velocità e con meno risorse.

Lo scopo di Java EE è quello di fornire agli sviluppatori un potente insieme di API che accorcino i tempi di sviluppo, riducano la complessità delle applicazioni e migliorino le prestazioni.

La piattaforma Java EE viene sviluppata attraverso il Java Community Process (JCP), che è il responsabile di tutte le tecnologie Java. Gruppi di esperti hanno creato Java Specification Requests (JSR) per definire le diverse tecnologie Java EE. Il lavoro della comunità Java nel programma JCP contribuisce a garantire gli standard di stabilità e la compatibilità tra piattaforme della tecnologia Java.

Il deployment descriptor in XML è opzionale. Uno sviluppatore può semplicemente inserire le informazioni come annotazioni, direttamente nel codice sorgente. Queste annotazioni sono generalmente utilizzate per incorporare in un programma dati che altrimenti sarebbero forniti in un deployment descriptor. (Le annotazioni sono più semplici, ma il file XML è più potente. Inoltre usando l'XML possiamo effettuare facilmente dei cambiamenti, mentre per effettuare modifiche con le annotazioni bisogna accedere al codice sorgente. L'XML sovrascrive le annotazioni).

Nella piattaforma Java EE, la dependency-injection può essere applicata a tutte le risorse necessarie a un componente, nascondendo la creazione e il lookup (ricerca) di risorse dal codice applicativo. La dependency-injection può essere utilizzata dagli Enterprise JavaBeans (EJB) container, web container e application client. La dependency-injection consente al container Java EE di inserire automaticamente riferimenti ad altri componenti o risorse necessari, utilizzando le annotazioni.

1.1 Java EE 7 Platform Highlights

L'obiettivo più importante della piattaforma Java EE 7 è quello di semplificare lo sviluppo fornendo una base comune per i diversi tipi di componenti della piattaforma Java EE. Gli sviluppatori traggono vantaggio dai miglioramenti della produttività con più annotazioni e meno configurazione XML, più Plain Old Java Object (POJO), e packaging semplificato.

1.2 Java EE Application Model

Java EE è progettato per supportare applicazioni che implementano servizi aziendali per clienti, dipendenti, fornitori, partner e altri. Sono applicazioni intrinsecamente complesse, potenzialmente accedono a dati provenienti da una varietà di fonti e distribuiscono applicazioni a una vasta gamma di client.

Per migliorare il controllo e la gestione di queste applicazioni, le funzioni aziendali vengono condotte nel middle tier. Il middle tier rappresenta un ambiente strettamente controllato dal dipartimento informatico aziendale. Il middle tier è generalmente eseguito su un server dedicato e ha accesso ai servizi completi dell'enterprise.

Il modello di applicazione Java EE definisce un'architettura per l'implementazione di servizi come applicazioni multitier che offrono la scalabilità, l'accessibilità e la gestibilità necessarie per le applicazioni a livello enterprise. Il lavoro necessario per implementare un servizio multitier è diviso in:

- La logica di business e di presentazione che lo sviluppatore deve implementare
- I servizi di sistema standard forniti dalla piattaforma Java EE

Lo sviluppatore può contare sulla piattaforma per fornire soluzioni a problemi relativi allo sviluppo di un servizio multitier.

1.3 Distributed Multitiered Applications

La piattaforma Java EE utilizza un modello di applicazione distribuito multilivello distribuito per le applicazioni enterprise. La logica dell'applicazione è divisa in componenti in base alla funzione e i componenti dell'applicazione che compongono un'applicazione Java EE sono installati su varie macchine a seconda del livello nell'ambiente multiplo Java EE cui appartiene il componente dell'applicazione.

La logica dell'applicazione è suddivisa in componenti a seconda della loro funzione. I componenti che compongono un'applicazione Java EE vengono installati in varie macchine a seconda del tier nell'ambiente Java EE multitier a cui il componente dell'applicazione appartiene.

La figura mostra due applicazioni Java EE multitier suddivise nei livelli descritti nell'elenco seguente.

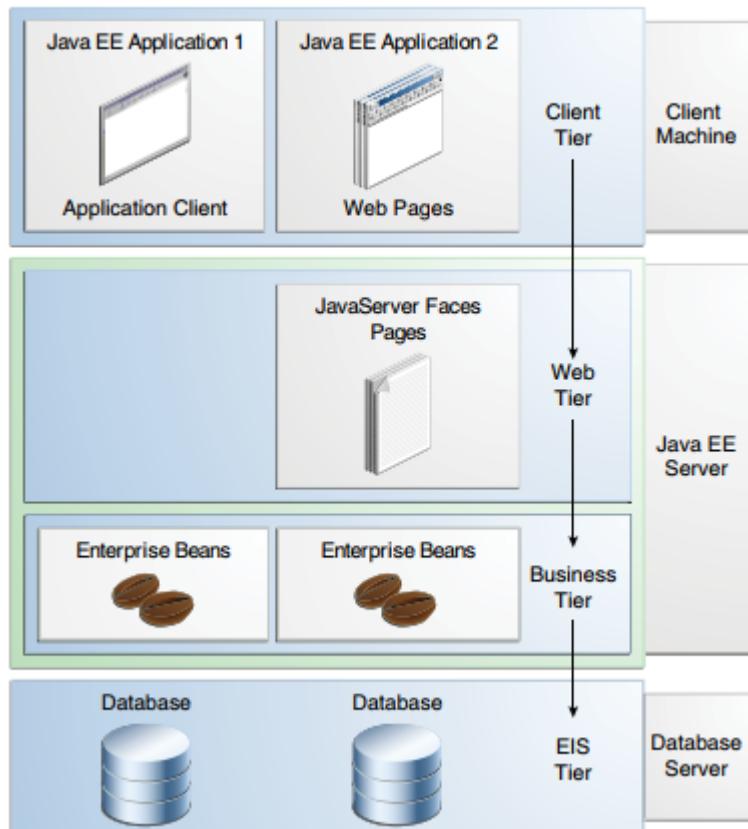
- Componenti del livello Client (Client-Tier) eseguiti sulla macchina client.
- Componenti del livello Web (Web-tier) eseguiti sul server Java EE.
- Componenti del livello Business (Business-Tier) eseguiti sul server Java EE.
- Enterprise Information System (EIS)-Tier: software eseguito sul server EIS.

Sebbene un'applicazione Java EE può essere costituita da tutti i livelli mostrati le applicazioni multitiered (multilivello) sono considerate threee-tier in quanto distribuite in tre locazioni:

1. macchine client
2. macchina server Java EE
3. database o macchine legacy sul back-end.

In questo modo viene esteso il modello client-server standard a due livelli, posizionando un multithreaded application server tra l'applicazione client e lo storage di back-end.

Figure 1–1 Multitiered Applications



1.3.1 Security

Java EE security environment consente di definire i vincoli di sicurezza al momento del deployment. La piattaforma Java EE rende le applicazioni portabili a una vasta gamma di implementazioni di sicurezza e fa in modo che gli sviluppatori non debbano implementare funzionalità di protezione.

Java EE fornisce un controllo dichiarativo degli accessi e meccanismi standard di login in modo che gli sviluppatori non debbano implementare questi meccanismi. La stessa applicazione funziona in vari security environment senza modificare il codice sorgente.

1.3.2 Java EE Components

Le applicazioni Java EE sono costituite da Components.

Un Java EE component è un'unità software funzionale self-contained assemblata in una Java EE application con classi e file, e che comunica con altre components.

La specifica Java EE definisce i seguenti componets:

- Application clients e Applets sono components eseguiti sul client
- Java Servlet, JavaServer Faces, e JavaServer Pages (JSP) sono web components eseguiti sul server
- EJB components (enterprise beans) sono business components eseguiti sul server.

I Components Java EE sono scritti nel linguaggio di programmazione Java e sono compilati allo stesso modo di qualsiasi programma di quel linguaggio. Le differenze tra i Java EE components e le classi Java "standard" sono che i components Java EE vengono assemblati in un'applicazione Java EE, viene verificato che siano conformi alla specifica Java EE, ne viene fatto il deploy e poi vengono eseguiti e gestiti dal server Java EE.

1.3.3 Java EE Clients

Un client Java EE è solitamente un Web Client o un'Application Client.

1.3.3.1 Web Clients

Un client web consiste di due parti:

1. Pagine web dinamiche, contenenti vari tipi di linguaggi di markup(HTML, XML, ecc), generate dai web components eseguiti nel Web Tier
2. Un Web Browser che fa il rendering delle pagine ricevute

Un client web è detto **thin client** perché non fa operazioni pesanti come query ad un database che invece sono a carico degli EJB eseguiti sul server.

1.3.3.2 Application Clients

Un application client viene eseguito su una macchina client e fornisce agli utenti per gestire le attività che richiedono un'interfaccia utente più ricca di quanto possa essere fornito da un linguaggio di markup. Un Application Client ha in genere un'interfaccia utente grafica (GUI)

Gli application client accedono direttamente agli Enterprise Beans eseguiti nel Business-Tier. Tuttavia, un application client può aprire una connessione http per stabilire la comunicazione con una servlet in esecuzione nel Web-Tier. Gli application

client 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 non java.

1.3.3.3 Applets

Scritta in Java, un applet è una piccola applicazione client eseguita nella JVM installata nel browser.

Per essere eseguito ha bisogno di un Plug-in java e di un file di security policy. I Web component sono preferiti per la creazione di un programma client web perché non sono necessari plug-in o file di policy. Inoltre, forniscono un modo per separare la programmazione dell'applicazione dal design delle pagine web.

1.3.3.4 The JavaBeans Component Architecture

Server e client tier includono components basati su JavaBeans component architecture per gestire il flusso tra :

- Application Client o Applet e i Component eseguiti sul server Java EE.
- Server Components e database

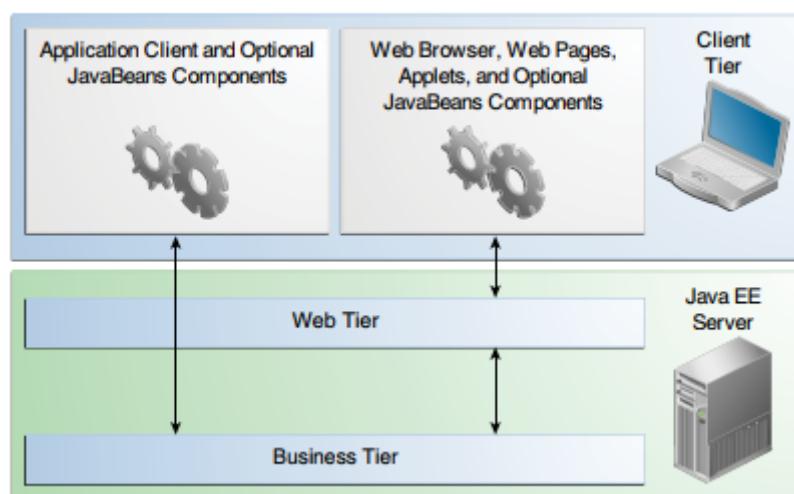
JavaBeans components non sono considerati Java EE components.

I componenti JavaBeans hanno attributi e hanno metodi get e set per l'accesso a quegli attributi.

1.3.3.5 Java EE Server Communications

La figura mostra gli elementi del Client-Tier. Il client comunica con il Business Tier o direttamente o attraverso pagine web e servlet eseguite nel web tier.

Figure 1–2 Server Communication



1.3.4 Web Components

Un web component Java EE è una servlet o una pagina web creata usando JSP o tecnologia JavaServer Faces.

Le **Servlet** sono classi in linguaggio Java che elaborano dinamicamente le richieste e costruiscono le risposte.

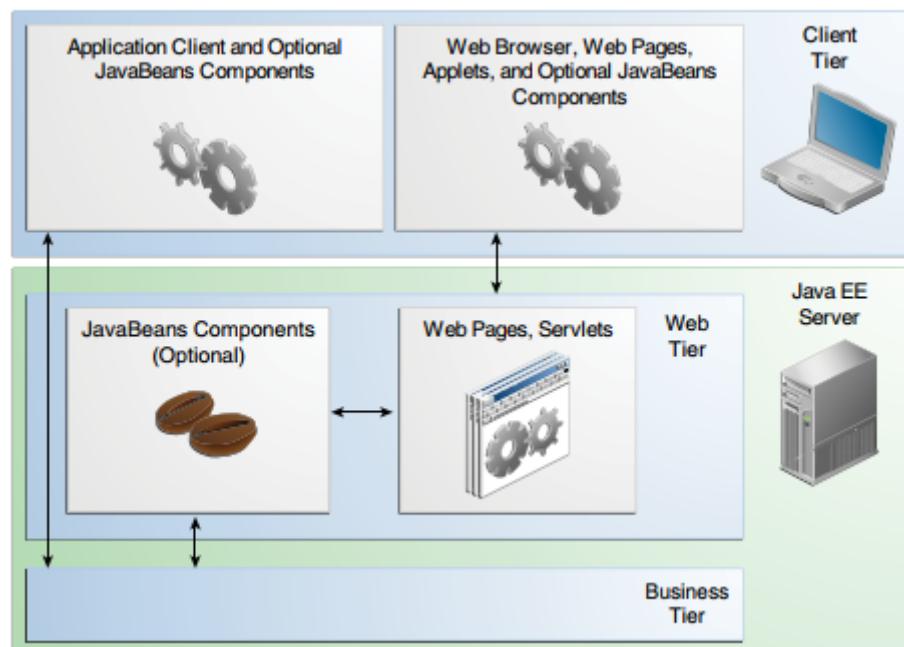
Le **Pagine JSP** sono documenti basati sul testo che vengono eseguite come servlet ma che consentono un approccio più naturale per la creazione di un contenuto statico.

La **JavaServer Faces** technology si basa su Servlet e JSP e fornisce un framework di componenti di interfaccia utente per applicazioni web.

Le pagine statiche HTML o le Applets sono raggruppate con le Web Components durante l'assemblaggio dell'applicazione, ma non sono considerate come Web Components da Java EE. Stessa cosa per le Server-Side utility classes.

Come mostra la figura il Web-Tier, come il Client-Tier, può includere JavaBean component per gestire l'input e mandarlo agli enterprise bean eseguiti nel Business-Tier.

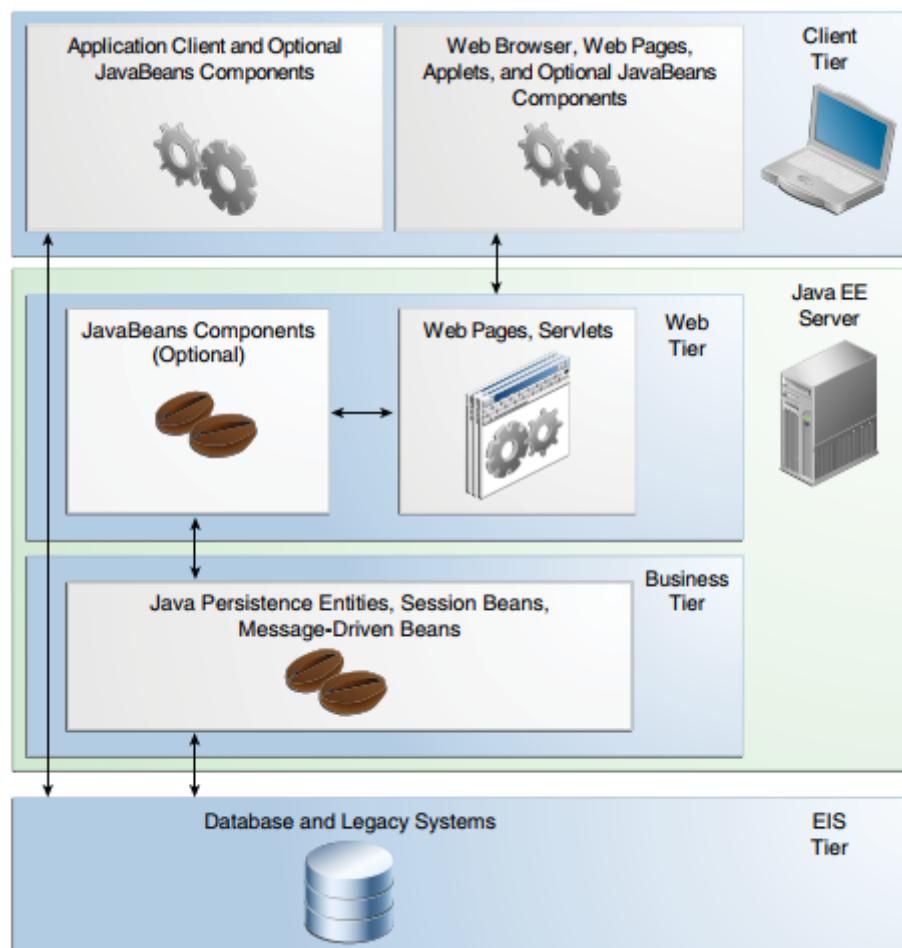
Figure 1–3 Web Tier and Java EE Applications



1.3.5 Business Components

Il business code è la logica che risolve o soddisfa le esigenze di un particolare dominio di business, come quello bancario, vendita al dettaglio o finanza, viene gestito dagli enterprise beans eseguiti nel Business Tier o nel Web Tier. La figura mostra come un enterprise bean riceve i dati dal client li processa e li manda all'Enterprise Information System Tier (EIS) per la memorizzazione. Inoltre un EJB riceve i dati dallo storage li processa e li rimanda al client

Figure 1–4 Business and EIS Tiers



1.3.6 Enterprise Information System Tier

L' enterprise information system tier gestisce l' EIS software e include, Enterprise Resource Planning, database, mainframe Transaction processing e altri sistemi legacy.

1.4 Java EE Containers

Normalmente, le applicazioni multilivello thin client sono difficili da scrivere perché coinvolgono molte linee di codice per gestire Transaction e la gestione dello stato, multithreading, pool di risorse e altri dettagli complessi di basso livello.

L'architettura Java EE rende le applicazioni semplici da scrivere perché la business logic è organizzata in componenti riutilizzabili. Inoltre, il server Java EE fornisce i servizi sottostanti sotto forma di un **Container** per ogni tipo di Component. Così lo sviluppatore è libero di concentrarsi sulla soluzione del problema aziendale.

1.4.1 Container Services

I container sono l'interfaccia tra un Component e la funzionalità di basso livello specifiche per la piattaforma.

Per essere eseguito, un component (web, enterprise bean, application client) dev'essere assemblato in un modulo Java EE e dev'essere fatto il deploy nel suo container.

Il processo di assemblaggio prevede la specifica delle impostazioni del Container per ciascun component nell'applicazione Java EE e per l'applicazione Java EE stessa. Le impostazioni del container personalizzano il supporto fornito dal server Java EE, come ad esempio sicurezza, gestione delle transaction, Java Naming and Directory Interface (JNDI), look-up, connessione remota.

Ecco i punti più importanti:

- Java EE security model consente di configurare un web component o un enterprise bean in modo che le risorse siano accessibili solo da utenti autorizzati.
- Java EE transaction model fa in modo che tutti i metodi in una transazione siano trattati come una singola unità.
- I servizi di Lookup JNDI forniscono un'interfaccia unificata a più servizi di naming in modo che i component possano accedere a questi servizi.
- Java EE remote connectivity gestisce comunicazioni a basso livello tra client ed enterprise bean. Dopo aver creato un bean, un client invoca metodi su di esso come se fossero nella stessa macchina virtuale.

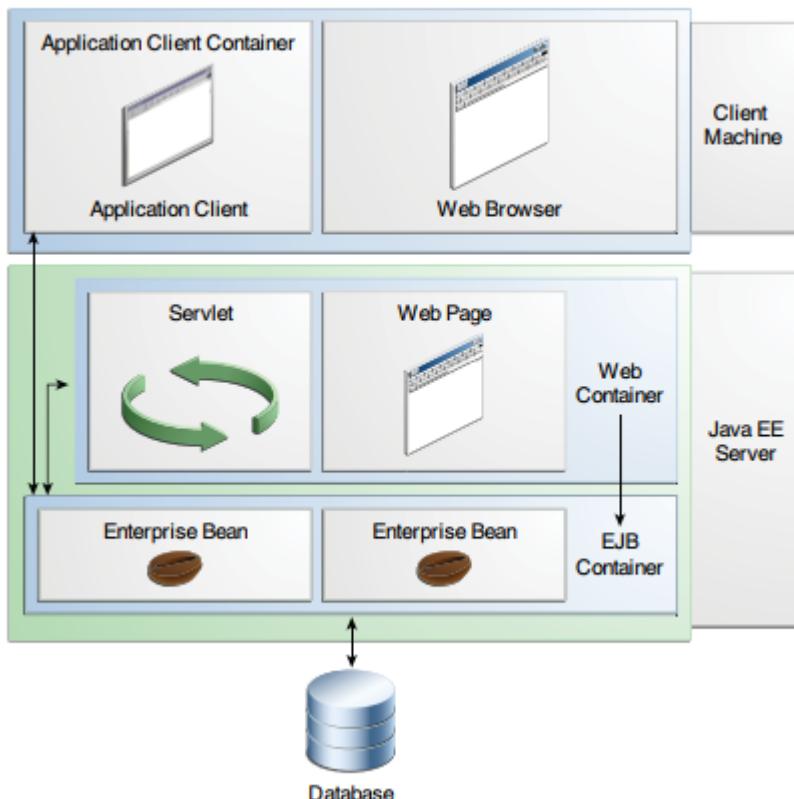
Poiché i servizi sono configurabili, i component possono comportarsi in modo diverso in base a dove vengono distribuiti. Ad esempio, un bean può avere impostazioni di protezione che consentono un certo livello di accesso al database in un ambiente e un altro livello di accesso al database in un altro ambiente.

Il contenitore gestisce anche servizi non configurabili, come i cicli di vita di bean e servlet, database connection pool, la persistenza dei dati e l'accesso alle API della piattaforma Java EE.

1.4.2 Container Types

Il processo di **deployment** installa i Java EE components nei Java EE container come illustrato in figura:

Figure 1–5 Java EE Server and Containers



I server e i container sono i seguenti:

- **Java EE Server:** La parte runtime di un prodotto Java EE. Un server Java EE fornisce EJB e web containers.
- **EJB container:** gestisce l'esecuzione degli enterprise beans. I beans e il loro container sono eseguiti sul server Java EE .
- **Web container:** gestisce l'esecuzione di web pages, servlets, e alcuni EJB components. I Web components e il container sono eseguiti sul server Java EE.
- **Application client container:** gestisce l'esecuzione dell'application client components. Application clients e relativo container sono eseguiti sul client.
- **Applet container:** gestisce l'esecuzione applets. Consiste di un browser e un Plug-in java eseguiti insieme sul client.

1.6 Java EE Application Assembly and Deployment

Un'applicazione Java EE è packaged in una o più unità per il deployment su una Java EE platform. Ogni unità contiene:

- Uno o più functional component come enterprise bean, web page, servlet, o applet.
- Un deployment descriptor opzionale che ne descrive il contenuto.

Il deployment tipicamente prevede l'uso di un deployment tool per specificare informazioni come la lista di utenti locali a cui è consentito l'accesso e il nome del database locale. Una volta fatto il deploy si può eseguire l'applicazione.

Context and Dependency Injection

La prima versione di Java EE ha introdotto il concetto di inversion of control (IoC), ciò significa che il container assume il controllo del business code e fornisce servizi tecnici (come transazione o gestione della sicurezza). Prendere il controllo ha significato gestire il ciclo di vita dei componenti, dependency injection e la configurazione dei componenti.

Questi servizi sono stati integrati nel container ed i programmatore hanno dovuto attendere fino alle versioni successive di Java EE per potervi accedere. La configurazione dei componenti è stata resa possibile nelle prime versioni con i deployment descriptors XML, ma abbiamo dovuto attendere Java EE5 e Java EE6 per avere un'API semplice e robusta per la gestione del ciclo di vita e la dependency injection.

Java EE6 ha introdotto Context and Dependency Injection per semplificare alcuni di questi problemi, ma soprattutto per renderla una specifica centrale che lega insieme tutti questi concetti. Oggi CDI 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.

Allo stesso tempo, CDI unisce il livello web e il back-end, omogeneizzando gli ambiti.

Understanding Beans (Introduzione)

I POJO sono semplicemente classi java che vengono eseguite nella Java Virtual Machine (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 componenti di JavaEE seguono un determinato pattern (ad esempio Enterprise JavaBean) e sono eseguiti in un container che fornisce dei servizi (ad esempio, il container EJB).

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.

I Beans sono oggetti CDI basati sul modello dei Managed Beans. I Beans:

- hanno un ciclo di vita migliorato per oggetti stateful
- sono legati a contesti ben definiti.
- hanno un approccio typesafe alla dependency injection, interception e decoration
- specializzati con annotazioni qualifier
- possono essere usati in expression language (EL).

Infatti, con pochissime eccezioni, potenzialmente ogni classe Java che ha un costruttore di default e viene eseguito all'interno di un container è un bean. Quindi Java Beans e Enterprise Java Beans possono naturalmente trarre vantaggio da questi servizi CDI.

Dependency Injection

La Dependency Injection (DI) è un design pattern che disaccoppia componenti dipendenti e fa parte dell'inversione del controllo. Un modo per pensare alla DI è quello di pensare a un JNDI invertito. Invece di un oggetto che fa il lookup ad altri oggetti (li cerca), il container inietta questi oggetti dipendenti. È il cosiddetto Principio di Hollywood. “Don’t call us (lookup objects), We’ll call you” (inject objects).

Java EE è stato creato alla fine degli anni '90 e la prima versione aveva già EJB, Servlets e JMS. Questi componenti potevano utilizzare JNDI per cercare risorse gestite da container come JDBC DataSource. Questo permetteva le dipendenze dei componenti e consentiva al EJB container di affrontare le complessità della gestione del ciclo di vita delle risorse (istanziando, inizializzando e fornendo riferimenti di risorse ai client come richiesto).

Java EE 5 ha introdotto la Dependency Injection. Ha consentito agli sviluppatori di iniettare risorse del container come EJB, entity managers e data sources in un insieme di componenti definiti (Servlets e EJB). A questo scopo, Java EE 5 ha introdotto un nuovo set di annotazioni (@Resource, @PersistenceContext, @PersistenceUnit, @EJB e @WebServiceRef).

Questo primo passo non era sufficiente, quindi Java EE 6 ha creato due specifiche: Dependency Injection e Contexts and Dependency Injection.

Life-Cycle Management

Il ciclo di vita di un POJO è semplice: il programmatore può creare un’istanza di una casse usando la parola chiave **new** aspetta che il Garbage Collector si sbarazzi di esso e liberi la memoria. Ma se vogliamo eseguire un EJB in un container non possiamo usare new.

E’ necessario iniettare il bean e il container fa il resto, cioè 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 il responsabile della creazione dell’istanza (usando la parola new). 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.

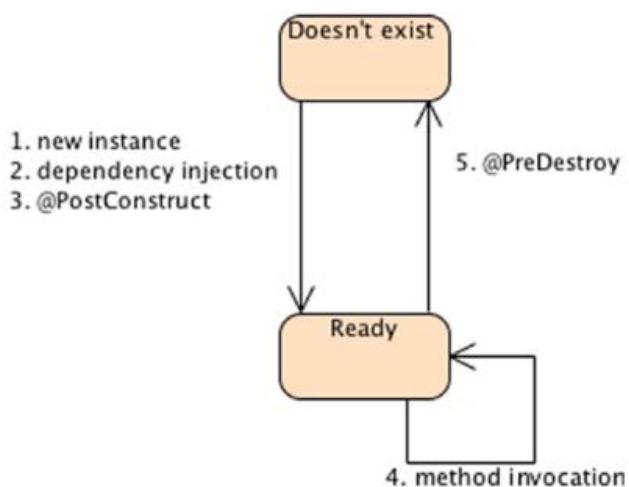


Figure 2-1. Managed Bean life cycle

Scope and Context

I CDI Beans possono essere stateful e sono contestuali, il che significa che vivono in uno scope ben definito (CDI è dotato di scope predefiniti: request, session, application e conversation). Ad esempio, uno scope di sessione e i suoi bean esistono durante tutta la durata di una sessione HTTP. Durante questo tempo, gli injected references ai bean sono anche consapevoli del contesto, cioè l'intera catena delle dipendenze del bean è contestuale. Il container gestisce automaticamente tutti i bean all'interno dello scope e, alla fine della sessione, li distrugge automaticamente.

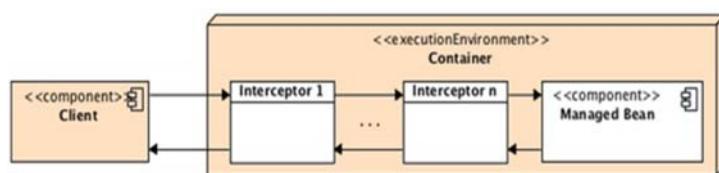
A differenza dei componenti stateless (ad esempio, stateless session beans) o singleton (ad esempio, Servlets o singleton), i diversi client di un bean stateful vedono il bean in stati diversi. Quando un bean è stateful (session, application e conversation scope), è importante quale istanza del bean ha il client. I clienti eseguiti nello stesso contesto vedranno la stessa istanza del bean. Ma client in un contesto diverso potrebbero vedere un'altra istanza.

In ogni caso non è il client a controllare il ciclo di vita dell'istanza, creandola e distruggendola: lo fa il container in base allo scopo.

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. La maggior parte delle applicazioni ha codice comune che viene ripetuto tra i componenti. Questi potrebbero essere problemi come registrare l'entrata e l'uscita di ciascun metodo, registrare la durata di invocazione di un metodo, memorizzare statistiche sull'uso del metodo, ecc. o business concerns (ad esempio 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 procedure possono essere applicate automaticamente tramite AOP all'intera applicazione o a un suo sottoinsieme. 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.



La Figura mostra un numero di Interceptor che vengono chiamati tra il client e i Managed Bean.

Si potrebbe pensare a un contenitore 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 (gestione del ciclo di vita, transazione, sicurezza, ecc.). Con gli Interceptor, si aggiungono i propri meccanismi in modo trasparente al business code.

Loose Coupling and Strong Typing

Gli Interceptors sono un modo molto efficace per disaccoppiare i problemi tecnici dalla logica di business. Contextual life-cycle management disaccoppia anche i bean dalla gestione dei propri cicli di vita. Con l'injection, un bean non è a conoscenza dell'implementazione di qualsiasi bean con cui interagisce. I Beans possono usare event notifications per disaccoppiare i produttori di eventi dai consumatori di eventi o Decorators per disaccoppiare i problemi di business.

In altre parole, il basso accoppiamento è il DNA su cui è stato costruito CDI.

E tutte queste strutture sono consegnate in modo tipicamente sicuro. CDI non si basa mai su identificatori basati su stringhe per determinare come gli oggetti si incastrano.

CDI, invece, utilizza annotazioni fortemente tipizzate (ad es. Qualifiers, Stereotypes, e Interceptor bindings) per legare insieme i bean. L'utilizzo dei descrittori XML è ridotto ad informazioni veramente specifiche dell'implementazione

Deployment Descriptor

Quasi ogni specifica JavaEE ha un XML Deployment Descriptor opzionale. Di solito descrive come bisogna configurare un component, un modulo o un'applicazione.

Con CDI, il Deployment Descriptor si chiama **beans.xml** ed è obbligatorio. Può essere utilizzato per configurare alcune funzionalità (Interceptors, Decorators, Alternatives, ecc.), ma è essenziale per abilitare CDI, questo perché il CDI ha bisogno di identificare i Beans nel class path.

È durante la fase di Bean Discovery che avviene la magia: CDI trasforma un POJO in un CDI Bean.

Al momento di deploy, il CDI controlla tutti i file jar e war dell'applicazione e ogni volta che trova un beans.xml gestisce tutti i POJOs facendoli diventare CDI Beans. Senza un file beans.xml nel class path (sotto la directory META-INF o WEB-INF), CDI non sarà in grado di utilizzare Injection, Interception, Decoration e così via. Se l'applicazione web contiene diversi file jar e si desidera avere CDI abilitati in tutta l'applicazione, ogni jar avrà bisogno del proprio beans.xml per attivare il rilevamento di CDI e bean discovery per ogni jar.

⊕ Come scrivere un Bean CDI

Un CDI Bean può essere qualsiasi tipo di classe che contiene business logic. Può essere chiamato direttamente dal codice Java tramite l'Injection o può essere richiamato tramite EL da una pagina JSF. Un bean:

- è un POJO che non eredita o estende qualcosa
- può iniettare riferimenti ad altri bean (@Inject),
- ha il suo ciclo di vita gestito dal container (@PostConstruct)
- e i suoi metodi possono essere intercettati (qui @Transaction è un legame di interceptor).

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 CDI Bean

Il container tratta ogni classe che soddisfa le seguenti condizioni come un CDI bean:

- non è una classe interna non static
- è una classe concreta o è annotata con @Decorator
- ha un costruttore di default senza parametri o dichiara un costruttore annotato con @Inject

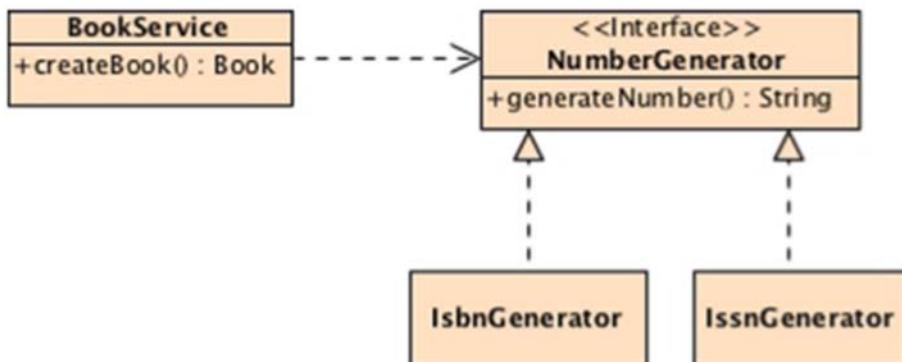
Un bean può avere uno scope opzionale, un nome EL opzionale, un insieme di interceptor bindings e una gestione opzionale del ciclo di vita.

Dependency Injection

Java è un linguaggio di programmazione orientato agli oggetti, il che significa che il mondo reale viene rappresentato usando gli oggetti. Ad esempio, abbiamo una classe Book, un Cliente e un PurchaseOrder che 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.

Il processo di creazione di un libro (BookService) può essere ridotto all'istanziazione di un Book, generando un numero univoco usando un NumberGenerator e rendendo persistente il libro in un database. Il NumberGenerator può generare un ISBN (13 cifre) o un ISSN (8 cifre). Il BookService finirebbe quindi per dipendere da un IsbnGenerator o da un IssnGenerator in base a ciò che viene richiesto.

La Figura 2-3 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 per un libro.



Come collegheresti un BookService all'implementazione ISBN dell'interfaccia NumberGenerator? Uno la soluzione è usare la parola chiave new come mostrato di seguito.

```
public class BookService {

    private NumberGenerator numberGenerator;

    public BookService() {
        this.numberGenerator = new IsbnGenerator();
    }

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

Il codice è piuttosto semplice, e fa il suo lavoro. Nel costruttore, il BookService crea un'istanza di IsbnGenerator e influisce sull'attributo numberGenerator. Invocando numeroGenerator.generateNumber () il metodo genererebbe un numero di 13 cifre. Ma cosa succede se si desidera scegliere tra le implementazioni e non essere collegati solo a IsbnGenerator? Una soluzione è passare l'implementazione al costruttore e lasciare ad una classe esterna di scegliere quale implementazione vuole usare (vedi codice seguente).

```
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;  
    }  
}
```

So now an external class could use the BookService with the implementation it needs.

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

Ciò illustra l'inversione del controllo: il controllo della creazione della dipendenza tra BookService e NumberGenerator viene invertito perché è assegnato ad una classe esterna, non alla classe stessa. Dal momento che colleghiamo le dipendenze da soli si parla di construction by hand

Nel codice precedente abbiamo usato il costruttore per scegliere l'implementazione (constructor injection), ma un altro modo comune è usare setter (setter injection). Tuttavia, invece di costruire dipendenze a mano, puoi lasciarlo fare a un iniettore (cioè, CDI).

@Inject

Poiché Java EE è un ambiente managed, non è necessario costruire manualmente le dipendenze, ma è possibile lasciare che sia il container a fare un inject. In poche parole, CDI Dependency Injection consiste nel fare l'inject di beans dentro altri beans in modo sicuro, il chè significa non nell'XML ma con le annotazioni.

L'injection già esisteva in Java EE 5 con le annotazioni @Resource, @PersistentUnit o @EJB, ad esempio. Ma era limitato a determinate risorse (datasource, EJB ...) e in determinati component (Servlets, EJBs ...). Con CDI è possibile iniettare quasi tutto ovunque grazie all'annotazione **@Inject**. Si noti che in Java EE 7 è ancora possibile utilizzare gli altri meccanismi di injection (@Resource ...) ma si dovrebbe considerare l'utilizzo di @Inject quando possibile.

Il Listato 2-4 mostra come iniettare un riferimento del NumberGenerator nel BookService usando il CDI @Inject.

```
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;  
    }  
}
```

L'annotazione @Inject informa il container che ha bisogno di iniettare un riferimento ad un'implementazione di NumberGenerator all'interno di numberGenerator. Questo è chiamato **injection point** (cioè il punto in cui l'annotazione @inject si trova).

Il seguente codice mostra come implementare IsbnGenerator. Non ci sono annotazioni speciali e la classe implementa l'interfaccia NumberGenerator.

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

Inject Point

L'annotazione `@Inject` definisce un punto di inject che viene iniettato durante l'istanziazione del bean. L'inject può avvenire tramite tre diversi meccanismi: proprietà(attributi), setter o costruttore. Fino ad ora, in tutti gli esempi di codice precedenti, hai visto l'annotazione `@Inject` sugli attributi (proprietà).

```
@Inject
private NumberGenerator numberGenerator;
```

Si noti che non è necessario creare getter e setter per un attributo che usa l'injection. CDI può accedere ai campi inject direttamente (anche se è private), e ciò aiuta ad evitare ridondanza di codice. Si può anche usare `@Inject` sul costruttore:

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

Ma si può avere solo un costruttore con `@Inject`. È il container che istanzia i bean quindi è permesso un solo costruttore.

L'altra scelta è quella di usare setter injection, simile a un constructor injection. C'è solo bisogno di annotare il setter con `@Inject`.

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

Default Injection

Supponiamo che NumberGenerator abbia solo un'implementazione (IsbnGenerator). Il CDI sarà quindi in grado di iniettarlo semplicemente usando `@Inject`.

```
@Inject
private NumberGenerator numberGenerator;
```

Questa è chiamata **default injection**. Ogni volta che un bean o punto di iniezione non dichiara esplicitamente un qualificatore, il container assume il qualificatore `@javax.enterprise.inject.Default`.

Infatti, il codice seguente è identico al precedente:

```
@Inject @Default
private NumberGenerator numberGenerator;
```

`@Default` è un qualificatore incorporato che informa CDI di iniettare l'implementazione bean di default. Se si definisce un bean senza qualificatore, il bean ha automaticamente il qualificatore `@Default`. Quindi il codice seguente è identico a quello precedente.

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

```
@Default
public class IsbnGenerator implements NumberGenerator {

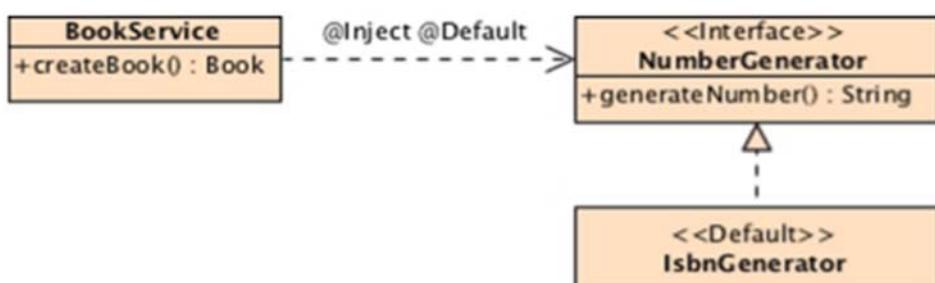
    public String generateNumber() {
        return "13-84356-" + Math.abs(new Random().nextInt());
    }
}
```

Quindi:

`@Inject` da solo equivale a `@Inject @Default`.

Se generiamo un Bean senza qualificatori viene considerato `@Default`.

Se dobbiamo scegliere tra più implementazioni servono i qualificatori.

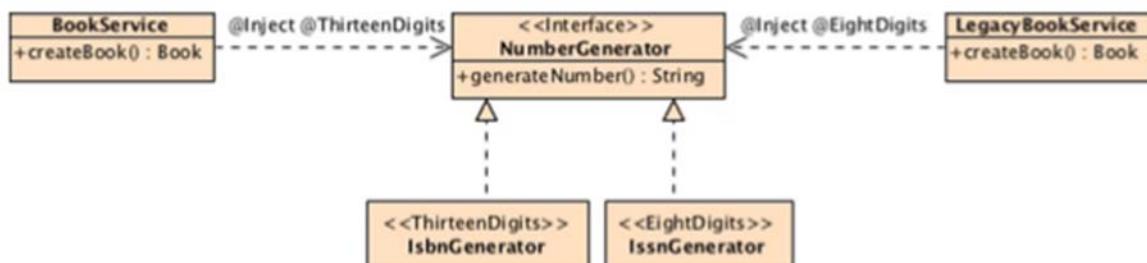


Qualifiers

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 conteiner ti informerà di unsatisfied dependency 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 contenitore 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 (IsbnGenerator o ISSNGenerator) 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. Diciamo ad esempio, di avere un'applicazione con un BookService che crea libri con un numero ISBN di 13 cifre e a LegacyBookService che crea libri con un numero ISSN a 8 cifre. Come potete vedere nella Figura 2-5, entrambi i servizi si iniettano un riferimento della stessa interfaccia NumberGenerator.

Le implementazioni dei due NumberGenerator dell'esempio si distinguono utilizzando i qualificatori.



Un qualificatore rappresenta una semantica associata a un tipo che è soddisfatto da alcune implementazioni di questo genere. 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 entrambi 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 si può 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 `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;
    }
}
```

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;  
    }  
}
```

Affinché ciò funzioni, non è necessaria la configurazione esterna; è per questo che si dice che CDI usa una strong typing. Puoi rinominare le implementazioni in base a ciò che si desidera, rinominare il qualificatore: il punto di iniezione non cambierà (questo è un loose coupling). Come puoi vedere, CDI è un modo elegante per fare un'iniezione di tipo typesafe. Ma se inizi a creare annotazioni ogni volta che devi iniettare qualcosa, la tua applicazione finirà per essere molto prolissa. Ed è in quei casi che i qualificatori con i membri possono aiutarti.

Qualifiers with Members

Ogni volta che si deve scegliere tra le varie implementazioni, viene creato un qualificatore. Ad esempio, oltre alle 13 e 8 cifre, potremmo volere generatori di numeri a due o 10 cifre (quindi annotazioni aggiuntive `@TwoDigits`, `@TenDigits`) o ancora, i numeri generati possono essere sia pari che dispari (quindi ulteriori annotazioni: `@TwoOddDigits`, `@TwoEvenDigits`, ecc).

Un modo per evitare tutte queste annotazioni, consiste nell'utilizzare i membri.

Nell'esempio, sostituiamo tutti questi qualificatori utilizzando solo `@NumberOfDigits` con un'enumerazione come valore e un Booleano per la parità.

```
@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 viene usato un qualificatore con membri è uguale a quello visto precedentemente.

L'injection point qualifica l'implementazione necessaria impostando i membri di annotazione come segue:

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

E l'implementazione interessata farà lo stesso:

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

Multiple Qualifiers

Un altro modo per qualificare un bean e un punto di iniezione è specificare più qualificatori. Quindi, invece di avere più qualificatori per parità (@TwoOddDigits, @TwoEvenDigits.). o con 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;
```

Quindi solo un bean con entrambe le annotazioni del qualificatore sarebbe idoneo per l'injection. I qualificati dovrebbero essere significativi. Avere i nomi e la giusta granularità dei qualificatori è importante per un'applicazione.

Alternatives

I qualificatori ti consentono di scegliere tra più implementazioni di un'interfaccia in fase di sviluppo. Ma a volte si desidera usare un'implementazione in base ad un particolare scenario. Ad esempio, potresti voler usare un generatore di numeri fintizi in un ambiente di test. Alternatives sono i bean annotati con il qualificatore speciale javax.enterprise.inject.Alternative. Di default le alternative sono disabilitate e devono essere abilitate nel descrittore beans.xml per renderle disponibili per l'istanziazione e l'inject. Il Listato 2-14 mostra un'alternativa del generatore di numeri fintizi.

```
@Alternative
public class MockGenerator implements NumberGenerator {

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

Come puoi vedere nel Listato 2-14, MockGenerator implementa l'interfaccia NumberGenerator come al solito. È annotato con **@Alternative**, il che significa che CDI lo considera come l'alternativa predefinita del NumberGenerator. Come in Listato 2-6, questa alternativa predefinita avrebbe potuto usare il qualificatore incorporato **@Default** come segue:

```
@Alternative @Default
public class MockGenerator implements NumberGenerator {...}
```

Invece di un'alternativa predefinita, è possibile specificare l'alternativa utilizzando i qualificatori. Ad esempio, il seguente codice dice a CDI che l'alternativa di un generatore di numeri a 13 cifre è il mock:

```
@Alternative @ThirteenDigits
public class MockGenerator implements NumberGenerator {...}
```

Di Default, i bean **@Alternative** sono disabilitati ed è necessario abilitarli esplicitamente nel descrittore beans.xml come mostrato:

```
<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

Abbiamo visto come fare l'inject di Beans CDI in altri Beans CDI. Grazie ai Producers, è anche possibile fare l'inject delle primitive(ad es. Int, long, float. . .), i tipi di array e qualsiasi POJO che non è abilitato CDI. Con CDI abilitato intendo qualsiasi classe racchiusa in un archivio contenente un file beans.xml.

Di Default, non è possibile inserire classi come java.util.Date o java.lang.String. Questo perché tutti questi le classi sono impacchettate nel file rt.jar (le classi dell'ambiente runtime Java) e questo archivio non contiene a un deployment descriptor beans.xml. Se un archivio non ha bean.xml sotto la directory META-INF, CDI non si può fare bean discover e i POJOs non potranno essere trattati come bean e, quindi, iniettabili. (Non sono classi che abbiamo scritto noi quindi non possiamo andare ad aggiungere i qualificatori nel codice) L'unico modo per fare l'inject di POJO è utilizzare i campi producers o i metodi producers come mostrato nel Listato 2-16.

```
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());  
    }  
}
```

La classe NumberProducer nel Listato 2-16 ha diversi attributi e metodi tutti con annotati javax.enterprise.inject.Produces. Ciò significa che ora è possibile iniettare tutti i tipi e le classi prodotte con @Inject usando un qualificatore (@ThirteenDigits, @EightDigits o @Random). Il metodo Produces (random () nel Listato 2-16) è un metodo che funge da fabbrica di istanze bean. Permette il valore di ritorno da iniettare.

Nel Listato 2-9 IsbnGenerator genera un numero ISBN con la formula "13-84356-" + Math.abs (nuovo Random () . NextInt ()). Usando il NumberProducer (Listato 2-16) possiamo usare i tipi prodotti per cambiarlo formula.

Nel Listato 2-17 IsbnGenerator ora inietta sia una stringa che un intero con @Inject @ThirteenDigits che rappresenta il prefisso ("13-") e l'identificatore dell'editor (84356) di un numero ISBN. Il numero casuale è iniettato con @Inject @Random e restituisce un doppio.

```
@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;  
    }  
}
```

Nel Listato 2-17 puoi vedere una forte digitazione. Utilizzando la stessa sintassi (@Inject @ThirteenDigits), CDI sa che è necessario iniettare una stringa, un intero o un'implementazione di un NumberGenerator. Il vantaggio di usare tipi iniettati (Listato 2-17) piuttosto che una formula fissa (Listato 2-9) per generare numeri è che puoi usare tutto il Funzionalità CDI come alternative (e se necessario un algoritmo generatore di numeri ISBN alternativo).

Disposer

Negli esempi precedenti abbiamo utilizzato i producers per creare tipi di dati o POJO da iniettare. Li abbiamo creati e non dovevamo distruggerli o chiuderli una volta usati. Ma alcuni metodi dei prodcers restituiscono oggetti che richiedono una distruzione esplicita, ad esempio una connessione Java Database Connectivity (JDBC).

Per la creazione, CDI utilizza i producer, e per la distruzione, i disposer(dissipatori). Il codice mostra una classe di utilità che crea e chiude una connessione JDBC.

Il metodo createConnection richiede a Derby Driver JDBC, crea una connessione con un URL specifico, gestisce le eccezioni e restituisce una connessione. Questo metodo è annotato con @Produces.

Il metodo closeConnection termina la connessione JDBC. È annotato con @Dispose.

```
@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();
}
```

La distruzione può essere eseguita con un metodo dispose definito dalla stessa classe del metodo del producer. Ogni metodo annotato con @Disposes, deve avere esattamente un parametro dispose dello stesso tipo (qui java.sql.Connection) e qualificatori (@Default) come il tipo restituito dal metodo corrispondente del producer (annotato @Produces). Il metodo del disposer(closeConnection()) viene chiamato automaticamente quando il contesto del client termina (nell'esempio il contesto è @ApplicationScoped).

Listing 2-20. JDBC Connection Producer and Disposer

```
@ApplicationScoped  
public class DerbyPingService {  
  
    @Inject  
    private Connection conn;  
  
    public void ping() throws SQLException {  
        conn.createStatement().executeQuery("SELECT 1 FROM SYSIBM.SYSDUMMY1");  
    }  
}
```

Scopes

CDI non riguarda solo la Dependency Injection ma anche il contesto (la "C" in CDI). Ogni oggetto gestito da CDI ha uno scope ben definito e un ciclo di vita vincolato ad un contesto specifico. In Java, l'ambito di un POJO è piuttosto semplice: si crea un'istanza di una classe utilizzando new e si affida al garbage collector per liberare la memoria. Con CDI, un bean è legato a un contesto e rimane in quel contesto finché viene distrutto dal container. Non esiste alcun modo per rimuovere manualmente un bean da un contesto. Mentre il web tier dispone di ambiti ben definiti (application, session, request), non esiste una cosa simile per il service tier. Ciò è dovuto al fatto che quando i session bean o i POJO vengono utilizzati nelle applicazioni Web, non sono consapevoli dei contesti delle applicazioni web. CDI ha messo insieme service e web tier associandoli a scope significativi. 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. Il bean viene creato una sola volta per tutta la durata dell'applicazione e viene scartato quando l'applicazione è chiusa. Questo ambito è utile per le classi di utilità o di supporto o per gli oggetti che memorizzano dati condivisi dall'intera applicazione (ma si dovrebbe prestare attenzione ai problemi di concorrenza quando i dati devono essere accessibili da più thread).
- **Session scope (@SessionScoped):** comprende diverse richieste 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 eliminato quando la sessione termina. Questo scope è per gli oggetti che sono necessari per la durata della sessione (esempi: preferenze di un utente o le credenziali di accesso).
- **Request scope (@RequestScoped):** corrisponde a una singola richiesta HTTP o a un'invocazione di metodo. Il bean viene creato per la durata dell'invocazione del metodo e viene eliminato quando il metodo termina.

- **Conversation scope (@ConversationScoped):** si estende tra più invocazioni all'interno dei confini della sessione con i punti di start ed end determinati dall'applicazione. Le conversazioni vengono utilizzate in più pagine come partedi un flusso di lavoro a più fasi.
- **Dependent pseudo-scope (@Dependent):** il ciclo di vita è lo stesso del client. Un dependent bean viene creato ogni volta che viene iniettato e il riferimento viene rimosso quando viene rimosso l'oggetto in cui è iniettato. Questo è lo **scope di default per CDI**.

Tutti gli scope hanno un'annotazione che puoi usare sui Beans CDI (tutte queste annotazioni sono in il pacchetto javax.enterprise.context). I primi tre ambiti sono ben noti. Un esempio si scope di sessione è un bean del carrello, il bean verrà creato automaticamente all'avvio della sessione (ad es., la prima volta che un utente accede) e vengono automaticamente distrutti al termine della sessione.

@SessionScoped

```
public class ShoppingCart implements Serializable {...}
```

Un'istanza del bean ShoppingCart è associata a una sessione utente ed è condivisa da tutte le richieste che vengono eseguite nel contesto di quella sessione. Se non si desidera che il bean rimanga nella sessione per tempo indefinito, bisognerebbe prendere in considerazione l'utilizzo di un altro scioe con una durata più breve, come request o conversation scope. Si noti che i bean con scope @SessionScoped o @ConversationScoped devono essere serializzabile, poiché il contenitore li passa di volta in volta.

Se un ambito non viene specificato esplicitamente, il bean appartiene al Dependent pseudo-scope (@Dependent). Beans con questo scope non vengono mai condivisi tra client diversi o punti di iniezione diversi. Sono dipendenti da qualcun altro Bean, il che significa che il loro ciclo di vita è legato al ciclo di vita di quel Bean. Un bean dipendente viene istanziato quando l'oggetto a cui appartiene viene creato e distrutto quando l'oggetto a cui appartiene viene distrutto. Il codice che segue mostra un generatore ISBN con scope dipendente con un qualificatore:

@Dependent @ThirteenDigits

```
public class IsbnGenerator implements NumberGenerator {...}
```

Essendo lo scope di default, puoi omettere l'annotazione @Dependent e scrivere quanto segue:

@ThirteenDigits

```
public class IsbnGenerator implements NumberGenerator {...}
```

Gli scope possono essere mescolati. Un bean @SessionScoped può essere iniettato in un @RequestScoped o @ApplicationScoped Bean e viceversa.

Conversation

L'ambito della conversazione è leggermente diverso dall'applicazione, dalla sessione o dallo scope della richiesta. Tiene lo stato associato con un utente, si estende su più richieste ed è demarcato a livello di codice dall'applicazione. Un bean `@ConversationScoped` può essere utilizzato per un processo a lungo termine in cui esiste un inizio e una fine precisi come ad esempio la navigazione tramite un wizard o l'acquisto di oggetti e su un negozio online. La richiesta di oggetti con scope ha una durata molto breve che di solito dura per una singola richiesta (richiesta o metodo http invocazione) mentre gli oggetti con scope di sessione durano per l'intera durata della sessione dell'utente. Ma ci sono molti casi in cui si ricade tra questi due estremi.

Ci sono alcuni oggetti del layer di presentazione che possono essere utilizzati in più di una pagina ma non in tutta la sessione. Per questo, CDI ha uno scope conversazionale speciale (`@ConversationScoped`).

Gli oggetti oggetto di conversazione hanno un ciclo di vita ben definito che inizia e termina esplicitamente utilizzando l'API `javax.enterprise.context.Conversation`.

Ad esempio, pensa a un customer che crea un'applicazione tramite un wizard web. La procedura è composta da tre passaggi. Nel primo passaggio, il cliente inserisce le informazioni di accesso. Nel secondo passaggio, il cliente inserisce dettagli di account come il nome, il cognome, l'indirizzo e l'indirizzo di posta elettronica. Nella tappa finale conferma tutte le informazioni raccolte e crea l'account.

Il Listato 2-21 mostra il Conversation Scope bean che implementa la procedura guidata del creatore del cliente.

```
@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 a CustomerService, per creare il Cliente, ma più importante, inietta una Conversation. Questa interfaccia consente il controllo programmatico sul ciclo di vita del Conversation Scope. Si noti che quando il metodo saveLogin è invocato, la conversazione inizia (conversation.begin()). La conversazione è ora iniziata e viene utilizzata per durata della procedura guidata. Una volta richiamato l'ultimo passo della procedura guidata, viene richiamato il metodo createCustomer e il metodo la conversazione termina (conversation.end()).

✚ Interceptors

Gli interceptors consentono di aggiungere cross-cutting concernes (soluzioni trasversali) ai bean. Come mostrato nella figura, quando un client invoca un metodo su un Managed Bean (e quindi un CDI Bean, un EJB, un servizio web RESTful ...), 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. Interceptor associati a un costruttore della classe Target (@AroundConstruct)
- **Method-level interceptors:** Interceptors a livello di metodi. Interceptor associati a uno specifico metodo business (@AroundInvoke).
- **Timeout method interceptors:** Interceptors di un metodo Timeout. Interceptor che si interpone sui metodi di timeout con @AroundTimeout (utilizzato solo con il servizio Timer EJB).
- **Life-cycle callback interceptors:** Interceptors richiamati sul ciclo di vita. Interceptor che si interpone sul ciclo di vita dell'istanza di destinazione event callback (@PostConstruct e @PreDestroy).

Target Class Interceptors

Ci sono diversi modi per definire l'interception. Il più semplice è quello di aggiungere interceptors al bean stesso come mostrato in figura. CustomerService annota logMethod () con @AroundInvoke.

logMethod() è utilizzato per visualizzare un messaggio quando si entra o si esce da un metodo. Una volta che viene fatto il deploy di questo Managed Bean, ad ogni invocazione di

```
@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());
        }
    }
}
```

createCustomer () o findCustomerById () verrà applicato logMethod (). Si noti che l'ambito di questo interceptor è limitato al bean stesso (la classe target).

Nonostante sia annotato con @AroundInvoke, logMethod () deve avere il seguente schema di firma:

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

Le seguenti regole si applicano al metodo around-invoke (ma anche al costruttore, al time-out, o interceptors lyfe-cycle)

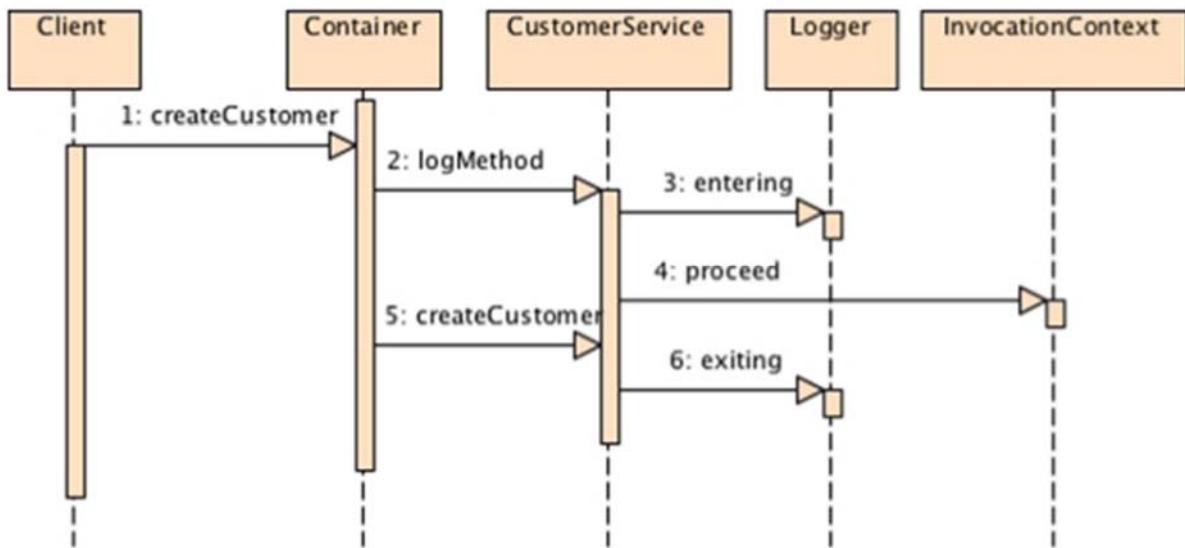
- ⑩ Il metodo può avere accesso public, private, protected, o package-level, ma non deve essere static o final
- ⑩ Il metodo deve avere un parametro javax.interceptor.InvocationContext e deve restituire un Object, il quale è il risultato del metodo target invocato.
- ⑩ il metodo può lanciare una checked exception

L'oggetto InvocationContext permette all'interceptor di controllare il comportamento della catena di invocazione. Se molti interceptors sono in catena, la stessa instanza di InvocationContext è passato ad ogni interceptor che può aggiungere contextual data.

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.

Per spiegare come funziona il codice nel listato 2-23, diamo un'occhiata al diagramma di sequenza mostrato nella Figura 2-6 per vedere cosa succede quando un client richiama il metodo createCustomer (). Prima di tutto, il contenitore intercetta il chiamata e, invece di elaborare direttamente createCustomer (), richiama prima il metodo logMethod (). logMethod () utilizza l'interfaccia InvocationContext per ottenere il nome del bean richiamato (ic.getTarget ()) e il metodo richiamato (ic.getMethod ()) per registrare un messaggio di entrata (logger.entering ()). Quindi, viene chiamato il metodo proceed (). Chiamare InvocationContext.proceed () è estremamente importante in quanto indica al contenitore che dovrebbe procedere alla successiva interceptors o chiamare il metodo di business del bean. Non chiamare proceed () fermebbe la catena degli interceptor e lo farebbe evitare di chiamare il business method. Il createCustomer () viene infine invocato e, una volta restituito, l'intercettatore termina la sua esecuzione registrando un messaggio di uscita (logger.exiting ()).



Class Interceptors

Nell'esempio di prima avevamo un metodo che intercettava le chiamate, ma possiamo anche avere l'interceptor in una classe separata per poter intercettare i metodi di molti beans (es.: logging). Per specificare una classe interceptor si deve sviluppare una classe separata e istruire il container affinché applichi questa classe a specifici beans o ai loro metodi.

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

```

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());
        }
    }
}

```

Il LoggingInterceptor può ora essere avvolto in modo trasparente da qualsiasi bean interessato a questo intercettore. Per fare questo, il bean deve informare il contenitore con `@javax.interceptor.Interceptors.annotation`. Nel Listato 2-25, l'annotazione è impostata sul metodo `createCustomer ()`. Ciò significa che qualsiasi invocazione di questo metodo sarà intercettato dal contenitore e la classe LoggingInterceptor verrà invocata (registrazione di un messaggio in entrata e in uscita del metodo)

```
@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);  
    }  
}
```

Nel Listato 2-25, `@Interceptors` è solo collegato al metodo `createCustomer ()`. Questo significa che se un cliente invoca `findCustomerById ()`, il contenitore non intercetterà la chiamata. Se si desidera che le chiamate ad entrambi i metodi vengano intercettate, è possibile aggiungere l'annotazione `@Interceptors` su entrambi i metodi o sul bean stesso. Quando lo fai quindi, l'interceptor viene attivato se viene invocato uno dei due metodi. E poiché l'intercettore ha un `@AroundConstruct`, anche la chiamata al costruttore verrà intercettata (Qua viene chiamato su tutti i metodi della classe).

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

Se il tuo bean ha diversi metodi e vuoi applicare un intercettore all'intero bean eccetto per uno specifico metodo, è possibile utilizzare l'annotazione `javax.interceptor.ExcludeClassInterceptors` per escludere un metodo dall'essere intercettati. Nel seguente codice, la chiamata a `updateCustomer ()` non verrà intercettata, ma tutti gli altri si:

```

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

```

Life-Cycle Interceptor

I life-cycle interceptor permettono di isolare del codice in una classe ed invocarla quando avviene un life-cycle event.

Con un'annotazione di richiamata, 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 necessità di catturare gli eventi del ciclo di vita attraverso molti tipi di Bean? Gli intercettori del ciclo di vita consentono di isolare del codice in una classe e invocarla quando viene attivato un evento del ciclo di vita. Il Listato 2-26 mostra la classe ProfileInterceptor con due metodi: logMethod () , usato per la postcostruzione (@PostConstruct) e profile () , utilizzati 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");
        }
    }
}

```

Come puoi vedere nel Listato 2-26, gli intercettori del ciclo di vita prendono un parametro InvocationContext e restituiscono void invece di Object. Per applicare l'intercettore definito nel Listato 2-26, è necessario il bean CustomerService (Listato 2-27) per utilizzare l'annotazione @Interceptors e definire ProfileInterceptor. Quando il bean viene istanziato dal container, il logMethod () sarà invocato prima del metodo init (). Quindi, se un client chiama createCustomer () o findCustomerById (), il metodo profile () verrà richiamato.

Listing 2-27. CustomerService Using an Interceptor and a Callback Annotation

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

Chaining and Excluding Interceptors

Si possono chiamare anche più interceptors indicandoli nell'annotazione @Interceptors separati da una virgola. L'ordine in cui sono chiamati è quello con cui compaiono nell'annotazione. Si può anche indicare di non invocare gli interceptors definiti su tutto il bean con l'annotazione @ExcludeClassInterceptors.

Listing 2-28. CustomerService Chaining Several 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) {...}  
}
```

Quando un client chiama il metodo updateCustomer (), non viene richiamato l'interceptor perché il metodo è annotato con @ExcludeClassInterceptors. Quando viene chiamato il metodo createCustomer (), viene eseguito l'interceptor I1 dall'intercettore I2. Quando viene invocato il metodo findCustomerById (), gli interceptor I1, I2, I3 e I4 vengono eseguiti in quest'ordine.

InterceptorBinding

Se si esamina il codice riportato sopra si nota come è necessario specificare l'implementazione dell'intercettore direttamente nell'implementazione del bean (ad esempio, @Interceptors (LoggingInterceptor.class)). Questo è typesafe, ma non debolmente accoppiato. CDI fornisce l'interceptor binding che introduce un livello di riferimento indiretto e diminuisce l'accoppiamento. Un interceptor binding è un'annotazione definita dall'utente annotata con @InterceptorBinding che lega la classe interceptor al bean senza alcuna dipendenza diretta tra le due classi. Un interceptor binding può avere dei qualificatori

Il Listato 2-29 mostra un binding intercettore chiamato Loggable. Come puoi vedere, questo codice è molto simile a un qualificatore. Un Interceptor Bindong 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 { }
```

Una volta che hai un Binding Interceptor, devi collegarlo all'intercettore stesso. Questo viene fatto annotando sia l'interceptor con @Interceptor e il Binding Interceptor con (@Loggable nel Listato 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());  
        }  
    }  
}
```

Ora è possibile applicare l'intercettore a un bean annotando la classe bean con lo stesso intercettore come mostrato nel Listato 2-31. Questo ti dà un accoppiamento lento (poiché la classe di implementazione dell'intercettore non è esplicitamente dichiarato) e un buon livello di riferimento indiretto.

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);  
    }  
}
```

Nel Listato 2-31 il Binding Interceptor è sul bean, il che significa che ogni metodo verrà intercettato e registrato. Ma come gli intercettori, è possibile applicare un legame di intercettatore a un metodo anziché a un intero bean.

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

Gli intercettori sono specifici dell'implementazione e sono disabilitati per default. Come alternative, gli intercettori devono essere abilitati utilizzando il Deployment Descriptor CDI beans.xml del jar o del modulo Java EE come mostrato nel Listato 2-32.

```
<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

L'interceptor binding non ci dà la possibilità di ordinare gli interceptor come facevamo con (@Interceptors({I1.class, I2.class})). Per questo CDI definisce la priorità degli interceptors usando l'annotazione @Priority come mostrato di seguito.

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 interceptor con valori di priorità minore sono Eseguiti per prima. Java EE 7 definisce le priorità a livello di piattaforma e quindi è possibile chiamare gli intercettori prima o dopo determinati event. Javax.interceptor. L'annotazione javax.interceptor.Interceptor definisce la seguente serie di costanti:

- **PLATFORM_BEFORE = 0**: inizio del range per early interceptor definito dalla piattaforma Java EE,
- **LIBRARY_BEFORE = 1000**: inizio del range per early interceptor definito dalle librerie di estensione,
- **APPLICATION = 2000**: inizio del range per interceptor definiti dalle applicazioni,
- **LIBRARY_AFTER = 3000**: Inizio del range per late interceptor definiti dalle librerie di estensione e
- **PLATFORM_AFTER = 4000**: inizio del range per late interceptor definiti dalla piattaforma Java EE.

Quindi, se si vuole che l'interceptor venga eseguito prima di ogni interceptor di applicazione, ma dopo qualsiasi interceptor della piattaforma, si può scrivere quanto segue:

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

Decorators

Gli intercettori eseguono compiti trasversali. Per natura gli interceptors non sono consapevoli della semantica reale delle azioni che intercettano e quindi non sono appropriati per separare i problemi business-related. Per i decorators è vero il contrario.

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.

Interceptors e Decorators, però 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 decorator. La classe FromEightToThirteenDigitsDecorator è annotato con javax.decorator.Decorator, implementa le interfacce di business (il NumberGenerator definito in Figura 2-3) e sostituisce il metodo generateNumber (un decoratore può essere dichiarato come una classe astratta in modo che lo faccia non è necessario implementare tutti i metodi di business delle interfacce se ce ne sono molti). Il metodo generateNumber () invoca 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;
    }
}
```

Un decorator deve avere un injection point annotato con @Delegate che è dello stesso tipo del bean che decora. Questo permette al decoratore di invocare business method su quell'oggetto. Come alternatives e interceptor i decorators sono disabilitati e vanno abilitati nel bean.xml

```
<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">

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

⊕ Eventi

Mentre DI, alternatives, interceptors 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. Un bean può definire un evento, un altro bean può attivare l'evento e un altro bean può gestirlo. Tutto ciò, seguendo uno schema che segue l'Observer pattern.

L'event producer fa partire gli eventi tramite l'interfaccia javax.enterprise.event.Event. Un producer provoca gli eventi chiamando fire(), passa l'oggetto dell'event e non dipende dall'observer.

Nel Listato 2-36 il BookService genera un evento (bookAddedEvent) ogni volta che viene creato un libro. Il codice bookAddedEvent.fire (libro) lancia l'evento e lo notifica a tutti i metodi Observer che osservano questo particolare evento. Il contenuto di questo evento è il libro oggetto stesso che sarà 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;  
    }  
}
```

BookService fa partire l'evento ogni volta che un libro viene creato e notifica ogni metodo observer che sta osservando questo evento.

Gli eventi sono fatti partire dagli event producer e sottoscritti dall'osservatore di eventi. L'observer è un bean con uno o più metodi observer. Questi observer prendono un evento 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. Quando l'evento viene chiamato, il CDI container interrompe l'esecuzione e passa all'observer registrato, per poi riprendere da dove era stato interrotto. Gli eventi in CDI non sono quindi asincroni.

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);  
    }  
}
```

Come la maggior parte dei CDI, la produzione e la sottoscrizione degli eventi sono tipicamente sicuri e consentono ai qualificatori di determinare quale evento gli observer osserveranno. Ad un evento possono essere assegnati uno o più qualificatori (con o senza membri), che consente agli osservatori di distinguere da altri eventi dello stesso tipo. Il Listato 2-38 rivisita il BookService bean aggiungendo un evento extra. Quando viene creato un libro, viene generato un libroAddedEvent e quando viene rimosso un libro, viene attivato un libroRemovedEvent, sia di tipo Book. Per distinguere entrambi gli eventi, ognuno è qualificato da @Aggiunto o da @Rimosso.

Il codice di questi qualificatori è identico al codice nel Listato 2-7:
un'annotazione senza membri e annotata con @Qualifier.

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);
    }
}
```

InventoryService nel Listato 2-39 osserva entrambi gli eventi dichiarando due metodi separati che osservano entrambi l'evento aggiunto al libro (@Observes @Added Book) o l'evento rimosso dal libro (@Observes @Removed 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.warning("Adding book " + book.getTitle() + " to inventory");
        inventory.add(book);
    }

    public void removeBook(@Observes @Removed Book book) {
        logger.warning("Removing book " + book.getTitle() + " to inventory");
        inventory.remove(book);
    }
}
```

Poiché il modello di eventi utilizza i qualificatori, si può approfittare di avere membri su tali qualificatori o aggregandoli. Il codice che segue osserva tutti i libri aggiunti che hanno un prezzo superiore a 100 :

```
void addBook(@Observes @Added @Price(greaterThan=100) Book book)
```

Java Persistence API

Le applicazioni sono costituite da business logic, interazione con altri sistemi, interfacce utente... e dati. La maggior parte dei dati che le nostre applicazioni manipolano devono essere archiviati in database, recuperati e analizzati. I database sono importanti: memorizzano i business data, fungono da punto centrale tra le applicazioni e elaborano i dati tramite i trigger o stored procedure. I dati persistenti sono ovunque e il più delle volte usano database relazionali per essere memorizzati. I database relazionali memorizzano i dati in tabelle fatte di righe e colonne. I dati sono identificati da chiavi primarie, che sono colonne speciali con vincoli di unicità e, a volte, indici. Le relazioni tra le tabelle utilizzano chiavi esterne e uniscono tabelle con vincoli di integrità. Tutto questo vocabolario è completamente sconosciuto in un linguaggio orientato agli oggetti come Java. In Java, manipoliamo oggetti che sono istanze di classi. Gli oggetti ereditano dagli altri, hanno riferimenti a collezioni di altri oggetti, e a volte indicano se stessi in modo ricorsivo. Abbiamo classi concrete, astratte, interfacce, enumerazioni, annotazioni, metodi, attributi e così via. Gli oggetti incapsulano lo stato e il comportamento in un modo pulito, ma questo stato è accessibile solo quando la Java Virtual Machine (JVM) è in esecuzione: se la JVM si ferma o il garbage collector pulisce il suo contenuto di memoria, gli oggetti scompaiono, così come il loro stato. Alcuni oggetti devono essere persistenti. Con dati persistenti, intendo dati che sono memorizzati deliberatamente in forma permanente su supporti magnetici, memorie flash e così via. Un oggetto che può memorizzare il suo stato per essere riutilizzato in seguito si dice persistente.

Il principio della mappatura oggetto-relazionale (ORM, object-relational mapping) è quello di riunire il mondo del database e degli oggetti. Esso comporta la delega dell'accesso ai database relazionali a strumenti o framework esterni, che a loro volta forniscono un oggetto orientato agli oggetti vista dei dati relazionali e viceversa. Gli strumenti di mappatura hanno una corrispondenza bidirezionale tra il database e oggetti. Diversi framework ottengono questo risultato, come Hibernate, TopLink e Java Data Objects (JDO), ma Java Persistence API (JPA) è la tecnologia preferita e fa parte di Java EE 7. Questo capitolo è un'introduzione all'APP, e nei due capitoli seguenti mi concentrerò su ORM e su query e gestione di oggetti persistenti.

Understanding Entities

Quando si parla della mappatura di oggetti in un database relazionale, di oggetti persistenti o di query su oggetti,

dovrebbe essere usato il termine "entità" piuttosto che "oggetto". Gli oggetti sono istanze che vivono solo nella memoria. Le entità sono oggetti che vivono brevemente nella memoria ma in modo persistente in un database.

in memoria e persistentemente in un database. Hanno la capacità di essere mappati su un database; possono essere concreti o astratto; e supportano l'ereditarietà, le relazioni e così via. Queste entità, una volta mappate, possono essere gestite da JPA. È possibile mantenere un'entità nel database, rimuoverla e interrogarla utilizzando un linguaggio di query Java Persistence Query Linguaggio (JPQL). ORM consente di manipolare le entità, mentre dietro le quinte si accede al database. E,

come vedrai, un'entità segue un ciclo di vita definito. Con i metodi di callback e i listeners, JPA ti consente di collegarne alcuni business code agli eventi del ciclo di vita.

Come primo esempio, iniziamo con l'entità più semplice che possiamo avere. Nel modello di persistenza JPA. 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 (il suo stato) che possono essere manipolati tramite getter e setter.

Il Listato 4-1 mostra un'entità semplice.

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
}
```

L'esempio nel Listato 4-1 rappresenta un'entità del libro da cui ho omesso i getter e i setter per chiarezza.

Come si vede, ad eccezione di alcune annotazioni, questa entità appare esattamente come qualsiasi classe Java: ha diversi attributi (id, titolo, prezzo, ecc.) di diversi tipi (Long, String, Float, Integer e Boolean), un costruttore predefinito e getter e setter per ogni attributo.

Quindi, come facciamo a mapparlo in una tavola? La risposta è grazie alle annotazioni.

Anatomy of an Entity

Affinché una classe diventi un'entità deve essere annotata con `@ javax.persistence.Entity`, che consente al persistence provider di riconoscerla come una classe persistente e non solo come un semplice POJO. Quindi, l'annotazione `@ javax.persistence.Id` definisce l'identificativo univoco di questo oggetto. Perché JPA riguarda il mapping degli oggetti su tabelle relazionali, gli oggetti necessitano di un ID che verrà mappato su una chiave primaria. Gli altri attributi nel Listato 4-1 (titolo, prezzo, descrizione, ecc.) non sono annotati, quindi verranno resi persistenti applicando una mappatura di default.

Questo esempio di codice ha solo attributi, ma, come vedrai in seguito, un'entità può anche avere business method.

Nota che questa entità Book è una classe Java che non implementa alcuna interfaccia o estende alcuna classe.

In effetti, per essere un'entità, una classe deve seguire queste regole:

- La classe entità deve essere annotata con `@ javax.persistence.Entity` (o denotata nel descrittore XML come entità).
- L'annotazione `@ javax.persistence.Id` deve essere utilizzata per indicare una semplice chiave primaria.
- La classe entità deve avere un costruttore senza parametri (vuoto) che deve essere pubblico o protetto. È possibile avere anche altri costruttori.
- 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 (ad es. Attraverso un'interfaccia remota), la classe entità deve implementare l'interfaccia Serializzabile.

Object-Relational Mapping

L'ORM delega ad un tool esterno o ad un framework (nel nostro caso, JPA) il compito di creare una corrispondenza tra oggetti e tabelle. Le classi, gli oggetti e gli attributi vengono quindi mappati in un database relazionale costituiti da tabelle contenenti righe e colonne. Il Mapping offre una vista object-oriented agli sviluppatori che possono dunque utilizzare in modo trasparente entità anziché tabelle.

JPA mappa gli oggetti in un Database attraverso i Metadata.

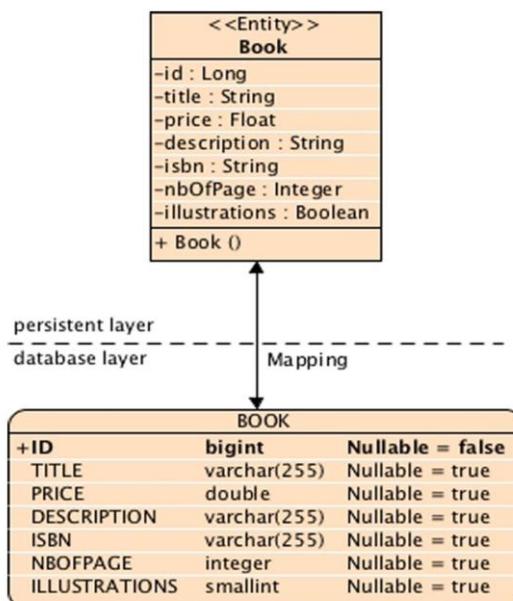
Associato ad ogni entità v'è il metadata che ne descrive il mapping. I metadati permettono al persistance provider di riconoscere le entità e interpretare il mapping.

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. Ciò può essere utile quando la configurazione del database cambia a seconda dell'ambiente.

L'entità vengono mappate in una tabella e ogni colonna prende il nome dall'attributo della classe.

Come mostra la Figura 4-1, l'entità Book è mappata in una tabella BOOK e ogni colonna prende il nome dall'attributo di la classe (ad esempio, l'attributo isbn di tipo String è mappato a una colonna denominata ISBN di tipo VARCHAR).



Java EE 5 ha introdotto l'idea di **configurazione per eccezione**, in cui il container o il provider applicano regole di default, a meno che non venga specificato diversamente. Di conseguenza, basta scrivere una minima quantità di codice per poter avviare l'applicazione, facendo affidamento ai defaults del container e del provider. Se non si desidera che il provider applichi le regole di default, è comunque possibile personalizzare il mapping usando i metadata. Senza alcuna annotazione, l'entità Book nel Listato 4-1 verrebbe trattata come un POJO e non sarebbe persistente. Questa è la regola: se non viene fornita alcuna configurazione speciale, è necessario applicare l'impostazione di default e l'impostazione predefinita per il persistence provider è che la classe Book non viene rappresentata in database.

Ma poichè è necessario modificare questo comportamento di default, bisogna annotare la classe con `@Entity`. È lo stesso vale per l'identificatore. Hai bisogno di un modo per dire al persistence provider che l'attributo id deve essere mappato su una chiave primaria, quindi va annotato con `@Id` e il valore di questo identificatore viene **generato automaticamente dal persistence provider, utilizzando l'annotazione facoltativa `@GeneratedValue`.**

Ciò significa che, per tutti gli altri attributi, si applicano le seguenti regole di mappatura predefinite:

- L'entità è mappata in una tabella dallo stesso nome (l'entity Book è mappata su una tavola che si chiama Book). Se si desidera mapparlo su un'altra tabella, si usa la notazione `@Table`.
- Gli attributi sono mappati su una colonna che hanno lo stesso nome (ad esempio, l'attributo id o il metodo `getId()` sono mappati su una colonna ID). **Se si desidera modificare questa mappatura di default, si usa `@Column`.**

Le regole JDBC sono applicate per mappare le primitive Java in tipi di dati relazionali. Una stringa sarà mappato a VARCHAR, a Long to a BIGINT, a Boolean a SMALLINT e così via.

Listing 4-2. Script Creating the BOOK Table Structure

```
CREATE TABLE BOOK (
    ID BIGINT NOT NULL,
    TITLE VARCHAR(255),
    PRICE FLOAT,
    DESCRIPTION VARCHAR(255),
    ISBN VARCHAR(255),
    NBOFPAGE INTEGER,
    ILLUSTRATIONS SMALLINT DEFAULT 0,
    PRIMARY KEY (ID)
)
```

JPA 2 ha un'API e un meccanismo standard per generare automaticamente il database dalle entità e generare script come il Listato 4-2. Questa funzione è molto comoda quando si è in modalità sviluppo. Però, il più delle volte è necessario connettersi a un database legacy già esistente.

Come vedrai nel prossimo capitolo, la mappatura può essere molto più ricco, consentendo di mappare di tutto, dagli oggetti alle relazioni.

Querying Entities

JPA ti permette di mappare le entità in un database e interrogare quest'ultimo usando diversi criteri. La potenza di JPA sta nel fatto che consente di interrogare le entità e le loro relazioni in modo object-orientend senza dover utilizzare direttamente le chiavi esterne (foreign key) o le colonne. Il pezzo centrale responsabile dell'organizzazione delle entità è `javax.persistence.EntityManager`.

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 and delate) e query complesse che utilizzano JPQL. Il seguente snippet di codice mostra come ottenere un Entity Manager e rendere persistente un'entity Book.

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

Nella Figura 4-2, è possibile vedere come l'interfaccia EntityManager può essere utilizzata da una classe (qui Main) per manipolare entità (in questo caso, Book). Con metodi come persist () e find (), l'Entity Manager nasconde le chiamate JDBC al database e le istruzioni INSERT o SELECT SQL (Structured Query Language).

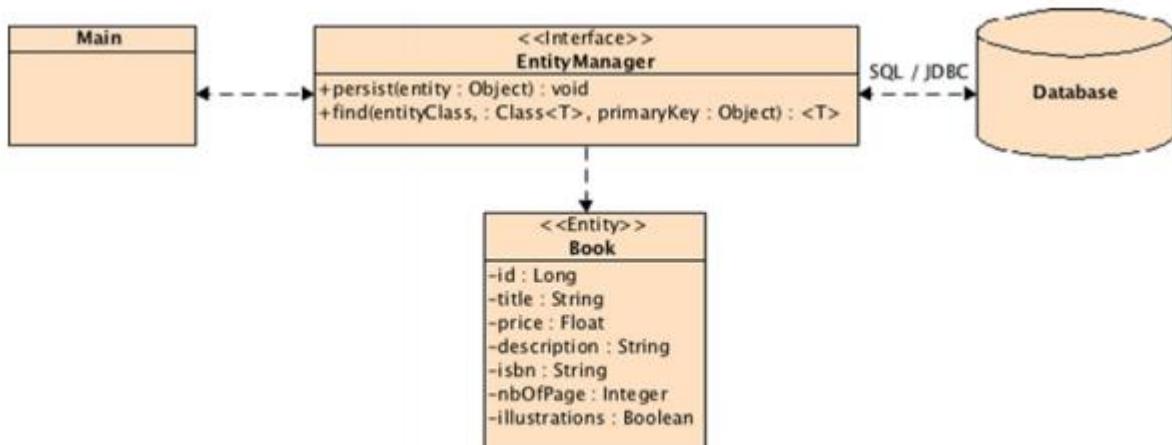


Figure 4-2. The entity manager interacts with the entity and the underlying database

L'Entity Manager consente anche di interrogare le entità. Una query in questo caso è simile a una query di database, ad eccezione che, invece di usare SQL, si utilizza JPQL. La sua sintassi utilizza la nota notazione dell'oggetto punto (.) .

Per recuperare tutti i libri che hanno il titolo H2G2, è possibile scrivere quanto segue:

```
SELECT b FROM Book b WHERE b.title = 'H2G2'
```

Si noti che title è il nome dell'attributo Book, non il nome di una colonna in una tabella. Dichiarazioni JPQL manipolano oggetti e attributi, non tabelle e colonne. Un'istruzione JPQL può essere eseguita con query dinamiche (creato dinamicamente aruntime) o query statiche (definite staticamente al momento della compilazione). Le **query statiche, note anche come Named Query**, vengono definite utilizzando le annotazioni (@NamedQuery) o con i metadati XML.

Il Listato 4-3 mostra un'entità Book che definisce la query denominata findBookH2G2 usando il l'annotazione @NamedQuery.

Listing 4-3. Book Entity with a findBookH2G2 Named Query

```
@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
}
```

L'Entity Manager si ottiene in una classe Java standard usando un factory. La seguente classe mostra la creazione di una istanza di entità Book, fornendogli persistenza e chiamando un named query. Essa segue i seguenti cinque step

1. Creare una istanza di una entità Book: si crea tramite “new” come un POJO
2. Ottenere un' EntityManager e una transaction: Questa parte è fondamentale, in quanto un Entity Manager è necessario per manipolare le entità. Per prima cosa, viene creata una entityManager factory per la persistance unit “chapter04PU”. Il factory è dunque usato per ottenere un Entity Manager (variabile em) che verrà usato per ottenere una transazione (tx) e per dare persistenza e prendere un Book.
3. Dare persistenza ad un book in un database: Il codice inizia una transaction (tx.begin()) e usa il metodo EntityManager.persist() per inserire un'istanza di Book. Quando la transaction è commit (tx.commit()), i dati sono portati nel database.
4. Eseguire la Named Query: l'entity manager permette di ottenere un book usando la named query findBookH2G2.
5. Chiudere l'Entity Manager e l'Entity Manager factory.

Listing 4-4. A Main Class Persisting and Retrieving a Book Entity

```
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();
    }
}
```

Si noti l'assenza di query SQL o chiamate JDBC. Come mostrato nella Figura 4-2, la classe principale interagisce con il database sottostante tramite l'interfaccia EntityManager, che fornisce un insieme di metodi standard che permettono di eseguire operazioni sull'entity Book. Dietro le quinte, EntityManager fa affidamento sul persistence provider per interagire con i database

Persistance Unit

Quale driver JDBC bisogna usare? Come ci si connette al database? Qual è il nome del database? Queste informazioni non sono presenti nel codice precedente. Quando la classe Main (Listato 4-4) crea una EntityManagerFactory, gli passa come parametro il nome di una Persistence Unit (in questo caso, si chiama chapter04PU). La Persistence Unit indica all'Entity Manager il tipo di database da utilizzare e i parametri di connessione, che sono definiti nel file `persistence.xml`, mostrato nel Listato 4-5, che deve essere accessibile attraverso il path della classe.

Listing 4-5. The `persistence.xml` File Defining the Persistence Unit

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
    version="2.1">

    <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>
```

Il chapter04PU persistance unit definisce una connessione JDBC per il chapter04DB Derby database avviato sul localhost e sulla porta 1527. Si connette tramite un user (APP) e password (APP) ad un URL dato. Il tag `<class>` comunica al persistance provider di dover gestire la classe Book.

Entity Life Cycle and Callbacks

Le entità sono solo POJO. Quando l'Entity Manager gestisce i POJO, queste hanno un'identità di persistenza (una chiave che le identifica in modo univoco; equivalente ad una chiave primaria) e il database sincronizza il loro stato. Quando sono non gestiti (cioè, sono detached dall'Entity Manager), possono essere utilizzati come qualsiasi altra classe Java. Questo significa che tali entità hanno un ciclo di vita, come mostrato nella Figura 4-3.

Quando si crea una istanza di Book tramite “new”, l’oggetto esiste nella memoria e la JPA non ne conosce nulla a riguardo. Quando poi l’entity manager gestisce l’oggetto, esso viene mappato e il suo stato viene sincronizzato, finché il metodo l’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

CHAPTER 4 ■ JAVA PERSISTENCE API

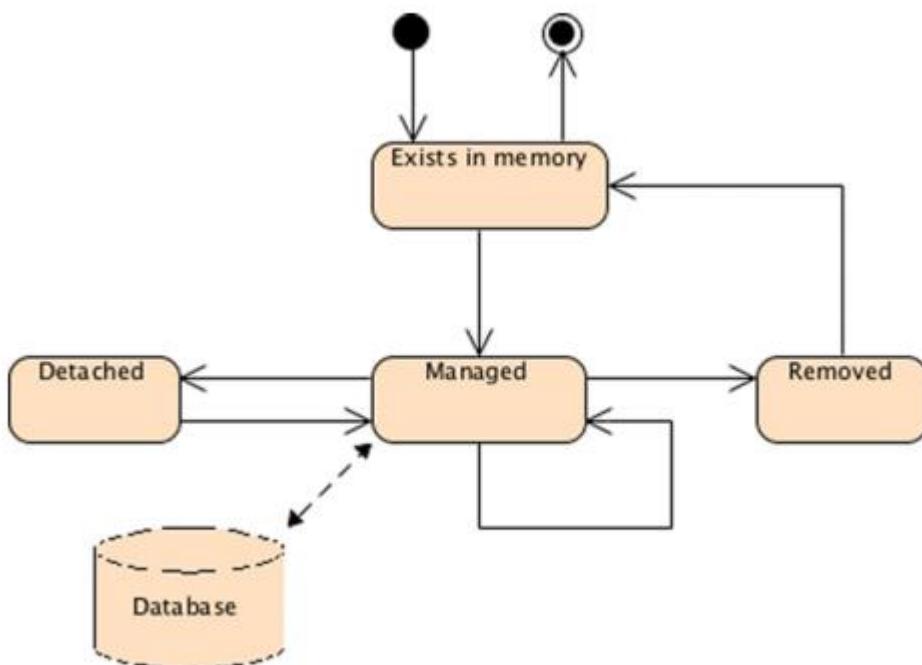


Figure 4-3. The life cyle of an entity

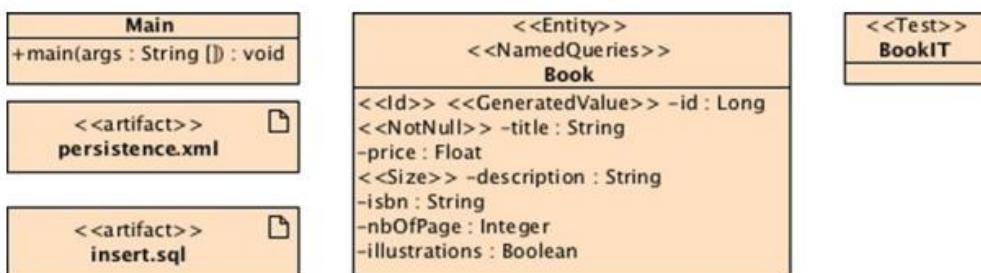
Le operazioni effettuate alle entità rientrano in quattro categorie: persisting, updating, removing and loading, che corrispondono alle operazioni del database di inserimento, aggiornamento, cancellazione e selezione, rispettivamente.

Ogni operazione ha un evento "pre" e "post" (ad eccezione del caricamento, che ha solo un evento "post") che può essere intercettato dall'Entity Manager per invocare un metodo di business.

Putting it all together

L'idea è quella di scrivere un'entità di Book semplice con **vincoli di Bean Validation** e una classe Main che rende persistente un libro.

- src/main/java contiene Book e la classe Main
- src/main/resources contiene persistence.xml e insert.sql
- src/test/java con la classe BookIT (usata per il testing)
- pom.xml: For the Maven POM, which describes the project and its dependencies on other external modules and components



Writing the Book Entity

Spieghiamo le annotazioni usate dalla classe Book.

- `@Entity` informa il persistence provider che la classe è un entità e che dovrebbe gestirla.
- `@NamedQueries` e `@NamedQuery` definiscono delle query che usano JPQL per trovare un book nel database
- `@Id` dice che l'attributo è una chiave primaria
- `@GeneratedValue` informa il persistence provider che deve autogenerare la chiave usando utility id.

```
package org.agoncal.book.javaee7.chapter04;
@Entity
@NamedQueries({
    @NamedQuery(name = "findAllBooks", query = "SELECT b FROM Book b"),
    {"SELECT b FROM Book b WHERE b.title = 'H2G2'"}
})
public class Book {

    @Id @GeneratedValue
    private Long id;
    @NotNull
    private String title;
    private Float price;
    @Size(min = 10, max = 2000)
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;

    // Constructors, getters, setters
}
```

Writing the Main Class

Il main crea un nuovo libro e ne setta alcuni attributi. Poi usa la classe Persistence per ottenere un' EntityManagerFactory che si riferisce ad una persistence unit "chapter04PU". Questa factory cre un'istanza di un EntityManager. Questo è il pezzo centrale di JPA in quanto è in grado di creare una transazione, e rendere persistente l'oggetto book con il metodo persist() e fare il commit della transazione.

```
package org.agoncal.book.javaee7.chapter04;
public class Main {

    public static void main(String[] args) {

        // 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);

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

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

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

Writing the Persistence Unit

Il main ha bisogno di una persistence unit chiamata chapter04PU. Questa va definita in src/main/resources/META-INF nel file persistence.xml. Questo file, richiesto nella specifica JPA, è importante perché collega il provider JPA al database.

Contiene tutte le informazioni necessarie per connettersi al database (URL, driver JDBC, utente e password) e informa il provider della modalità di generazione dello schema del database (drop-and-create significa che le tabelle verranno eliminate e quindi create). L'elemento <provider> definisce il provider di persistenza. Le unità di persistenza elencano tutte le entità che devono essere gestite dal gestore di entità. Qui, il tag <class> si riferisce all'entità Book.

// qua sul libro ci sta un'altra persistence unit
(insert.sql ci fa fare l'inizializzazione del db. Per esempio)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
    version="2.1">

    <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-and-scripts" />
            <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"/>
            <property name="javax.persistence.sql-load-script-source" value="insert.sql" />
        </properties>
    </persistence-unit>
</persistence>
```

```
INSERT INTO BOOK(ID, TITLE, DESCRIPTION, ILLUSTRATIONS, ISBN, NBOFPAGE, PRICE) values ↵
(1000, 'Beginning Java EE 6', 'Best Java EE book ever', 1, '1234-5678', 450, 49)
INSERT INTO BOOK(ID, TITLE, DESCRIPTION, ILLUSTRATIONS, ISBN, NBOFPAGE, PRICE) values ↵
(1001, 'Beginning Java EE 7', 'No, this is the best ', 1, '5678-9012', 550, 53)
INSERT INTO BOOK(ID, TITLE, DESCRIPTION, ILLUSTRATIONS, ISBN, NBOFPAGE, PRICE) values ↵
(1010, 'The Lord of the Rings', 'One ring to rule them all', 0, '9012-3456', 222, 23)
```

Object-Relational Mapping

Nel capitolo precedente abbiamo visto le basi del mapping object-relational (ORM), che fondamentalmente è il mapping di entità su tabelle e attributi su colonne. Abbiamo la configurazione per eccezione che consente al provider JPA per mappare un'entità su una tabella di database utilizzando tutti i valori di default. Ma le impostazioni predefinite non sono sempre adatte, soprattutto se si esegue il mapping del tuo modello di dominio su un database esistente.

JPA viene fornito con un ricco set di metadati in modo da poter personalizzare il mapping.

✚ Elementary Mapping

Ci sono sostanziali differenze tra il modo in cui Java tratta i dati e il modo in cui questi vengono trattati dai database relazionali. In Java usiamo le classi per descrivere sia gli attributi per conservare i dati che i metodi per l'accesso e la manipolazione di quei dati. Una volta definita una classe, possiamo creare tutte le istanze di cui abbiamo bisogno con la parola new. In un Database relazionale, i dati sono archiviati in una struttura non a oggetti (righe e colonne) e i comportamenti sono **table triggers e stored procedures**, che non sono strettamente vincolati alle strutture di dati, così come lo sono con gli oggetti.

A volte il mapping di oggetti Java nel database può essere semplice e dunque si applicano le regole di default. A volte invece bisogna soddisfare le proprie esigenze e si vuole personalizzare il mapping.

Le annotazioni dell'Elementary Mapping si concentrano sul personalizzare le tabelle, le chiavi primarie, le colonne e consente di modificare determinate convenzione di naming o di tipi (not-null column, lenght, ecc).

@Table

Nelle regole della configuration-by-exception il nome dell'entità è lo stesso della tabella. Se vogliamo cambiarlo (ad esempio perché si desidera mappare i dati su una tabella diversa o associare una singola entità a più tabelle) si può usare l'annotazione **@Table**. Si può anche definire un vincolo di unicità con l'annotazione **@UniqueConstraint** insieme all'annotazione **@Table**.

Ad esempio, se vogliamo cambiare il nome in T_BOOK invece di BOOK:

Listing 5-1. The Book Entity Being Mapped to a T_BOOK Table

```
@Entity  
@Table(name = "t_book")  
public class Book {  
  
    @Id  
    private Long id;  
    private String title;  
    private Float price;  
    private String description;  
    private String isbn;  
    private Integer nbOfPage;  
    private Boolean illustrations;  
  
    // Constructors, getters, setters  
}
```

@SecondaryTable

Fino ad ora, abbiamo assunto che un'entità venga mappata su una singola tabella, nota anche come tabella primaria. Ma a volte quando si dispone di un modello dati esistente, è necessario distribuire i dati su più tabelle o tabelle secondarie. Per fare questo, è necessario utilizzare l'annotazione **@SecondaryTable** per associare una tabella secondaria a un'entità o **@SecondaryTables** (con una "s") per diversi tabelle secondarie.

Listing 5-2. Attributes of the Address Entity Mapped in Three Different Tables

```
@Entity  
@SecondaryTables({  
    @SecondaryTable(name = "city"),  
    @SecondaryTable(name = "country")  
})  
public class Address {  
  
    @Id  
    private Long id;  
    private String street1;  
    private String street2;  
    @Column(table = "city")  
    private String city;  
    @Column(table = "city")  
    private String state;  
    @Column(table = "city")  
    private String zipcode;  
  
    @Column(table = "country")  
    private String country;  
  
    // Constructors, getters, setters  
}
```

Le annotazioni definiscono due tabelle secondarie. Di default gli attributi sono mappati nella tabella primaria ma quelli con l'annotazione **@Column** vanno nella tabella corrispondente

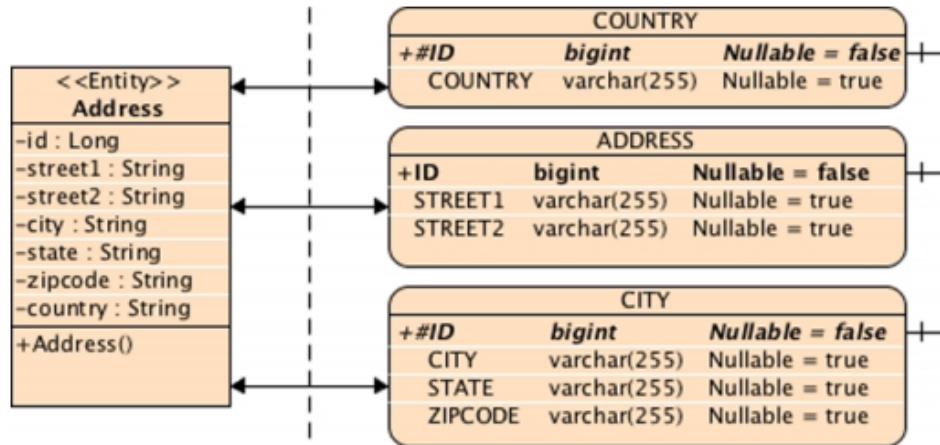


Figure 5-1. The `Address` entity is mapped to three tables

As you probably understand by now, you can have several annotations in the same entity. If you want to rename the primary table, you can add the `@Table` annotation as demonstrated in Listing 5-2.

Listing 5-2. The Primary Table Is Renamed to `T_ADDRESS`

```

@Entity
@Table(name = "t_address")
@SecondaryTables({
    @SecondaryTable(name = "t_city"),
    @SecondaryTable(name = "t_country")
})
public class Address {

    // Attributes, constructor, getters, setters
}

```

Primary Keys

Nei database relazionali, una chiave primaria identifica in modo univoco ogni riga di una tabella. Comprende una singola colonna o set di colonne. Le chiavi primarie devono essere univoche, in quanto identificano una singola riga (non è consentito un valore nullo). JPA richiede che le entità abbiano un identificatore mappato a una chiave primaria, che seguirà la stessa regola: identificare univocamente un'entità con un singolo attributo o un set di attributi (chiave composta). Il valore della chiave primaria di questa entità non può essere aggiornato una volta assegnato.

`@Id` and `@GeneratedValue`

Una chiave primaria semplice (vale a dire, non composta) deve corrispondere a un singolo attributo della classe entità. L'annotazione `@Id` è usato per denotare una semplice chiave primaria. `@ javax.persistence.Id` annota un attributo come un identificatore univoco. Può essere uno dei seguenti tipi:

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

Quando si crea un'entità, il valore di questo identificatore può essere generato manualmente dall'applicazione o automaticamente dal persistence provider utilizzando l'annotazione `@GeneratedValue`.

Questa annotazione può avere quattro valori possibili:

- SEQUENCE and IDENTITY specificano rispettivamente l'uso di un' SQL sequence o l'identità di una colonna.
- TABLE dice al persistence provider di conservare il sequence name e il suo valore in una tabella e di incrementarlo ogni volta che viene istanziata una nuova entità.
- AUTO : la generazione di una chiave viene eseguita automaticamente dal persistence provider, che sceglierà la strategia appropriata per il database. AUTO è il valore di default dell'annotazione `@GeneratedValue`

Se l'annotazione `@GeneratedValue` non è definita, l'applicazione deve creare il proprio identificatore applicando qualsiasi algoritmo che restituirà un valore univoco. Il codice nel Listato 5-3 mostra come avere una generazione automatica identificatore. GenerationType.AUTO essendo il valore predefinito, avrei potuto omettere l'elemento strategia. Si noti che l'attributo ID è annotato due volte, una volta con `@Id` e una volta con `@GeneratedValue`.

```
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String title;
    private Float price;
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;

    // Constructors, getters, setters
}
```

Relational Mapping

Il mondo della programmazione orientata agli oggetti abbonda di classi e associazioni tra le classi. Queste associazioni sono strutturali in quanto collegano oggetti di un genere a oggetti di un altro, permettendo ad un oggetto di far invocare ad un altro oggetto azioni su di sé.

Possono esistere diversi tipi di associazioni tra le classi.

Prima di tutto, un'associazione ha una direzione. Può essere unidirezionale (ossia un oggetto può navigare verso un altro) o bidirezionale (cioè un oggetto può navigare verso l'altro e viceversa). In Java, si utilizza la sintassi dot (.) per navigare tra gli oggetti. Ad esempio, quando si scrive `customer.getAddress().getCountry()`, si passa dall'oggetto Cliente a Indirizzo e quindi a Paese.

Nel linguaggio di modellazione unificata (UML), per rappresentare un'associazione unidirezionale tra due classi, si utilizza una freccia per indicare l'orientamento.



Figure 5-5. A unidirectional association between two classes

To indicate a bidirectional association, no arrows are used. As demonstrated in Figure 5-6, Class1 can navigate to Class2 and vice versa. In Java, this is represented as `Class1` having an attribute of type `Class2` and `Class2` having an attribute of type `Class1`.



Figure 5-6. A bidirectional association between two classes

An association also has a *multiplicity* (or *cardinality*). Each end of an association can specify how many referring objects are involved in the association. The UML diagram in Figure 5-7 shows that one `Class1` refers to zero or more instances of `Class2`.



Una relazione ha una ownership (cioè il proprietario della relazione). In una relazione unidirezionale, la proprietà è implicita: nella Figura 5-5, non c'è dubbio che il proprietario sia Classe 1, ma in una relazione bidirezionale come in Figura 5-6, il proprietario deve essere specificato esplicitamente. Quindi va indicato il proprietario, che specifica il mapping fisico, e il lato inverso (il lato non-owning).

Relationships in Relational Databases

Un database relazionale è una raccolta di relazioni (tabelle). Per modellare un'associazione, non si dispone di elenchi, set o mappe (come nel object orientation). 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 (una colonna di join) 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.

Ad esempio, si consideri che un cliente ha un indirizzo. In Java, si avrebbe una classe di clienti con un attributo Indirizzo. In un mondo relazionale, si potrebbe avere una tabella CLIENTI che indirizza ad un indirizzo tramite una colonna chiave esterna (o unione colonna), come descritto nella figura.

Customer				Address			
Primary key	Firstname	Lastname	Foreign key	Primary key	Street	City	Country
1	James	Rorisson	11				
2	Dominic	Johnson	12				
3	Maca	Macaron	13	11	Aligre	Paris	France
				12	Balham	London	UK
				13	Alfama	Lisbon	Portugal

Esiste anche un secondo modo per fare la modellazione di una relazione one-to-one: usare una join table.

La tabella CUSTOMER in Figura 5-9 non memorizza la chiave esterna dell'indirizzo ADDRESS. Viene creata una tabella intermedia per conservare le informazioni sulla relazione memorizzando le chiavi esterne di entrambe le tavole.

Customer			Address			
Primary key	Firstname	Lastname	Primary key	Street	City	Country
1	James	Rorisson	11	Aligre	Paris	France
2	Dominic	Johnson	12	Balham	London	UK
3	Maca	Macaron	13	Alfama	Lisbon	Portugal

Join table	
Customer PK	Address PK
1	11
2	12
3	13

Figure 5-9. A relationship using a join table

Questo metodo viene utilizzato per lo più quando si hanno relazioni molti a uno o molti a molti.

Entity relationship

Torniamo a JPA.

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. Come per le annotazioni Elementary Mapping, anche per le associazioni JPA usa la configuration by exception.

Per specificare le cardinalità si possono usare le annotazioni `@OneToOne`, `@OneToMany`, `@ManyToOne` o `@ManyToMany`. Ogni annotazione può essere utilizzata in modo unidirezionale o bidirezionale.

Table 5-1. All Possible Cardinality-Direction Combinations

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

Unidirectional and Bidirectional

Dal punto di vista della modellazione di oggetti, la direzione tra le classi è naturale. In un'associazione unidirezionale, l'oggetto A punta solo sull'oggetto B; in un'associazione bidirezionale, entrambi gli oggetti si riferiscono l'un l'altro.

Tuttavia, alcuni accorgimenti sono necessari quando si esegue il mapping di una relazione bidirezionale con un database relazionale, come illustrato dall'esempio seguente.

In una relazione unidirezionale, un'entità Customer ha un attributo di tipo Adress (vedi Figura 5-10). Il rapporto

è a senso unico, navigando da una parte all'altra. Customer è il proprietario della relazione. In termini di

database, questo significa che la tabella CUSTOMER avrà una chiave esterna (colonna join) che punta a ADDRESS e, quando si possiede una relazione, sei in grado di personalizzare il mapping di questa relazione. Ad esempio, se è necessario modificare il nome della chiave esterna, la mappatura verrà eseguita nell'entità Custom(ad esempio, owner).

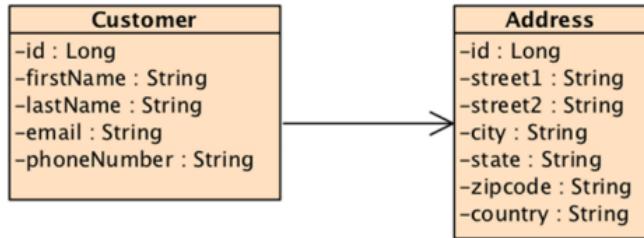


Figure 5-10. A unidirectional association between Customer and Address

Le relazioni possono essere anche bidirezionali. In termini di codice java è simile ad avere due relazioni one to one sparate, una in ogni direzione.

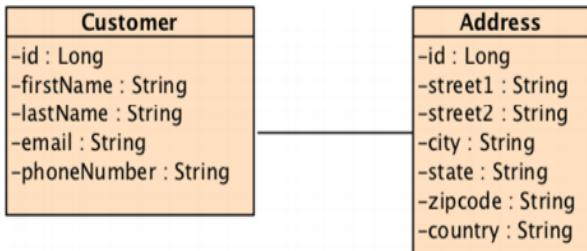


Figure 5-11. A bidirectional association between Customer and Address

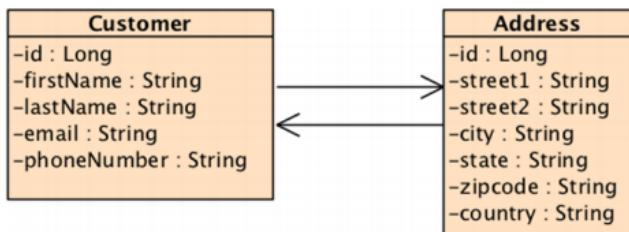
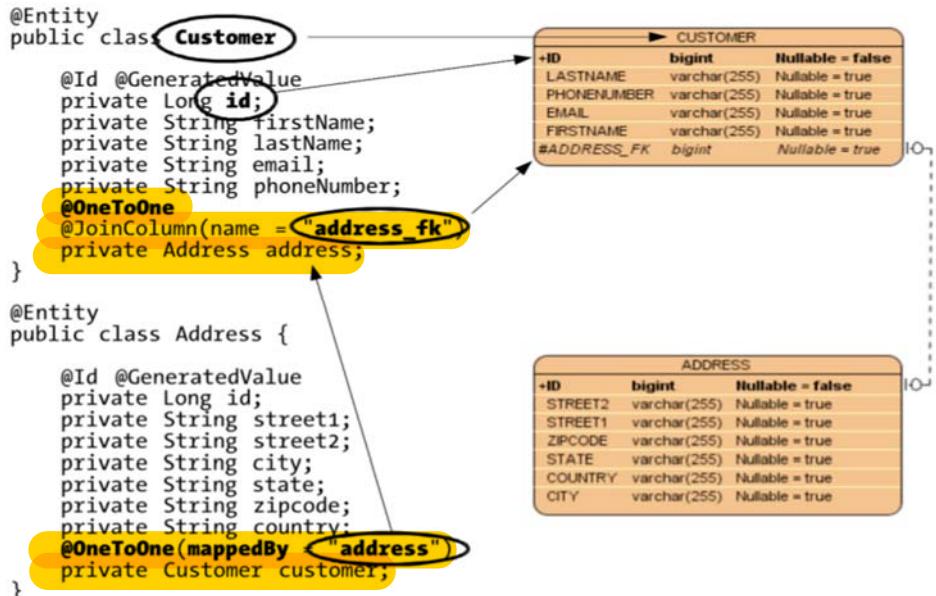


Figure 5-12. A bidirectional association represented with two arrows

Come si mappa una coppia di relazioni unidirezionali? Chi è il proprietario di questa relazione bidirezionale? Chi possiede le informazioni di mapping della colonna di join o della tabella di join? Se le relazioni unidirezionali hanno un owning side le bidirezionali hanno un owning e un inverse side, che devono essere specificati esplicitamente con l'elemento mappedBy delle annotazioni `@OneToOne`, `@OneToMany` e `@ManyToMany`. MappedBy identifica l'attributo che possiede la relazione ed è richiesto per le relazioni bidirezionali. A titolo di spiegazione, confrontiamo il codice Java (da un lato) e la mappatura del database (dall'altro). Come potete vedere sul lato sinistro della figura entrambe le entità puntano l'un l'altro attraverso attributi: il Customer dispone di un attributo Address annotato con `@OneToOne` e l'entità Address dispone di un attributo Customer con un'annotazione. Sul lato destro, il mapping del database mostra una tabella CUSTOMER e una ADDRESS. Il CUSTOMER è il proprietario della relazione perché contiene la chiave esterna ADDRESS.



L'entità Address utilizza l'elemento mappedBy nella sua annotazione @OneToOne.

L'indirizzo è chiamato inverse owner della relazione perché ha un elemento mappato.

L'elemento mappedBy indica che la colonna di join (address) è specificata all'altra estremità della relazione. Infatti, all'altra estremità, l'entità del cliente definisce la colonna di join utilizzando l'annotazione @JoinColumn e rinomina la chiave esterna a address_fk.

Il customer è quello che definisce il mapping della colonna di join.

Address è l'inverse side dove la tabella contiene la chiave esterna.

Esiste un elemento mappato sulle annotazioni @OneToOne, @OneToMany e @ManyToMany, ma non sull'annotazione @ManyToOne. Non è possibile avere un attributo mappato su entrambi i lati di un'associazione bidirezionale. Sarebbe anche sbagliato non averlo da entrambe le parti, il provider lo considererebbe come due relazioni unidirezionali indipendenti. Ciò implica che ciascun lato è il proprietario e può definire una colonna di join.

@OneToOne Unidirectional

Una relazione unidirezionale one-to-one tra entità ha un riferimento di cardinalità 1, che può essere raggiunto solo in una direzione. Facendo riferimento all'esempio di un Customer e il suo Adress, supponiamo che il cliente abbia un solo Adress, (cardinalità 1). È importante navigare dal cliente (la fonte) verso l'indirizzo (l'obiettivo) per sapere dove il cliente vive. Tuttavia, per qualche ragione, nel nostro modello mostrato nella Figura 5-14, non è necessario essere in grado di navigare nella direzione opposta (ad esempio, non è necessario sapere quale cliente abita a un dato indirizzo).

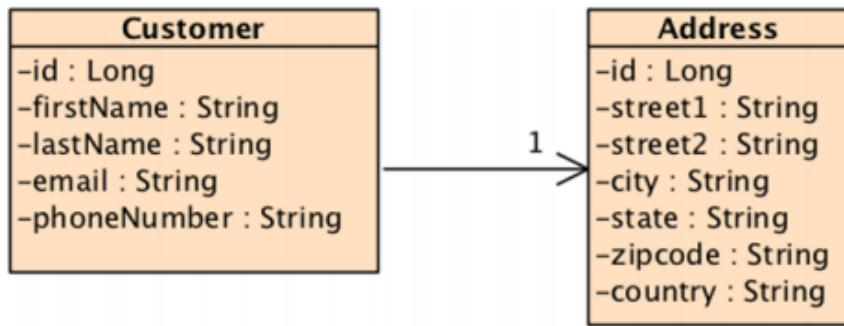


Figure 5-14. One customer has one address

In Java, una relazione unidirezionale significa che il Customer avrà un attributo Adress (Listato 5-34) ma L'Adress non avrà un attributo Customer (Listato 5-35).

Listing 5-34. A Customer with One Address

```

@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
}

```

Listing 5-35. An Address Entity

```

@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
}

```

//TODO ... ma in pratica metto semplicemente l'attributo Adress al Customer e fa tutto java. (Di default il nome della colonna foreign key è il nome dell'attributo concatenato al nome della chiave primaria della tabella target). Per customizzare il mapping possiamo usare le annotazioni `@OneToOne` e `@JoinColumn`

Listing 5-39. The Customer Entity with Customized Relationship Mapping

```
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @OneToOne (fetch = FetchType.LAZY)
    @JoinColumn(name = "add_fk", nullable = false)
    private Address address;

    // Constructors, getters, setters
}
```

@OneToMany Unidirectional

Una relazione uno a molti prevede che un oggetto abbia i riferimenti ad un insieme di oggetti target.



La navigabilità è da Order a OrderLine. In java questa molteplicità viene descritta usando Collections, List e Set.

```
@Entity
public class Order {

    @Id @GeneratedValue
    private Long id;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
    private List<OrderLine> orderLines;

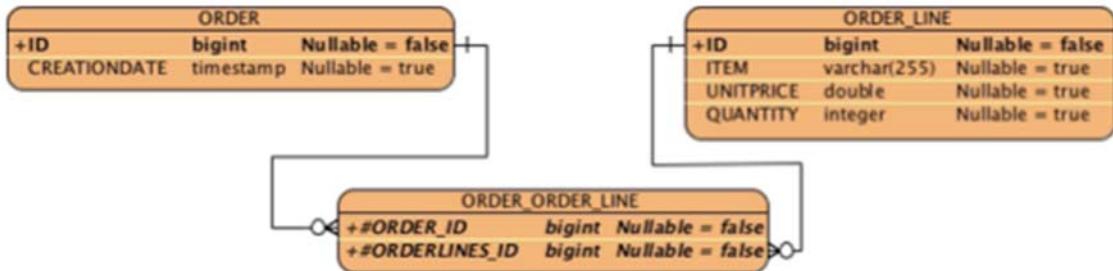
    // Constructors, getters, setters
}

@Entity
@Table(name = "order_line")
public class OrderLine {

    @Id @GeneratedValue
    private Long id;
    private String item;
    private Double unitPrice;
    private Integer quantity;

    // Constructors, getters, setters
}
```

Di default per le relazioni unidirezionali uno a molti viene usata una join table con due colonne una che si riferisce alla chiave primaria di Order e una che si riferisce alla chiave primaria di order line. Il nome della join table viene dall'unione dei nomi delle due tabelle



Se non ti piace la tabella join e i nomi delle chiavi esterne, o se stai mappando su una tabella esistente, puoi usare JPA per ridefinire i valori di default. Si può usare `@JoinTable`.

```

@Entity
public class Order {

    @Id @GeneratedValue
    private Long id;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
    @OneToMany
    @JoinTable(name = "jnd_ord_line",
               joinColumns = @JoinColumn(name = "order_fk"),
               inverseJoinColumns = @JoinColumn(name = "order_line_fk") )
    private List<OrderLine> orderLines;

    // Constructors, getters, setters
}

```

Se non vogliamo usare una join table possiamo usare l'annotazione `@JoinColumn`

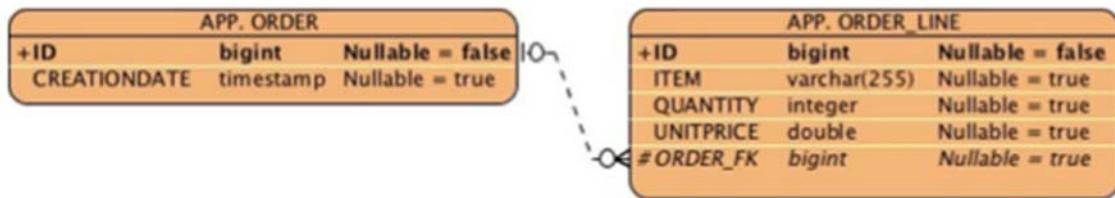
```

@Entity
public class Order {

    @Id @GeneratedValue
    private Long id;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
    @OneToMany(fetch = FetchType.EAGER)
    @JoinColumn(name = "order_fk")
    private List<OrderLine> orderLines;

    // Constructors, getters, setters
}

```



@ManyToMany Bidirectional

Esiste una relazione bidirezionale molti-a-molti quando un oggetto sorgente fa riferimento a molti target e quando un target si riferisce a molte fonti.

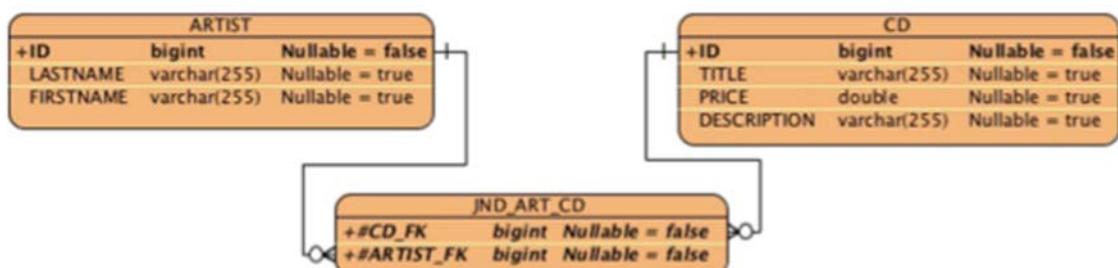
Ad esempio, un album CD viene creato da diversi artisti e un artista appare su diversi album.

In java ogni entità ha una collezione di target entity, Nei database relazionali l'unico modo per mappare una relazione many-to-many è di usare una join table. Inoltre bisogna esplicitamente definire il proprietario della relazione usando l'elemento mappedBy.

```
@Entity  
public class Artist {  
  
    @Id @GeneratedValue  
    private Long id;  
    private String firstName;  
    private String lastName;  
    @ManyToMany  
    @JoinTable(name = "jnd_art_cd", ~  
              joinColumns = @JoinColumn(name = "artist_fk"), ~  
              inverseJoinColumns = @JoinColumn(name = "cd_fk"))  
    private List<CD> appearsOnCDs;  
  
    // Constructors, getters, setters  
}
```

Nell'esempio assumiamo che Artist sia il proprietario della relazione.

```
@Entity  
public class CD {  
  
    @Id @GeneratedValue  
    private Long id;  
    private String title;  
    private Float price;  
    private String description;  
    @ManyToMany(mappedBy = "appearsOnCDs")  
    private List<Artist> createdByArtists;  
  
    // Constructors, getters, setters  
}
```



Si noti che nelle relazioni molti a molti e uno ad uno bidirezionali, entrambi i lati possono essere proprietari. Non importa quale sia il lato designato come proprietario, l'altro lato dovrebbe includere l'elemento mappedBy. In caso contrario, il provider pensa che entrambi i lati siano i proprietari e li tratterà come due relazioni uno a molti unidirezionali. Ciò potrebbe portare a quattro tabelle: ARTIST e CD, più due tabelle di join, ARTIST_CD e CD_ARTIST.

MAPPING PERSISTENT OBJECTS

JPA ha due parti. La prima è la possibilità di mappare oggetti a un database relazionale. La configuration by exception consente ai persistence provider di fare la maggior parte del lavoro senza molto codice, ma la ricchezza dell'API consente anche il mapping personalizzato dagli oggetti alle tabelle utilizzando annotazioni o XML descriptor.

L'altro aspetto dell'API è la capacità di interrogare questi oggetti. Il servizio centralizzato che consente di manipolare istanze di entità è L'Entity Manager. Fornisce un'API per creare, individuare, rimuovere e sincronizzare gli oggetti con il database. Consente inoltre l'esecuzione di query JPQL, query statiche, dinamiche ecc.

Il mondo del database si basa su SQL. Questo linguaggio di programmazione è progettato per la gestione dei dati relazionali (recupero, inserimento, aggiornamento e cancellazione), e la sua sintassi è orientata alle tabelle. Nel mondo Java, dove manipoliamo oggetti, un linguaggio fatto per le tabelle (SQL) deve essere adattarsi a un linguaggio che tratta oggetti (Java). JPQL è il linguaggio definito in JPA per interrogare le entità memorizzate in un database relazionale. 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

Il gestore di entità è un pezzo centrale di JPA. Gestisce lo stato e il ciclo di vita delle entità nonché le query delle entità all'interno di un persistence context. L'entity manager è responsabile della creazione e della rimozione di istanze di entità persistenti e dell'individuazione delle entità tramite la chiave primaria. Può lockare le entità per proteggerle dall'accesso concorrente utilizzando lock ottimistico o pessimistico e può utilizzare le query JPQL per recuperare entità secondo determinati criteri. 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. La forza di JPA è che le entità possono essere utilizzate come oggetti normali da diversi livelli di un'applicazione e diventare gestiti dall'Entity Manager quando è necessario. Quando un'entità viene gestita l'Entity Manager sincronizza automaticamente lo stato dell'entità con il database. Quando l'entità è detached (cioè non gestita) ritorna ad essere un semplice POJO e può quindi essere utilizzato da altri livelli (ad esempio, un layer di presentazione di JavaServer Faces o JSF) senza sincronizzare lo stato con il database.

Per quanto riguarda la persistenza, il vero lavoro inizia con l'Entity Manager. EntityManager è un'interfaccia implementata da un provider di persistenza che genererà ed eseguirà istruzioni SQL. javax.persistence.EntityManager interface fornisce l'API per manipolare le entità (sottoinsieme mostrato nel Listato 6-1).

Obtaining an Entity Manager

Per usare un entity manager bisogna ottenerlo. In un container-managed environment le transazioni sono gestite dal container, mentre in un application managed environment l'applicazione è responsabile di ottenere esplicitamente un'istanza di entity manager e di gestirne il ciclo di vita.

Listing 6-2. A Main Class Creating an EntityManager with an EntityManagerFactory

```
public class Main {  
  
    public static void main(String[] args) {  
  
        // 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);  
  
        // 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();  
  
        // Closes the entity manager and the factory  
        em.close();  
        emf.close();  
    }  
}
```

In JavaEE il modo più comune di ottenere un entity manager è con l'annotazione `@PersistenceContext`

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

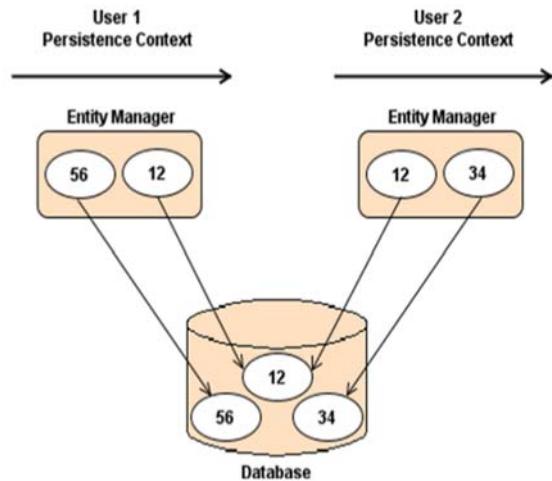
```
@Stateless  
public class BookEJB {  
  
    @PersistenceContext(unitName = "chapter06PU")  
    private EntityManager em;  
  
    public void createBook() {  
  
        // Creates an instance of book  
        Book book = new Book("H2G2", "The Hitchhiker's Guide to the Galaxy", 12.5F, ←  
                           "1-84023-742-2", 354, false);  
  
        // Persists the book to the database  
        em.persist(book);  
    }  
}
```

Persistence Context

Un persistence context è un insieme di entità gestite in un certo istante di tempo per una certa transazione: solo le istanze di entità nello stesso **persistence identity** esistono nello stesso persistence context. 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 mette le entità prima di fare il flush nel database.

Esempio:

Ogni utente ha un proprio persistence context che dura per la durata della sua transazione. L'utente 1 ottiene le entità del libro con ID pari a 12 e 56 dal database, per cui entrambi vengono memorizzati nel suo contesto di persistenza. L'utente 2 ottiene le entità 12 e 34. Come si può vedere, l'entità con ID = 12 viene memorizzata nel contesto di persistenza di ciascun utente. Mentre la transazione viene eseguita, il contesto di persistenza funziona come una cache di primo livello che memorizza le entità gestibili dall'EntityManager. Una volta terminata la transazione, il contesto di persistenza termina e le entità vengono cancellate.



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.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
    version="2.1">

    <persistence-unit name="chapter06PU" transaction-type="RESOURCE_LOCAL">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <class>org.agoncal.book.javaee7.chapter06.Book</class>
        <class>org.agoncal.book.javaee7.chapter06.Customer</class>
        <class>org.agoncal.book.javaee7.chapter06.Address</class>
        <properties>
            <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
            <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.EmbeddedDriver"/>
            <property name="javax.persistence.jdbc.url" value="jdbc:derby:memory:chapter06DB;create=true"/>
            <property name="eclipselink.logging.level" value="INFO"/>
        </properties>
    </persistence-unit>
</persistence>
```

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

Manipulating Entities

Usiamo l'entity manager sia per manipolazioni semplici che per complesse query JPQL. Quando manipoliamo singole entità l'interfaccia dell'entity manager ci consente di effettuare le CRUD operation

Table 6-2. EntityManager Interface Methods to Manipulate Entities

Method	Description
<code>void persist(Object entity)</code>	Makes an instance managed and persistent
<code><T> T find(Class<T> entityClass, Object primaryKey)</code>	Searches for an entity of the specified class and primary key
<code><T> T getReference(Class<T> entityClass, Object primaryKey)</code>	Gets an instance, whose state may be lazily fetched
<code>void remove(Object entity)</code>	Removes the entity instance from the persistence context and from the underlying database

(continued)

Table 6-2. (continued)

Method	Description
<code><T> T merge(T entity)</code>	Merges the state of the given entity into the current persistence context
<code>void refresh(Object entity)</code>	Refreshes the state of the instance from the database, overwriting changes made to the entity, if any
<code>void flush()</code>	Synchronizes the persistence context to the underlying database
<code>void clear()</code>	Clears the persistence context, causing all managed entities to become detached
<code>void detach(Object entity)</code>	Removes the given entity from the persistence context, causing a managed entity to become detached
<code>boolean contains(Object entity)</code>	Checks whether the instance is a managed entity instance belonging to the current persistence context

Listing 6-5. The Customer Entity with a One-Way, One-to-One Address

```
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    @OneToOne (fetch = FetchType.LAZY)
    @JoinColumn(name = "address_fk")
    private Address address;

    // Constructors, getters, setters
}
```

Listing 6-6. The Address Entity

```
@Entity
public class Address {

    @Id @GeneratedValue
    private Long id;
    private String street1;
    private String city;
    private String zipcode;
    private String country;

    // Constructors, getters, setters
}
```

La mappatura in database del codice di sopra è il seguente.

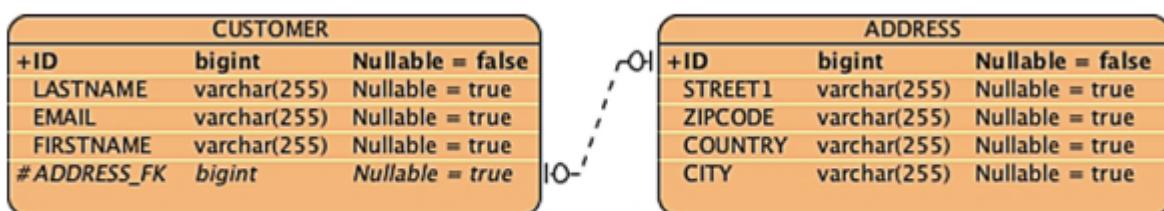


Figure 6-2. CUSTOMER and ADDRESS tables

Persisting an Entity

Rendere persistente un'entità significa inserire in un database che non esistono già (altrimenti viene lanciata un EntityExistsException). Per fare questo è necessario creare una nuova entità (parola new), settare i valori degli attributi, fare il bind delle entità (associare un entità ad un'altra) e chiamare il metodo persist().

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE", "UK");
customer.setAddress(address);

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

assertNotNull(customer.getId());
assertNotNull(address.getId());
```

I due oggetti diventano managed entities quando l'entity manager li rende persistenti. (em.persist()) Quando si fa il commit viene fatto il flush dei dati nel database e viene creata una nuova riga nella tabella CUSTOMER e una nuova riga nella tabella ADDRESS. L'espressione assertNotNull controlla che entrambe le entità abbiano ricevuto un identificatore generato.

NOTA: Se si ha bisogno di passare le entità per valore queste devono implemetare Serializable (questo perché potrei star usando RMI).

Finding by ID

Per trovare un'entità in base al suo identificativo, si possono utilizzare due metodi diversi.

- 1) Il primo è il metodo EntityManager.find (), che ha due parametri: la classe entità e l'identificatore univoco. Se l'entità viene trovata, viene restituito; se non viene trovato, viene restituito un valore nullo.

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

- 2) Il secondo è il metodo getReference (). Molto simile all'operazione di find. Fornisce un riferimento all'entità ma non recupera i suoi dati. Può essere pensato come ad un proxy di un'entità piuttosto che all'entità stessa. Questo può essere utile se non abbiamo bisogno di

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

accedere ai dati contenuti nell'entità. Se non viene trovata l'entità si lancia una EntityNotFoundException.

Removing an Entity

Un'entità può essere rimossa con il metodo EntityManager.remove (). Una volta rimossa un'entità è cancellata dal database è detached dall'entity manager e non può essere sincronizzata con il database. Come oggetto, però, è ancora accessibile finché non esce dallo scope e viene rimossa dal garbage collector.

Esempio di come rimuovere un'entità.

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE", "UK");
customer.setAddress(address);

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

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

// The data is removed from the database but the object is still accessible
assertNotNull(customer);
```

Nel database, la riga del cliente è collegata all'indirizzo attraverso una chiave esterna; più avanti nel codice, solo il customer viene cancellato

Quando il customer viene eliminato il fatto che l'address venga rimosso o no dipende dalla politica di cascade. L'address potrebbe rimanere una riga orfana.

Orphan Removal

Per la coerenza dei dati, gli orfani non sono desiderabili, poiché comportano l'inserimento di righe in un database a cui non viene fatto riferimento da qualsiasi altra tabella.

Con JPA è possibile informare il persistence provider di rimuovere automaticamente le entità orfane o fare il cascade su un'operazione di remove. Ad esempio se un'entità target (address) è di proprietà privata da un'entità source (customer) quando l'entità source viene rimossa il provider deve cancellare anche l'entità target. Per indicare ciò si usa un parametro dell'annotazione @OneToOne

```
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    @OneToOne (fetch = FetchType.LAZY, orphanRemoval=true)
    private Address address;

    // Constructors, getters, setters
}
```

Synchronizing with the Database

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

Fino ad ora, la sincronizzazione con il database è stata eseguita al momento del commit. L'entity manager è una cache di primo livello, ma che succede quando address e customer devono essere inseriti?

Tutte le modifiche in sospeso richiedono un'istruzione SQL; qui due istruzioni insert vengono prodotte e rese permanenti solo quando si fa commit. Per la maggior parte delle applicazioni, questa sincronizzazione automatica dei dati è sufficiente.

Anche se non si sa esattamente in che momento il provider effettua effettivamente le modifiche al database, si può essere sicuri che accada quando si fa il commit della transaction. Il database viene sincronizzato con le entità nel persistence context, ma si può fare esplicitamente il flush dei dati nel database o aggiornare le entità con i dati del database (refresh). Se si fa il flush dei dati nel database in un punto e se più tardi nell'applicazione chiama il metodo rollback(), i dati flushed verranno tolti dal database.

Flushing an Entity

Chiamando il metodo flush() si forza il provider a svuotare il persistence context. Viene generata un'insert e viene eseguita quando si chiama flush.

```
tx.begin();
em.persist(customer);
em.flush();
em.persist(address);
tx.commit();
```

Due cose interessanti accadono nel codice precedente. Il primo è che em.flush() non attenderà la commit della transaction e costringerà il provider a svuotare il context provider.

Il secondo è che questo codice non funzionerà a causa del vincolo di integrità referenziale. Senza un esplicito

flush, il gestore delle entità memorizza nella cache tutte le modifiche e poi li esegue in un ordine che sia coerente per il database. Con un flush esplicito, l'istruzione insert per CUSTOMER verrà eseguita, ma verrà violata l'integrità referenziale della foreing key Adress. (la colonna ADDRESS_FK in CUSTOMER). Ciò condurrà la transazione a rollback. Flussi esplicativi dovrebbero essere usati con attenzione e solo quando necessario

Refreshing an Entity

Il metodo refresh() è usato per la sincronizzazione dei dati nella direzione opposta al flush. Un uso tipico è per rimuovere i cambiamenti fatti ad un entità solo in memoria.

Nell'esempio, viene trovato trova un Cliente per ID, si cambia il suo nome e poi si annulla questa modifica usando il metodo refresh().

```

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

customer.setFirstName("William");

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

```

Content of the Persistence Context

Il Context Persistence contiene le entità gestite. Con l'interfaccia EntityManager, è possibile verificare se un'entità viene gestita, scollegata o se tutte le entità vengono cancellate dal persistence context.

Contains

Il metodo EntityManager.contains() restituisce un booleano che permette di controllare se una particolare entità è gestita dall'entity manager all'interno del contesto di persistenza corrente.

```
(em.contains(customer)).
```

Clear and Detach

Il metodo clear svuota il persistence context, staccando tutte le entità. Il metodo detach(Object entity) rimuove invece dal contesto di persistenza l'oggetto passato come parametro. Eventuali modifiche successive non verranno quindi sincronizzate col database.

```

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

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

assertTrue(em.contains(customer));

em.detach(customer);

assertFalse(em.contains(customer));

```

Merging di entità

Un' entità detached non è più associata al contesto di persistenza, e bisogna quindi “riattaccarla” per poterla gestire nuovamente, ad esempio per apportare nuove modifiche. Si usa il metodo merge() per poter effettuare il merging delle due entità, cioè sostituire l'entità con quella modificata.

Il merge è necessario se l'entità non è sincronizzata con quella del database e non è attaccata al contesto di persistenza.

Questo è quello che succede con customer.setFirstName ("William"); questo viene eseguito su un'entità distaccata e i dati non vengono aggiornati nel DataBase. Per replicare questa

modifica al database, è necessario ricollegare l'entità con em.merge (ccustomer) all'interno di una transazione.

```
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();
```

Updating an Entity

Se un'entità è managed qualsiasi cambiamento apportato all'entità si rifletterà nel database automaticamente. (In caso contrario, dovrà chiamare esplicitamente merge().)

Ad esempio nel codice seguente quando viene chiamato setFirstName() lo stato dell'entità cambia. L'entity manager fa il caching dei cambiamenti che avvengono a partire da tx.begin() e li sincronizza quando si fa il commit

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

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

customer.setFirstName("Williman");

tx.commit();
```

Eventi a cascata

Di default, le operazioni dell'entity manager si applicano solo all'entità argomento dell'operazione, ma talvolta la stessa operazione potrebbe influire anche su un'entità associata a quella cui l'operazione è applicata, ottenendo degli “eventi a cascata”. Di conseguenza è possibile modificare anche una entità che è stata collegata ad un'altra entità tramite una sola chiamata del metodo persist.

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE", "UK");
customer.setAddress(address);
tx.begin();
em.persist(customer);
tx.commit();
```

E non con due, sulle varie entità:

Listing 6-17. Persisting a Customer with an Address

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE", "UK");
customer.setAddress(address);

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

Le notazioni `@OneToOne`, `@OneToMany` e `@ManyToMany` hanno un attributo cascade che prende un array di eventi da gestire in cascata, come ad esempio un evento Persist può essere reso cascade così come evento Remove, come da esempio.

```
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    @OneToOne (fetch = FetchType.LAZY, cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
    @JoinColumn(name = "address_fk")
    private Address address;

    // Constructors, getters, setters
}
```

Si può scegliere tra differenti eventi da mettere in cascata, ed è persino possibile metterli tutti in cascata usando `CascadeType.ALL`

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

JPQL 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 tra 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 ad oggetti. Ciò lo rende più comprensibile a coloro le cui conoscenze sono limitate ai linguaggi ad oggetti, infatti non è visibile il modo in cui JPQL applica dei meccanismi per modificare le proprie query in un linguaggio comprensibile ad un database SQL.

La query JPQL più semplice seleziona tutte le istanze di una singola entità.

```
SELECT b  
FROM Book b
```

Invece di selezionare da una tabella, JPQL seleziona le entità, qui Book.

La query di selezione più semplice consiste di due parti obbligatorie: la SELECT e la clausola FROM. SELECT definisce il formato dei risultati della query. La clausola FROM definisce l'entità o le entità da cui saranno ottenuti i risultati e le clausole facoltative WHERE, ORDER BY, GROUP BY e HAVING possono essere utilizzate per limitare o ordinare il risultato di una query.

```
SELECT b  
FROM Book b  
WHERE b.title = 'H2G2'
```

Si può anche ritornare una lista di oggetti con i valori che risultano dalla query. La classe in questione non deve per forza essere un'entità.

```
SELECT NEW org.agoncal.javaee7.CustomerDTO(c.firstName, c.lastName, c.address.street1)  
FROM Customer c
```

Per scrivere delle query parametriche si possono usare parametri posizionali designati da un ‘?’ seguito da un numero o parametri identificati da un nome.

```
SELECT c  
FROM Customer c  
WHERE c.firstName = ?1 AND c.address.country = ?2
```

```
SELECT c  
FROM Customer c  
WHERE c.firstName = :fname AND c.address.country = :country
```

Per cancellare una lista di entità si può fare una select, iterare e cancellare le entità, ma a livello di performance è molto meglio usare DELETE. (È come SELECT e può avere la clausola WHERE).

```
DELETE FROM Customer c  
WHERE c.age < 18
```

Si può anche fare l'update di una serie di entità che soddisfano una determinata condizione settando uno o più parametri.

```
UPDATE Customer c  
SET c.firstName = 'TOO YOUNG'  
WHERE c.age < 18
```

Queries

Come si integra una dichiarazione JPQL nella propria applicazione? Attraverso le queries. JPA ha 5 diversi tipi di queries:

- ⑩ **Dynamic queries:** la forma più semplice, costituita da una stringa query JPQL specificata dinamicamente a runtime.
- ⑩ **Named queries:** Le named query sono statiche e non modificabili.
- ⑩ **Criteria API:** introducono il concetto di object-oriented query API.
- ⑩ **Native queries:** per eseguire una query in SQL anziché in JPQL.
- ⑩ **Stored procedure queries:** introduce una nuova API per chiamare stored procedure.

Il punto centrale per scegliere tra questi cinque tipi di query è l'interfaccia EntityManager, che ha diversi metodi di factory, elencati nella Tabella 6-4, restituendo un'interfaccia Query, TypedQuery o StoredProcedureQuery (sia TypedQuery che StoredProcedureQuery estendono la query). L'interfaccia Query viene utilizzata nei casi in cui il risultato type è Object e TypedQuery viene utilizzato quando si preferisce un risultato tipizzato. StoredProcedureQuery viene utilizzato per il controllo stored procedure di esecuzione della query.

Table 6-4. EntityManager Methods for Creating Queries

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

I metodi maggiormente utilizzati in questa API sono quelli che eseguono la query stessa. Per eseguire una query SELECT, devi scegliere tra due metodi a seconda del risultato richiesto:

- ⑩ Il metodo `getResultSet()` esegue la query e restituisce un elenco di risultati (entità, attributi, espressioni, ecc.).
- ⑩ Il metodo `getSingleResult()` esegue la query e restituisce un singolo risultato (genera una `NonUniqueResultException` se viene trovato più di un risultato).

Per eseguire un aggiornamento o un'eliminazione, il metodo `executeUpdate()` esegue la query e restituisce il numero di entità interessate dall'esecuzione della query. Una query può utilizzare i parametri che sono posizionali (ad esempio, 1) o che hanno un nome (ad esempio: `myParam`). L'API definisce diversi metodi `setParameter` per impostare i parametri..

Quando si esegue una query, è possibile restituire un gran numero di risultati. A seconda dell'applicazione, questi possono essere elaborati insieme o in blocchi (ad esempio, un'applicazione Web visualizza solo dieci righe contemporaneamente).

Per controllare la paginazione, l'interfaccia Query definisce metodi

- ⑩ `setFirstResult()`: per specificare il primo risultato da ricevere (numerato da zero)
- ⑩ `setMaxResults()`: per il numero massimo di risultati da restituire rispetto a quel punto.

La modalità flush indica al persistence provider come gestire le modifiche e le query in sospeso. Esistono due possibili impostazioni di modalità di scorrimento: AUTO e COMMIT.

- ⑩ AUTO (predefinita) indica che il persistence provider è responsabile di garantire che le modifiche in sospeso siano visibili all'elaborazione della query.
- ⑩ Con COMMIT l'effetto degli aggiornamenti alle entità non si sovrappone ai dati modificati nel persistence context.

Le query possono essere locked utilizzando il metodo `setLockMode(LockModeType)`. I lock sono destinate a fornire una struttura che consente l'effetto di ripetibile leggere se ottimisticamente o pessimisticamente.

Dynamic Queries

Le query dinamiche sono create al volo quando richiesto dall'applicazione.

Per creare una query dinamica si usa il metodo `EntityManager.createQuery()` che prende come parametro una stringa che indica la query.

Nel seguente codice, la query JPQL seleziona tutti i clienti dal database. Il risultato di questa query è una lista, quindi, quando invochi il metodo `getResultSet()`, restituisce un elenco di entità cliente (Elenco <Cliente>). Però, se sai che la tua query restituisce solo una singola entità, usa il metodo `getSingleResult()`. Restituisce una singola entità ed evita il lavoro di recupero dei dati come lista.

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

Questa query JPQL restituisce un oggetto Query. Quando invochi il metodo `query.getResultSet()`, restituisce un elenco di oggetti non tipizzati. Se si desidera che la stessa query restituisca un elenco di tipo Cliente, è necessario utilizzare `TypedQuery` come segue:

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

Per settare i parametri si usa il metodo setParameter(si può passare un nome o una posizione):

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

query = em.createQuery("SELECT c FROM Customer c where c.firstName = ?1");
query.setParameter(1, "Betty");
List<Customer> customers = query.getResultList();
```

Se abbiamo query statiche si può evitare l'overhaed dovuto ad una dynamic query usando una named query.

Per costruire la stringa che costituisce la query dinamica si può usare la concatenazione. Ad esempio:

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

Se voglio visualizzare al massimo 10 risultati si può usare il metodo setMaxResult():

```
query = em.createQuery("SELECT c FROM Customer c", Customer.class);
query.setMaxResults(10);
List<Customer> customers = query.getResultList();
```

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 (deployment time?) e non ogni volta che la query viene eseguita.

Le Named Query sono query statiche espresse in metadati all'interno di un XML o con @NamedQuery.

Per definire queste query riutilizzabili, annotare un'entità con l'annotazione @NamedQuery, che richiede due elementi: il nome della query e il suo contenuto. Quindi cambiamo l'entità cliente e definiamo staticamente tre query usando le annotazioni (vedi il Listato 6-22).

```
@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")
})
```

```

public class Customer {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private Integer age;
    private String email;
    @OneToOne
    @JoinColumn(name = "address_fk")
    private Address address;

    // Constructors, getters, setters
}

```

Poichè stiamo definendo più annotazioni usiamo l'annotazione `@NamedQueries` che prende un array di `@NamedQuery`.

Ogni `@NamedQuery` prende due parametri: il nome della query e la query stessa.

```

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

```

Poichè la maggior parte dei metodi della Query API ritornano un oggetto di tipo `query` o `TypedQuery` è possibile scrivere:

```

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

```

```
Query query = em.createNamedQuery("findAll");
```

```
TypedQuery query = em.createNamedQuery("findAll", Customer.class);
```

Le query con nome sono utili per l'organizzazione delle definizioni di query e potenti per migliorare le prestazioni dell'applicazione.

Le named query sono definite staticamente sulle entità e in genere vengono posizionate sulla classe di entità che corrisponde direttamente al risultato della query (qui la query `findAll` restituisce i clienti, quindi dovrebbe essere definito nell'entità `Cliente`).

C'è una restrizione nel nome di una query. Lo scope del nome è gestito dalla persistence unit e deve essere unico all'interno dello scope. Ciò significa che non può esistere un'altra query con il nome `finadAll`. Una pratica comune è quella di mettere come prefisso il nome dell'entità.

Un altro problema è che il nome della query, che è una stringa, viene manipolato e, se si effettua un typo o un refactor del codice, è possibile ottenere alcune eccezioni che indicano che la query non esiste. Per limitare i rischi, puoi sostituire il nome di una query con una costante.

```

@Entity
@NamedQuery(name = Customer.FIND_ALL, query="select c from Customer c"),
public class Customer {

    public static final String FIND_ALL = "Customer.findAll";

    // Attributes, constructors, getters, setters
}

```

La costante FIND_ALL identifica la query findAll in modo non ambiguo anteponendo il nome della query al file nome dell'entità. La stessa costante viene quindi utilizzata nell'annotazione @NamedQuery e puoi utilizzare questa costante su eseguire la query come segue:

```
TypedQuery<Customer> query = em.createNamedQuery(Customer.FIND_ALL, Customer.class);
```

Criteria API (or Object-Oriented Queries)

Criteria API supporta tutto quello che può fare JPQL ma con una sintassi object based. (L'idea è che questo dovrebbe aiutare ad evitare errori di battitura).

L'idea è che tutte le parole chiavi (INSERT, UPDATE, DELETE, WHERE, LIKE, GROUP BY...) sono definite in questa API.

In JPQL avremmo SELECT c FROM Customer c WHERE c.firstName = 'Vincent' mentre usando Criteria API abbiamo:

```

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();

```

Le keyword di JPQL hanno una rappresentazione attraverso i metodi select() where() from() ecosì via. Le Criteria query sono costruite attraverso un'interfaccia CriteriaBuilder ottenuta dall' EntityManager. Contiene metodi per la costruzione della definizione delle query (desc(), asc(), avg(), sum(), max(), min(), count(), and(), or(), greaterThan(), lowerThan(...)) e definisce anche metodi come select(), from(), where(), orderBy(), groupBy() e having(),

```

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("age").as(Integer.class), 40));
Query query = em.createQuery(criteriaQuery).getResultList();
List<Customer> customers = query.getResultList();

```

In questo esempio abbiamo bisogno di effettuare il casting di age a intero perché non c'è altro modo per determinare se è maggiore di 40.

Native Queries

JPQL ha una sintassi molto ricca che consente di gestire le entità in qualsiasi forma e garantisce la portabilità tra database. JPA consente di utilizzare le funzionalità specifiche di un database utilizzando query native. Le query native prendono l'istruzione SQL nativa (SELECT, UPDATE o DELETE) come parametro e restituiscono un'istanza Query per l'esecuzione di tale istruzione SQL. Tuttavia, le query native non è previsto che siano portabili tra i database.

Se il codice non è portabile, perché non utilizzare le chiamate JDBC? Il motivo 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 del client dal database utilizzando SQL, è necessario utilizzare il metodo EntityManager.createNativeQuery () che ha come parametri la query SQL e la classe di entità che il risultato deve essere mappato.

```
Query query = em.createNativeQuery("SELECT * FROM t_customer", Customer.class);
List<Customer> customers = query.getResultList();
```

Di nuovo la stringa può essere creata dinamicamente e poiché il persistence provider non la conosce prima dovrà essere interpretata ogni volta. Si possono creare NamedNativeQuery con l'annotazione corrispondente e come le named query devono avere un nome unico.

```
@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 dalla propria applicazione al database che la eseguirà e restituirà un risultato.

Le stored procedures sono diverse perché sono conservate nel database e sono eseguite sul database.

Spesso sono scritte in linguaggi proprietari e non sono portabili tra i vari database. Ma ci sono molti vantaggi:

- Migliori prestazioni grazie alla precompilazione della stored procedure e alla riutilizzazione del suo piano di esecuzione,
- mantenere statistiche sul codice per mantenerlo ottimizzato,
- Riducendo la quantità di dati passati su una rete mantenendo il codice sul server,
- Modificare il codice in una posizione centrale senza replicare in più programmi diversi,

- Procedure memorizzate, che possono essere utilizzate da più programmi scritti in diversi linguaggi (non solo Java),
- nascondere i dati grezzi consentendo solo alle stored procedure di accedere ai dati e
- Migliorare i controlli di sicurezza concedendo all'utente l'autorizzazione a eseguire una stored procedure indipendentemente dalle autorizzazioni di tabella sottostanti.

Diamo un'occhiata ad un esempio pratico: archiviando vecchi libri e CD. Dopo un certo data i libri e i CD devono essere archiviati in un determinato magazzino, . L'archiviazione di libri e CD può essere un processo che richiede tempo perché diverse tabelle devono essere aggiornate . Quindi possiamo scrivere una stored procedure per raggruppare diverse dichiarazioni SQL e migliorare le prestazioni. La procedura memorizzata sp_archive_books prende una data di archiviazione e un codice di magazzino come parametri e aggiorna le tabelle T_Inventory e T_Transport.

```
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
```

Questa procedura è compilata nel database e può essere invocata tramite il nome e prende dei parametri. L'API permette solo di invocare una stored procedure, non di crearne una. Può essere invocata dinamicamente o tramite l'annotazione.

```
@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)
})
public class Book {

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

    // Constructors, getters, setters
}
```

Se non usiamo le annotazioni (@NamedStoredProcedureQuery) possiamo usare il metodo `createStoredProcedure()`. Ciò significa che i parametri e le informazioni sui set di risultati devono essere forniti a livello di programmazione. Questo può essere fatto usando il metodo `registerStoredProcedureParameter` del Interfaccia StoredProcedureQuery come mostrato nel Listato 6-32.

```
StoredProcedureQuery query = em.createStoredProcedureQuery("sp_archive_old_books");
query.registerStoredProcedureParameter("archiveDate", Date.class, ParameterMode.IN);
query.registerStoredProcedureParameter("maxBookArchived", Integer.class, ParameterMode.IN);

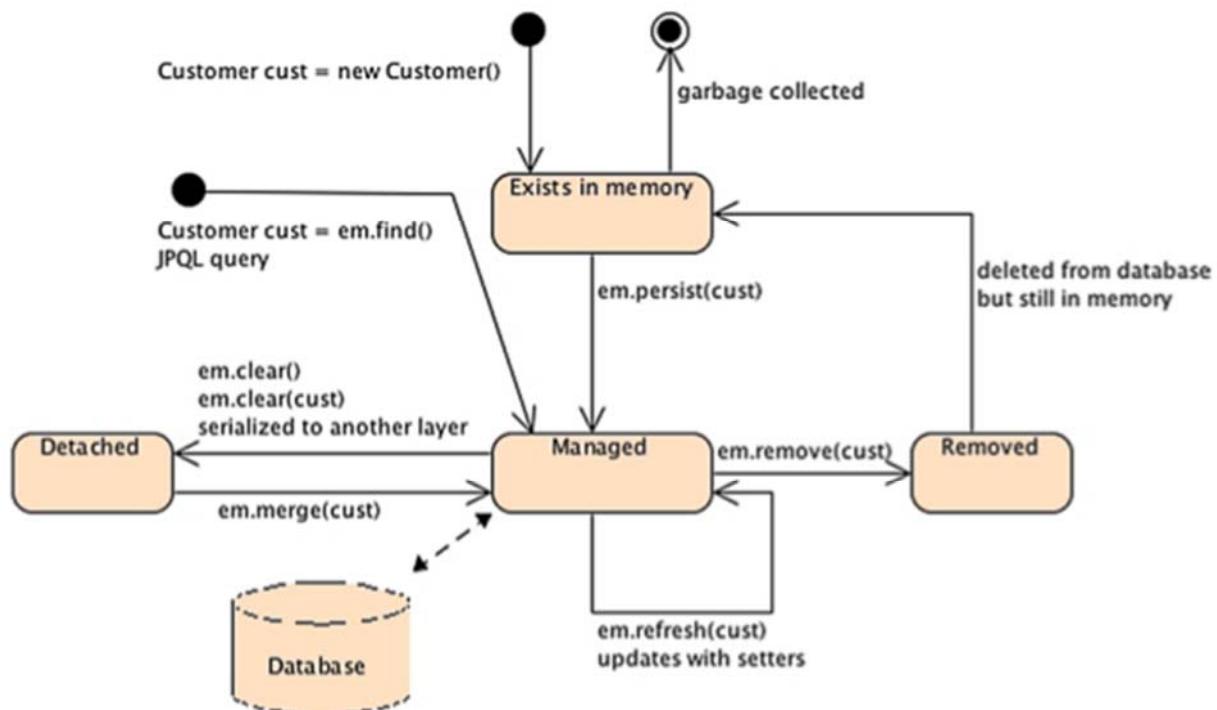
query.setParameter("archiveDate", new Date());
query.setParameter("maxBookArchived", 1000);
query.execute();

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

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.



Per creare un'istanza dell'entità di Customer , si usa l'operatore new. Questo oggetto esiste nella memoria, anche se l'API non ne sa nulla. Se non si fa niente con questo oggetto, andrà fuori dallo scope e finirà per essere rimosso dal garbage collector e sarà la fine del suo ciclo di vita. Quello che si può fare è rendere persistente un'istanza di Customer con il metodo EntityManager.persist(). In quel momento l'entità viene gestita e il suo stato viene sincronizzato con il database. Durante questo stato managed, è possibile aggiornare gli attributi utilizzando i metodi di setter 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 EntityManager.contains(customer), restituirà true. Un altro modo per gestire un'entità è quando viene caricata dal database. Quando si utilizza il metodo EntityManager.find() o si crea una query JPQL per recuperare un elenco di entità, tutte queste vengono gestite automaticamente e si può aggiornare o rimuovere i loro attributi.

Nello stato managed, è possibile chiamare il metodo EntityManager.remove () e l'entità viene eliminata dal database e non è più gestita. Ma l'oggetto Java continua a vivere in memoria, e si può utilizzarlo finché il garbage collector non lo elimina . Vediamo ora lo stato detached. Con i metodi EntityManager.clear () o EntityManager.detach (cliente) si eliminerà l'entità dal persistence context e diventerà detached . Ma c'è anche un altro modo di fare il detach un'entità: quando è serializzata. Se le entità necessitano di attraversare la rete per essere invocate in remoto o di attraversare layer per essere visualizzate nel presentation layer devono implementare Serializable. Quando un'entità managed viene serializzata, attraversa la rete e viene deserializzata , viene vista come un oggetto detached. Per fare il reattach di un'entità, è necessario chiamare il metodo EntityManager.merge(). Un caso di utilizzo comune è quando si utilizza un'entità in una pagina JSF. Una volta visualizzata, se esistono dati modificati che devono essere aggiornati, viene fatto il submit della form e l'entità viene inviata al server, deserializzata e deve essere fatto il merge per essere nuovamente attached. 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.

Callbacks

Il ciclo di vita di un'entità rientra in 4 categorie: persisting, updating, removing, e loading che corrispondono rispettivamente alle operazioni inserting, updating, deleting, e selecting. Per intercettare gli eventi del ciclo di vita i business methods possono essere annotati con le seguenti annotazioni:

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.

CHAPTER 6 ■ MANAGING PERSISTENT OBJECTS

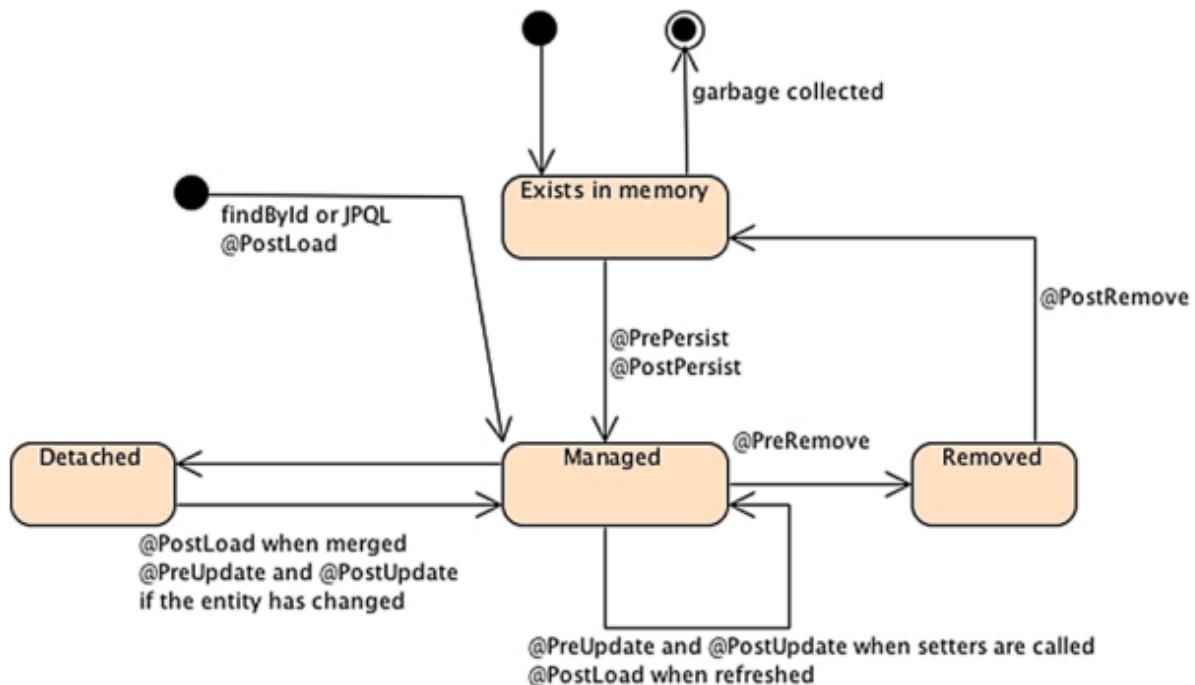


Figure 6-7. Entity life cycle with callback annotations

Prima di inserire un'entità nel database, l'Entity Manager chiama il metodo annotato con `@PrePersist`.

Se l'inserimento non genera un'eccezione, l'entità viene mantenuta, la sua identità viene inizializzata e con il metodo annotato `@PostPersist` viene quindi richiamato. Questo è lo stesso comportamento per gli aggiornamenti (`@PreUpdate`, `@PostUpdate`) e cancellazione (`@PreRemove`, `@PostRemove`). Un metodo annotato con `@PostLoad` viene chiamato quando un'entità viene caricata dal database

(tramite `EntityManager.find()` o una query JPQL). Quando l'entità è scollegata e deve essere unita, l'EM

deve prima verificare se ci sono differenze con il database (@PostLoad) e, in tal caso, aggiornare i dati (@PreUpdate, @PostUpdate).

Quindi le entità possono avere oltre che costruttori, getter e setter dei business methods per validare il proprio stato o calcolare il valore di alcuni attributi.

Esempio

Listing 6-38. The Customer Entity with Callback Annotations

```
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;

    @Transient
    private Integer age;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;

    @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;
    }

    // Constructors, getters, setters
}
```

Nel Listato 6-38, l'entità Cliente ha un metodo per convalidare i suoi dati (controlla gli attributi firstName e il lastName). Questo metodo è annotato con @PrePersist e @PreUpdate e verrà chiamato prima di inserire i dati nel database o di aggiornarli. Se i dati non sono validi, viene avviata un'eccezione di runtime e l'inserimento o l'aggiornamento

verrà ripristinato per garantire che i dati inseriti o aggiornati nel database siano validi. Il metodo calculateAge () calcola l'età del cliente. L'attributo età è transitorio e non viene mappato nel database. Dopo che l'entità viene caricata(@PostLoad), mantenuta(@PostPersist) o aggiornata(@PostUpdate), il metodo calculateAge () accetta il metodo data di nascita del cliente, calcola l'età e imposta l'attributo.

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, JMS, 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 Address.

Listeners

I metodi di callback in una entità funzionano quando la business logic è relativa solo a tale entità. I listeners sono usati per estrarre le business logic e condividerlo tra entità. Una entità listener è un POJO su cui puoi definire una o più metodi life-cycle callback. Per registrare una listener, l'entità ha bisogno di usare l'annotazione EntityListeners.

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);
    }
}
```

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");
    }
}
```

Le classi listeners devono:

1) avere un costruttore pubblico senza argomenti,

2) le firme dei metodi di callback devono avere accesso sullo stato dell'entità, e i metodi devono avere parametri di un tipo compatibile con quello dell'entità, dato che l'entità relativa all'evento è passato nel callback.

I metodi di callback definiti su un listener di entità possono avere due diversi tipi di firme:

1) Se il metodo deve essere usato su più entità, deve avere come parametro un Object.

```
void <METHOD>(Object anyEntity)
```

2) Se invece se deve essere usata su una entità o sottoclasse, il parametro può essere del tipo di tale entità.

```
void <METHOD>(Customer customerOrSubclasses)
```

L'annotazione @EntityListeners può notificare due o più entità listener di eventi life-cycle, infatti può prendere una sola entità listener come parametro o un array di essi. Quando avviene l'evento life-cycle, il provider di persistenza itera attraverso ogni listener nell'ordine in cui sono nell'array e ne invoca i metodi di callback, passandogli un riferimento all'entità su cui è avvenuto l'evento, evocando il callback method su quest'ultima se presente.

Listing 6-41. The Customer Entity Defining Two Listeners

```
@EntityListeners({DataValidationListener.class, AgeCalculationListener.class})
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
```

222

CHAPTER 6 ■ MANIFESTATION

```
@Transient
private Integer age;
@Temporal(TemporalType.TIMESTAMP)
private Date creationDate;

// Constructors, getters, setters
}
```

L'entità customer quindi validerà i propri dati prima di un inserimento o di un aggiornamento usando il metodo .validate() del primo listener, e quello .calculateage() del secondo. Le regole che i metodi una entità listener devono seguire sono simili a quelle dei metodi callback:

- Solo eccezioni non controllate possono essere lanciate.
- In una gerarchia da eredità, se più entità definiscono dei listener, quelle definite sulla superclasse vengono invocate prima di quelle definite sulla sottoclasse. Se una entità non vuole ereditare i listener della superclasse, può escluderlo con la notazione `@ExcludeSuperclassListeners`.

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");
    }
}
```

Quest'ultimo listener ha tutti i metodi che prendono un parametro di tipo `object`, quindi è possibile che sia usato da qualsiasi entità di una applicazione. Per evitare che si debba aggiungere all'annotazione di ciascuna entità manualmente, JPA offre la possibilità di aggiungere un listener di default, solo però tramite XML.

Listing 6-43. A Debug Listener Defined as the Default Listener

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" -->
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
        http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd"
    version="2.1">

    <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>

</entity-mappings>
```

`<persistence-unit-metadata>` definisce tutti i metadata le cui annotazioni non sono equivalenti, `<persistence-unit-defaults>` definisce tutti i defaults della persistence unit. Il `DebugListener` verrà quindi automaticamente invocato per ogni entità. Le entità listener di default verranno chiamate nell'ordine in cui sono presenti nell'XML mapping file, e prima di qualsiasi altro listener. Se una entità non vuole l'invocazione di un listener di default, può usare l'annotazione `@ExcludeDefaultListeners`.

Listing 6-44. The Customer Entity Excluding Default Listeners

```
@ExcludeDefaultListeners
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    @Transient
    private Integer age;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;

    // Constructors, getters, setters
}
```

Enterprise JavaBeans

Il layer di persistenza non è il layer appropriato per i processi di business, allo stesso modo l’interfaccia utente non dovrebbe effettuare business logic, specialmente quando ci sono più interfacce. Per separare il layer di persistenza dal layer di presentazione c’è bisogno del business layer che è implementato con gli EJB.

Il layering è importante per molte applicazioni. Le entità gestite con JPA in linguaggio naturale sono rappresentate dai nomi e i verbi modellano le azioni dell’applicazione gestite nel business layer. Il business layer può interagire con sistemi esterni, mandare messaggi asincroni, orchestrare le componenti del database verso sistemi esterni e servire il livello di presentazione.

Understanding JavaBeans

Gli EJB sono dei componenti server side che encapsulano business logic e si occupano delle transazioni e della sicurezza, hanno anche uno stack integrato per i messaggi , per fare scheduling...

Inoltre gli EJB sono integrabili con altri componenti integrati di JavaSE e JavaEE come JDBC, JPA, JMS, JNDI e RMI. Per questo sono usati nel business layer, sono al di sopra del database e orchestrano il business model layer. Agiscono come punto d’entrata per le tecnologie del presentation tier come JSF.

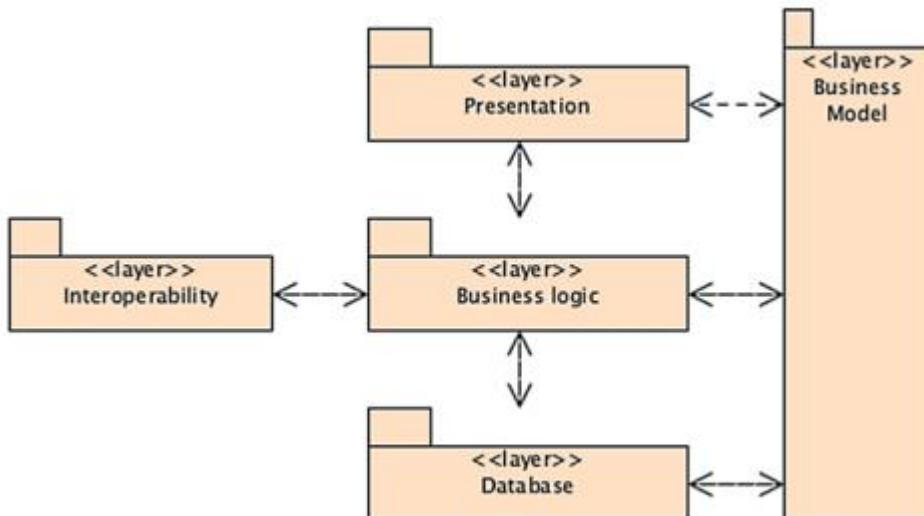


Figure 7-1. Architecture layering

Il modello di programmazione degli EJB combina facilità e robustezza. Oggi gli EJB sono dei semplici modelli di Java server-side development riducendo complessità e consentendo la riusabilità e scalabilità. 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. Gli sviluppatori di EJB possono quindi concentrarsi sull’implementazione della logica di business mentre il contenitore si occupa di tutto il resto.

Oggi più che mai, con la versione 3.2, gli EJB possono essere scritti una sola volta e distribuiti su qualsiasi contenitore che supporti la specifica

Types of EJBs

I session beans 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.

Un bean di sessione può avere un'interfaccia locale e/o remota, oppure non averne. I bean di sessione sono componenti gestiti dal container, quindi devono essere confezionati in un archivio e deployati in un container. I Message-driven beans sono usati per interagire con sistemi esterni ricevendo messaggi asincroni.

Process and Embedded Container

Da quando sono stati inventati gli EJB dovevano essere eseguiti in un contenitore che sarebbe stato eseguito su un JVM. Pensando a GlassFish, JBoss, Weblogic e così via ci si rende conto che il server deve prima essere avviato prima di deployare e utilizzare gli EJB. Questo in-process container è appropriato per un ambiente di produzione in cui il server esegue continuamente. Ma richiede tempo in un ambiente di sviluppo dove è spesso necessario avviare, distribuire, eseguire il debug e arrestare il container. Un altro problema con i server in esecuzione in un processo è che le capacità di test sono limitate. Per risolvere questi problemi, alcune implementazioni dell'application server avevano embedded container, ma questi erano specifici per l'implementazione. Da EJB 3.1 è stato standardizzato un embedded container portatile in tutti i server.

L'idea di un embedded container è di poter eseguire applicazioni EJB in un ambiente Java SE permettendo ai client di eseguire all'interno della stessa JVM e del class loader. Questo fornisce un migliore supporto per i test di integrazione, l'elaborazione offline e l'utilizzo dell'EJB nelle applicazioni desktop. L'API fornisce lo stesso ambiente gestito del container di runtime Java EE e include gli stessi servizi.

Ora è possibile eseguire il contenitore incorporato nello stesso JVM come il tuo IDE ed eseguire il debug del tuo EJB senza effettuare alcuna distribuzione a un server applicativo separato.

Services Given by the Container

Il container fornisce i seguenti servizi:

- *Remote client communication*: comunicazione remota con i client. senza scrivere codice complesso un client EJB può invocare metodi remoti.
- *Dependency injection*: il container inietta risorse in un EJB come in ogni altro POJO grazie a CDI.
- *State management*. Gestione dello stato. per i bean stateful il container gestisce lo stato in maniera trasparente.
- *Pooling*: per i bean stateless il container crea una pool di istanza che possono essere condivise da vari client. Una volta invocat, un EJB ritorna alla pool invece di essere distrutto.
- *Ciclo di vita dei componenti*: il container è responsabile di gestire il ciclo di vita di tutti i componenti
- *Messaging*: il container permette agli MDBs di ascoltare una destinazione e consumare messaggi.
- *Transaction management*. Gestione delle transazioni: Un EJB può usare annotazioni per specificare la transaction policy da usare e il container si occuperà di commit e rollback.
- *Interceptor*: i cross-cutting concerns possono esser inseriti negli interceptors che saranno invocati dal container
- *Invocazione asincrona di metodi*: Da EJB 3.1 si possono avere chiamate asincrone senza coinvolgere i messaggi.

Una volta che l'EJB è deployed, il programmatore può concentrarsi sulla business logic. Quando un client invoca un EJB, non funziona direttamente con un'istanza di quella EJB ma piuttosto con un proxy su un'istanza. Ogni volta che un client invoca un metodo su un EJB, la chiamata viene effettivamente proxyata e intercettata dal contenitore, che fornisce servizi per conto dell'istanza del bean. Naturalmente, questo è completamente trasparente per il client.

In un'applicazione Java EE, il container EJB interagisce con altri container: il container delle Servlet, il client application container, il broker di messaggi (per l'invio, la coda e la ricezione di messaggi), il persistence provider e così via.

I container forniscono agli EJB un insieme di servizi. D'altra parte, gli EJB non possono creare o gestire thread, accedere ai file utilizzando java.io, creare un ServerSocket, caricare una libreria nativa o utilizzare l'AWT (Abstract Window Toolkit) o le API Swing.

EJB Lite

Poiché EJB 3.2 definisce tecnologie complesse che sono raramente usate come l'interoperabilità con IIOP. Ogni vendor deve implementarle tutte e la cosa potrebbe risultare pesante anche agli sviluppatori che si avvicinano per la prima volta ad EJB. Per questo è stata definita una specifica EJB Lite che include un piccolo ma potente insieme di features:

Table 7-1. Comparison Between EJB Lite and Full EJB

Feature	EJB Lite	Full EJB 3.2
Session beans (stateless, stateful, singleton)	Yes	Yes
No-interface view	Yes	Yes
Local interface	Yes	Yes
Interceptors	Yes	Yes
Transaction support	Yes	Yes
Security	Yes	Yes
Embeddable API	Yes	Yes
Asynchronous calls	No	Yes
MDBs	No	Yes
Remote interface	No	Yes
JAX-WS web services	No	Yes
JAX-RS web services	No	Yes
Timer service	No	Yes
RMI/IIOP interoperability	No	Yes

Writing Enterprise Java Beans

I session bean encapsulano logica, sono transazionali, vivono in un container che fa pooling e multithread security e molto altro. Di cosa abbiamo bisogno per scrivere un EJB? Di una classe Java e delle annotazioni.

```
@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;
    }
}
```

Le versioni precedenti di J2EE richiedevano agli sviluppatori di creare diversi artefatti per creare un bean di sessione: un'interfaccia locale o remota (o entrambi), una **home locale** o **un'interfaccia home remota** (o entrambi) e un deployment descriptor.

Java EE 5 e EJB 3.0 hanno semplificato drasticamente il modello al punto in cui sono sufficienti solo una classe e una o più interfacce business e non è necessaria alcuna configurazione XML.

Anatomy of an EJB

L'esempio di sopra fa vedere un POJO annotato e senza interfacce. Ma un session bean può fornire un modello più ricco che permette di fare chiamate remote, dependency injection e chiamate asincrone. Un EJB è fatto dai seguenti elementi:

- *una classe bean*: che contiene l'implementazione di business methods e può implementare zero o più interfacce di business. Il Bean di Sessione deve essere annotato con @Stateless, @Stateful, o @Singleton
- *Business interfaces*: che contengono la dichiarazione di business methods visibili ai client. Un session bean può avere un'interfaccia remota, locale o nessuna delle due.

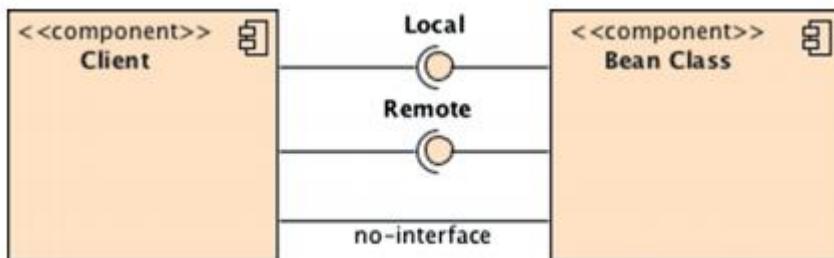


Figure 7-2. Bean class has several types of business interfaces

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
- li argomenti e i valori di ritorno devono essere tip 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.

Se la tua architettura ha un client che risiede fuori la JVM dell'EJB container deve implementare un'interfaccia remota. Questo si applica per i client che sono eseguiti in una JVM separata, in un application client container in un external web o un EJB container. In questo caso i metodi vanno invocati attraverso RMI.

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

È anche possibile che la tua applicazione usi entrambe le chiamate remote e locali sullo stesso bean di sessione.

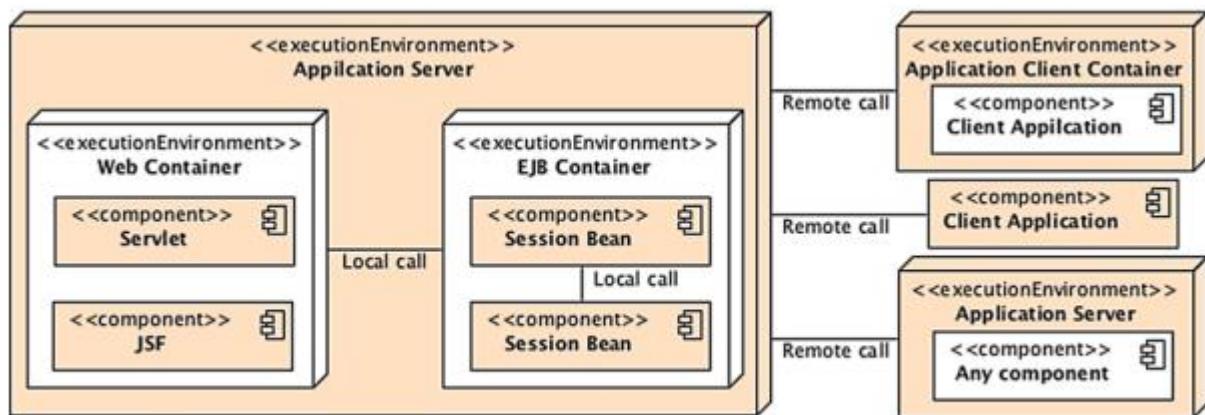


Figure 7-3. Session beans invoked by several types of client

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.

Non puoi annotare un'interfaccia con più di un'annotazione. Se non ha interfacce tutti i metodi sono accessibili localmente. Esempio:

Listing 7-2. Stateless Session Bean Implementing a Remote and Local Interface

```
@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 {  
    // ...  
}
```

In alternativa, è possibile specificare le interfacce all'interno del bean. Questo accade se si deve usare interfacce legacy in cui non è possibile aggiungere le annotazioni. Dunque @Local o @Remote possono essere specificate all'interno dei bean:

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 {  
    // ...  
}
```

Se un bean vuole esporre una vista senza interfacce ma implementa delle interfacce deve esplicitamente usare l'annotazione @LocalBean. Nel codice precedente, infatti, espone un'interfaccia remota, una locale e nessuna interfaccia.

Web Services Interface

Gli stateless bean possono essere invocati da remoto come SOAP web service o RESTful web service. Un bean può essere acceduto in maniera diversa solo usando delle annotazioni (@WebService per essere usato come SOAP e @Path per essere usati come RESTful)

Listing 7-4. A Stateless Session Bean Implementing Several Interfaces

```
@Local  
public interface ItemLocal {  
    List<Book> findBooks();  
    List<CD> findCDs();  
}  
  
@WebService  
public interface ItemSOAP {  
    List<Book> findBooks();  
    List<CD> findCDs();  
    Book createBook(Book book);  
    CD createCD(CD cd);  
}  
  
@Path(/items)  
public interface ItemRest {  
    List<Book> findBooks();  
}  
  
@Stateless  
public class ItemEJB implements ItemLocal, ItemSOAP, ItemRest {  
    // ...  
}
```

Portable JNDI Name

Prima i nomi JNDI non erano portabili, ed erano specifici dell'implementazione. Da EJB 3.1 i nomi sono stati specificati e sono diventati portabili. Ogni volta che si fa il deploy di un session bean si crea automaticamente un nome JNDI che segue la seguente sintassi:

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

Ogni parte del nome of the JNDI ha il seguente significato:

• <scope> defines a series of standard namespaces that map to the various scopes of a Java EE application:

- global: The java:global prefix allows a component executing outside a Java EE application to access a global namespace.

- app: The java:app prefix allows a component executing within a Java EE application to access an application-specific namespace.

- module: The java:module prefix allows a component executing within a Java EE application to access a module-specific namespace.

- comp: The java:comp prefix is a private component-specific namespace and is not accessible by other components.

• <app-name> is only required if the session bean is packaged within an ear or war file. If this is the case, the <app-name> defaults to the name of the ear or war file (without the .ear or .war file extension).

• <module-name> is the name of the module in which the session bean is packaged. It can be an

EJB module in a stand-alone jar file or a web module in a war file. The <module-name> defaults

to the base name of the archive with no file extension.

- <bean-name> is the name of the session bean.

- <fully-qualified-interface-name> is the fully qualified name of each defined business interface. For the no-interface view, the name can be the fully qualified bean class name.

Per illustrare la convenzione ecco degli esempi riferiti a ItemEJB che ha un'interfaccia locale, una remota e una vista senza interfacce:

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 {
    // ...
}
```

Once deployed, the container will create three JNDI names so an external component will be able to access the ItemEJB using the following global JNDI names:

```
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
```

Note that, if the ItemEJB was deployed within an ear file (e.g., myapplication.ear), you would have to use the <app-name> as follow:

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

The container is also required to make JNDI names available through the java:app and java:module namespaces. So a component deployed in the same application as the ItemEJB will be able to look it up using the following JNDI names:

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

This portable JNDI name can be applied to all session beans: stateless, stateful, and singleton.

Stateless Beans

Cosa vuol dire stateless? Un task deve essere concluso in una singola invocazione di metodo.

I servizi stateless sono ideali se si deve implementare un task che può essere concluso in una sola invocazione di metodo e non richiedono informazioni o stato tra una richiesta e l'altra. Se abbiamo bisogno ad esempio di creare un Book, settare dei parametri e renderlo persistente potremmo ad esempio fare così.

Nel codice seguente, puoi vedere che, dalla prima riga all'ultimo, l'oggetto libro è chiamato più volte e mantiene il suo stato:

```
Book book = new Book();
book.setTitle("The Hitchhiker's Guide to the Galaxy");
book.setPrice(12.5F);
book.setDescription("Science fiction comedy series created by Douglas Adams.");
book.setIsbn("1-84023-742-2");
book.setNbOfPage(354);
statelessService.persistToDatabase(book);
```

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. Ciò significa che è necessario un numero limitato di bean per gestire diversi client, come mostrato nella Figura 7-4.

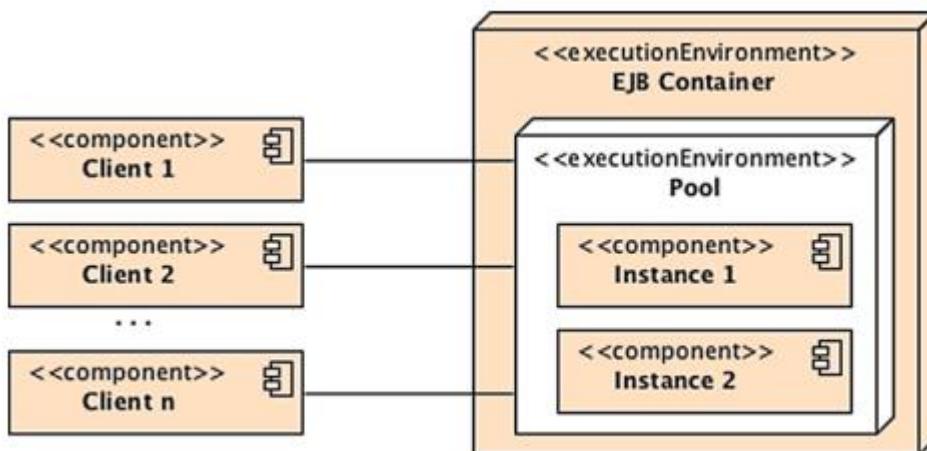


Figure 7-4. Clients accessing stateless beans in a pool

Un bean satateless è annotato con @Statelss e poiché vive in un container può usare dei servizi del container come la dependency injection. Per gli statelss session bean il persistence context è transazionale, questo significa che ogni metodo invocato in questo EJB (createBook(), createCD(), etc.) è transazionale.

```

@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;
    }
}

```

Nell'API l'annotazione è descritta così:

```

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

```

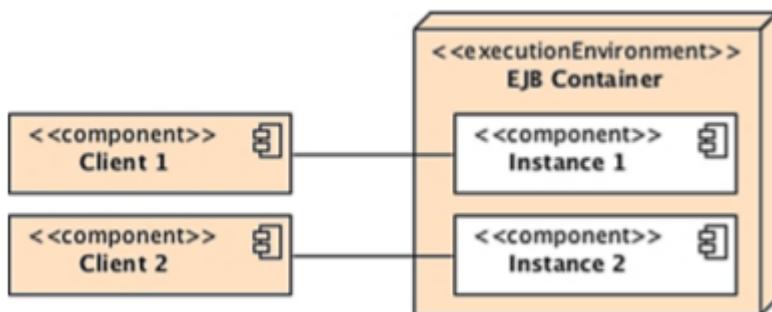
il parametro nome specifica il nome del bean ed è settato di default al nome della classe. Description può essere usato per descrivere l'EJB e mappedName è il nome JNDI assegnato dal container. Si noti che questo dipende dal vendor, non è portabile e non ha niente a che fare con il nome JNDI portabile descritto prima.

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. Prendiamo l'esempio di un carrello in un sito web di e-commerce. Un cliente accede (inizia la sessione), sceglie un primo libro, lo aggiunge al suo carrello, sceglie un secondo libro e lo aggiunge al suo carrello. Alla fine, il cliente controlla i libri, paga e si disconnette. Il carrello mantiene lo stato di quanti libri il cliente ha scelto durante l'interazione.

```
Book book = new Book();
book.setTitle("The Hitchhiker's Guide to the Galaxy");
book.setPrice(12.5F);
book.setDescription("Science fiction comedy series created by Douglas Adams.");
book.setIsbn("1-84023-742-2");
book.setNbOfPage(354);
statefulComponent.addBookToShoppingCart(book);
book.setTitle("The Robots of Dawn");
book.setPrice(18.25F);
book.setDescription("Isaac Asimov's Robot Series");
book.setIsbn("0-553-29949-2");
book.setNbOfPage(276);
statefulComponent.addBookToShoppingCart(book);
statefulComponent.checkOutShoppingCart();
```

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 uno a uno tra bean e cliente. La Figura 7-5 mostra una relazione uno-a-uno tra un'istanza bean e un client. Per quanto riguarda lo sviluppatore, nessun codice aggiuntivo è necessario, poiché il contenitore EJB gestisce automaticamente questa correlazione uno a uno.



Quindi se hai un milione di clienti servono un milione di bean. Per evitare questo incommodo 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 significa rimuovere l'istanza dalla memoria e salvarla in memoria persistente(file, db). Attivazione è il passaggio inverso, cioè significa fare il restore dello stato e applicarlo all'istanza. Entrambi sono fatti dal container in automatico. Ciò di cui ci si dovrebbe preoccupare è di liberare qualsiasi risorsa (ad esempio, connessione di database, connessione di stabilimenti JMS, ecc.) prima che il bean sia passivato. Dal EJB 3.2, è possibile anche disattivare passivazione.

```

@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();
}
}

```

L'unica annotazione necessaria è `@ javax.ejb.Stateful`, che ha la stessa API di `@Stateless`.

Si noti l'opzione `@ javax.ejb.StatefulTimeout` e `@ javax.ejb.Remove`. `@Remove` decora il metodo `checkout()`. In questo modo l'istanza del bean viene rimossa definitivamente dalla memoria dopo aver richiamato `checkout()`. `@StatefulTimeout` assegna un valore di timeout, che è la durata che il bean è autorizzato a rimanere inattivo (senza ricevere alcuna invocazione del client) prima di essere rimosso dal container. L'unità di misura di questa annotazione è un `java.util.concurrent.TimeUnit`, quindi può passare da `DAYS`, `HOURS` ... a `NANOSECONDS` (il valore predefinito è `MINUTES`). In alternativa, è possibile evitare queste annotazioni e fare affidamento sul container eliminando automaticamente un'istanza quando la sessione del client termina o scade. Tuttavia, assicurarsi che il bean stateful sia rimosso nel momento opportuno potrebbe ridurre il consumo di memoria.

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 la applicazione che può essere acceduta da qualsiasi punto della stessa. Per creare un singleton, si deve definire il costruttore privato, in modo tale che non si possano creare altre istanze dell'oggetto. Il metodo getInstance restituisce quell'unica istanza della classe, permettendone la modifica dei variabili di stato e la chiamata dei metodi. Il keyword synchronized previene l'interferenza di altri thread e inconsistenza dei dati.

Listing 7-8. A Java Class Following the Singleton Design Pattern

```
public class Cache {  
  
    private static Cache instance = new Cache();  
    private Map<Long, Object> cache = new HashMap<>();  
  
    private Cache() {}  
  
    public static synchronized Cache getInstance() {  
        return instance;  
    }  
  
    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;  
    }  
}
```

Una volta istanziato, il container fa in modo che ci sia una sola istanza del singleton per tutta la durata dell'applicazione, ed è condivisa tra molti client.

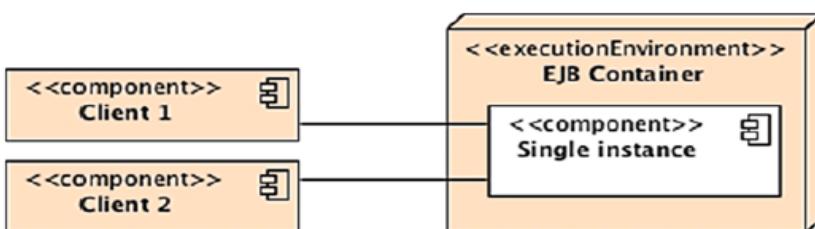


Figure 7-6. Clients accessing a singleton bean

Per trasformare una classe java singleton in un session bean singleton, basta aggiungere la annotazione `@Singleton` alla classe, e sarà poi il container ad assicurarsi che non vi siano altri istanze di tale classe. `@javax.ejb.Singleton` ha le stesse API della annotazione `@Stateless`.

Listing 7-9. Singleton Session Bean

```
@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;
    }
}
```

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'ultime, ogni superclasse o superinterfaccia necessaria, le eccezioni, le classi di aiuto e un deployment descriptor opzionale (ejb-jar.xml). Una volta che hai creato un file jar con tutti questi, puoi effettuare il deploy direttamente in un container. Un'altra opzione è di fare l'embed del jar in un file ear e effettuare il deploy su quest'ultima.

Un ear file è usato per inserire uno o più moduli (EJB o web app) in un singolo archivio così che il deployment in un'applicazione server possa avvenire in simultaneo e coerentemente. Per esempio, se hai bisogno di fare il deploy di una web app, potresti fare il package dei tuoi EJB in file jar separati, quello delle servlets in un war file, e il tutto in un ear file. Effettuare quindi il deploy del file ear in una applicazione server, e potrai quindi manipolare le entità del server usando l'EJB.

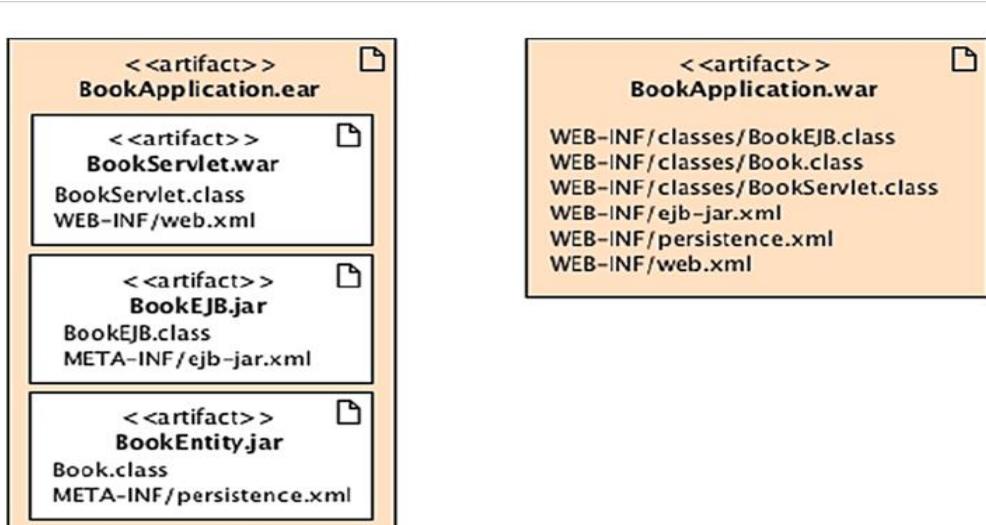


Figure 7-7. Packaging EJBs

I session bean sono componenti gestiti da un container, il quale tratta con ogni sorta di servizio, permettendo di concentrarsi solo sul codice di business. L'unico problema è quindi quello di dover necessariamente eseguire i EJB in un container, ma EJB 3.1 ha portato degli API standard per eseguire EJB in un ambiente Java SE. L'API del package javax.ejb.embeddable permette di instanziare un container EJB che può effettuare il run all'interno della propria JVM. Questo container fornisce un ambiente che supporta gli stessi servizi di base che esistono in un Java EE container (injection, transaction, life cycle e così via) e ha quindi le stesse capacità di un EJB lite container (ma non full EJB).

Listing 7-21. A Main Class Using the Embeddable Container

```
public class Main {  
  
    public static void main(String[] args) throws NamingException {  
  
        // Sets the container classpath  
        Map<String, Object> properties = new HashMap<>();  
        properties.put(EJBContainer.MODULES, new File("target/classes"));  
  
        // Creates an Embedded Container and get the JNDI context  
        try (EJBContainer ec = EJBContainer.createEJBContainer(properties)) {  
  
            Context ctx = ec.getContext();  
  
            // Creates an instance of book  
            Book book = new Book();  
            book.setTitle("The Hitchhiker's Guide to the Galaxy");  
            book.setPrice(12.5F);  
            book.setDescription("Science fiction comedy book");  
            book.setIsbn("1-84173-742-2");  
            book.setNbOfPage(354);  
            book.setIllustrations(false);  
  
            // Looks up the EJB with the no-interface view  
            ItemEJB itemEJB = (ItemEJB) ctx.lookup("java:global/classes/ItemEJB ");  
  
            // Persists the book to the database  
            itemEJB.createBook(book);  
  
            // Retrieves all the books from the database  
            for (Book aBook : itemEJB.findBooks()) {  
                System.out.println(aBook);  
            }  
        }  
    }  
}
```

Come è possibile vedere, Il metodo `createEJBContainer()` crea l'istanza di un container, che di default cerca il class path del client per ottenere un set di EJB per l'inizializzazione (si può selezionare il class path usando anche le proprietà). Una volta inizializzato il container, l'applicazione ottiene il contesto JNDI del container (`.getContext()` che ritorna un `javax.naming.Context`) per cercare l' ItemEJB.

Invoking Enterprise Java Beans

Per un invocare un metodo su un session bean, un client ha bisogno solo di un riferimento al bean, o alla sua interfaccia, e può ottenerlo sia tramite dependency injection (Annotazioni `@EJB` o `@Inject`) o tramite ricerca JNDI.

Invoke con Injection

La annotazione `@javax.ejb.EJB` serve a fare l'injection dei riferimenti ai session bean in un codice del client. DI è possibile solo all'interno di un ambiente gestito come l'EJB container, web container e application-cliente container.

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

Nel caso un session bean implementi più interfacce il client deve specificare quale riferimento vuole tra i loro.

```
@Stateless  
@Remote(ItemRemote.class)  
@Local(ItemLocal.class)  
@LocalBean  
public class ItemEJB implements ItemLocal, ItemRemote {...}  
  
// Client code injecting several references to the EJB or interfaces  
@EJB ItemEJB itemEJB;  
@EJB ItemLocal itemEJBLocal;  
@EJB ItemRemote itemEJBRemote;
```

L'API @EJB ha molti attributi, tra cui il nome JNDI dell'EJB di cui si vuole fare l'Inject. Questo può essere utile quando l'EJB remoto è presente su un server differente.

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

Invocare con CDI

@EJB non fornisce un meccanismo simile alle CDI Alternatives, per esempio. Per questo abbiamo bisogno di usare @Inject. Questo pone un problema: per gli EJB remoti serve il parametro con il nome JNDI ma @Inject non prende un parametro stringa, quindi abbiamo bisogno di produrre un EJB e poi iniettarlo.

Si utilizza la notazione @Inject al posto di @EJB, e nel lookup si aggiunge @Produces.

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

Invocare con JNDI

JNDI è usato soprattutto per un accesso remoto quando un client non è gestito da un container e non può usare la dependency injection. Con JNDI l'injection avviene a tempo di deployment, quindi se c'è la possibilità che i dati non vengano usati, il bean può evitare il costo dell'injection delle risorse.

JNDI è un API per l'accesso a differenti tipi di servizi di directory, permettendo i client di cercare e collegare gli oggetti tramite un nome indipendentemente dall'implementazione, che sia in Lightweight Directory Access Protocol (LDAP) directory o DNS, usando API standard.

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

CALLBACKS, TIMER SERVICE AND AUTHORIZATON

Tra i servizi che il container offre agli EJB ci sono: gestione del ciclo di vita, scheduling e autorizzazioni. La gestione del ciclo di vita vuol dire che i bean attraversano determinati stati in maniera dipendente dal tipo di bean (stateless, stateful, singleton). Ogni volta che il bean cambia stato il container invoca i metodi con le annotazioni di callback.

Anche la sicurezza è importante. Vogliamo che il business tier autorizzi alcune azioni ad un certo gruppo di utenti e rifiuti l'accesso ad altri.

Session Beans 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.

Ciò significa che né il cliente né il bean sono responsabili di determinare quando viene creata l'istanza bean, quando vengono iniettate le dipendenze o quando l'istanza viene distrutta. È il container ad occuparsi del ciclo di vita di un session Bean.

Tutti i bean hanno due fasi ovvie: creazione e distruzione. In più, i bean stateful hanno anche attivazione e passivazione

Stateless and Singleton

Stateless e Singleton beans hanno in comune il fatto che non mantengono lo stato di conversazione con un client. Hanno lo stesso ciclo di vita rappresentato in figura.

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 anche quando in container viene avviato (usando l'annotazione @Startup)
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 finché il container non deve liberare spazio nella pool, i singleton finché il container viene spento.
5. il container non ha più bisogno dell'istanza, invoca i metodi @PreDestroy se ci sono e termina la vita dell'istanza.

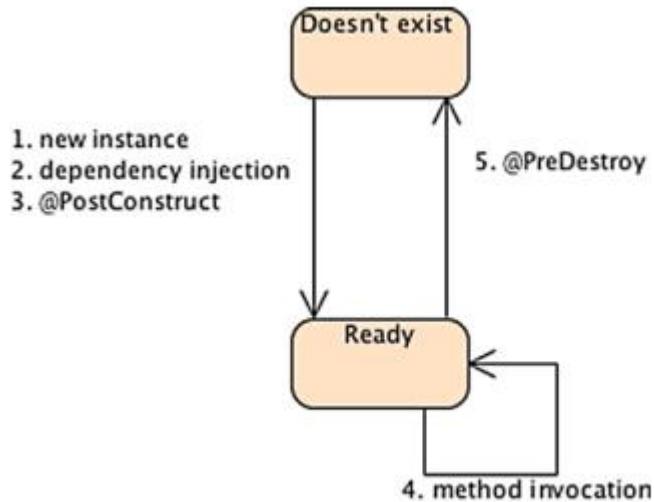


Figure 8-1. Stateless and singleton bean life cycle

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 contenitore 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 contenitore non ha più bisogno dell'istanza (solitamente quando il contenitore vuole ridurre il numero di istanze nel pool), 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 sono programmaticamente simili dai Stateless o Singleton: solo l'annotazione è diversa (`@Stateful` anziché `@Stateless` o `@Singleton`). Ma la vera differenza è che gli Stateful mantengono lo stato della conversazione con il proprio 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. Seguendo questo principio e in base alla tua applicazione, potresti ritrovarti ad es. con 1000 utenti simultanei potrebbero produrre 1000 Statefull Bean.

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, cioè se uno stateful bean resta inattivo per molto tempo e il container ha bisogno di memoria, il bean viene passivato (mettendolo in memoria persistente) e viene riattivato quando ne ha bisogno.

La passivazione avviene quando il container serializza l'istanza del bean su un supporto di memorizzazione permanente invece di tenerlo in memoria. L'attivazione, che è l'opposto, viene eseguita quando l'istanza del bean è nuovamente necessaria dal client. Il container deserializza il bean dalla memoria permanente e lo riattiva in memoria. Ciò significa che gli attributi del bean devono essere serializzabili.

La figura mostra il ciclo di vita di un Bean Stateful, ed 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 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 passivata 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.

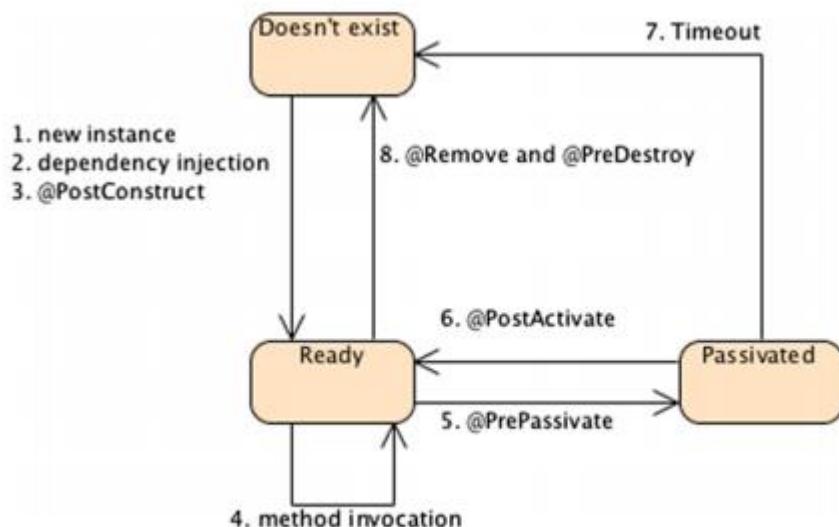


Figure 8-2. Stateful bean life cycle

Spesso uno stateful bean apre risorse come socket o connessioni al db. Poichè 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. Si può anche disattivare il meccanismo di attivazione e passivazione con `@Stateful(passivationCapable=false)`.

Callbacks

Come hai appena visto, ogni tipo di bean di sessione ha il proprio ciclo di vita gestito dal container. Il contenitore consente di inserire il proprio business code quando lo stato del bean cambia.

I cambiamenti di stato vengono intercettati dal container che chiama i metodi annotati con le seguenti annotazioni:

Le seguenti regole si applicano ai metodi di callback:

Table 8-1. Life-Cycle Callback Annotations

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 causerà il rollback della transazione se ne esiste una.
- Il metodo può avere accesso public, private, protected, o package-level, ma non deve essere static o final.
- Il metodo può essere annotato con più annotazioni. 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).
- Il metodo di callback può accedere alle voci dell'ambiente dei bean

Questi metodi di callback vengono in genere utilizzati per allocare e/o rilasciare le risorse del bean. Ad esempio, il codice sotto mostra il bean singleton CacheEJB usando un'annotazione `@PostConstruct` per inizializzare la sua cache. Subito dopo aver creato la singola istanza di CacheEJB, il container 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;
    }
}
```

Il codice sotto riporta un bean stateful.

Il container mantiene lo stato della conversazione, che può includere risorse pesanti come una connessione al database. Perché è costoso aprire una connessione al database, la connection al database viene condivisa tra le chiamate dei metodi e rilasciata quando il bean è passivato.

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();
    }
}
```

Dopo aver creato un'istanza di un bean stateful, il contenitore inietta il riferimento di un'origine dati predefinita all'attributo ds. Una volta eseguita l'iniezione, il contenitore chiamerà il metodo `@PostConstruct` che crea una connessione al database. Se al contenitore capita di passivare l'istanza, verrà invocato il metodo `close()` (`@PrePassivate`). Lo scopo è chiudere la connessione JDBC, che contiene risorse native e non è più necessaria durante la passivazione. Quando il cliente invoca un metodo di business sul bean, il container lo attiva e chiama di nuovo `init()` (`@PostActivate`). Quando il client richiama il metodo `checkout()` (annotato con `@Remove`), il contenitore rimuove l'istanza, ma prima chiamerà di nuovo il metodo `close()` (`@PreDestroy`). Si noti che per leggibilità si è omessa la gestione delle eccezioni SQL.[È stata la connection al db ad esempio per motivi legacy]

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. [Notare bene la differenza autenticazione/ autorizzazione] L'autorizzazione può essere fatta sia in modo dichiarativo che 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(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 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 usata 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 <code>launchNuclearWar()</code> 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).
@DeclareRoles	X		Defines roles for security checking.
@RunAs	X		Temporarily assigns a new role to a principal.

L'annotazione @RolesAllowed viene utilizzata per autorizzare un elenco di ruoli per accedere a un metodo. Può essere applicato a un metodo particolare o all'intero bean. Questa annotazione può prendere una singola stringa (solo un ruolo può accedere al metodo) o una matrice di String (qualsiasi ruolo può accedere a qualsiasi ruolo il metodo). L'annotazione @DeclareRoles può essere utilizzata per dichiarare altri ruoli.

Nel codice che segue @RolesAllowed indica che i metodi sono accessibili a user, employee e admin. Il metodo `deleteBook()` è accessibile solo agli admin.

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));  
    }  
}
```

`@RolesAllowed` definisce un elenco di ruoli a cui è consentito accedere a un metodo. Le annotazioni `@PermitAll` e `@DenyAll` vengono applicate per qualsiasi ruolo.

`@PermitAll` può essere acceduto da qualsiasi ruolo, `@DenyAll` è negato a tutti.

Come puoi vedere sotto poiché il metodo `findBookById()` è annotato con `@PermitAll`, ora può essere accessibile a qualsiasi ruolo, non solo all'utente, al dipendente e all'amministratore. D'altra parte, il metodo `findConfidentialBook()` non è accessibile a nessuno(`@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.persist(book);
        return book;
    }

    @RolesAllowed("admin")
    public void deleteBook(Book book) {
        em.remove(em.merge(book));
    }

    @DenyAll
    public Book findConfidentialBook(Long secureId) {
        return em.find(Book.class, secureId);
    }
}

```

L'annotazione `@DeclareRoles` dichiara i ruoli per l'intera applicazione. Quando si distribuisce l'EJB del codice sopra, il container dichiarerà automaticamente i ruoli utente, dipendente e amministratore controllando l'annotazione `@RolesAllowed`. Ma potresti voler dichiarare altri ruoli 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. 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 deployment descriptor che con l'annotazione `@DeclareRoles`. Quando si distribuisce ItemEJB nel codice sotto vengono dichiarati i cinque ruoli HR, salesDpt, user, employee e admin.

Listing 8-7. A Stateless Bean Declaring Roles

```
@Stateless  
@DeclareRoles({"HR", "salesDpt"})  
@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));  
    }  
}
```

@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. Ad esempio, ItemEJB sotto autorizza l'accesso al ruolo utente, dipendente e amministratore. Quando uno di questi ruoli accede a un metodo, il metodo viene eseguito con il ruolo temporaneo inventoryDpt (@RunAs ("inventoryDpt")).

Ciò significa che quando si richiama il metodo createBook (), il metodo InventoryEJB.addItem () verrà richiamato con un ruolo inventoryDpt.

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;  
    }  
}
```

Come puoi vedere, 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? È qui che entra in gioco l'autorizzazione programmatica.

Programmatic Authorization

L'autorizzazione dichiarativa copre la maggior parte dei casi di sicurezza necessari per un'applicazione. Ma 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, ecc.). In questo caso è possibile utilizzare l'autorizzazione programmatica per consentire o bloccare in modo selettivo l'accesso. Si fa grazie all' accesso diretto all'interfaccia `java.security.Principal` di JAAS.

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 ha 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 l'errore all' utente sulla violazione dell'autorizzazione.

Il metodo `createBook ()` esegue una business logic utilizzando i ruoli e il context. Si noti che il metodo `getCallerPrincipal ()` restituisce un oggetto `Principa(Context)l`, che ha un nome. Il metodo controlla se il nome dell'input è `paul`, quindi imposta il valore "utente speciale" sull'entità `book`.

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

Le transactions permettono di avere dati consistenti (coerenti) che possono essere poi processati in modo affidabile. Non essendo una questione che uno sviluppatore deve trattare, EJB provvede questi servizi in modo molto semplice: programmaticamente tramite un alto livello di astrazione o in modo dichiarativo usando i metadata.

La maggior parte del lavoro di un'applicazione enterprise riguarda la gestione dei dati: la memorizzazione (in genere in un database), il loro recupero, l'elaborazione, e così via. Spesso questo viene fatto contemporaneamente da diverse applicazioni che tentano di accedere agli stessi dati. Un database ha meccanismi di basso livello per preservare l'accesso simultaneo, come il lock pessimistico, utilizza dunque le transaction per garantire che i dati rimangano in uno stato coerente. I bean fanno uso di questi meccanismi.

Understanding Transactions

Una transaction è usata per assicurare che tutti i data sono tenuti in uno stato consistente. Rappresenta un gruppo logico di operazioni che vengono eseguite come una sola unità, chiamata “unit of work”. Fanno parte di queste operazione il rendere persistenti i dati su uno o più database, mandare messaggi a un Message-Oriented-Middleware (MOM), o invocare un servizio web.

Le aziende fanno affidamento sulle transazioni ogni giorno per loro applicazioni bancarie e di e-commerce

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'unità di lavoro. Alla conclusionedella 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 una transaction devono essere visibili ad altre applicazioni.

Read Conditions

L'isolation delle transaction può essere definito usando differenti read condition, che descrivono cosa accade quando due o più transaction operano sugli stessi dati nello stesso momento.

- Dirty reads: avviene quando una transaction legge i cambiamenti di cui una precedente transaction di cui non ha fatto il commit.
- Repeatable reads: Avviene quando si garantisce che il dato letto non cambi per tutta la durata della transaction.
- Phantom reads: avviene quando dei nuovi record aggiunti al database sono rilevabili dalle transaction iniziate prima dell'inserimento. Le query includeranno i record aggiunti da altre transaction dopo che le loro transaction sono iniziate.

Transaction Isolation Levels

I database usano diverse tecniche di locking che controllano come le transaction accedono ai dati in modo concorrente. Queste tecniche hanno un impatto sulle read condition descritte precedentemente. I livelli di isolamento sono comunemente usati nei database per descrivere come viene applicato il lock ai dati all'interno di una transazione.

Vi sono quattro livello di isolamento.

- Read uncommitted (il più basso livello di isolamento): La transaction può leggere dati sui cui non è stato effettuato il commit. Possono verificarsi Dirty, Non Repeatable e Phantom read.
- Read committed: La transaction non può leggere i dati su cui non è stato effettuato il committed (Non sono stati salvati). Possono avvenire i nonrepeatable e i phantom read.
- Repeatable read: Una transaction non può effettuare modifiche su dati letti da una differente transaction in esecuzione in quel momento. Possono avvenire solo i Phantom reads.
- Serializable (il più alto livello di isolamento): Una transaction ha lettura esclusiva. Le altre transaction non possono ne leggere ne scrivere gli stessi dati.

Ovviamente più aumenta il livello di isolamento più diminuiscono le performance di un sistema, poiché le transaction non possono accedere agli stessi dati nello stesso momento.

JTA Local Transactions

Prendiamo l'esempio di un'applicazione che esegue diverse modifiche a una singola risorsa (ad es. un database).

Quando vi è solo una risorsa transactional, è necessaria solo una transaction JTA locale. Una transaction locale è una transaction che hai con una singola risorsa la quale utilizza le proprie API specifiche.

La Figura 9-1 mostra l'applicazione interagire con una risorsa attraverso un gestore delle transazioni e un gestore delle risorse.

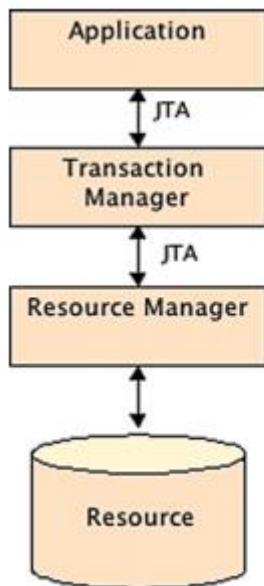


Figure 9-1. A transaction involving one resource

I componenti mostrati nella Figura 9-1 riassumono la maggior parte delle specifiche di una transaction:

- Transaction Manager: è il componente centrale responsabile delle operazioni delle transaction. Esso crea le transaction secondo la applicazione, informa il Resource Manager che è coinvolto in una transaction (enlistment), e gestisce il commit o il rollback sul Resource Manager.
- Il Resource Manager è responsabile della gestione delle risorse e le registra con il Transaction Manager. Un esempio di resource manager è un driver per un database relazionale, una risorsa JSM, ecc.
- La Resource è lo storage persistente da cui si legge o si scrive (database, una destinazione di un messaggio, etc..)

Non è responsabilità dell'applicazione conservare le proprietà ACID. L'applicazione decide solo su il commit o rollback della transazione e il Manager Transaction prepara tutte le risorse per eseguirla correttamente.

Transaction distribuite e XA

Una transaction su una singola risorsa è detta transazione locale, ma molte applicazioni enterprise usano più di una risorsa. Serve quindi la gestione delle transaction su più risorse o su risorse distribuite sulla rete. Questo richiede un coordinamento che coinvolge XA e JTS.

La Figura 9-2 mostra un'applicazione che utilizza la demarcazione delle transazioni su più risorse. Ciò significa che, nella stessa unit of work, l'applicazione può mantenere i dati in un database e inviare un messaggio JMS, ad esempio.

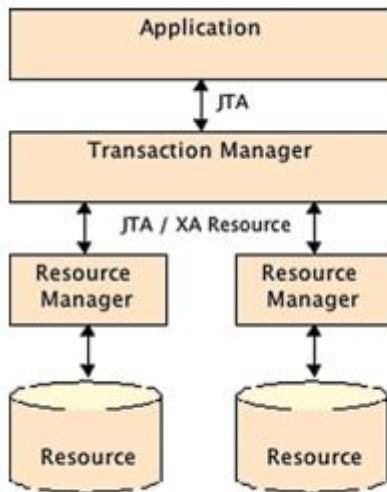


Figure 9-2. An XA transaction involving two resources

Per avere una transaction affidabile tra differenti risorse (non JTA local), il transaction manager ha bisogno di usare una interfaccia del resource manager XA (eXtended Architecture). XA è uno standard specifico per il processo di transaction distribuite (DTP) che preserva le proprietà ACID. È supportato da JTA e permette a Resource Manager eterogenei di interagire tramite una interfaccia comune. Usa un **two-phase commit** (2pc) che assicura che tutte le risorse possono fare o il commit o il roll back di qualsiasi transaction simultaneamente. Nella prima fase, si notifica tramite un comando “prepare” ad ogni resource manager che un commit sta per avvenire. Questo permette ai Resource Manager di dichiarare se si possono applicare le modifiche o meno. Se tutti dichiarano di essere preparati, la transaction può avvenire, e a quel punto si chiedere a tutti i Resource Manager di fare il commit nella seconda fase.

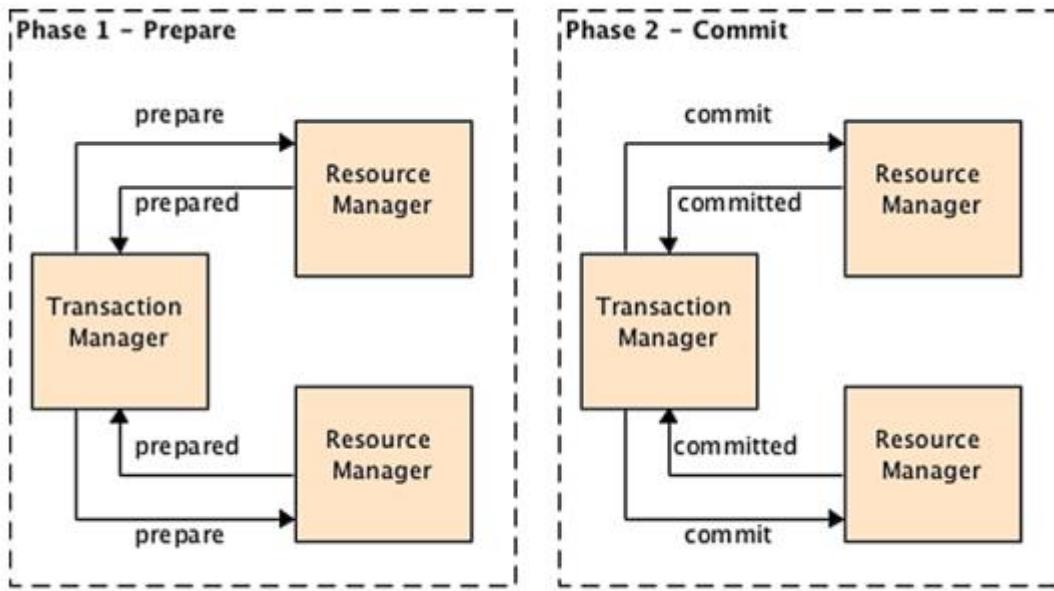


Figure 9-3. Two-phase commit

Essendo solitamente le risorse distribuite per tutta la rete. Si utilizza JTS, che implementa la specifica Object Transaction Service (OTS) di Object Management Group (OMG) che permette ai Transaction Manager di partecipare alle transaction distribuite tramite il protocollo Internet Inter-ORB (IIOP).

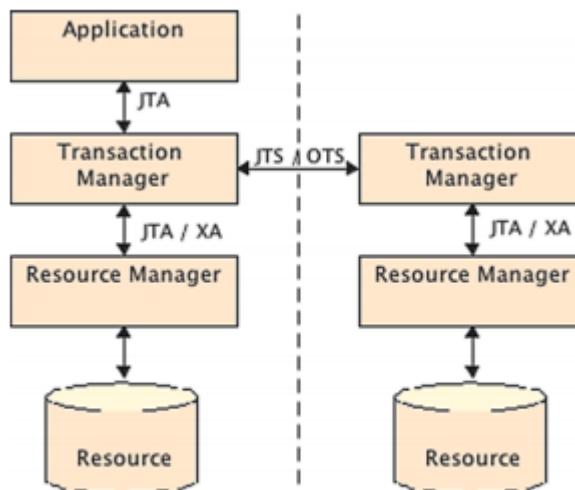


Figure 9-4. A distributed XA transaction

Come sviluppatore di EJB, non devi preoccuparti di questo; basta usare JTA, che si interfaccia con JTS ad un livello più alto.

Transaction Support In EJBs

Quando sviluppi business logic con EJB, non bisogna preoccuparsi della struttura interna del Transaction Manager o dei Resource Manager perché JTA astrae la maggior parte della complessità sottostante.

Con gli EJBs, si può sviluppare facilmente una applicazione transactional, lasciando al container il compito di implementare i protocolli transaction di basso livello, come il 2pc(**two-phase commit**) ad esempio. Dalla sua creazione, il modello EJB è stato progettato per gestire le transazioni. Infatti, l'EJBs di default include ciascun metodo in una transaction. Questo comportamento di default prende il nome di container-managed transaction (CMT), poiché le transazioni sono gestite dal contenitore EJB.

È anche possibile scegliere di gestire le transaction usando un bean-managed transaction (BMT), chiamato anche “programmatic transaction demarcation”. I transaction demarcation determinano quando le transaction iniziano e finiscono.

Container-Managed Transaction

Quando gestisci le transaction in modo dichiarativo, la politica riguardante le demarcation la si delega al container. Non è necessario usare il JTA nel codice, basta lasciare al container che automaticamente iniziare e farà il commit delle transaction basandosi sui metadata. In un'impresa bean con una transazione gestita dal contenitore, il contenitore EJB impone i limiti delle transazioni.

Il Listato 9-1 mostra il codice di un bean di sessione stateless usando CMT.

Come puoi vedere, non c'è un'annotazione aggiuntiva o qualsiasi interfaccia speciale da implementare. Gli EJB sono per natura transazionali.

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;
    }
}
```

Per comprendere cosa rende questo codice transactional, bisogna comprendere cosa accade quando un cliente invoca “createBook()”. Infatti, 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 createbook. Una volta che il metodo è terminato, il container esegue il commit o il roll back.

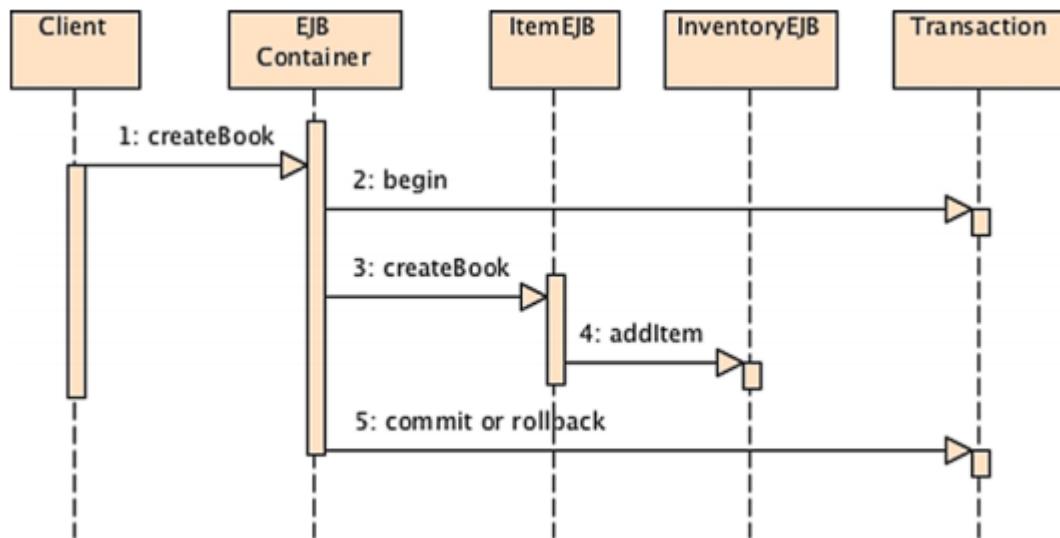


Figure 9-5. The container handles the transaction

Il comportamento di default può essere modificato usando i metadata (annotazione o XML deployment descriptor), in cui a seconda dell’attributo scelto (REQUIRED, REQUIRES_NEW, SUPPORTS, MANDATORY, NOT_SUPPORTED, o NEVER) puoi cambiare il modo in cui il container demarca le transaction.

Attributi della Transaction (CMT attributes)

- REQUIRED: questo attributo (valore di default) indica che il metodo deve essere sempre richiamato all’interno di una transaction. Se proviene da un client non transactional, il container crea una nuova transaction.
- REQUIRES_NEW: Il container crea sempre una nuova transaction prima di eseguire un metodo, indipendentemente se il client viene eseguito all’interno di una transaction. Se il cliente è eseguito una transaction, il container sospende tale transaction temporaneamente e ne crea una nuova, ne fa il commit o il roll back, e poi riprende la transaction sospesa.
- SUPPORTS: Il metodo EJB eredita il contesto della transaction del client. Se è disponibile un transaction context, è usato dal metodo. Se non è disponibile, il container invoca il metodo senza transaction context.
- MANDATORY: Il container necessita di una transaction prima di invocare un metodo ma non deve creare una nuova. Se non è disponibile un transaction context nel client, si invoca una eccezione javax.ejb.EJBTransactionRequiredException.

- NOT_SUPPORTED: Il metodo EJB non può essere invocato in una transaction context. Se è presente nel client, il container sospende la transaction e invoca il metodo.
- NEVER: il metodo EJB non deve essere invocato da un transactional client. Se il client è in esecuzione in un transaction context, il container lancia un'eccezione javax.ejb.EJBException.

Bean_Managed Transactions

Con CMT, si lascia il conteiner per fare la demarcazione della transazione semplicemente specificando un attributo di transazione e utilizzando il session context o le eccezioni per contrassegnare una transazione per il rollback. 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). Come si risolve? Gli EJB offrono un modo programmatico per gestire le demarcazioni delle transazioni con BMT. BMT ti permette di gestire esplicitamente i limiti delle transazioni (inizio, commit, rollback) usando JTA.

BMT ti permette di gestire esplicitamente i confini di una transaction(inizio, commit, rollback), usando JTA, al posto di lasciarlo fare al container, come nel CMT.

Per disattivare la demarcazione di default CMT e passare alla modalità BMT, si usa la annotazione @javax.ejb.TransactionManagement (o nel file XML)

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

Con il BMT, 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 javax.transaction.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 status della transaction corrente
- setTransactionTimeout: Modifica il timeout della transaction corrente

Il Listato 9-6 mostra come sviluppare un bean BMT.

Prima di tutto, otteniamo un riferimento di UserTransaction usando iniezione tramite l'annotazione @Resource. Il metodo oneItemSold () inizia la transazione, svolge alcune operazioni e, a seconda della logic business, fa il commit o esegue il rollback della transazione.

Si noti anche che la transazione per il rollback è contrassegnata nel blocco catch (ho semplificato la gestione delle eccezioni per una migliore leggibilità)

Listing 9-6. A Stateless Bean with BMT

```
@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();
    }
}
```

La differenza con il codice CMT mostrato nel Listato 9-3 è che con CMT il container inizia la transazione prima dell'esecuzione del metodo e ne fa il commit immediatamente dopo. Con il codice BMT mostrato nel Listato 9-6, si definiscono manualmente i limiti della transazione all'interno del metodo stesso.

MESSAGING

La maggior parte delle comunicazioni tra componenti viste fino ad ora sono sincrone: una classe chiama un'altra, un bean richiama un EJB, che chiama un'entità e così via. In questi casi, l'invocatore e il target devono essere attivi e in esecuzione affinché la comunicazione abbia successo e l'invocatore deve attendere che il target sia completato prima di procedere. Ad eccezione delle chiamate asincrone in EJB (grazie all'annotazione `@Asynchronous`), la maggior parte delle componenti Java EE utilizzano le chiamate sincrone (locali o remote). Quando parliamo di messaggistica, intendiamo generalmente comunicazioni asincrone tra componenti.

Middleware orientato ai messaggi (MOM, Message-Oriented Middleware) è un software (un provider) 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.

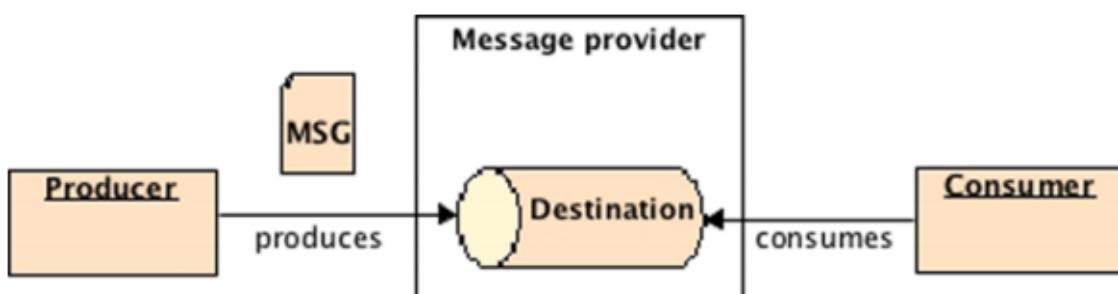
MOM si basa su un modello di interazione asincrona, quindi consente a queste applicazioni di funzionare in modo indipendente e, allo stesso tempo, di far parte di un processo di flusso di lavoro delle informazioni. La messaggistica è una buona soluzione per l'integrazione di esistenti e nuove applicazioni in un modo accoppiato, asincrono, purché il produttore e il consumatore siano d'accordo sul formato del messaggio e la destinazione intermedia. Questa comunicazione può essere locale all'interno di un'organizzazione o distribuita tra diversi servizi esterni.

Understanding Messaging

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

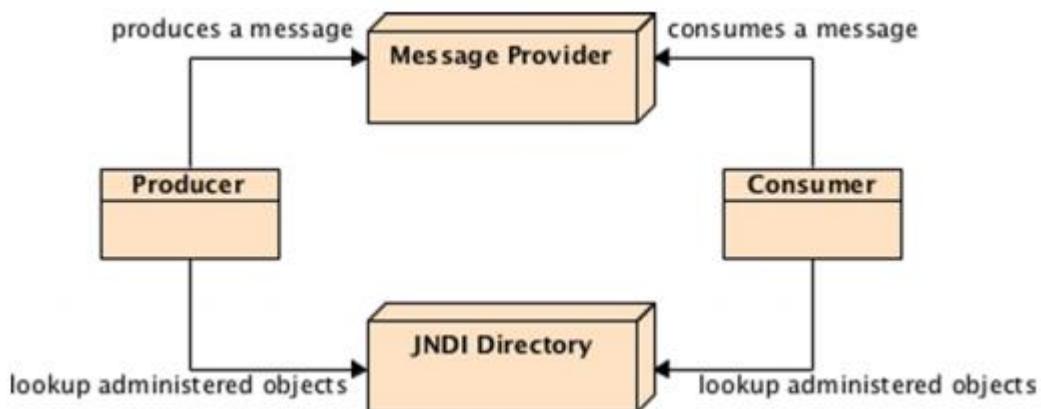
Il mittente del messaggio è chiamato **Producer** e il percorso in cui è archiviato il messaggio è chiamato **destinazione**.

Il componente che riceve il messaggio è chiamato **consumatore**. Qualsiasi componente 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** (coda). Quando si utilizza la messaggistica point-to-point, un client 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-subscribe (pub-sub)**: la destinazione è chiamata **topic**. Quando si utilizza la pubblicazione / sottoscrivitura dei messaggi, un client pubblica un messaggio su un topic argomento 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 destinatari. 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.

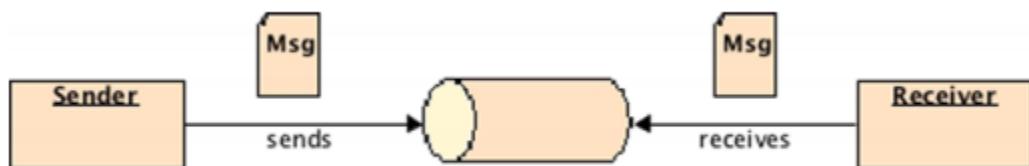


Figure 13-3. P2P model

Ogni messaggio viene inviato a una coda specifica e il destinatario estrae i messaggi dalla coda. Le code mantengono tutto messaggi inviati fino a quando non vengono consumati o fino alla loro scadenza. Il modello P2P viene utilizzato se c'è un solo 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.

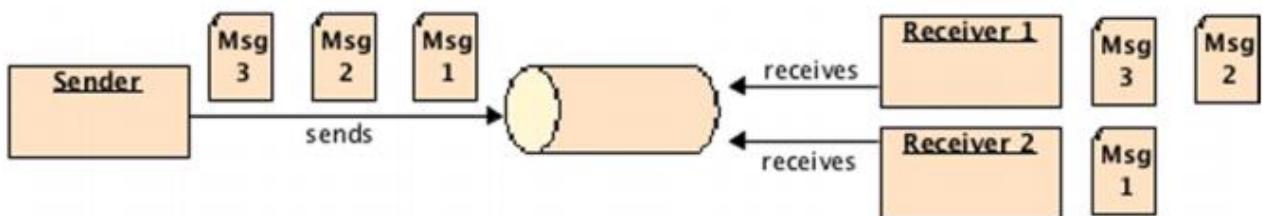


Figure 13-4. Multiple receivers

Si noti che P2P non garantisce che i messaggi vengano consegnati in un ordine particolare (l'ordine non è definito). Un fornitore potrebbe selezionarli in ordine di arrivo, in modo casuale, o in un altro modo.

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 e ciò si verifica in modo dinamico.

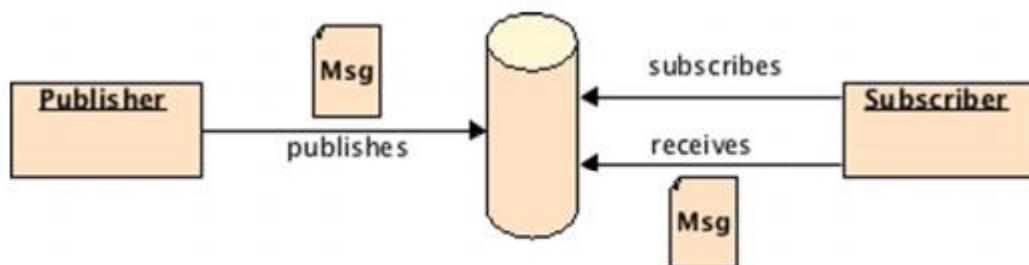


Figure 13-5. Pub-sub model

Il Topic conserva i messaggi fino a quando non vengono distribuiti a tutti i sottoscritti. A differenza del modello P2P, c'è a dipendenza temporale tra publishers e subscribers; i subscribers non ricevono i messaggi inviati prima della loro iscrizione e, se sono inattivo per un periodo specificato, non riceveranno i messaggi fin quando non ritornano attivi. Vedremo che questo può essere evitato, con l'API JMS.

Più subrscribers possono consumare lo stesso messaggio. Il modello pub-sub può essere utilizzato per applicazioni di tipo broadcast, in cui un singolo messaggio viene consegnato a più consumatori.

Nella seguente figura, il publisher invia tre messaggi che ciascun utente riceverà (in un ordine non definito).

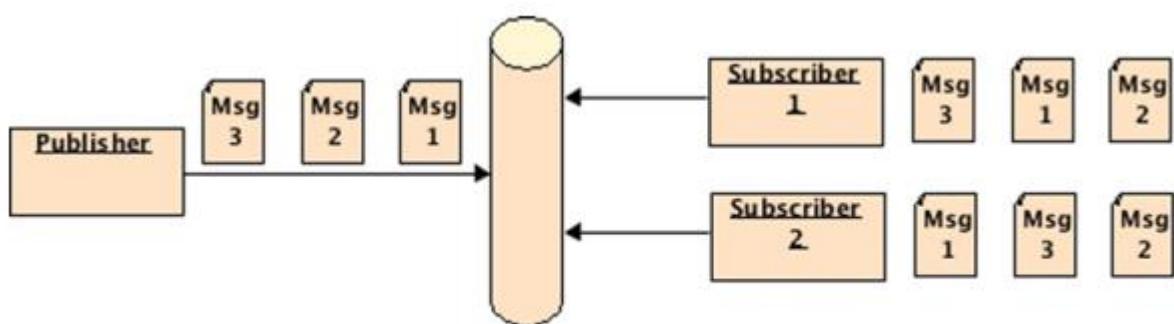


Figure 13-6. Multiple subscribers

Administered Objects

Gli Administered 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 oggetti 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. Le destinazioni possono essere code (P2P) o argomenti (pub-sub).

I client accedono a questi oggetti tramite interfacce portabili facendo il lookup in JNDI namespace o attraverso injection. In GlassFish, ci sono diversi modi per creare questi oggetti: usando la console administration console, la riga dei comandi asadmin o l'interfaccia REST. Con JMS 2.0 è possibile utilizzare anche `@JMSConnectionFactoryDefinition` e `@JMSSDestinationDefinition` per definire a livello di codice questi oggetti.

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 acknowledgement, ecc.), 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'unico 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 stateles, gli MDB non mantengono lo stato tra invocazioni separate da un messaggio ricevuto a quello successivo. MDB risponde ai messaggi ricevuti dal contenitore, mentre i bean di sessione stateless rispondono alle richieste del client attraverso un'interfaccia appropriata (locale, remota o senza interfaccia).

Java Messaging Service API

JMS è una API standard che permette alle applicazioni di creare, inviare, ricevere e leggere messaggi in modo asincrono. Definisce un comune set di interfacce e classi che permettono ai programmi di comunicare con altri message providers. Simile a JDBC il quale è capace di collegarsi con diversi database, JMS è capace di connettersi con diversi provider. L'API di JMS si è evoluta fin dalla sua creazione. Per motivi storici JMS offre 3 set alternativi di interfacce per produrre e consumare messaggi e sono note oggi come legacy API, classic API e simplified API.

JMS 1.0 ha fatto una chiara distinzione tra i modelli point-to-point e publish-subscribe. Ha definito due API di dominio specifico, una per il point-to-point (queues) e una per il pub-sub (topics). Ecco perché è possibile trovare QueueConnectionFactory e TopicConnectionFactory invece di una API generica ConnectionFactory. L'API 1.1 (riferito alla classic API) ha provveduto ad un set unificato di interfacce da poter utilizzare con entrambi i modelli. La tabella mostra relazione tra modello ed interfacce

Table 13-1. Interfaces Depending on JMS Version

Classic API	Simplified API	Legacy API (P2P)	Legacy API (Pub-Sub)
ConnectionFactory	ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	JMSCContext	QueueConnection	TopicConnection
Session	JMSCContext	QueueSession	TopicSession
Destination	Destination	Queue	Topic
Message	Message	Message	Message
MessageConsumer	JMSConsumer	QueueReceiver	TopicSubscriber
MessageProducer	JMSProducer	QueueSender	TopicPublisher
JMSException	JMSRuntimeException	JMSException	JMSException

Ma JMS 1.1 era ancora una API prolissa e di basso livello, messa a confronto con JPA o EJB. JMS 2.0 ha introdotto la simplified API che offre tutte le funzionalità della classic API ma richiede meno interfacce ed è più semplice da usare. La Tabella mostra la relazione tra le API. Non si discuterà la legacy API ma è stato necessario introdurre la classic API per il largo uso che se ne fa ancora oggi ed anche perché la simplified API si basa su di essa

Classic API

La JMS classic API offre classi ed interfacce per applicazioni che richiedono un sistema di messaggistica (mostrato in figura 13-7). Questa API permette una comunicazione asincrona tra client fornendo una connessione al provider, e una sessione dove i messaggi possono essere creati e inviati o ricevuti. Questi messaggi possono contenere testo o altri tipi di oggetti

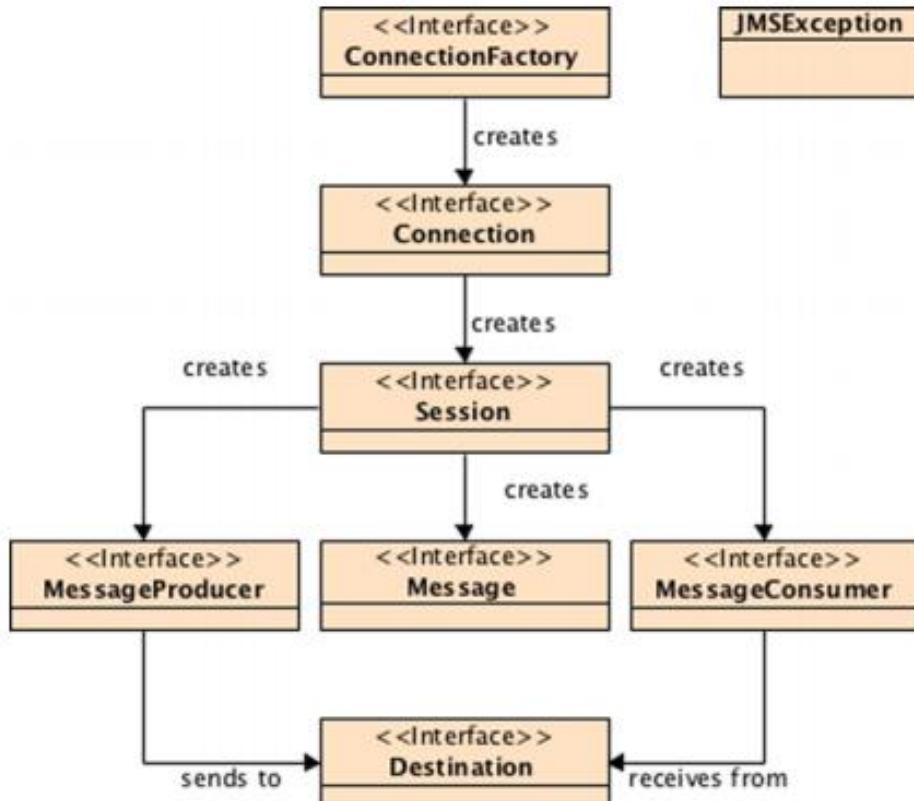


Figure 13-7. JMS Classic API

Connection Factory

Le connection factories sono administered objects che permettono a un'applicazione di connettersi a un provider. `javax.jms.ConnectionFactory` è un'interfaccia che incapsula i parametri di configurazione definiti dall'amministratore. Per usare un administered object il client ha bisogno di fare un JNDI lookup (o usare l'injection). Per esempio, il seguente frammento di codice ottiene l'InitialContext object e lo usa per la look up a `ConnectionFactory` by its JNDI name:

```
Context ctx = new InitialContext();
ConnectionFactory ConnectionFactory = (ConnectionFactory)ctx.lookup("jms/javaee7/ConnectionFactory");
```

I metodi disponibili in questa interfaccia sono metodi `createConnection` che restituiscono un `Connection` object e i nuovi metodi `createContext` di JMS 2.0 che restituiscono un `JMSContext`. È possibile creare una connessione o un `JMSContext` con l'identità user di default o specificando username e password

```

public interface ConnectionFactory {

    Connection createConnection() throws JMSException;
    Connection createConnection(String userName, String password) throws JMSException;
    JMSContext createContext();
    JMSContext createContext(String userName, String password);
    JMSContext createContext(String userName, String password, int sessionMode);
    JMSContext createContext(int sessionMode);
}

```

Destination

La destinazione è un administered object configurato specificamente con informazioni come l'indirizzo di destinazione. Ma questa configurazione è nascosta dai JMS client usando l'interfaccia standard javax.jms.Destination. JNDI lookup è necessario per restituire determinati oggetti:

```

Context ctx = new InitialContext();
Destination queue = (Destination) ctx.lookup("jms/javaee7/Queue");

```

Connection

L'oggetto javax.jms.Connection, creato usando il metodo `createConnection()` di connection factory, incapsula una connessione al JMS provider. Le connessioni sono thread-safe e sviluppate per essere condivisibili. Comunque, una sessione (`javax.jms.Session`), offre un contesto single-thread per inviare e ricevere messaggi, usando una connessione per creare una o più sessioni. Una volta avuto un connection factory, puoi usarlo per creare una connessione nel modo seguente:

```
Connection connection = connectionFactory.createConnection();
```

Prima che un ricevitore possa consumare messaggi, deve chiamare il metodo `start()`, e se si ha bisogno di bloccare temporaneamente la ricezione di emssaggi senza chiudere la connessione, si può invocare il metodo `stop()`.

```

connection.start();
connection.stop();

```

Quando l'applicazione ha terminato, si deve chiudere qualsiasi connessione creata, chiudendo così, anche le sue sessioni e i suoi produttori e consumatori:

```
connection.close();
```

Session

Si crea una sessione da una connessione usando il metodo `createSession()`. Una sessione offre un contesto transazionale nel quale un set di messaggi da inviare o ricevere viene raggruppato in unità atomiche di lavoro, intendendo, che se vengono mandati diversi messaggi durante la stessa sessione, JMS assicurerà che saranno inviati tutti o nessuno. Questo comportamento è impostato alla creazione della sessione:

```
Session session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
```

Il primo parametro da specificare è se la connessione è transazionale o meno. Nel codice il parametro è settato a true se la richiesta di invio non sarà realizzata finché viene chiamato `commit()`, o chiusa la sessione. Se il parametro è settato a false, la sessione non sarà transazionale e i messaggi saranno inviati non appena il metodo `send()` sarà invocato. Il secondo parametro stabilisce se la sessione debba ritornare acknowledgement quando i messaggi vengono ricevuti con successo. Una sessione è single-threaded ed è usata per creare messaggi, produttori e consumatori.

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 encapsulano informazioni e sono divisi in 3 parti:

- *Un 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

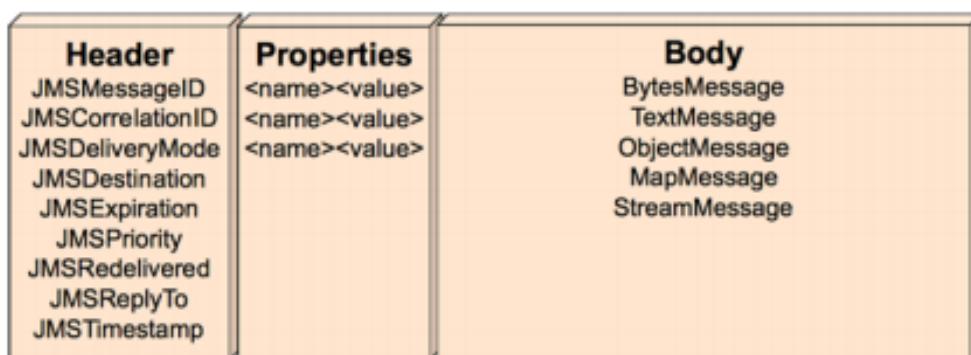


Figure 13-8. Structure of a JMS message

Header

L'header ha coppie di nomi-valori predefinite, comuni a tutti i messaggi ed usate sia dai client che dai provider per instradare correttamente i messaggi. Possono essere visti come metadati siccome forniscono informazioni sul messaggio. Ogni campo ha i metodi getter e

setter associati definiti in javax.jms.Message interface. Alcuni campi dell'header sono fatti per essere settati dal client ma molti vengono impostati dai metodi send() e publish()

Table 13-2. Fields Contained in the Header

Field	Description	Set By
JMSDestination	This indicates the destination to which the message is being sent.	send() or publish() method
JMSDeliveryMode	JMS supports two modes of message delivery. PERSISTENT mode instructs the provider to ensure the message is not lost in transit due to a failure. NON_PERSISTENT mode is the lowest-overhead delivery mode because it does not require the message to be logged to a persistent storage.	send() or publish() method
JMSMessageID	This provides a value that uniquely identifies each message sent by a provider.	send() or publish() method
JMSTimestamp	This contains the time a message was handed off to a provider to be sent.	send() or publish() method
JMSCorrelationID	A client can use this field to link one message with another such as linking a response message with its request message.	Client
JMSReplyTo	This contains the destination where a reply to the message should be sent.	Client
JMSRedelivered	This Boolean value is set by the provider to indicate whether a message has been redelivered.	Provider
JMSType	This serves as a message type identifier.	Client
JMSExpiration	When a message is sent, its expiration time is calculated and set based on the time-to-live value specified on the send() method.	send() or publish() method
JMSPriority	JMS defines a 10-level priority value, with 0 as the lowest priority and 9 as the highest.	send() or publish() method

Properties

Oltre ai campi dell'header la javax.jms.Message interface supporta anche valori proprietà (property values) I quali sono simili a quelli dell'header, ma sono esplicitamente creati dall'applicazione, invece di essere standard tra messaggi. Ciò offre un meccanismo per aggiungere campi opzionali dell'header ai messaggi che il client sceglierà di ricevere, tramite selettori. I property values possono essere booleani, short, int, long, float, double e String. Il codice per impostare ed ottenere proprietà è:

```
message.setFloatProperty("orderAmount", 1245.5f);
message.getFloatProperty("orderAmount");
```

Body

Il corpo del messaggio è opzionale, e contiene i dati da inviare o ricevere. A seconda dell'interfaccia utilizzata, può contenere differenti formati di dati

Table 13-3. Types of Messages

Interface	Description
StreamMessage	A message whose body contains a stream of Java primitive values. It is filled and read sequentially.
MapMessage	A message whose body contains a set of name-value pairs where names are strings and values are Java primitive types.
TextMessage	A message whose body contains a string (for example, it can contain XML).
ObjectMessage	A message that contains a serializable object or a collection of serializable objects.
BytesMessage	A message that contains a stream of bytes.

E' possibile creare il proprio formato se si estenda la javax.jms.Message interface. Da notare che se un messaggio è ricevuto, il suo corpo viene soltanto letto. In base al tipo di messaggio si hanno differenti metodi per accedere al suo contenuto. Un messaggio di testo avrà i metodi getText() e setText(), un messaggio oggetto avrà getObject() e setObject() e così via:

```
textMessage.setText("This is a text message");
textMessage.getText();
bytesMessage.readByte();
objectMessage.getObject();
```

Da notare che da JMS 2.0, il nuovo metodo <T> T getBody(Class<T> c) restituisce il corpo del messaggio come un oggetto di tipo specificato

Io 428-433

Sending and Receiving a Message with Classic API

Ora guardiamo un esempio per aver un'idea di come si usa l'API JMS classica. Il producer manda un messaggio alla destination dove il consumer aspetta che arrivi il messaggio. Le destinazioni possono essere queue o topic. Nell'esempio sotto usiamo una queue.

```

public class Producer {
    public static void main(String[] args) {
        try {
            // Gets the JNDI context
            Context jndiContext = new InitialContext();

            // Looks up the administered objects
            ConnectionFactory connectionFactory = (ConnectionFactory) →
                jndiContext.lookup("jms/javaee7/ConnectionFactory");
            Destination queue = (Destination) jndiContext.lookup("jms/javaee7/Queue");

            // Creates the needed artifacts to connect to the queue
            Connection connection = connectionFactory.createConnection();
            Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            MessageProducer producer = session.createProducer(queue);

            // Sends a text message to the queue
            TextMessage message = session.createTextMessage("Text message sent at " + new Date());
            producer.send(message);

            connection.close();
        } catch (NamingException | JMSException e) {
            e.printStackTrace();
        }
    }
}

```

Il codice rappresenta una classe Producer che ha solo un metodo main(). La prima cosa da fare è istanziare un contesto JNDI utilizzato per ottenere ConnectionFactory e una destinazione. Le factory e le destination sono chiamate administrated objects: devono essere creati e dichiarati nel provider (nel nostro caso, OpenMQ in GlassFish). La classe Producer utilizza una ConnectionFactory per creare una connessione da cui viene ottenuta una sessione. Con questa sessione, un MessageProducer e un messaggio vengono creati nella coda di destinazione (session.createProducer (queue)). Il produttore invia quindi questo messaggio (di tipo testo). Si noti che viene fatto il catch sia di JNDI NamingException che dell'eccezione checked JMSEException. Il codice per ricevere il messaggio sembra quasi lo stesso. Infatti, le prime righe della classe Consumer (vedi sotto)sono esattamente le stesse: creare un contesto JNDI, fare lookup della factory e della destinazione, quindi connettersi. Le uniche differenze sono che viene utilizzato un MessageConsumer al posto di un MessageProducer e che il ricev entra in un ciclo infinito per ascoltare la coda (in seguito vedremo che questo ciclo può essere evitato usando un listener). Quando arriva il messaggio, viene consumato e il contenuto viene visualizzato.

```

public class Consumer {

    public static void main(String[] args) {
        try {
            // Gets the JNDI context
            Context jndiContext = new InitialContext();

            // Looks up the administered objects
            ConnectionFactory connectionFactory = (ConnectionFactory) jndiContext.lookup("jms/javaee7/ConnectionFactory");
            Destination queue = (Destination) jndiContext.lookup("jms/javaee7/Queue");

            // Creates the needed artifacts to connect to the queue
            Connection connection = connectionFactory.createConnection();
            Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            MessageConsumer consumer = session.createConsumer(queue);

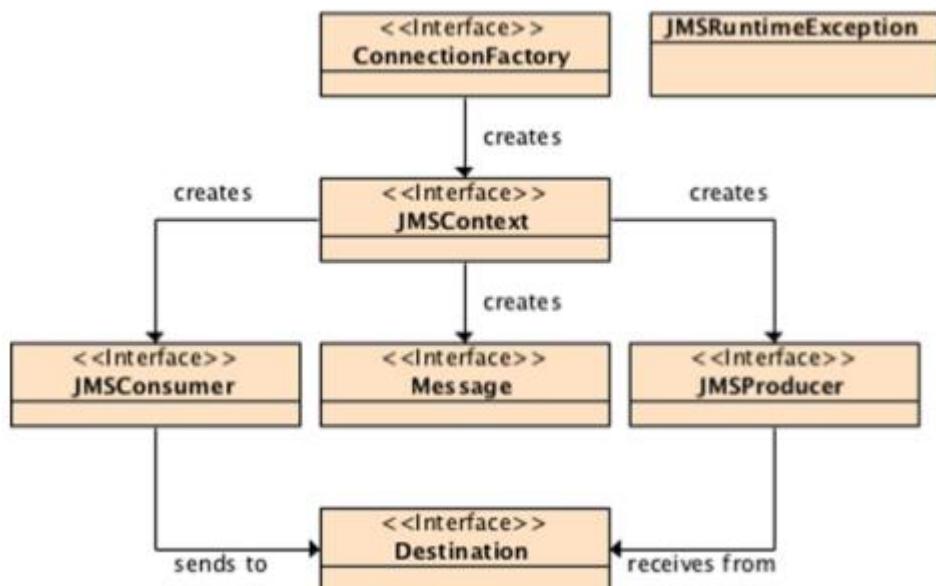
            connection.start();

            // Loops to receive the messages
            while (true) {
                TextMessage message = (TextMessage) consumer.receive();
                System.out.println("Message received: " + message.getText());
            }
        } catch (NamingException | JMSEException e) {
            e.printStackTrace();
        }
    }
}

```

Simplified API

Come si può notare il codice riportato sopra è abbastanza verboso e di basso livello. JMS 2.0 introduce un'API semplificata che consente principalmente di tre nuove interfacce (JMSContext, JMSProducer and JMSConsumer). Grazie alla nuova eccezione JMSRuntimeException, che è un'eccezione non controllata, il codice per inviare o ricevere un messaggio è molto più facile da scrivere e leggere. La figura mostra un class diagram semplificato di questa nuova API. Le API legacy, classiche e semplificate si trovano tutte nel pacchetto javax.jms.



Le tre interfacce principali sono:

- JMSContext: attiva la connessione ad un provider JMS e un contesto a single-thread per l'invio e la ricezione di messaggi
- JMSProducer: oggetto creato da un JMSContext che viene utilizzato per l'invio di messaggi a una Queue o ad un Topic
- JMSConsumer: oggetto creato da un JMSContext che viene utilizzato per ricevere messaggi inviati a una Queue o a un Topic

JMSContext

JMSContext è l'interfaccia principale dell'API JMS 2.0. Combina la funzionalità di due oggetti dell'API classica JMS 1.1: una Connection e una Session.

Un JMSContext può essere creato dall'applicazione chiamando uno dei diversi metodi createContext su ConnectionFactory e quindi chiuso. In alternativa, se l'applicazione è in esecuzione in un contenitore (EJB o Web), JMSContext può essere iniettato utilizzando l'annotazione @Inject.

Quando un'applicazione deve inviare messaggi, utilizza il metodo createProducer per creare un JMSProducer, che fornisce metodi per configurare e inviare messaggi. I messaggi possono essere inviati sia in modo sincrono che asincrono. Per ricevere messaggi, un'applicazione può utilizzare uno dei numerosi metodi createConsumer per creare un JMSConsumer. La tabella mostra un sottoinsieme dell'API JMSContext.

Property	Description
void start()	Starts (or restarts) delivery of incoming messages
void stop()	Temporarily stops the delivery of incoming messages
void close()	Closes the JMSContext
void commit()	Commits all messages done in this transaction and releases any locks currently held
void rollback()	Rolls back any messages done in this transaction and releases any locks currently held
BytesMessage createBytesMessage()	Creates a BytesMessage object
MapMessage createMapMessage()	Creates a MapMessage object
Message createMessage()	Creates a Message object
ObjectMessage createObjectMessage()	Creates an ObjectMessage object
StreamMessage createStreamMessage()	Creates a StreamMessage object
TextMessage createTextMessage()	Creates a TextMessage object
Topic createTopic(String topicName)	Creates a Topic object
Queue createQueue(String queueName)	Creates a Queue object
JMSConsumer createConsumer(Destination destination)	Creates a JMSConsumer for the specified destination
JMSConsumer createConsumer(Destination destination, String messageSelector)	Creates a JMSConsumer for the specified destination, using a message selector
JMSProducer createProducer()	Creates a new JMSProducer object which can be used to configure and send messages
JMSContext createContext(int sessionMode)	Creates a new JMSContext with the specified session mode

JMSProducer

Un JMSProducer viene utilizzato per inviare messaggi. Fornisce vari metodi per inviare un messaggio a una destinazione specificata. Un'istanza di JMSProducer viene creata chiamando il metodo `createProducer` su un `JMSContext`. Fornisce, inoltre, metodi per specificare le opzioni di invio, le proprietà dei messaggi e gli header dei messaggi. La tabella mostra un sottoinsieme dell'API JMSProducer.

Property	Description
<code>get/set[Type]Property</code>	Sets and returns a message property where [Type] is the type of the property and can be Boolean, Byte, Double, Float, Int, Long, Object, Short, String
<code>JMSProducer clearProperties()</code>	Clears any message properties set
<code>Set<String> getPropertyNames()</code>	Returns an unmodifiable Set view of the names of all the message properties that have been set
<code>boolean propertyExists(String name)</code>	Indicates whether a message property with the specified name has been set
<code>get/set[Message Header]</code>	Sets and returns a message header where [Message Header] can be <code>DeliveryDelay</code> , <code>DeliveryMode</code> , <code>JMSCorrelationID</code> , <code>JMSReplyTo</code> , <code>JMSType</code> , <code>Priority</code> , <code>TimeToLive</code>
<code>JMSProducer send(Destination destination, Message message)</code>	Sends a message to the specified destination, using any send options, message properties and message headers that have been defined
<code>JMSProducer send(Destination destination, String body)</code>	Sends a <code>TextMessage</code> with the specified body to the specified destination

JMSConsumer

Un JMSConsumer viene utilizzato per ricevere messaggi da una Queue o un Topic. Viene creato con uno dei metodi `createConsumer` su un `JMSContext` passando una queue o un topic. Come vedremo in seguito, un JMSConsumer può essere creato con un selettore di messaggi in modo da limitare i messaggi consegnati.

Un client può ricevere un messaggio in modo sincrono o asincrono al loro arrivo. Per la consegna asincrona, un client può registrare un oggetto `MessageListener` con un `JMSConsumer`. All'arrivo dei messaggi, il provider li consegna chiamando il metodo `onMessage` di `MessageListener`. La tabella mostra un sottoinsieme dell'API JMSConsumer.

Property	Description
<code>void close()</code>	Closes the <code>JMSConsumer</code>
<code>Message receive()</code>	Receives the next message produced
<code>Message receive(long timeout)</code>	Receives the next message that arrives within the specified timeout interval
<code><T> T receiveBody(Class<T> c)</code>	Receives the next message produced and returns its body as an object of the specified type
<code>Message receiveNoWait()</code>	Receives the next message if one is immediately available
<code>void setMessageListener(MessageListener listener)</code>	Sets the <code>MessageListener</code>
<code>MessageListener getMessageListener()</code>	Gets the <code>MessageListener</code>
<code>String getMessageSelector()</code>	Gets the message selector expression

Writing Message Producers

La nuova API semplificata JMS consente di scrivere i producers e i consumers in modo molto meno dettagliato rispetto all'API classica. Ma necessita ancora di entrambi gli oggetti amministrati: ConnectionFactory e Destination. In base a se vengono eseguiti fuori o dentro un container (EJB, Web o ACC) viene utilizzato l'JNDI lookups o l'injection.

Come visto precedentemente, l'API JMSContext è l'API principale per produrre e consumare messaggio. Se un'applicazione viene eseguita fuori dal container bisognerà gestire il ciclo di vita del JMSContext (creandolo e chiudendolo programmaticamente). Se viene eseguita all'interno del container, esso può essere semplicemente iniettato e lasciare al container la gestione del suo ciclo di vita.

z

Producing a Message outside a Container

Un producer di messaggi (JMSProducer) è un oggetto creato da JMSContext e viene usato per spedire messaggi ad una destinazione. Per creare un producer che invia un messaggio ad una coda al di fuori di un qualsiasi container bisogna:

- Ottenere una connection factory e una coda utilizzando una JNDI lookups;
- Creare un oggetto JMSContext usando il connection factory (da notare la dichiarazione try-with-resources che lo andrà a chiudere automaticamente);
- Creare un javax.jms.JMSProducer usando l'oggetto JSMContext;
- Inviare un messaggio di testo alla coda usando il metodo JMSProducer.send().

Listing 13-4. The Producer Class Produces a Message into a Queue

```
public class Producer {  
  
    public static void main(String[] args) {  
  
        try {  
            // Gets the JNDI context  
            Context jndiContext = new InitialContext();  
            // Looks up the administered objects  
            ConnectionFactory connectionFactory = (ConnectionFactory) ←  
                jndiContext.lookup("jms/javaee7/ConnectionFactory");  
            Destination queue = (Destination) jndiContext.lookup("jms/javaee7/Queue");  
  
            // Sends a text message to the queue  
            try (JMSContext context = connectionFactory.createContext()) {  
                context.createProducer().send(queue, "Text message sent at " + new Date());  
            }  
  
        } catch (NamingException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Producing a Message inside a Container

Le connection factories e le destinazioni sono oggetti amministrati che risiedono in un provider di messaggi e devono essere dichiarati all'interno degli spazi dei nomi di JNDI, motivo per cui viene utilizzata un'API JNDI per cercarli. Quando il codice del client viene

eseguito all'interno di un container, invece, può essere utilizzata una dependency injection. Java EE7 mette a disposizione diversi container: EJB, servlet e application client container (ACC). Se il codice viene eseguito in uno di questi container, l'annotazione `@Resource` può essere utilizzata per iniettare un riferimento a tale risorsa dal container. Con Java EE 7, utilizzare risorse è molto più semplice, in quanto non si ha la complessità di JNDI o non è richiesta la configurazione dei riferimenti alle risorse nei deployment descriptors. Si fa affidamento semplicemente alle capacità di injection del container.

La tabella 13-7 elenca gli attributi di `@Resource`.

Tabella 13-7. API delle `@javax.annotation.Resource Annotation`

Elemento	Descrizione
name	Il nome JNDI della risorsa (il nome è un'implementazione specifica e non è portabile)
type	Il tipo java della risorsa (es. <code>javax.sql.DataSource</code> o <code>javax.jms.Topic</code>)
authenticationType	Il tipo di autenticazione da utilizzare per la risorsa (container o applicazione)
shareable	Se la risorsa può essere condivisa
mappedName	Un nome specifico del prodotto a cui deve essere mappata la risorsa
lookup	Il nome JNDI di una risorsa a cui verrà associata la risorsa in stato di definizione. Può riferirsi a qualsiasi risorsa compatibile utilizzando i nomi portabili JNDI.
description	Descrizione della risorsa

Per utilizzare l'annotazione `@Resource` prendiamo l'esempio del producer in Listing 13-4, modificandolo in un statless session bean e utilizziamo l'injection anziché le JNDI lookups. Nell'esempio precedente, sia la connection factory che la coda erano ricercati tramite la JNDI. Nel Listing 13-5, il name JNDI si trova sull'annotazione `@Resource`. Quando il ProducerEJB viene eseguito in un container, i riferimenti alla ConnectionFactory e alla coda vengono iniettati durante l'inizializzazione.

Listing 13-5. The ProducerEJB Running inside a Container and using `@Resource`.

```

@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());
        }
    }
}

```

Producing a Message inside a Container with CDI

Quando il producer viene eseguiti in un container (EJB o Servlet container) con CDI abilitato, può iniettare il JMSContext. Il container gestirà, quindi, il suo ciclo di vita (non vi è necessità di creare o chiudere il JMSContext). Ciò può essere fatto grazie alle annotazioni @Inject e @JMSConnectionFactory.

L'annotazione javax.jms.JMSConnectionFactory può essere utilizzata per specificare il nome della lookup JNDI della ConnectionFactory utilizzato per creare il JMSContext (vedi Listing 13-6). Se l'annotazione JMSConnectionFactory viene omessa, allora verrà utilizzata una factory connection di JMS predefinita.

Listing 13-6. A Managed Bean Producing a Message using @Inject

```
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());  
    }  
}
```

Nel codice presentato è il container che inietta i componenti necessari e ne gestisce il ciclo di vita. Così sviluppato si necessita di un'unica riga di codice per inviare un messaggio.

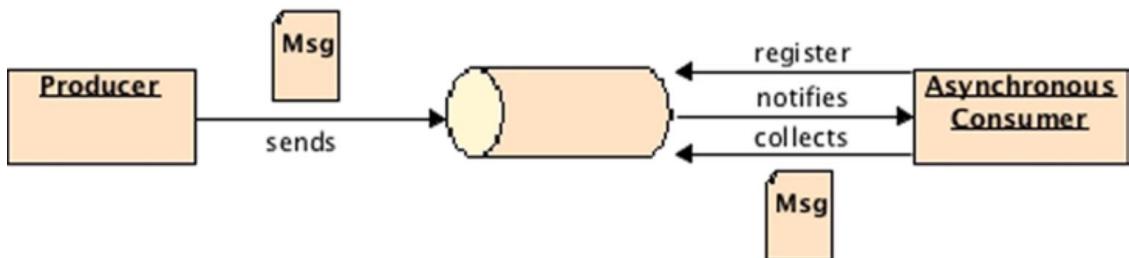
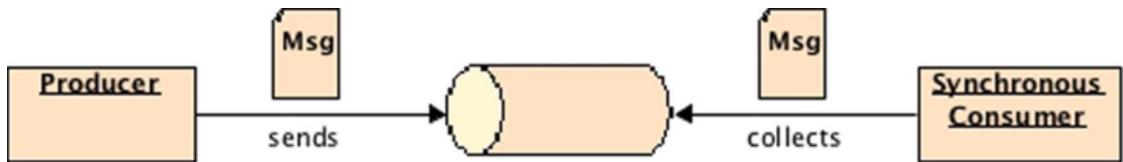
Con l'annotazione javax.jms.JMSPasswordCredential si può specificare un nome utente e password per quando viene creato il JMSContext.

```
@Inject  
@JMSConnectionFactory("jms/connectionFactory")  
@JMSPasswordCredential(userName="admin",password="mypassword")  
private JMSContext context;
```

Writing Message Consumers

Per ricevere messaggi da una destinazione, un client utilizza un JMSConsumer. Esso viene creato passando una coda o un argomento al metodo createConsumer() di JMSContext. La messaggistica è intrinsecamente asincrona, ovvero non vi è dipendenza temporale tra producer e consumer. Tuttavia il client può consumare i messaggi in due modi:

- In modo sincrono: un ricevitore preleva il messaggio in modo esplicito dalla destinazione attraverso il metodo receive();
- In modo asincrono: un destinatario decide di registrarsi ad un evento che viene attivato quando arriva un messaggio. Deve essere implementata l'interfaccia MessageListener e, ogni qualvolta giunge un messaggio, il provider lo consegna utilizzando il metodo onMessage().



Synchronous Delivery

Un consumer sincrono deve avviare un JMSContext, eseguire un loop per attendere fino all'arrivo di un nuovo messaggio e richiedere il messaggio appena arrivato tramite uno dei suoi metodi receive() (vedi Tabella 13-6). Esistono diverse varianti di receive() che consentono ad un client di estrarre o attendere il prossimo messaggio. I seguenti passaggi spiegano come creare un consumer sincrono che preleva messaggi da una coda (vedi Listing 13-7):

- Ottenere una factory di connessione e un argomento attraverso le JNDI lookups (o injection);
- Creare un oggetto JMSContext utilizzando la connecton factory;
- Creare una javax.jms.JMSConsumer utilizzando l'oggetto JMSContext creato;
- Effettuare il loop e la chiamata al metodo receive() (o in questo caso receiveBody) sull'oggetto consumer. I metodi receive() sono bloccati se la coda è vuota e attendono l'arrivo di un messaggio. Qui, il ciclo infinito attende l'arrivo di altri messaggi.

Listing 13-7. The Consumer Class Consumes Messages in a Synchronous Manner

```
public class Consumer {  
  
    public static void main(String[] args) {  
  
        try {  
            // Gets the JNDI context  
            Context jndiContext = new InitialContext();  
  
            // Looks up the administered objects  
            ConnectionFactory connectionFactory = (ConnectionFactory) jndiContext.lookup("jms/javaee7/ConnectionFactory");  
            Destination queue = (Destination) jndiContext.lookup("jms/javaee7/Queue");  
  
            // Loops to receive the messages  
            try (JMSContext context = connectionFactory.createContext()) {  
                while (true) {  
                    String message = context.createConsumer(queue).receiveBody(String.class);  
                }  
            }  
        } catch (NamingException e) {  
            e.printStackTrace();  
        }  
    }  
}  
//437-442
```

Asynchronous 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 onMessage() del listener, che prende come argomento un tipo Message.

Per creare un listener asincrono di messaggi:

- La classe deve implementare l'interfaccia javax.jms.MessageListener, che definisce un unico metodo onMessage()
- Ottieni una connection factory e un topic usando il lookup JNDI (o injection)
- Crea un javax.jms.JMSConsumer usando un oggetto JSMContext
- Chiama il metodo setMessageListener, passando un'istanza dell'interfaccia di MessageListener
- Implementa il metodo onMessage() e processa il messaggio ricevuto. Ogni volta un messaggio arriva, il provider chiamerà questo metodo, passandogli il messaggio.

Listing 13-8. The Consumer Is a Message Listener

```
public class Listener implements MessageListener {  
  
    public static void main(String[] args) {  
  
        try {  
            // Gets the JNDI context  
            Context jndiContext = new InitialContext();  
  
            // Looks up the administered objects  
            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));  
    }  
}
```

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
- Creare iscrizioni durature: Assicurare che i messaggi siano consegnati a un iscritto disponibile in un modello pub-sub
- Definire priorità

Filtraggio messaggi

Quando si invia un messaggio in broadcast, è utile creare dei meccanismi per permettere solo ad alcuni di ricevere il messaggio. I messaggi sono composti da tre parti: header, proprietà e body. L'header contiene un determinato numero di campi, le proprietà invece sono set di coppie nome-valore che l'applicazione può usare per assegnare qualsiasi valore. La selezione può esser fatta su queste due aree. Il producers set una o più valori nelle proprietà o campi nell'header, e il consumer specifica i criteri di selezione dei messaggi usando le selector expression. Solo i messaggi che combaciano con i selector sono consegnati.

Un message selector è una stringa che contiene una expression, la cui sintassi è:

```

context.createConsumer(queue, "JMSPriority < 6").receive();
context.createConsumer(queue, "JMSPriority < 6 AND orderAmount < 200").receive();
context.createConsumer(queue, "orderAmount BETWEEN 1000 and 2000").receive();

```

Con queste righe di codice si creano i consumer e si passa una selector string che può contenere un campo dell'header o una proprietà. Il producer setta queste proprietà in questo modo:

```

context.createTextMessage().setIntProperty("orderAmount", 1530);
context.createTextMessage().setJMSPriority(5);

```

Time-to-live

Il provider eliminerà i messaggi quando diventano obsoleti

```
context.createProducer().setTimeToLive(1000).send(queue, message);
```

Persistence

JMS supporta due modalità di spedizione messaggi: persistenti e non persistenti. La spedizione persistente assicura che un messaggio è spedito una sola volta ad un consumer, mentre non persistente richiede che un messaggio sia spedito una sola volta al massimo. La prima è più affidabile, in quanto previene la perdita dei messaggi quando avviene un failure del provider. Il modo di consegna può esser specificato usando il setDeliveryMode() dell'interfaccia JMSProducer:

```
context.createProducer().setDeliveryMode(DeliveryMode.NON_PERSISTENT).send(queue, message);
```

Controllare Acknowledgment

Quando un messaggio è stato ricevuto, alcune volte si vuole ottenerne l'acknowledgment. Una fase di acknowledgment può essere iniziata sia dal provider JMS o dal client, a seconda della modalità di acknowledgment.

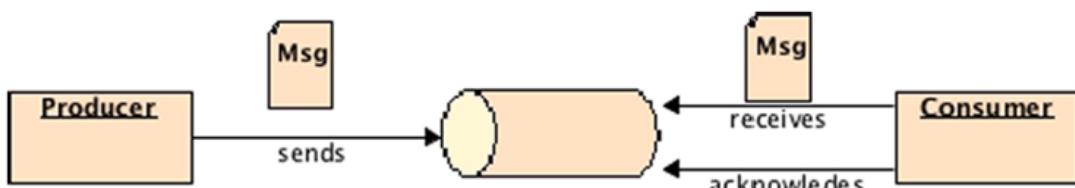


Figure 13-11. A consumer acknowledging a message

In una sessione transactional, l'acknowledgment avviene automaticamente quando si fa il commit di una transaction. Se una transaction è annullata, tutti i messaggi sono riconsegnati. Ma in una sessione nontransactional, la modalità di acknowledgment deve essere specificata:

- AUTO_ACKNOWLEDGE: La sessione fa automaticamente l'acknowledgment della ricezione del messaggio
- CLIENT_ACKNOWLEDGE: Un client fa l'acknowledge del messaggio esplicitamente chiamando il metodo Message.acknowledge()

- DUPS_OK_ACKNOWLEDGE: Questa opzione dice alla sessione di fare il lazy acknowledge dei messaggi. Quindi se il JMS Provider fallisce ci possono essere duplicati nella consegna dei messaggi. Se il messaggio viene riconsegnato il provider detta il valore dell'header JMSRedelivered a true

Nel codice seguente è usato l'annotazione @JMSSessionMode per definire la modalità di acknowledgment:

```
// Producer
@Inject
@JMSConnectionFactory("jms/connectionFactory")
@JMSSessionMode(JMSContext.AUTO_ACKNOWLEDGE)
private JMSContext context;
...
context.createProducer().send(queue, message);

// Consumer
message.acknowledge();
```

Creating Durable Consumers

Usando consumers durevoli, le API JMS definiscono un modo per tenere i messaggi in topic fin quando ogni consumer iscritto non li riceve. Il consumer può quindi essere offline e ricevere il messaggio quando si riconnette. Si usa JMSContext:

```
context.createDurableConsumer(topic, "javaee7DurableSubscription").receive();
```

Ogni consumer durevole deve avere un id univoco.

Priorità

Si possono definire dei livelli di priorità per poter spedire prima i messaggi urgenti, con i valori da 0 a 9 usando il setPriority() di JMSProducer

```
context.createProducer().setPriority(2).send(queue, message);
```

Scrivere un Message-Driven bean

Un MDB è un consumer asincrono invocato dal container all'arrivo di un messaggio. Sono simili agli stateless session bean e hanno accesso alle risorse gestite dal container. Vengono usati al posto dei JMS clients grazie al container, che gestisce il multithreading, la sicurezza e le transaction, semplificando il codice. Inoltre gestisce l'arrivo di vari messaggi tra multipli MDB.

Listing 13-9. A Simple MDB

```
@MessageDriven(mappedName = "jms/javaee7/Topic")
public class BillingMDB implements MessageListener {

    public void onMessage(Message message) {
        System.out.println("Message received: " + message.getBody(String.class));
    }
}
```

I client quindi non possono invocare metodi direttamente su i MDB, comunque, quest'ultimo hanno un ricco modello di programmazione, che include il life-cycle, le

annotazioni di callback, interceptor, injection e transaction, ma, non facendo parte dell'EJB lite model, necessitano comunque dell'intero stack Java EE.

I requisiti per sviluppare un MDB sono:

- La classe deve avere l'annotazione @javax.ejb.MessageDriven o l'equivalente in XML in un deployment descriptor
- La classe deve implementare l'interfaccia MessageListener
- Deve essere pubblica, non final né abstract
- La classe deve avere un costruttore pubblico senza argomenti che il container userà per creare le istanze dell'MDB
- La classe non deve definire il metodo finalize()

443-447 Catello

@ActivationConfigProperty

JMS consente la configurazione di alcune proprietà come selettori di messaggi, modalità di riconoscimento, durable subscribers e così via. In un MDB, queste proprietà possono essere impostate utilizzando l'annotazione @ActivationConfigProperty. Questa annotazione facoltativa può essere fornita come uno dei parametri per l'annotazione @MessageDriven e, rispetto all'equivalente JMS, @ActivationConfigProperty è molto semplice, costituito da una coppia nome-valore (vedere il Listato 13-11).

```
@Target({}) @Retention(RUNTIME)
public @interface ActivationConfigProperty {
    String propertyName();
    String propertyValue();
}
```

La proprietà activationConfig consente di fornire una configurazione standard e non standard (specifica del fornitore). Il codice nell'immagine che segue imposta la modalità di conferma e il selettore di messaggi.

Listing 13-12. Setting Properties on MDBs

```
@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));
    }
}
```

Ogni proprietà di attivazione è una coppia nome-valore che il provider di messaggistica sottostante comprende e utilizza per impostare l'MDB. La Tabella 13-8 elenca alcune proprietà standard che è possibile utilizzare.

Table 13-8. Activation Properties for OpenMQ

Property	Description
acknowledgeMode	The acknowledgment mode (default is AUTO_ACKNOWLEDGE)
messageSelector	The message selector string used by the MDB
destinationType	The destination type, which can be TOPIC or QUEUE
destinationLookup	The lookup name of an administratively-defined Queue or Topic
connectionFactoryLookup	The lookup name of an administratively defined ConnectionFactory
destination	The name of the destination.
subscriptionDurability	The subscription durability (default is NON_DURABLE)
subscriptionName	The subscription name of the consumer
shareSubscriptions	Used if the message-driven bean is deployed into a clustered
clientId	Client identifier that will be used when connecting to the JMS provider

Dependencies Injection

Come tutti gli altri bean EJB che hai visto nel Capitolo 7, gli MDB possono utilizzare dependency injection per acquisire riferimenti a risorse come dati JDBC, EJB o altri oggetti. L'iniezione è il mezzo con cui il container inserisce automaticamente le dipendenze dopo aver creato l'oggetto. Queste risorse devono essere disponibili nel conteiner oppure nell'environment context, quindi il codice seguente è consentito in un MDB:

```
@PersistenceContext  
private EntityManager em;  
@Inject  
private InvoiceBean invoice;  
@Resource(lookup = "jms/javaee7/ConnectionFactory")  
private ConnectionFactory connectionFactory;
```

L' MDB context può anche essere iniettato usando l'annotazione @Resource:

```
@Resource private MessageDrivenContext context;
```

MDB Context

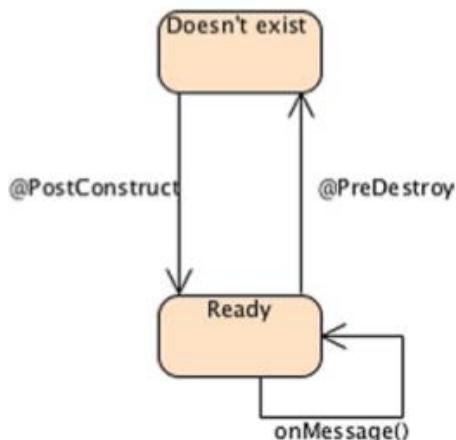
L'interfaccia MessageDrivenContext fornisce l'accesso al context di runtime che il conteiner fornisce per un'istanza MDB. Il conteiner passa l'interfaccia MessageDrivenContext a questa istanza, che rimane associata per tutta la durata del MDB. Ciò fornisce all'MDB la possibilità di eseguire il rollback esplicito di una transazione, ottenere l'utente principale e così via. L'interfaccia MessageDrivenContext estende l'interfaccia javax.ejb.EJBContext senza aggiungere ulteriori metodi.

Se MDB inietta un riferimento al suo contesto, sarà in grado di richiamare i metodi elencato nella tabella

Method	Description
getCallerPrincipal	Returns the <code>java.security.Principal</code> associated with the invocation
getRollbackOnly	Tests whether the current transaction has been marked for rollback
getTimerService	Returns the <code>javax.ejb.TimerService</code> interface
getUserTransaction	Returns the <code>javax.transaction.UserTransaction</code> interface to use to demarcate transactions. Only MDBs with bean-managed transaction (BMT) can use this method
isCallerInRole	Tests whether the caller has a given security role
Lookup	Enables the MDB to look up its environment entries in the JNDI naming context
setRollbackOnly	Allows the instance to mark the current transaction as rollback. Only MDBs with BMT can use this method

Life Cycle and Callback Annotations

Il ciclo di vita di un MDB (vedere la Figura 13-12) è identico a quello del bean di sessione stateless: l'MDB esiste ed è pronto a consumare messaggi oppure non esiste. Prima di esistere, il container crea un'istanza di MDB e, se applicabile, inietta le risorse necessarie come specificato dalle annotazioni sui metadati (`@Resource`, `@Inject`, `@EJB`, ecc.) o dal deployment descriptor. Il container chiama quindi il metodo di callback `@PostConstruct` del bean, se presente. Dopo questo, MDB è pronto e attende di consumare qualsiasi messaggio in arrivo. Il callback `@PreDestroy` si verifica quando l'MDB viene rimosso dal pool o distrutto.



Come spiegato nella sezione "Writing Message Consumers" in questo capitolo, i consumatori possono ricevere un messaggio in modo sincrono, eseguendo il ciclo e attendendo che arrivi un messaggio o in modo asincrono, implementando l'interfaccia `MessageListener`. Per natura, gli MDB sono progettati per funzionare come consumatori di messaggi asincroni. Gli MDB implementano un'interfaccia listener di messaggi, che viene attivata dal container quando arriva un messaggio. Un MDB può essere un consumatore sincrono? Sì, ma questo non è raccomandato. I consumatori di messaggi sincroni bloccano e vincolano le risorse del server (gli EJB rimarranno in loop senza eseguire alcun lavoro e il

conteiner non sarà in grado di liberarli). Gli MDB, come i bean di sessione stateless, vivono in un pool di una certa dimensione. Quando il conteiner ha bisogno di un'istanza, ne prende uno dal pool e lo usa. Se ogni istanza entra in un ciclo infinito, il pool finirà per svuotarsi e tutte le istanze disponibili saranno in loop. Il conteiner EJB può anche iniziare a generare più istanze MDB, far crescere il pool e consumare sempre più memoria. Per questo motivo, i bean di sessione e gli MDB non dovrebbero essere usati come consumatori di messaggi sincroni. La Tabella 13-10 mostra le diverse modalità di ricezione per MDB e bean di sessione.

Enterprise Beans	Producer	Synchronous Consumer	Asynchronous Consumer
Session beans	Yes	Not recommended	Not possible
MDB	Yes	Not recommended	Yes

MDB as a Producer

Gli MDB sono in grado di diventare produttori di messaggi, cosa che spesso accade quando sono coinvolti in un flusso di lavoro, poiché ricevono un messaggio da una destinazione, lo elaborano e lo inviano a un'altra destinazione. Per aggiungere questa funzionalità, è necessario utilizzare l'API JMS.

Una destinazione e una factory connection possono essere iniettate utilizzando le annotazioni `@Resource` e `@JMSConnectionFactory` o tramite la ricerca JNDI, quindi i metodi sull'oggetto `javax.jms.JMSContext` possono essere richiamati per creare e inviare un messaggio.

```

@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");
    }
}

```

Questo MDB utilizza la maggior parte dei concetti presentati finora. Innanzitutto, utilizza l'annotazione `@MessageDriven` per definire il nome JNDI del topic che sta ascoltando (`mappedName = "jms / javaee7 / topic"`). In questa stessa annotazione, definisce un insieme di proprietà, come la modalità di conferma e un selettore di messaggi utilizzando una

matrice di annotazioni `@ActivationConfigProperty` e implementa `MessageListener` e il suo metodo `onMessage()`.

Questo MDB deve anche produrre un messaggio. Pertanto, viene iniettato con i due oggetti amministrati richiesti: una factory di connessione (utilizzando `JMSContext`) e una destinazione (la coda denominata `jms / javaee7 / Queue`). Infine, il metodo di business che invia i messaggi (il metodo `sendPrintingMessage()`) assomiglia a quello che hai visto prima: un `JMSProducer` viene creato e utilizzato per creare e inviare un messaggio di testo. Per una migliore leggibilità, la gestione delle eccezioni è stata omessa nell'intera classe.

Transactions

Gli MDB possono utilizzare BMT o transazioni gestite dal container (CMT); possono eseguire il rollback esplicito di una transazione usando il metodo `MessageDrivenContext.setRollbackOnly()` e così via. Tuttavia, ci sono alcune specifiche riguardanti MDB che vale la pena di spiegare.

Quando parliamo di transazioni, pensiamo sempre ai database relazionali. Tuttavia, altre risorse sono anche transazionali, come i sistemi di messaggistica. Se due o più operazioni devono avere esito positivo o negativo, formano una transazione. Con la messaggistica, se vengono inviati due o più messaggi, devono avere successo (commit) o fallire (rollback) insieme. Come funziona in pratica? La risposta è che i messaggi non vengono rilasciati ai consumatori fino a quando la transazione non viene eseguita.

Il container avvierà una transazione prima che il metodo `onMessage()` venga richiamato e farà il commit della transazione quando il metodo restituirà un valore (a meno che la transazione non sia stata contrassegnata per il rollback con `setRollbackOnly()`).

Anche se gli MDB sono transazionali, non possono essere eseguiti nel contesto della transazione del client, in quanto non hanno un client. Nessuno invoca esplicitamente metodi su MDB, semplicemente ascoltano una destinazione e consumano messaggi.

Non è stato passato alcun contesto da un client a un MDB e, analogamente, il contesto di transaction del client non può essere passato al metodo `onMessage()`. La Tabella 13-11 confronta i CMT con i bean di sessione e gli MDB.

Transaction Attribute	Session Beans	MDB
NOT_SUPPORTED	Yes	Yes
REQUIRED	Yes	Yes
MANDATORY	Yes	No
REQUIRES_NEW	Yes	No
SUPPORTS	Yes	No
NEVER	Yes	No

Nei frammenti di codice di questo capitolo, la gestione delle eccezioni è stata omessa, poiché l'API JMS può essere verbose (a dire di sara, significa che devi scrivere tanto) nell'affrontare le eccezioni. La classica API definisce 12 diverse eccezioni, tutte ereditate da `javax.jms.JMSEException`. L'API semplificata definisce 10 eccezioni di runtime tutte ereditate da `javax.jms.JMSRuntimeException`.

È importante notare che `JMSEException` è un'eccezione controllata e `JMSRuntimeException` è non controllata. La specifica EJB delinea due tipi di eccezioni:

- Eccezioni dell'applicazione: eccezioni controllate che estendono eccezioni e non provocano il rollback del container
- Eccezioni di sistema: eccezioni non verificate che estendono `RuntimeException` e causano il rollback del container

Lanciare una JMSRuntimeException farà sì che il container esegua il rollback, ma il lancio di una JMSEException non lo farà. Se è necessario un rollback, il setRollBackOnly() deve essere chiamato esplicitamente.

```
public void onMessage(Message message) {  
    try {  
        System.out.println("Message received: " + message.getBody(String.class));  
    } catch (JMSEException e) {  
        context.setRollBackOnly();  
    }  
}
```

Web Services

Il termine web service indica "qualcosa" accessibile sul "web" che ti dà un "servizio". Il primo esempio che ci viene in mente è una pagina HTML: è accessibile online e, una volta letta, ti dà le informazioni che stavi cercando. Un altro tipo di Web service sono le Servlets. Sono legati a un URL, quindi accessibili sul Web, e eseguono qualsiasi tipo di elaborazione. Il termine "servizi web" divenne rapidamente una parola chiave, assimilato a Service Oriented Architecture (SOA) e oggi fanno parte della nostra vita architetturale quotidiana. Le applicazioni dei Web service 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 "debolmente accoppiati" perché il client, ovvero il consumatore, di un servizio web non deve conoscere i dettagli dell'implementazione (come il linguaggio usato per svilupparlo, il metodo firma o la piattaforma su cui gira). Il consumatore è in grado di invocare un servizio Web SOAP utilizzando un'interfaccia intuitiva che descrive i metodi business 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 saranno ancora in grado di scambiare dati in un modo debolmente 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. Il protocollo di rete predefinito è HTTP, un protocollo stateless noto e robusto. I servizi Web SOAP sono ovunque. Possono essere richiamati da un semplice desktop o utilizzati per l'integrazione business-to-business (B2B) in modo che le operazioni che in precedenza richiedevano l'intervento manuale venissero eseguite automaticamente. I SAOP Web Service integrano le applicazioni gestite da varie organizzazioni tramite Internet o all'interno della stessa azienda (che è nota come Enterprise Application Integration o EAI). In tutti i casi, forniscono un modo standard per collegare diversi software.

Understanding SOAP Web Services

In poche parole, 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 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). Nella Figura è possibile vedere un'immagine di alto livello dell'interazione di un Web Service SOAP. 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 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.

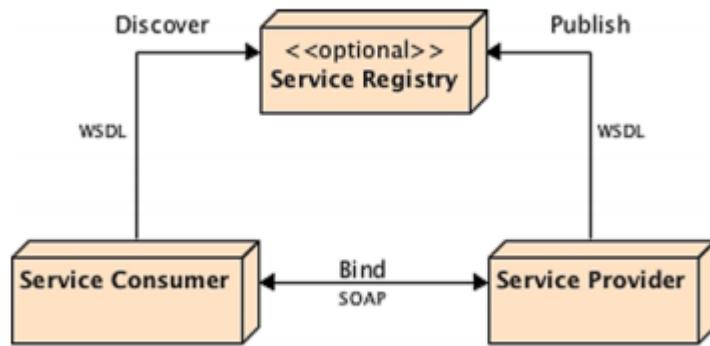


Figure 14-1. The consumer discovers the service through a registry

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. Quelli che incontrerai più spesso sono i seguenti:

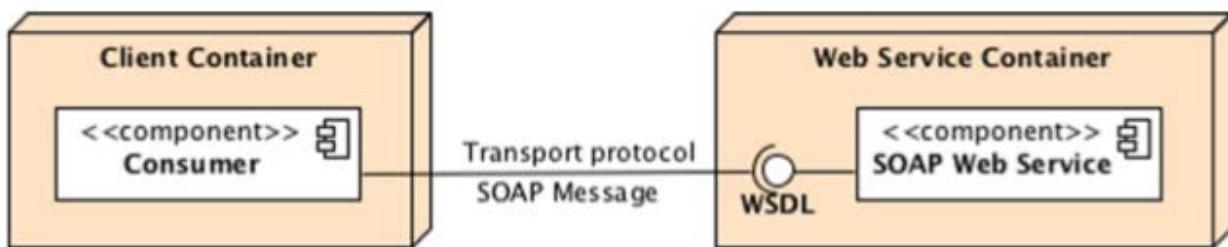
- L'XML (Extensible Markup Language) è la base su cui vengono creati e definiti i SOAP web services (SOAP, WSDL e UDDI).
- WSDL (Web Services Description Language) definisce il protocollo, l'interfaccia, i tipi di messaggi e le interazioni tra il consumatore e il fornitore.
- Il protocollo SOAP (Simple Object Access Protocol) è un protocollo di codifica dei messaggi basato su tecnologie XML, che definisce una busta (envelope) per la comunicazione di servizi Web.
- I messaggi vengono scambiati utilizzando un protocollo di trasporto. Sebbene Hypertext Transfer Protocol (HTTP) sia il protocollo di trasporto più diffuso, è possibile utilizzarne altri come SMTP o JMS.
- Universal Description Discovery, 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). 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 dalla base idea di "RPC (Remote Procedure Call) usando XML."

WSDL

WSDL è il linguaggio di definizione dell'interfaccia (IDL) che definisce le interazioni tra consumatori e SOAP web services(vedere la Figura). È fondamentale 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 per consentire a un consumatore e a un produttore di condividere e comprendere un messaggio. Questo è il ruolo di WSDL. Il Listato 14-1 mostra un esempio di WSDL che rappresenta un SOAP web service 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 trasporto protocollo (soap:binding transport).

(questo te lo metto a titolo di esempio ma è una di quelle cose che la guardi e fai hahahaha e passi avanti)

```

<?xml version="1.0" encoding="UTF-8" ?>
<definitions >
    xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata" -->
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" -->
    xmlns:tns="http://chapter14.javaee7.book.agoncal.org/" -->
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" -->
    xmlns="http://schemas.xmlsoap.org/wsdl/" -->
    targetNamespace="http://chapter14.javaee7.book.agoncal.org/" -->
    name="CardValidatorService">
<types>
    <xsd:schema>
        <xsd:import namespace="http://chapter14.javaee7.book.agoncal.org/" -->
            schemaLocation="http://localhost:8080/chapter14/CardValidatorService?xsd=1"/>
    </xsd:schema>
</types>
<message name="validate">
    <part name="parameters" element="tns:validate"/>
</message>
<message name="validateResponse">
    <part name="parameters" element="tns:validateResponse"/>
</message>
<portType name="CardValidator">
    <operation name="validate">
        <input message="tns:validate"/>
        <output message="tns:validateResponse"/>
    </operation>
</portType>
<binding name="CardValidatorPortBinding" type="tns:CardValidator">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="validate">
        <soap:operation soapAction="" />
        <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>

```

(si proprio una pagina inutile)

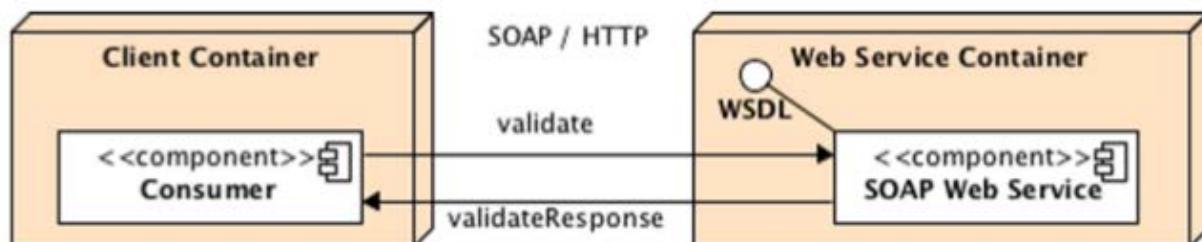
La tabella mostra un sottoinsieme degli elementi di wsdl

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

(lo schema del file wsdl te lo risparmio)

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 . SOAP è il protocollo standard per i Web services. Fornisce il meccanismo di comunicazione per connettere servizi Web, scambiando dati XML 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 envelope, un'header, un body, 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 (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.

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>
```

Il consumatore invia tutti i dati della carta di credito all'interno di una SOAP envelope (Listato 14-3) al metodo di convalida della carta di credito. Il servizio restituisce un altro envelope SOAP (Listato 14-4) con il risultato della convalida (vero o falso). La Tabella 14-2 elenca un sottoinsieme degli elementi e degli attributi SOAP. Come WSDL, SOAP è definito dal corpo standard del W3C.

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 Web service 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, 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 web services. Queste

informazioni forniscono ulteriori metadati sul servizio, descrivendo il suo comportamento e la posizione effettiva del documento WSDL.

Protocollo di trasporto

Per consentire a un utente di comunicare con un Web service, è necessario un modo per inviare messaggi. I messaggi SOAP possono essere trasportati su una rete utilizzando un protocollo supportato da entrambe le parti. Dato che i servizi Web vengono utilizzati principalmente sul Web, in genere utilizzano HTTP, ma possono anche utilizzare altri protocolli di rete come HTTPS (HTTP Secure), TCP / IP, SMTP (Simple Mail Transport Protocol), FTP (File Transfer Protocol) , e così via.

SOAP Web Services Specifications Overview

(io questo di solito lo salto)

Writing SOAP Web Services

Finora hai visto molti concetti di basso livello come protocollo HTTP, documenti WSDL, messaggi SOAP o XML. Ma come si scrive un servizio web SOAP con tutte le specifiche che sono state presentate in precedenza? Puoi o iniziare dal WSDL o passare direttamente alla codifica di Java. 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 contract first . Questo approccio inizia con il contratto (WSDL) definendo operazioni, messaggi e così via. Quando sia il consumatore che il fornitore si accordano sul contratto, è 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.

Con un modello di sviluppo semplice e alcune annotazioni, è possibile modificare la mappatura da Java a WSDL. Ma attenzione, l'approccio bottom-up, può comportare applicazioni molto inefficienti, poiché i metodi e le classi Java non hanno alcuna relazione con l'ideale granularità dei messaggi che attraversano la rete. Se la latenza è elevata e/o la larghezza di banda bassa, si vorrebbe utilizzare i messaggi meno numerosi e questo può essere fatto in modo più efficiente utilizzando l'approccio contract first. Scrivere e consumare un servizio web nell'approccio dbottom-up è molto semplice. I Web services SOAP seguono il paradigma “ease of development” di Java EE 7 e non richiedono la scrittura di alcun WSDL o SOAP ma solo di un POJO annotato. Il Listato 14-5 mostra il codice di un servizio web che convalida una carta di credito.

Listing 14-5. The CardValidator Web Service

```
@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;
        }
    }
}
```

Come entità o EJB, un Web service SOAP utilizza il modello POJO annotato con il criterio di configuration-by-exception. Ciò significa che un Web service può essere solo una classe Java annotata con `@ javax.jws.WebService`. Questo servizio Web SOAP CardValidator prende una carta di credito come parametro e restituisce vero o falso a seconda che la carta sia valida o meno. In questo caso, si presuppone che le carte di credito con un numero pari siano valide e quelle con un numero dispari non lo siano. Un oggetto CreditCard (vedere il Listato 14-6) viene scambiato tra il consumatore e il web service.

```
@XmlRootElement
public class CreditCard {

    @XmlAttribute(required = true)
    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
}
```

Anatomy of a SOAP Web Service

Come la maggior parte dei componenti Java EE 7, i servizi Web SOAP si basano 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 `@ javax.jws.WebService` o l'equivalente XML in un deployment descriptor (webservices.xml).
- 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 @ javax.ejb.Stateless o @ javax.ejb.Singleton .
- Un servizio deve essere un oggetto stateless e non deve salvare lo stato specifico del client attraverso le chiamate di metodo.

La specifica WS-Metadata (JSR 181) afferma che, purché soddisfi questi requisiti, è possibile utilizzare un POJO per implementare un servizio Web distribuito nel container servlet. Questo è comunemente definito come servlet endpoint.

Un bean di sessione stateless o singleton può anche essere utilizzato per implementare un servizio che verrà distribuito in un contenitore EJB (a.k.a. un endpoint EJB).

SOAP Web Service Endpoints

JAX-WS consente sia le normali classi Java che gli EJB di essere esposti come servizi web. Chiamiamo interfacce service endpoint (SEI). Il confronto tra il codice di un POJO e quello di un servizio web EJB non rivela quasi nessuna differenza, con l'eccezione che il servizio web EJB ha l'annotazione extra @Stateless (o @Singleton).

```
@WebService
@Stateless
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;
        }
    }
}
```

Entrambi gli endpoint hanno un comportamento quasi identico, ma alcuni vantaggi extra derivano dall'utilizzo degli endpoint EJB.

Poiché il Web service è anche un EJB, i vantaggi della transazione e della sicurezza gestiti dal container sono automatici e possono essere utilizzati gli interceptors, il che non è possibile con gli endpoint servlet. Il codice aziendale può essere esposto contemporaneamente come servizio Web e EJB, il che significa che la logica aziendale può

essere esposta tramite SOAP e anche tramite RMI aggiungendo un'interfaccia remota.

WSDL mapping

A livello di servizio, i sistemi sono definiti in termini di messaggi XML, operazioni WSDL e messaggi SOAP. A livello di Java, le applicazioni sono definite in termini di oggetti, interfacce e metodi. È necessaria una traduzione dagli oggetti Java alle operazioni WSDL. Il runtime JAXB utilizza annotazioni per determinare come eseguire il marshall / unmarshal di una classe in / da XML. Allo stesso modo, JWS utilizza annotazioni per mappare le classi Java in WSDL e determinare come eseguire il marshalling di una chiamata di metodo a una richiesta SOAP e annullare una risposta SOAR in un'istanza del tipo di ritorno del metodo. Le specifiche JAX-WS e WS-Metadata definiscono due diversi tipi di annotazioni:

- Annotazioni di mapping WSDL: queste annotazioni consentono di modificare il mapping WSDL/Java. `@WebMethod`, `@WebResult`, `@WebParam` e `@OneWay` le annotazioni vengono utilizzate sul servizio Web per personalizzare la firma dei metodi esposti.
- Annotazioni di binding SOAP: queste annotazioni consentono la personalizzazione del binding SOAP (`@SOAPBinding` e `@SOAPMessageHandler`).

@WebService

L'annotazione `@ javax.jws.WebService` contrassegna una classe o interfaccia Java come un servizio Web. Se usato direttamente sulla classe il container genererà l'interfaccia, quindi i seguenti snippet di codice sono equivalenti. Ecco l'annotazione sulla classe:

```
@WebService
public class CardValidator {...}
```

E il seguente snippet mostra l'annotazione su un'interfaccia implementata da una classe. Come potete vedere, l'implementazione deve definire il nome completo dell'interfaccia nell'attributo `endpointInterface`:

```
@WebService
public interface Validator {...}

@WebService(endpointInterface = "org.agoncal.book.javaee7.chapter14.Validator")
public class CardValidator implements Validator {...}
```

L'annotazione `@WebService` ha una serie di attributi che ti permettono di personalizzare il nome del servizio web nel file WSDL (l'elemento `<wsdl: portType>` o `<wsdl: service>`) nonché modificare la posizione del WSDL stesso (l'attributo `wsdlLocation`).

```

@Retention(RUNTIME) @Target(TYPE)
public @interface WebService {
    String name() default "";
    String targetNamespace() default "";
    String serviceName() default "";
    String portName() default "";
    String wsdlLocation() default "";
    String endpointInterface() default "";
}

@WebService(portName = "CreditCardValidator", serviceName = "ValidatorService")
public class CardValidator {...}

```

If you compare with the default WSDL described in Listing 14-1 you'll see the following changes:

```

<service name="ValidatorService">
    <port name="CreditCardValidator" binding="tns:CreditCardValidatorBinding">
        <soap:address location="http://localhost:8080/chapter14/ValidatorService"/>
    </port>
</service>

```

@WebMethod

Per impostazione predefinita, tutti i metodi pubblici di un servizio Web SOAP vengono esposti nel WSDL e utilizzano tutte le regole di mapping predefinite. Per personalizzare alcuni elementi puoi usare l'annotazione `@ javax.jws.WebMethod` sui metodi. L'API di questa annotazione è abbastanza semplice in quanto consente di rinominare un metodo o di escluderlo dal WSDL. Il codice mostra come il servizio Web CardValidator rinomina i primi due metodi ed esclude l'ultimo.

```

@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
    }
}

```

Come puoi vedere nel frammento WSDL, solo i due metodi ValidateCreditCard e ValidateCreditCardNumber sono definiti. L'ultimo metodo è stato escluso dal WSDL.

```

<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>
    <operation name="ValidateCreditCardNumber">
        <input message="tns:ValidateCreditCardNumber"/>
        <output message="tns:ValidateCreditCardNumberResponse" />
    </operation>
</portType>

```

@WebResult

L'annotazione @ javax.jws.WebResult controlla il nome generato del valore restituito dal messaggio nel WSDL. Nel codice sotto il risultato restituito del metodo validate () viene rinominato in IsValid.

```

@WebService
public class CardValidator {

    @WebResult(name = "IsValid")
    public boolean validate(CreditCard creditCard) {
        // Business logic
    }
}

```

Per impostazione predefinita, il nome del valore restituito nel WSDL è impostato a return. Ma con l'annotazione @WebResult puoi essere più specifico e avere un contratto più espressivo:

```

<!-- Default -->
<xss:element name="return" type="xs:boolean"/>
<!-- Renamed to IsValid -->
<xss:element name="IsValid" type="xs:boolean"/>

```

@WebParam

L'annotazione @ javax.jws.WebParam è simile a @WebResult in quanto personalizza i parametri per i metodi del servizio web. La sua API consente di cambiare il nome del parametro nel WSDL lo spazio dei nomi e il tipo. I tipi validi sono IN, OUT o INOUT (entrambi), che determinano il modo in cui il parametro scorre (l'impostazione predefinita è IN).

```

@WebService
public class CardValidator {

    public boolean validate(@WebParam(name= "Credit-Card", mode = IN) CreditCard creditCard) {
        // Business logic
    }
}

```

Di nuovo, se si confronta questo con l'XSD predefinito definito nel Listato 14-2 noterete che, di default, il nome del parametro è arg0. L'annotazione @WebParam sovrascrive questo valore predefinito con il nome Carta di credito:

```

<!-- Default -->
<xss:element name="arg0" type="tns:creditCard" minOccurs="0"/>
<!-- Renamed to Credit-Card -->
<xss:element name="Credit-Card" type="tns:creditCard" minOccurs="0"/>

```

@OneWay

L'annotazione @OneWay può essere utilizzata su metodi che non hanno un valore di ritorno come i metodi che restituiscono void. Questa annotazione non ha elementi e può essere vista come un'interfaccia di markup che informa il contenitore su tale chiamata si può ottimizzare, in quanto non vi è alcun ritorno.

@SOAPBinding

Un binding descrive come il web service è associato a un protocollo di messaggistica, in particolare il protocollo di messaggistica SOAP. Esistono due stili di programmazione per l'associazione SOAP definiti in WSDL:

- Documento: il messaggio SOAP contiene il documento. Viene inviato come un documento nell'elemento <soap: Body> senza regole di formattazione aggiuntive; contiene qualsiasi cosa il mittente e il destinatario siano d'accordo. Lo stile documento è la scelta predefinita.
- RPC: il messaggio SOAP contiene i parametri e i valori di ritorno. Il <soap:Body> contiene un elemento con il nome del metodo o la procedura remota invocata. Questo elemento contiene un elemento per ogni parametro di quella procedura.

Un binding SOAP (Documento o RPC) deve quindi scegliere tra due diversi formati di serializzazione / deserializzazione:

- Letterale: i dati sono serializzati secondo uno schema XML.
- Codificato: la codifica SOAP specifica come oggetti, strutture, matrici e grafici di oggetti dovrebbero essere serializzati.

Questo ti dà quattro modelli stile / uso:

- Documento / Letterale (predefinito)
- Documento / Codificato (non conforme a WS- *)

- RPC / Letterale
- RPC / Encoded

Per impostazione predefinita, il WSDL generato che hai visto finora utilizza lo stile documento/ letterale. Specificare l'annotazione `@SOAPBinding` sulla classe come mostrato nel Listato 14-14 può cambiarlo.

```

@WebService
@SOAPBinding(style = RPC, use = LITERAL)
public class CardValidator {

    public boolean validate(CreditCard creditCard) {
        // Business logic
    }
}

<!-- 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="arg0" 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"/>
```

Putting the Mapping All Together

Per avere una migliore comprensione di quale influenza hanno queste annotazioni sui SOAP web services, mettiamoli tutti insieme e osserviamo i diversi artefatti. Useremo il web service `CardValidator` e aggiungeremo la maggior parte delle annotazioni che abbiamo visto finora.

```

@WebService(portName = "CreditCardValidator", serviceName = "ValidatorService")
@SOAPBinding(style = RPC, use = LITERAL)
public class CardValidator {

    @WebResult(name = "IsValid")
    @WebMethod(operationName = "ValidateCreditCard")
    public boolean validate(@WebParam(name = "Credit-Card") CreditCard creditCard) {
        // Business logic
    }

    @WebResult(name = "IsValid")
    @WebMethod(operationName = "ValidateCreditCardNumber")
    public void validate(@WebParam(name = "Credit-Card-Number") String creditCardNumber) {
        // Business logic
    }

    @WebMethod(exclude = true)
    public void validate(Long creditCardNumber) {
        // Business logic
    }
}

```

(poi qua fa vedere tutto l'xml come viene)

Handling Exceptions

Finora tutto funziona bene: i dati scambiati tra il consumatore e il fornitore sono validi, il web service non si blocca e la rete è affidabile. Ma non è sempre così. 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 consumatore e il servizio potrebbero non essere scritti nello stesso linguaggio e potrebbero essere separati dalla rete. Quindi l'idea è di usare un errore SOAP nel messaggi. Il runtime JAX-WS converte automaticamente le eccezioni Java in messaggi di errore SOAP restituiti al client. Questa funzione consente di risparmiare un sacco di tempo ed energia eliminando la necessità di scrivere codice che associa le eccezioni del servizio agli errori SOAP.

Se si osserva il metodo di convalida del servizio Web CardValidator si noterà che se il parametro CreditCard è nullo, la convalida si arresta con una NullPointerException. Quando questo accade,

il runtime JAX-WS rileva l'eccezione NullPointerException sul server, crea un messaggio di errore SOAP e lo rimanda al consumatore.

Listing 14-21. A SOAP Fault is Sent in the SOAP Response

```

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>java.lang.NullPointerException</faultstring>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>

```

Come si può vedere il runtime JAX-WS imposta automaticamente la stringa di errore sul nome qualificato dell'eccezione Java. La specifica fornisce anche un meccanismo per distinguere tra i tipi di errore usando l'elemento faultcode. In questo caso è impostato su soap: Server che indica che il server è responsabile dell'errore (soap:client è l'altra opzione). Un'altra opzione per restituire un errore SOAP è lanciare un'eccezione dell'applicazione come mostrato sotto. Qui il web service genera una checked exception (ma lo stesso meccanismo si applica per le unchecked) se il numero della carta di credito è dispari. Questa eccezione verrà automaticamente convertita nel messaggio di errore, inserita nel SOAP body e restituito al client.

```
@WebService
public class CardValidator throws CardValidatorException {

    public boolean validate(CreditCard creditCard) {
        Character lastDigit = creditCard.getNumber().charAt(
            creditCard.getNumber().length() - 1);

        if (Integer.parseInt(lastDigit.toString()) % 2 == 0) {
            return true;
        } else {
            throw new CardValidatorException("The credit card number is invalid");
        }
    }
}
```

L'eccezione può ereditare da Exception, RuntimeException o da un'eccezione del SOAP web service, come javax.xml.ws.WebServiceException o una delle sue sottoclassi. Queste eccezioni possono anche essere annotate con @WebFault per avere una busta SOAP più esplicita come mostrato sotto.

```
@WebFault(name = "CardValidationFault")
public class CardValidatorException extends Exception {

    public CardValidatorRTException() {
        super();
    }

    public CardValidatorRTException(String message) {
        super(message);
    }
}

<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>
```

Ma se si desidera produrre un messaggio di errore SOAP più accurato con un diverso codice di errore e così via, è possibile utilizzare l'API javax.xml.soap.SOAPFactory per creare un oggetto javax.xml.soap.SOAPFault.

```
@WebService
public class CardValidator {

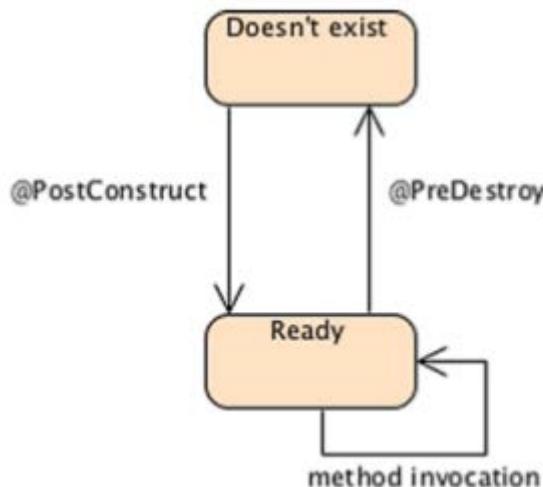
    public boolean validate(CreditCard creditCard) {
        Character lastDigit = creditCard.getNumber().charAt( ~
            creditCard.getNumber().length() - 1);

        if (Integer.parseInt(lastDigit.toString()) % 2 == 0) {
            return true;
        } else {
            SOAPFactory soapFactory = SOAPFactory.newInstance();
            SOAPFault fault = soapFactory.createFault("The credit card number is invalid", ~
                new QName("ValidationFault"));
            throw new CardValidatorException(fault);
        }
    }

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <soap:Fault>
            <faultcode>ValidationFault</faultcode>
            <faultstring>The credit card number is invalid</faultstring>
        </soap:Fault>
    </soap:Body>
</soap:Envelope>
```

Life Cycle and Callback

Come potete vedere nella Figura, i servizi Web SOAP hanno anche un ciclo di vita che ricorda i managed beans. È lo stesso ciclo di vita dei componenti che non hanno nessuno stato: o non esistono o sono pronti per elaborare una richiesta. Il container gestisce questo ciclo di vita.



Sia le servlet che gli endpoint EJB supportano la dependency injection (poiché vengono

eseguite in un contenitore) e i metodi del ciclo di vita come `@PostConstruct` e `@PreDestroy`. Il container chiama il metodo di callback `@PostConstruct`, se presente, quando crea un'istanza di un servizio e chiama la callback `@PreDestroy` quando lo distrugge. Una differenza tra gli endpoint EWB e servlet è che gli EJB possono utilizzare gli interceptors.

WebServiceContext

Un SOAP web service ha un contesto di ambiente e può accedervi inserendo un riferimento di `javax.xml.ws.WebServiceContext` con l'annotazione di `@resource`. In questo contesto, il webservice 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. Il codice sotto utilizza il `WebServiceContext` per verificare se il chiamante ha il ruolo di amministratore per convalidare un credito carta. La Tabella elenca i metodi definiti nell'interfaccia `WebServiceContext`.

Listing 14-27. SOAP Web Service Using the `WebServiceContext`

```
@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
    }
}
```

Table 14-6. Methods of the `WebServiceContext` Interface

Method	Description
<code>getHttpContext</code>	Returns the <code>MessageContext</code> 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.
<code>getUserPrincipal</code>	Returns the <code>Principal</code> that identifies the sender of the request currently being serviced.
<code>isUserInRole</code>	Returns a Boolean indicating whether the authenticated user is included in the specified logical role.
<code>getEndpointReference</code>	Returns the <code>EndpointReference</code> associated with this endpoint.

Deployment Descriptor

Come la maggior parte delle tecnologie Java EE 7, i web services consentono di definire i metadati utilizzando annotazioni e XML. Situato sotto la directory WEB-INF, il file `webservices.xml` sovrascrive le annotazioni. Come la maggior parte dei descrittori `webservices.xml` è facoltativo.

Packaging

I SOAP web services possono essere impacchettati in un file jar war o EJB. Quelli confezionati in un file war possono utilizzare gli endpoint EJB Lite o servlet. I servizi Web SOAP contenuti in un file jar possono utilizzare solo bean di sessione Stateless /Singleton. Lo sviluppatore è responsabile per il packing:

- Il bean di implementazione del servizio e le sue classi dipendenti
- Le interfacce dell'endpoint del servizio (facoltativo)
- Il file WSDL per contenimento o riferimento (non richiesto quando vengono utilizzate le annotazioni perché il WSDL può essere generato automaticamente dal runtime JAX-WS)
- Manufatti generati per la richiesta e la risposta SOAP (facoltativi in quanto vengono generati automaticamente dal runtime JAX-WS)
- Un descrittore opzionale

Publishing a SOAP Web Service

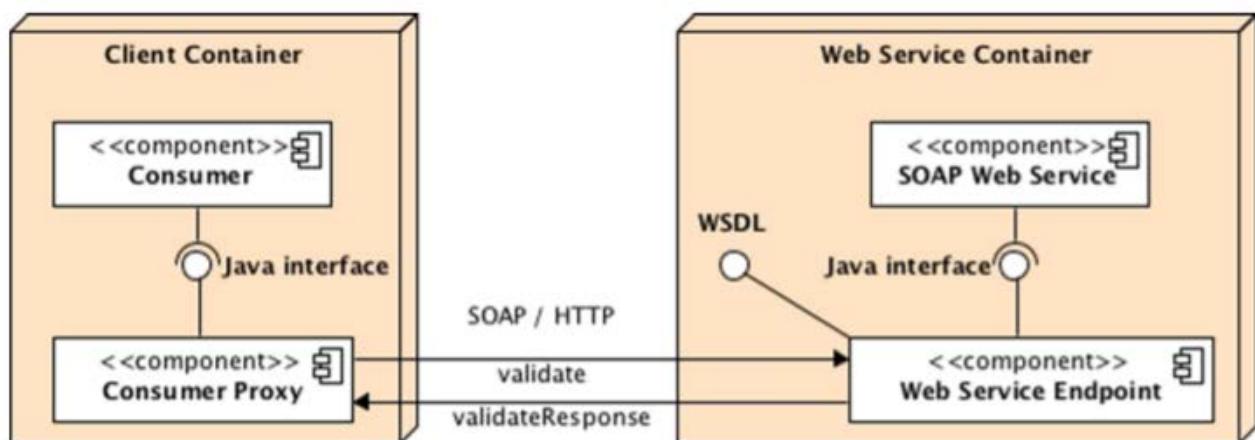
Una volta confezionato in un file war / jar, la pubblicazione di un servizio Web SOAP è solo una questione di distribuzione dell'archivio in un contenitore Java EE come GlassFish o JBoss. Ma è anche possibile pubblicare un servizio Web SOAP in un Web container come Tomcat o Jetty se si incorpora un'implementazione JAX-WS come Metro, CXF o Axis2. Ma ricorda che JAX-WS è disponibile anche in Java SE e a volte non hai bisogno di tutta la potenza di un servlet o di un EJB container (ad esempio durante il test del tuo web service). In questo caso è possibile utilizzare l'API javax.xml.ws.Endpoint per pubblicare in modo programmatico un servizio Web. Il metodo Endpoint.publish utilizza per impostazione predefinita un server HTTP leggero incluso nella JVM di Oracle.

```
@WebService  
public class CardValidator {  
  
    public boolean validate(CreditCard creditCard) {  
        // Business logic  
    }  
  
    public static void main(String[] args) {  
        Endpoint.publish("http://localhost:8080/cardValidator", new CardValidator());  
    }  
}
```

Invoking SOAP Web Services

Finora hai visto come scrivere e pubblicare un SOAP web service. Ora diamo un'occhiata a come invocare un tale servizio. Con WSDL e alcuni strumenti per generare gli stub client Java (o proxy), è possibile richiamare facilmente un servizio Web senza preoccuparsi di HTTP o SOAP. Chiamare un web service è simile al chiamare un oggetto distribuito con 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 (si noti che è anche possibile richiamare codice non Java utilizzando RMI-IIOP). Metro fornisce uno strumento di utilità WSDL-to-Java (wsimport) che genera interfacce e classi Java da un WSDL. Tali proxy forniscono una rappresentazione Java di un endpoint del servizio Web (servlet o EJB). Questo proxy inoltra quindi la chiamata locale Java al servizio Web remoto utilizzando HTTP. Quando viene richiamato un metodo su questo proxy (vedere la Figura), 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. Non è necessario comprendere il lavoro interno del proxy né osservare il codice. Prima di compilare il consumer del cliente, è necessario generare il SEI per ottenere la classe del proxy per chiamarlo nel codice.



Quando si sviluppano gli utenti SOAP utilizzando contract first, il client deve ottenere il WSDL, generare gli artefatti necessari, richiamare il servizio CardValidator per convalidare una carta di credito (il messaggio SOAP valido) e ricevere una risposta (il messaggio SOAP validateResponse).

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 una 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. Per iniettare un servizio Web, è necessario utilizzare l'annotazione @javax.xml.ws.WebServiceRef o un CDI producer.

Invoking Programmatically

Se il tuo consumer è in esecuzione all'esterno di un container, è necessario richiamare programmaticamente il web service. Come puoi vedere nel codice sotto, il servizio web CardValidator non viene invocato direttamente. Il consumatore utilizza un'istanza di CardValidatorService (che è stata generata dal WSDL grazie a wsimport) utilizzando new. Quindi deve ottenere la classe CardValidator proxy (getCardValidatorPort ()) per invocare localmente i metodi di business. Una chiamata locale viene effettuata sul metodo validate () del proxy, che a sua volta invocherà il servizio Web remoto, creerà la richiesta SOAP, eseguirà il marshalling dei messaggi della carta di credito e così via. Il proxy trova il

servizio di destinazione perché l'URL dell'endpoint predefinito è incorporato nel file WSDL e successivamente integrato nell'implementazione del proxy.

```
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);  
    }  
}
```

Invoking with Injection

D'altra parte, se il consumer viene eseguito in un contenitore, è possibile utilizzare l'injection per ottenere un riferimento al proxy . Il codice mostra una semplice classe Java in esecuzione in un application client container (ACC) che usa l'annotazione @WebServiceRef. Quando questa annotazione viene applicata su un attributo (o un metodo getter), il contenitore inietterà un'istanza del proxy client del servizio Web quando l'applicazione viene inizializzata.

```
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);  
    }  
}
```

Invoking with CDI

La specifica JAX-WS non è stata aggiornata in Java EE 7 e quindi non include tutte le funzionalità CDI. Ad esempio, non è possibile iniettare direttamente un riferimento al proxy del servizio Web utilizzando `@Inject`. Ma puoi usare i producers.

Grazie alla classe di utility WebServiceProducer qualsiasi EJB o Servlet può ora iniettare CardValidatorService con `@Inject` e richiamare su di esso un metodo di business. Con questo meccanismo è anche possibile utilizzare le alternatives.

```
public class WebServiceProducer {  
  
    @Produces  
    @WebServiceRef  
    private CardValidatorService cardValidatorService;  
@Stateless  
public class EJBConsumerWithCDI {  
  
    @Inject  
    private CardValidatorService cardValidatorService;  
  
    public boolean validate(CreditCard creditCard) {  
  
        CardValidator cardValidator = cardValidatorService.getCardValidatorPort();  
        return cardValidator.validate(creditCard);  
    }  
}
```