1. INTRO A JAVA EE

Java Enterprise Edition (**Java EE**) è un insieme di tecnologie integrate, con l'obiettivo di ridurre il costo e la complessità di sviluppare e gestire applicazioni basate su un'architettura multi-tier, dove tra i client (front end) e i dati (back end) vengono posti alcuni livelli. Java EE introduce la logica a componenti, quali sono unità di software categorizzabili come componenti client, web e business. Introduce una serie di nuove funzioni, tra le quali la **Context and Dependency Injection** (**CDI**), la quale permette l'iniezione automatica di risorse. Vengono introdotti i **containers**, i quali provvedono determinati servizi ai componenti del sistema, come la gestione del ciclo di vita, dependency injection, concorrenza ed altro.

1.1 LIVELLI DI JAVA EE

Abbiamo già preannunciato l'architettura di Java EE definendola **multi-tier**. I tier che introduce sono i seguenti:

- un tier di presentazione client (browser web o client applicativo);
- un tier HTTP server (Web server / Web container) che fornisce l'assemblaggio delle informazioni fornite dall'applicazione mediante determinate tecnologie come JSP, Servlet, ecc.;
- un tier server di logica di business (server applicazione / EJB container);
- un ultimo tier per l'accesso ai dati (Database container).

I tier vengono accomunati come unico tier, detto anche middle-layer o middle-tier.

1.2 COMPONENTI

Java EE definisce quattro tipologie:

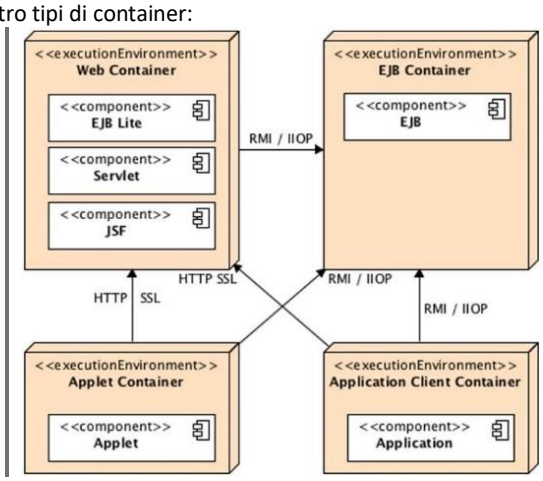
- **Applets**, sono applicazioni con GUI che vengono eseguite nel web browser;
- **Applicazioni**, sono programmi eseguiti su un client che sfruttano principalmente il middle- tier;
- **Applicazioni Web**, sono programmi fatti di servlet, filtri, listener, pagine JSP, ecc.; tali programmi vengono eseguiti in un web container e rispondono ad http requests da parte dei web client;
- **Applicazioni Enterprise**, sono programmi fatti di EJB (Enterprise Java Beans), Java Message Service, Java Transaction, ecc.; tali programmi vengono eseguiti in un EJB container e possono essere accessibili sia localmente che in remoto tramite RMI.

1.3 CONTAINER

L'infrastruttura di Java EE è partizionata in diversi domini denominati **container**, i quali offrono servizi alle componenti supportate (sicurezza, accesso al database, transazioni, servizio di naming, resource injection) incrementando la modularità che il sistema offre.
Ogni tipologia di container gestisce determinate tipologie di componenti. Java EE definisce quattro tipi di container:

1. **Applet container**, fornisce un ambiente in cui vengono delineati i limiti nel quale deve essere eseguito il codice;
2. **Application client container**, include molteplici classi Java, librerie ed altri file per gestire la sicurezza, il servizio di naming, ecc. Tale container comunica con il container EJB utilizzando RMI-IIOP, mentre con il web container utilizzando HTTP;
3. **Web container**, adatto al managing e all'esecuzione di componenti web come servlets, JSPs, filtri, listeners, ecc. Tale container è responsabile riguardo l'istanziazione, l'inizializzazione e l'invocazione delle servlets, utilizzando HTTP;
4. **EJB container**, adatto al managing e all'esecuzione degli Enterprise beans: crea nuove istanze di determinati EJBs, ne gestisce il loro ciclo di vita, provvede servizi e la possibilità di essere invocati asincronamente. Addirittura, in tale container non è il programmatore ad utilizzare la keyword new, siccome ciò viene effettuato dal container.

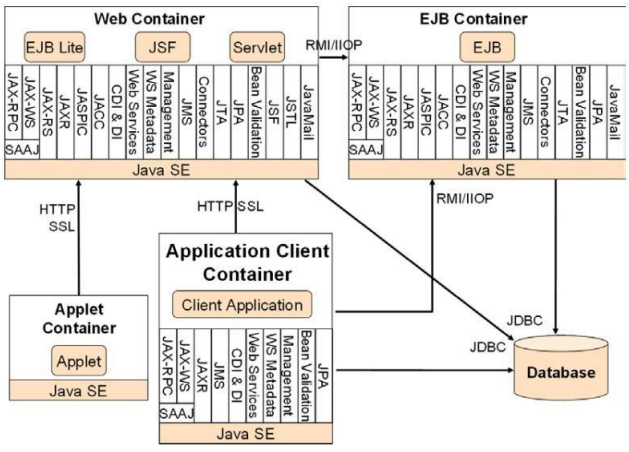
Nei container vengono mostrati i componenti supportati, mentre le frecce che collegano i container mostrano i protocolli che utilizzano per comunicare tra loro.



1.4 SERVIZI

I container provvedono determinati servizi alle componenti gestite, permettendo al programmatore di concentrarsi sul codice di business invece di risolvere problemi tecnici rilegati alle applicazioni enterprise. Vediamo ora i servizi offerti dai container: ovviamente ogni container offre determinati servizi, quindi il fatto che vediamo tutti i servizi non implica il fatto che ogni container offra ogni servizio.

- JTA – Java Transaction API: permette l'utilizzo delle transazioni, cioè serie di istruzioni le quali vengono eseguite tutte o nessuna;
- JPA – Java Persistence API: permette di rendere persistenti determinati dati e permette l'esecuzione delle query tramite Java Persistente Query Language (JPQL), in modo da gestire oggetti posti in un database;
- JMS – Java Message Service: permette alle componenti di comunicare tra loro asincronamente tramite messaggi;
- Naming: permette di associare e trovare oggetti tramite operazione di lookup. Tale servizio viene utilizzato anche tramite iniezione, ma in maniera trasparente;
- JavaMail: permette all'applicazione di mandare email;
- XML processing: effettua la manipolazione di codice XML;
- JSON processing: effettua la manipolazione di codice JSON;
- Servizi di sicurezza;
- Web services;
- Dependency Injection (DI);
- Deployment.



1.5 PACKAGING

Per esser deployati in un container, i componenti devono essere impacchettati in determinati archivi, il cui formato dipenderà dal container che gestisce quel determinato tipo di componente.

1.6 ANNOTAZIONI

È bene sapere che esistono due tipi di approccio alla programmazione, l'approccio imperativo, che tende a specificare l'algoritmo che risolve un determinato problema, e l'approccio dichiarativo, che tende a specificare come deve essere risolto un determinato problema. In Java EE, l'approccio dichiarativo utilizza metadati, **annotazioni** e deployment descriptors.

2. CONTEXT AND DEPENDENCY INJECTION (CDI)

La prima versione di Java EE introduceva il concetto di **inversion of control** (**IoC**), il quale consisteva nel fatto che il container avesse predo controllo del codice di business e che avesse provvisto diversi servizi tecnici, come transazioni o gestione della sicurezza. Il fatto che il container prenda controllo sta nel fatto che gestisca egli stesso il ciclo di vita delle componenti, che gestisca la dependency injection e la configurazione delle componenti. In Java EE 6 è stata introdotta la **Context and Dependency Injection** (**CDI**), la quale permette ai managed bean (bean gestiti dal container) di essere iniettabili, intercettabili e, appunto, gestibili. CDI è stata creata con lo scopo di assicurare *basso accoppiamento* e *forte tipizzazione* (loose coupling, strong typing).

2.1 PANORAMICA SUI BEAN

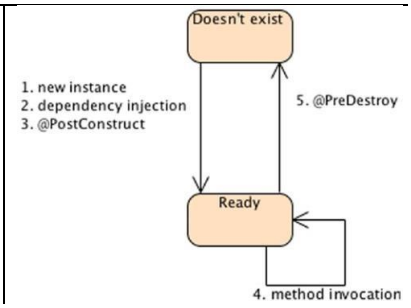
Mentre Java SE possiede solamente i Java Bean, Java EE possiede gli **Enterprise Java Bean** (**EJB**), che non sono gli unici componenti offerti da Java EE, difatti si hanno altri componenti come le servlet, web services, entità e managed bean. È bene sapere che i normali Java Bean offerti da Java SE sono detti **POJO** in Java EE (**Plain Old Java Object**) e seguono un determinato pattern che consiste in convenzioni per proprietà e costruttori. Anche altri componenti di Java EE seguono precisi pattern, ma con la differenza che vengono eseguiti in un container e usufruiscono di determinati servizi: tali componenti vengono detti **managed bean**, che sono bean gestiti dal container che supportano solo determinati servizi, quali la resource injection, la gestione del ciclo di vita e l'intercezione. Tali bean possono essere visti come una generalizzazione delle diverse componenti di Java EE, ad esempio un EJB può esser visto come un managed bean con servizi aggiuntivi. I CDI bean sono una categoria di managed bean con un ciclo di vita migliorato per oggetti con uno stato (*stateful objects*), sono legati ad un contesto ben definito, permettono la DI ben tipizzata, l'intercezione e l'uso delle annotazioni. Precisamente, tranne in alcune eccezioni, ogni classe Java con un costruttore di default ed eseguita in un container viene considerata come bean.

2.2 DEPENDENCY INJECTION (DI)

La **Dependency Injection** (**DI**) è un design pattern che permette di disaccoppiare componenti dipendenti tra loro. Si utilizza la DI in un ambiente gestito, quindi invece di cercare oggetti tramite servizi di lookup, il container inietta gli oggetti dipendenti in maniera del tutto automatica. Sostanzialmente il container effettua in automatico il lookup dell'oggetto per poi iniettarlo. È possibile utilizzare JNDI per cercare ed ottenere risorse gestite (vedremo JNDI più avanti, ma sostanzialmente è un servizio che permette di effettuare il lookup di oggetti), ma il container semplifica ciò nascondendo tali operazioni al programmatore. Ciò ha permetto ai programmatori di iniettare determinate risorse in determinati componenti.

2.3 CICLO DI VITA DI UN CDI BEAN

Il ciclo di vita di un POJO è: si crea l'istanza utilizzando la keyword "new" ed essa verrà eliminata quando il garbage collector lo riterrà giusto, in modo da liberare memoria. Per i CDI bean, la questione è diversa, se si vuole usare un CDI bean in un container, l'uso della keyword "new" è vietato.
I CDI bean devono essere necessariamente iniettati, quindi il ciclo di vita di tali bean passerà nelle mani del container. Per inizializza un bean, senza richiamare il suo costruttore, il container fornisce delle funzioni chiamate "callback", nelle quali è possibile effettuare determinate operazioni in determinati momenti.
Riguardo il ciclo di vita di un CDI bean o di un managed bean, vengono eseguite due callback: **@PostConstruct** e **@PreDestroy**.

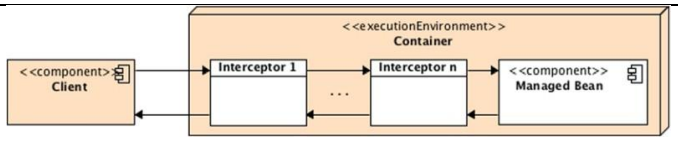


2.4 SCOPE E CONTESTO

I CDI bean sono contestuali, quindi hanno uno scope ben definito. Il bean opera entro determinati limiti quali possono essere una request, una sessione, l'intera applicazione, ecc. Il container gestisce tutti i bean posti nel proprio scope e, alla fine della sessione, li distrugge tutti.

2.5 INTRO AGLI INTERCEPTORS

Gli **interceptors** vengono utilizzati per interporre invocazioni a metodi di business, ciò permette di riutilizzare con estrema facilità eventuale codice ripetuto, magari, molteplici volte. Sostanzialmente, quindi, è possibile intercettare l'esecuzione di un managed bean per eseguire un interceptor.



2.6 INTRO AL DEPLOYMENT DESCRIPTOR

La specifica di Java EE ha un **deployment descriptor** in XML opzionale, descrive come un componente, un modulo o un'applicazione dovrebbe essere configurata. Con CDI, il deployment descriptor viene chiamato ***beans.xml*** ed è obbligatorio il suo utilizzo, tale file non fa altro che attivare CDI.

2.7 INTRO ALLA PRATICA SUI CDI BEAN

Un CDI bean non è altro che una classe Java contenente logica di business. I CDI bean non possono essere istanziati tramite keyword "new", quindi è necessario utilizzare l'iniezione che viene eseguita tramite apposita annotazione **@Inject**, tutto ciò che è eseguibile su tali bean. Partiamo, però, con l'anatomia di un CDI bean, in modo da sapere com'è fatto in precisione, per poi passare al come funziona l'iniezione.

2.8 ANATOMIA DI UN CDI BEAN

Per essere trattata come CDI bean, una classe Java deve soddisfare le seguenti condizioni:

- Non deve essere una classe interna;
- Deve essere una classe concreta o annotata con @Decorator;
- Deve avere un costruttore di default senza alcun parametro (non deve essere per forza l'unico costruttore della classe);
- Non deve implementare il metodo finalize.

2.9 DEPENDENCY INJECTION COI CDI BEAN

Vediamo come funziona la **Dependency Injection** nella pratica con i CDI bean. Poniamo di avere una classe BookService, la quale fa uso di un generatore di codici ISBN.
La classe BookService fa utilizzo di un generatore IsbnGenerator: in tal modo, l'accoppiamento è decisamente alto siccome la classe BookService dipende dall'utilizzo di IsbnGenerator.

```
public class BookService {

   private NumberGenerator generator = null;

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

   public Book createBook(String title, Float price, String description){
      Book b = new Book(title, price, description);
      b.setIsbn(this.generator.generateNumber());
      return b;
   }

}
```

Ciò è un problema, siccome un eventuale cambiamento della classe IsbnGenerator dovrebbe essere riportato anche in BookService.
Il problema si capirebbe meglio nel caso si voglia utilizzare un generatore di ISSN invece del generatore di ISBN, si dovrebbe cambiare la classe utilizzata in BookService a causa della dipendenza dovuta dall'alto accoppiamento.

Una soluzione riguarda il passaggio di un generatore al costruttore che implementi un'interfaccia comune, ma se il generatore di ISBN e l'ISSN implementano entrambi NumberGenerator, il problema è risolto ma c'è comunque dipendenza a causa dell'interfaccia, si ricorre all'iniezione.

2.9.1 INIEZIONE TRAMITE @Inject

In un ambiente gestito non è necessario costruire dipendenze, ma è possibile lasciare tale compito al container evitando l'alto accoppiamento.
La **dependency injection** è l'abilità di iniettare bean in un modo che assicuri forte tipizzazione. Con CDI è possibile iniettare qualsiasi risorsa tramite l'annotazione **@Inject**. Nel seguente esempio il container inietta un oggetto che implementi NumberGenerator, la variabile iniettata viene denominata come **injection point**. L'annotazione **@Inject**, quindi, definisce un **injection point** che viene iniettato durante l'istanziazione del bean. L'iniezione può avvenire in tre modi differenti: tramite proprietà, metodo setter e costruttore.

Nell'esempio utilizzante la @Inject viene mostrata l'iniezione tramite ***proprietà***:

```
public class BookService {

   @Inject
   private NumberGenerator generator;

   public Book createBook(String title, Float price, String description){
      Book b = new Book(title, price, description);
      b.setIsbn(this.generator.generateNumber());
      return b;
   }

}
```

L'iniezione tramite ***metodo*** setter funziona nel seguente modo: viene annotato il metodo setter e viene iniettato il parametro passato come argomento. Il container provvederà all'iniezione di un NumberGenerator al metodo setter:

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

L'iniezione tramite ***costruttore***, invece, permette di iniettare un bean come parametro del costruttore:

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

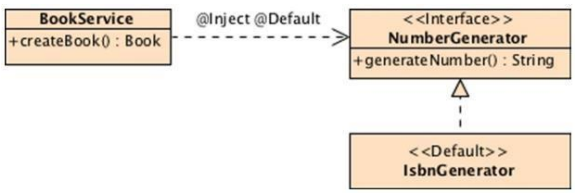
Nei modi appena visti, viene iniettato un oggetto che rappresenta un'implementazione di NumberGenerator: precisamente, però, quale delle tante? Ebbene, è possibile iniettare solo se presente un'unica implementazione dell'interfaccia. L'iniezione viene detta **default injection** (iniezione di default). È bene precisare il fatto che si possono avere più implementazioni della stessa interfaccia, cambierà il modo con cui viene effettuata l'iniezione, ma tra le molteplici implementazioni è possibile specificarne una di default.
Ricapitolando: se esiste un'unica implementazione dell'interfaccia, allora quell'unica implementazione viene detta di default, se esistono molteplici implementazioni dell'interfaccia, una di queste può essere specificata come default. Ovviamente, nel caso si voglia iniettare un'implementazione di default, è necessario specificarlo tramite apposita annotazione.

Specifica dell'implementazione di default:

```
@Default
public class IsbnGenerator implements NumberGenerator{
   @Override
   public int generateNumber(){
      return 0;
   }
}
```

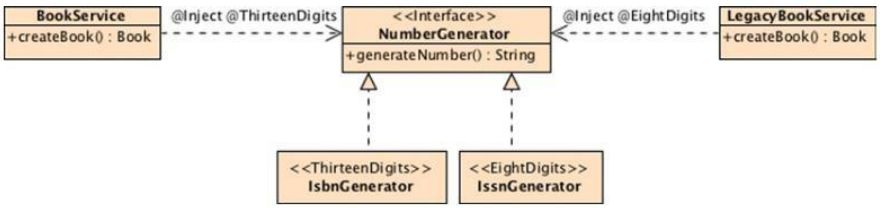
Iniezione dell'implementazione di default:

```
@Inject
@Default
private NumberGenerator generator;
```



2.9.2 QUALIFICATORI (PER SPECIFICARE L'IMPLEMENTAZIONE)

Il controllo della presenza di un'implementazione iniettabile viene fatto a tempo di inizializzazione, quindi viene controllato ogni injection point esistente. Ciò significa che nel caso non esista alcuna implementazione dell'interfaccia, il container ci informerà riguardo la mancanza di essa. Nel caso, invece, esista un'unica implementazione, sarà possibile effettuare l'iniezione tramite l'annotazione @Default.
Nel caso esistano molteplici implementazioni della stessa interfaccia, il container ci informerebbe riguardo la dipendenza ambigua siccome non saprebbe quale implementazione dell'interfaccia scegliere. È possibile specificare quale implementazione si preferisce utilizzare tramite i **qualificatori**, i quali precisano l'iniezione effettuata.
I **qualificatori** sono annotazioni che non fanno altro che mantenere la tipizzazione forte. Il seguente schema spiega come funzionano i qualificatori:



Il qualificatore è un'annotazione definita dal programmatore e specifica un tipo associato ad un'implementazione. È utile quando si hanno molteplici implementazioni della stessa interfaccia.
Si supponga di avere due implementazioni dell'interfaccia NumberGenerator: IsbnGenerator e IssnGenerator. Quando si effettua l'iniezione, il container non sa quale delle due implementazioni iniettare. Si specificano tanti qualificatori quante sono le implementazioni di quell'interfaccia.

Nel seguente modo viene specificato un qualificatore denominato come ThirteenDigits:

```
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.inject.Qualifier;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

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

Si specifica, ovviamente, nello stesso modo anche EightDigits.
Il qualificatore ThirteenDigits, alla scelta dell'implementazione da iniettare, suggerirà l'utilizzo di IsbnGenerator, mentre il qualificatore EightDigits suggerirà l'utilizzo di IssnGenerator.

A tal punto, quindi, si specifica che le implementazioni faranno riferimento a tali qualificatori semplicemente utilizzandoli come annotazioni alle classi:

```
@ThirteenDigits
public class IsbnGenerator implements NumberGenerator{
   @Override
   public int generateNumber(){
      return 0;
   }
}
```

Al momento dell'iniezione, quindi, va specificato quale implementazione iniettare:

```
public class BookService {

   @Inject
   @ThirteenDigits
   private NumberGenerator generator;

   public Book createBook(String title, Float price, String description){
      Book b = new Book(title, price, description);
      b.setIsbn(this.generator.generateNumber());
      return b;
   }

}
```

Ecco perché si dice che CDI utilizza forte tipizzazione: è possibile rinominare l'implementazione a proprio piacere, cambiarla o farci quello che si vuole, ma l'injection point non cambierà, assicurando basso accoppiamento. L'utilizzo di molteplici qualificatori, però, rende il codice più complesso e meno leggibile: si ricorre, quindi, ai qualificatori con membri.
È importante, comunque, specificare che una classe può essere comunque contraddistinta da molteplici qualificatori e non solamente da uno. Se una classe è contraddistinta da molteplici qualificatori, allora per essere iniettata dovranno essere utilizzati altrettanti qualificatori.

2.9.3 QUALIFICATORI CON MEMBRI

Ogni volta in cui c'è da scegliere una tra le molteplici implementazioni di un'interfaccia, è necessario l'utilizzo dei **qualificatori**. Come già visto, nel caso si vogliano distinguere molteplici implementazioni sarebbe necessario creare annotazioni extra (ad esempio @TwoDigits, @EightDigits, @TenDigits), rendendo il codice meno leggibile. È possibile evitare la creazione di un gran numero di annotazioni tramite i membri, i quali sono parametri che permettono di specificare quale implementazione utilizzare.

La seguente interfaccia NumberDigits permette tramite un membro denominato number di specificare quante cifre utilizza quell'implementazione, quindi le consente di contraddistinguersi dalle altre tramite un semplice parametro.

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

   int number();

}
```

È di semplice utilizzo ed evita che si creino molteplici qualificatori inutilmente, quando poi è possibile utilizzare lo stesso per effettuare una maggiore contraddistinzione tramite dei semplici parametri. In questo esempio viene mostrato l'utilizzo di un solo parametro, ma è bene sapere che è possibile utilizzare quanti parametri si vogliano.

Vediamo ora come viene utilizzata l'annotazione sulla classe per categorizzarla con tale qualificatore.

```
@NumberDigits(number = 13)
public class IsbnGenerator implements NumberGenerator{
   @Override
   public int generateNumber(){
      return 0;
   }
}
```

Oppure:

```
@NumberDigits(number = 8)
public class IssnGenerator implements NumberGenerator{
   @Override
   public int generateNumber(){
      return 0;
   }
}
```

La differenza circa il normale utilizzo è veramente minima, se non semplificata.

```
public class BookService {

   @Inject
   @NumberDigits(number = 13)
   private NumberGenerator generator;

   public Book createBook(String title, Float price, String description){
      Book b = new Book(title, price, description);
      b.setIsbn(this.generator.generateNumber());
      return b;
   }

}
```

2.9.4 PRODUCERS (INIEZIONE DI TIPI PRIMITIVI E POJO)

Abbiamo visto come iniettare CDI bean in semplici variabili. È, inoltre, possibile iniettare tipi primitivi e POJO, ciò è reso possibile dai **producers**.
Se non fosse per i producers, non sarebbe possibile iniettare classi come Date e String, perché nel package in cui sono situate non è presente il deployment descriptor beans.xml, se beans.xml non è presente, allora CDI non sarà attivo e gli oggetti non saranno trattati come beans, di conseguenza, non saranno nemmeno iniettabili. L'unico modo per iniettare tali oggetti è tramite l'utilizzo dei **producers**.
Viene stabilita una classe che produca risorse associate ad un qualificatore.
Nel seguente esempio, la classe NumberProducer dichiara due variabili private ed un metodo contrassegnati dai qualificatori (non è obbligatorio il loro utilizzo). I qualificatori utilizzati per "differenziare" tali variabili o metodi devono essere vuoti, quindi devono essere interfacce che non forniscano metodi. Sono qualificatori vuoti, atti solamente a differenziare la risorsa. In tal caso, se si inietta una stringa utilizzando l'annotazione @ThirteenDigits, si inietterà l'apposita risorsa stringa contrassegnata con tale annotazione nel momento della dichiarazione, quindi in questo caso verrà iniettato prefix13digits.
Notiamo che la stessa annotazione può essere utilizzata per la dichiarazione di molteplici risorse, si inietterà la risorsa che corrisponderà al tipo richiesto: se si richiede l'iniezione di una stringa con annotazione @ThirteenDigits, allora si otterrà la stringa dichiarata con tale annotazione. Come detto, però, l'iniezione di tipi primitivi è resa possibile tramite producers, quindi si utilizza anche l'annotazione @Produces, oltre all'annotazione per distinguere la risorsa.

```
public class NumberProducer {

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

   @Produces @ThirteenDigits
   private int editorNumber = 84356;

   @Produces @Random
   public double random(){
      return 0;
   }

}
```

Vediamo ora la classe IsbnGenerator che, a differenza dei precedenti esempi, conterrà variabili di istanza che verranno iniettate al momento dell'istanziazione, tali variabili sono tipi primitivi, quindi vengono iniettati grazie ai producers. Ovviamente, al momento dell'iniezione, per capire a quale stringa/intero/ecc si fa riferimento, si specifica il qualificatore che annota l'oggetto da iniettare al momento della sua dichiarazione.

```
@ThirteenDigits
public class IsbnGenerator implements NumberGenerator{
   @Inject @ThirteenDigits
   private String prefix;
   @Inject @ThirteenDigits
   private int editorNumber;
   @Inject @Random
   private double randomValue;
   @Override
   public int generateNumber(){
      return (int) randomValue;
   }
}
```

2.10 CONTESTO (SCOPE) E ANNOTAZIONI

Ogni CDI Bean ha un ciclo di vita rilegato ad un determinato contesto. In CDI, un bean è legato e rimane in tale contesto finché non viene distrutto dal container. Non esiste alcun modo per rimuovere manualmente un bean dal contesto, può esser rimosso solamente dal container.
CDI definisce alcuni contesti (detti anche scope):

- **Application scope** (**@ApplicationScoped**): si estende per tutta la durata dell'applicazione. Il bean da utilizzare viene creato una sola volta durante l'intera applicazione e viene eliminato dal container quando l'applicazione si conclude. Questo scope è utile per le classi "helper", cioè per oggetti che conservano dati condivisi dall'intera applicazione;
- **Session scope** (**@SessionScoped**): si estende tra diverse richieste http o tra diversi metodi utilizzati per la sessione di un singolo utente. Il bean da utilizzare viene creato per la sessione HTTP e viene eliminato quando la sessione scade. Utile, appunto, per oggetti che necessitano di essere nella sessione, come le credenziali di login;
- **Request scope** (**@RequestScoped**): corrisponde ad una singola richiesta HTTP o ad una singola invocazione di un metodo. Il bean da utilizzare viene creato per la durata dell'invocazione del metodo e viene eliminato quando l'invocazione si conclude. Viene utilizzato quando i bean hanno necessità di esistere per la durata di una request HTTP;
- **Conversation Scope** (**@ConversationScoped**): si estende tra diverse invocazioni di metodi, precisamente da una determinata invocazione ad un'altra (vengono dette starting ed ending point);
- **Dependent pseudo-scope** (**@Dependent**): un bean viene creato ogni volta che viene iniettato e il suo riferimento viene rimosso quando il target dell'iniezione viene rimosso. Questo è lo scope di default per CDI.

Vediamo un semplice esempio nel quale viene applicata l'annotazione per il contesto.

```
@SessionScoped
public class BookService implements Serializable {

   @Inject
   @NumberDigits(number = 13)
   private NumberGenerator generator;

   public Book createBook(String title, Float price, String description){
      Book b = new Book(title, price, description);
      b.setIsbn(this.generator.generateNumber());
      return b;
   }

}
```

Va data, però, particolare attenzione al **Conversation Scope**, siccome viene utilizzato tramite uno starting point e un ending point, si inietta un oggetto Conversation tramite semplice @Inject, il quale verrà utilizzato per eseguire i metodi definiti nell'intervallo dei due punti, i quali sono definiti tramite appositi metodi begin() ed end() di tale oggetto iniettato.

2.11 INTERCEPTOR

Gli **interceptor** permettono di aggiungere funzionalità ai nostri CDI bean, quando un client invoca un metodo su un CDI bean, su un EJB, ecc, il container intercetta la chiamata al metodo ed esegue l'interceptor, quindi le funzionalità aggiuntive, prima dell'esecuzione dell'invocazione.
Esistono quattro tipi di interceptor:

- **Constructor-level interceptor** (**@AroundConstruct**): interceptor eseguito all'utilizzo del costruttore della classe target;
- **Method-level interceptor** (**@AroundInvoke**): interceptor eseguito all'utilizzo di un metodo della classe target;
- **Timeout method interceptor** (**@AroundTimeout**): interceptor che si interpongono tra metodi di timeout;
- **Life-cycle callback interceptor** (**@PostConstruct** e **@PreDestroy**): interceptor che si interpongono tra gli eventi del ciclo di vita del bean target.

È bene sapere che gli interceptor hanno un proprio scope, quindi vengono a loro volta suddivisi in tre categorie circa il loro utilizzo:

- **Target class interceptor**: l'interceptor è disponibile e viene eseguito solo nella classe in cui è dichiarato, quindi solamente nella classe target (quindi lo scope è locale per la classe in cui l'interceptor è dichiarato);
- **Class interceptor**: l'interceptor viene isolato in una classe di supporto e viene utilizzato da altre classi specificando la classe di supporto in apposita annotazione (quindi lo scope è locale ed esterno alla classe in cui l'interceptor è dichiarato);
- **Life-Cycle interceptor**: riguarda i callback del ciclo di vita dei bean. Il callback viene visto come interceptor (effettivamente un callback è un interceptor) e viene richiamato nel momento adatto nel ciclo di vita del bean. Tali interceptor sono, però, isolati in una classe di supporto.

Se non fosse ancora chiaro, gli interceptor non sono altro che metodi eseguibili in determinati situazioni o eventi: come visto, possono essere eseguiti prima dell'esecuzione del costruttore, prima dell'invocazione di un metodo, durante il ciclo di vita, ecc.
I metodi candidati ad essere interceptor devono rispettare le seguenti regole:

- Il metodo non deve essere static o final;
- Il metodo deve avere un parametro InvocationContext e deve ritornare Object e non una precisa classe (è possibile non ritornare nulla, rendendo il metodo void);
- Il metodo deve poter lanciare una checked exception.

L'oggetto InvocationContext permette agli interceptor di controllare il comportamento della catena di invocazioni: se molteplici interceptor sono concatenati tra loro tramite un'invocazione, viene passata la stessa istanza di InvocationContext, la quale permette di gestire determinati dati tra i diversi interceptor.

Esempio di target class interceptor

Vediamo ora un esempio di **target class interceptor**:
Viene specificato un method-level interceptor, il quale verrà eseguito prima di ogni esecuzione di ogni chiamata a metodo appartenente alla classe in cui tale interceptor è dichiarato.

```
@Transactional
public class CustomerService {
    @Inject
    private EntityManager em;

    public void createBook(Book b){
        em.persist(b);
    }

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

    @AroundInvoke
    private void logMethod(InvocationContext ic) throws Exception{
        System.out.println("Entered in logMethod");
    }
}
```

Esempio di class interceptor

Vediamo ora un esempio di **class interceptor**: viene creata una classe contenente quanti interceptor si vogliano (in tal caso ne conterrà uno solo), i quali verranno utilizzati da classi esterne. In tale esempio dichiariamo una classe contenente un interceptor che viene eseguito all'invocazione di un determinato metodo di una classe esterna.
Qui vediamo la dichiarazione della classe contenente il metodo interceptor:

```
public class SupportClass {
    @AroundInvoke
    public void supportMethod(InvocationContext ic) throws Exception{
        System.out.println("It's the support method!");
    }
}
```

Per utilizzare l'interceptor dichiarato nella classe esterna, bisogna utilizzare l'annotazione **@Interceptors(…)**, la quale specifica la classe dalla quale andare a ricavare gli interceptor.

Il seguente esempio mostra come è possibile richiamare gli interceptor su un solo e specifico metodo, facendo in modo che l'annotazione faccia riferimento ad un unico metodo.

```
@Transactional
public class CustomerService {
    @Inject
    private EntityManager em;

    @Interceptors(SupportClass.class)
    public void createBook(Book b){
        em.persist(b);
    }

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

}
```

Il seguente esempio, invece, mostra come è possibile richiamare gli interceptor su qualsiasi metodo della classe:

```
@Transactional
@Interceptors(SupportClass.class)
public class CustomerService {
   @Inject
   private EntityManager em;

   public void createBook(Book b){
      em.persist(b);
   }

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

}
```

È possibile escludere l'esecuzione dell'interceptor da determinati metodi tramite l'utilizzo dell'annotazione **@ExcludeClassInterceptors**.
Altra nota riguarda l'utilizzo di molteplici class interceptor, è possibile includere molteplici classi tramite l'annotazione @Interceptors(…), utilizzando la seguente sintassi: **@Interceptors({C1.class, C2.class, …})**.
L'utilizzo di tale sintassi permette di definire una determinata priorità per gli interceptor, definendo quale venga eseguito per primo e quale dopo.

Interceptor nel Deployment Descriptor

Gli interceptor sono disabilitati di default e necessitano di essere abilitati utilizzando il deployment descriptor beans.xml.

```
<interceptors>
   <class>test_src.LoggableInterceptor</class>
</interceptors>
```

2.11.1 INTERCEPTOR BINDING

Precedentemente abbiamo visto come funzionano gli interceptor, specificando la classe in cui sono dichiarati per poterli usare: ciò non offre basso accoppiamento, siccome si necessita di specificare una determinata classe. Di conseguenza CDI offre l'**interceptor binding**, il quale introduce l'utilizzo di annotazioni definite dal programmatore che facciano riferimento alla classe contenente gli interceptor, in modo da offrire basso accoppiamento.

L'annotazione personalizzata interceptor binding viene creata nel seguente modo:

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

Ottenuta quindi l'annotazione personalizzata, la si applica alla classe contenente gli interceptor preceduta dall'annotazione @Interceptor.

```
@Interceptor
@LoggableInterceptor
public class SupportClass {
   @AroundInvoke
   public void supportMethod(InvocationContext ic) throws Exception{
      System.out.println("It's the support method!");
   }
}
```

Fatto ciò, si applica l'interceptor binding (precisamente, solamente l'annotazione personalizzata) ai metodi o alla classe i cui metodi verranno preceduti dall'esecuzione dell'interceptor.

```
@Transactional
@LoggableInterceptor
public class CustomerService {
   @Inject
   private EntityManager em;

   public void createBook(Book b){
      em.persist(b);
   }

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

}
```

Notiamo che tale definizione non regola quale interceptor viene eseguito prima dell'altro, quindi non definisce in qualche modo una priorità tra interceptor, è possibile definirla manualmente tramite apposita annotazione **@Priority(p)**, dove p è un intero che indica la priorità. Gli interceptor con priorità p minore vengono eseguiti prima.

2.12 EVENTI

Gli **eventi** sono un qualcosa di astratto, definiscono quando succede qualcosa e sono gestibili: seguono il pattern Observer della Gang of Four.
I bean possono definire eventi, lanciarli (viene detto "*fire*") e gestirli. Nell'ambito degli eventi abbiamo due attori: l'event producer, che definisce e lancia eventi, e l'observer, che osserva il comportamento degli eventi e li gestisce.

Nel seguente esempio definiamo un event producer, quindi una classe in grado di definire e lanciare eventi.

```
public class BookService implements Serializable {

   @Inject
   private NumberGenerator generator;

   @Inject
   private Event<Book> bookAddedEvent;

   public Book createBook(String title, Float price, String description){
      Book b = new Book(title, price, description);
      b.setIsbn(this.generator.generateNumber());
      bookAddedEvent.fire(b);
      return b;
   }

}
```

La gestione dell'evento lanciato passa all'observer, il quale è un semplice bean con metodi riguardanti la gestione degli eventi, ognuno di questi metodi richiede uno specifico parametro annotato con **@Observes** ed eventuali qualificatori.
Ciò fa in modo che il metodo venga notificato di un evento, controllando se il tipo di oggetto contenuto dall'evento coincida col parametro annotato.

Vediamo un esempio di observer:

```
public class BookContainer {

   List<Book> inventory = new ArrayList<>();

   public void addBook(@Observes Book b){
      System.out.println("Added book " + b.title);
      inventory.add(b);
   }

}
```

L'observer BookContainer contiene un metodo di gestione degli eventi addBook, riconoscibile dal fatto che abbia come parametro un oggetto annotato con **@Observes**.
È possibile applicare qualificatori agli eventi, in modo da poter distinguere gli eventi da altri eventi dello stesso tipo.

Nel seguente esempio, grazie ai qualificatori, dichiariamo un evento lanciabile quando un libro viene creato ed un evento lanciabile quando un libro viene eliminato. Il qualificatore, in realtà, non serve tanto al lancio degli eventi, bensì serve alla loro gestione siccome, altrimenti, con la semplice annotazione @Observes si gestirebbe qualsiasi evento contenente un oggetto Book. Ciò implica che l'utilizzo dei qualificatori non renderà più unico il controllo del tipo.

```
public class BookService implements Serializable {

   @Inject
   private NumberGenerator generator;

   @Inject @Added
   private Event<Book> bookAddedEvent;

   @Inject @Removed
   private Event<Book> bookRemovedEvent;

   public Book createBook(String title, Float price, String description){
      Book b = new Book(title, price, description);
      b.setIsbn(this.generator.generateNumber());
      bookAddedEvent.fire(b);
      return b;
   }

   public void removeBook(Book b){
      bookRemovedEvent.fire(b);
   }

}
```

Dopo questo esempio riguardante la dichiarazione degli eventi e il loro lancio, vediamo anche l'esempio in cui essi vengono gestiti tramite qualificatori.

```
public class BookContainer {

   List<Book> inventory = new ArrayList<>();

   public void addBook(@Observes @Added Book b){
      System.out.println("Added book " + b.title);
      inventory.add(b);
   }

   public void removeBook(@Observes @Removed Book b){
      System.out.println("Removed book " + b.title);
      inventory.remove(b);
   }

}
```

3. JAVA PERSISTENCE API (JPA)

Sappiamo bene che le applicazioni sono fatte di logica di business (logica di elaborazione che rende operativa un'applicazione), interazioni con altri sistemi, interfacce e dati. Soffermandoci sui dati, sappiamo che per la loro manipolazione è necessario che i dati siano conservati in un database, nel quale possono essere anche ripresi ed analizzati. I database conservano dati, fungendo da punto centrale alle applicazioni, e processano dati.
Nei database relazionali i dati sono conservati in tabelle composte da righe e colonne e sono distinti tra loro da chiavi primarie (primary key, PK), mentre sono collegati tra loro tramite chiavi esterne (foreign key, FK). Gli oggetti Java non rientrano in database relazionali siccome si dovrebbero manipolare istanze di classi e non dati. Certo, sarebbe possibile estrarre i dati dalle istanze, ma questo è un altro discorso. Il problema è che i database relazionali non possono trattare determinate cose come istanze di oggetti, interfacce, classi astratte, annotazioni, metodi, attributi, ecc.
Quel che abbiamo visto non implica il fatto che non esista alcun modo per rendere tali dati persistenti. Innanzitutto, i dati persistenti sono dati conservabili permanentemente su memorie magnetiche, memorie flash, ecc. Detto ciò, introduciamo l'**Object Relational Mapping** (**ORM**), fornito da JPA, il quale interfaccia il mondo dei database al mondo degli oggetti. In realtà di ORM ne vedremo solamente il suo funzionamento in maniera estremamente generale: ci concentreremo in particolare sull'utilizzo delle query e sulla Persistence Unit.

3.1 INTRO ALLE ENTITÀ

Quando si parla di oggetti in relazione ai database, sarebbe più corretto utilizzare il termine "entità" anziché "oggetto". Gli oggetti sono istanze che vivono temporaneamente in memoria, mentre le entità sono istanze che possono essere rese persistenti nel database. È possibile rendere persistente un'entità, rimuoverla dal database o effettuare una query su di essa utilizzando il linguaggio **JPQL** (**Java Persistence Query Language**).
Un'entità, sostanzialmente, è un normalissimo POJO, quindi dovrà essere dichiarata, istanziata e utilizzata come qualsiasi altra classe Java. Ovviamente l'entità, come qualsiasi altra classe Java, avrà un suo stato e le sue caratteristiche saranno manipolabili tramite getter e setter.

3.2 ANATOMIA DI UN'ENTITÀ

Per essere entità, una classe deve essere annotata con **@Entity**, la quale annotazione permette al gestore della persistenza di riconoscere tale classe come classe persistente e non come semplice POJO.
Inoltre, un'entità deve possedere un identificativo, il quale distingue l'entità dalle altre.
Questo identificativo è specificabile tramite l'annotazione **@Id** e raffigura la primary key dell'entità.

```
@Entity
public class Book implements Serializable {

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

   public Book(){}

   // Getters and setters

}
```

In sostanza un'entità, per essere tale, deve seguire le seguenti regole:

- La classe dell'entità deve essere annotata da @Entity;
- L'entità deve possedere un identificativo, dichiarabile con @Id;
- L'entità deve possedere un costruttore vuoto senza alcun argomento. Ciò non implica il fatto che non possano esserci altri costruttori;
- L'entità non può essere un'enumerazione o un'interfaccia;
- La classe dell'entità non deve essere final, come nessun suo metodo o variabile;
- Se l'istanza dell'entità viene passata per valore come oggetto detached, la classe dell'entità deve implementare Serializable.

3.3 OBJECT RELATIONAL MAPPING (ORM)

Il principio di **ORM** è quello di delegare a tools o framework esterni (nel nostro caso di JPA) il compito di creare una corrispondenza tra oggetti e database. Di norma, è possibile estrarre i dati dagli oggetti per poi salvarli nelle tabelle dei database relazionali, ma nel caso si vogliano salvare le entità al posto delle tabelle, JPA mappa gli oggetti nel database tramite annotazioni e descrittori XML (XML descriptors). Senza alcuna annotazione, l'entità verrebbe trattata come un normale POJO e non sarebbe adatta ad un database, per cambiare tale comportamento, si annota la classe con **@Entity**, rendendola persistente e adatta ad un database.
Una semplice annotazione stravolge il comportamento che avrebbe il gestore della persistenza verso tale oggetto. Altro esempio è l'annotazione **@Id**, dove un attributo viene reso chiave primaria dell'entità.
Tali cambiamenti raffigurano l'approccio "**configuration-by-exception**", cioè vengono fatti dei cambiamenti alle regole di mapping di default:

- Il nome dell'entità rappresenta il nome della tabella nel database relazionale. Se si vuole mappare l'entità in un'altra tabella, è necessario utilizzare l'annotazione **@Table**;
- I nomi degli attributi rappresentano i nomi delle colonne. Se si vuole il comportamento di default, basta utilizzare l'annotazione **@Column**.

Esistono ovviamente tante altre regole, ma ne abbiamo viste solo un paio in modo da renderci conto di come un'annotazione possa cambiare le regole di mapping di default.

3.4 INTRO ALLE QUERY E ALL'ENTITY MANAGER

JPA permette di mappare le entità in database e di effettuare query su entità e le loro relazioni secondo diversi criteri, trattando tutto ciò con un approccio object-oriented, senza avere a che fare con righe e colonne. Tali query utilizzano JPQL, il quale è molto simile al normale SQL, con la differenza che la sua sintassi tratta oggetti e utilizza la notazione con il punto (.) per trattare attributi.

SELECT b FROM Book WHERE b.title = "H2G2"

Come ben vediamo non si trattano più le colonne delle tabelle, bensì si ricavano gli oggetti e si trattano gli attributi.

Le query JPQL possono essere create dinamicamente a runtime o staticamente a tempo di compilazione. Il seguente esempio mostra la query statica, cioè la **named query**.

```
@Entity
@NamedQuery(name = "Book.findBook_H2G2",
            query = "SELECT b FROM Book b WHERE b.title = 'H2G2'")
public class Book implements Serializable {

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

   public Book(){}

   // Getters and setters

}
```

Il responsabile della gestione delle entità è l'Entity Manager. Il suo ruolo è quello di gestire le entità secondo le operazioni CRUD (Create, Read, Update, Delete) utilizzando JPQL. È definibile come un'interfaccia da utilizzare per la gestione delle entità, la cui implementazione è fornita dal gestore della persistenza. In poche parole, l'Entity Manager rende persistenti le entità, le aggiorna e le rimuove dal database, inoltre, può effettuare query sulle entità utilizzando JPQL.
L'Entity Manager è ottenibile tramite un factory, come vediamo nel seguentecodice.

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("testPU");
EntityManager em = emf.createEntityManager();
```

Essendo le query eseguibili solamente dall'Entity Manager, si necessita forzatamente di esso quando si parla di persistenza.

Lo scopo dell'esempio è quello di mostrare come un libro viene salvato e ripreso dal database. Avendo un oggetto Book (il quale è un'entità, quindi annotato con @Entity), lo si vuole salvare nel database e, dopo il salvataggio, riprenderlo.
Per fare ciò sappiamo che è necessario l'Entity Manager, il quale è ricavabile da apposito factory. Ottenuto l'Entity Manager, si apre una transazione in cui si rende persistente l'oggetto, tramite il metodo persist, si salva l'oggetto nel database, quindi viene reso persistente. Fatto ciò, si vuole riprendere l'oggetto dal database, quindi si esegue la named query, la si crea tramite apposito metodo e la si esegue facendosi restituire il risultato. Fatto ciò, chiudiamo l'Entity Manager ed il suo factory.

```
public class Main {

   public static void main(String[] args) {

      //Creation of the book
      Book b = new Book("H2G2", "A guide to the galaxy", "1-84023-742-2",
                                                          12.5F, 354, false);
      //Creation of the Entity Manager
      EntityManagerFactory emf =
                          Persistence.createEntityManagerFactory("testPU");
      EntityManager em = emf.createEntityManager();
      //Saving the book in the database
      EntityTransaction et = em.getTransaction();
      et.begin();
      em.persist(b);
      et.commit();
      //Executing the named query
      b = em.createNamedQuery("Book.findBook_H2G2", Book.class).
                                                          getSingleResult();
      System.out.println("Book from database: " + b.getTitle());
      //Closing managers
      em.close();
      emf.close();

   }

}
```

3.5 PERSISTENCE UNIT

La **Persistence Unit** (**PU**) indica all'Entity Manager il tipo di database da utilizzare e i parametri di connessione, definiti nel file persistence.xml posto nella cartella META-INF.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" 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">
   <persistence-unit name="testPU" transaction-type="RESOURCE_LOCAL">
      <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
      <class>bookexercise.Book</class>
      <properties>
         <property
            name="javax.persistence.jdbc.url"
            value="jdbc:derby://localhost:1527/sample;create=true"/>
         <property
            name="javax.persistence.jdbc.driver"
            value="org.apache.derby.jdbc.ClientDriver"/>
         <property
            name="javax.persistence.schema-generation.database.action"
            value="drop-and-create"/>
         <property
            name="eclipselink.target-database" value="DERBY"/>
         <property
            name="javax.persistence.jdbc.user" value="APP"/>
         <property
            name="javax.persistence.jdbc.password" value="APP"/>
      </properties>
   </persistence-unit>
</persistence>
```

È bene sapere che l'ambiente in cui si lavora può essere application-managed e container-managed, se application-managed, vanno trascritte nel persistence.xml tutte le informazioni riguardanti il collegamento al database, proprio come nell'esempio appena visto, se container-managed, il discorso cambia siccome non si specificano più le informazioni riguardanti il collegamento, bensì si specifica il data source (il quale serve per collegarsi al database). Vediamo anche un esempio di persistence.xml per ambienti container-managed.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" 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">
   <persistence-unit name="testPU" transaction-type="JTA">
      <jta-data-source>java:global/jdbc/sample</jta-data-source>
      <class>bookexercise.Book</class>
      <properties>
         <property
            name="eclipselink.target-database" value="DERBY"/>
         <property
            name="eclipselink.ddl-generation"
            value="drop-and-create-tables"/>
      </properties>
   </persistence-unit>
</persistence>
```

3.6 CENNI SUL CICLO DI VITA DELLE ENTITÀ E LE CALLBACKS

Le entità non sono altro che POJO gestiti dall'Entity Manager, il quale permette di assegnare identificativi alle entità e permette al database di salvarne il loro stato. Nel caso le entità non siano gestite dall'Entity Manager, allora saranno considerate come normalissimi oggetti Java, quindi saranno POJO, e non avranno nulla a che fare con il database.
Quando si crea un'istanza di una determinata classe con l'operatore *new*, essa esisterà in memoria e JPA non saprà nulla riguardo la sua esistenza. Dal momento in cui l'Entity Manager inizia a gestire l'istanza creata, essa verrà mappata nel database e ne verrà sincronizzato il suo stato.
Un'entità può essere resa dall'Entity Manager persistente, aggiornata, rimossa e caricata, tali operazioni corrispondono alle operazioni database di inserimento, aggiornamento, rimozione e selezionamento.
Ogni operazione è caratterizzata da due eventi "pre" e "post" (tranne per il caricamento, il quale possiede solo l'evento "post"): l'evento "pre" si manifesta prima dell'operazione; l'evento "post" si manifesta dopo l'operazione. L'evento manifestante può essere intercettato dall'Entity Manager in modo da poter invocare metodi di business annotati da determinate annotazioni, in base al tipo di operazione che si sta eseguendo, ad esempio, per la resa persistente vengono utilizzate le annotazioni @PrePersist e @PostPersist.

4. GESTIONE DEGLI OGGETTI PERSISTENTI

In JPA, il servizio che manipola le entità è l'Entity Manager, il quale crea, trova, rimuove e sincronizza oggetti con il database. Inoltre, l'Entity Manager esegue molteplici tipi di query JPQL, in modo da poter operare con completa libertà sulle entità. JPQL è il linguaggio definito in JPA per effettuare le query su entità conservate nei database. La sintassi di tale linguaggio ricorda SQL, ma con un approccio Object Oriented siccome vengono trattati gli oggetti. Le query non fanno altro che gestire le entità secondo le operazioni CRUD (create, read, update, delete).

4.1 ENTITY MANAGER

L'Entity Manager è il pezzo centrale di JPA, gestisce il ciclo di vita delle entità ed esegue le query, trattando le entità nel database. Tale componente di JPA è responsabile riguardo la creazione e la rimozione di entità persistenti (ricordiamo che un'entità è detta persistente se si trova nel database) e riguardo la loro ricerca in base alla loro chiave primaria (ID). Quando un Entity Manager gestisce un'entità, quindi ne ottiene un riferimento ad essa, quest'ultima viene detta "**managed**", se un'entità non risulta managed, viene detta "**detached**". La differenza tra un'entità managed ed una detached consiste nella sincronizzazione con il database, la quale avviene solo per le entità managed.
L'Entity Manager è un'interfaccia implementata da un persistence provider, il quale genera ed esegue statements in SQL.

4.1.1 OTTENERE UN ENTITY MENAGER

L'**Entity Manager** è un'interfaccia utilizzata per interagire con le entità, ma per essere utilizzata deve essere ottenuta dall'applicazione, il modo in cui viene ottenuta dipende dal tipo di ambiente, il quale può essere gestito dal container o dall'applicazione (container-managed environment, application-managed environment). Ad esempio, nell'ambiente gestito dal container non c'è il bisogno di utilizzo dei metodi commit e rollback, a differenza dell'ambiente gestito dall'applicazione. In sostanza, il tipo di ambiente impatta sulla gestione dell'istanza dell'Entity Manager e sul ciclo di vita.

Nel seguente esempio vediamo come il programmatore è responsabile riguardo la creazione e la chiusura dell'Entity Manager (ciò rientra nella gestione del ciclo di vita di quest'ultimo).

```
public static void main(String[] args) {

   //Creation of the book
   Book b = new Book("H2G2", "A guide to the galaxy", "1-84023-742-2",
                                                        12.5F, 354, false);
   //Creation of the Entity Manager
   EntityManagerFactory emf =
                        Persistence.createEntityManagerFactory("testPU");
   EntityManager em = emf.createEntityManager();
   //Saving the book in the database
   EntityTransaction et = em.getTransaction();
   et.begin();
   em.persist(b);
   et.commit();
   //Closing managers
   em.close();
   emf.close();

}
```

La creazione di un Entity Manager in un ambiente application-managed è relativamente semplice, ma nel container-managed è un po' diverso, lo vediamo subito in un esempio:

```
@Stateless
public class BookEJB {

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

   public void createBook(){
      Book b = new Book("H2G2", "A guide to the galaxy", "1-84023-742-2",
                                                        12.5F, 354, false);
      em.persist(b);
   }

}
```

L'ottenimento dell'Entity Manager in ambienti container-managed consiste nell'utilizzo dell'annotazione **@PersistenceContext** o tramite JNDI lookup. Inoltre, nel container-managed non ci sarà il bisogno di creare o chiudere l'Entity Manager, siccome il suo ciclo di vita è gestito dal container. L'annotazione **@PersistenceContext** richiede come parametro il nome della Persistence Unit.

4.1.2 ENTITY MENAGED, DETACHED E RELAZIONE CON L'APPARTENENZA AL DATABESE

Il fatto che un'entità sia **managed** o **detached** non c'entra con la sua appartenenza al database. Il fatto che un'entità appartenga al database non implica che sia **managed** o **detached**.
Un'entità è detta managed se gestita dall'Entity Manager, quindi se ne possiede un riferimento ad essa, nel senso che in una transazione (o in generale) sta operando su di essa. Un'entità è detta detached se non gestita dall'Entity Manager.
Il metodo **persist** non salva l'oggetto nel database, bensì lo rende managed, le entità managed sono raccolte in un insieme detto persistence context, le entità managed vengono salvate nel database solamente quando il persistence context viene svuotato, se un'entità non appartiene al persistence context, essa allora sarà detached.

```
// et is the EntityTransaction instance
et.begin();
// b is detached
em.persist(b);
// b is managed
et.commit(); //flushing persistence context
// b is saved into the database
// b is now detached
```

L'entità *b* viene salvata nel database se la transazione va a buon fine, effettuerà un flush automatico del persistence context, liberandolo di ogni entità e salvandolo nel database. Se la transazione fallisce, viene effettuato un rollback, quindi si ritorna allo stato precedente all'inizio della transazione.

4.1.3 PERSISTANCE CONTEXT

Il **persistence context** (contesto di persistenza) può essere visto allo stesso tempo sia come un livello posto tra Entity Manager e database, sia come un insieme di entità gestite dove non compaiono doppioni per identificativi, ad esempio, se esiste un libro con ID pari a 12 nel persistence context, non ne potrà esistere un altro con lo stesso ID. Il persistence context contiene tutte le entità che risultano managed, quindi gestite dall'Entity Manager, i cambiamenti di stato effettuati su tali entità verranno riportati automaticamente nel database. L'Entity Manager, all'invocazione di metodi, consulta ed eventualmente aggiorna il persistence context, ad esempio, se viene chiamato il metodo persist, l'entità passata come argomento dovrà essere aggiunta al persistence context se non esista già al suo interno. Tale controllo viene effettuato, ovviamente, in base all'ID dell'entità.
Il **persistence context** può essere visto come una memoria cache che contiene tutte le entità in attesa di essere inserite nel database, le entità sono conservate in tale spazio di memoria per la durata di una transazione. Conclusa la transazione, viene effettuato un flush sul persistence context e tutte le sue entità vengono riposte nel database. L'annotazione **@PersistenceContext** viene utilizzata per ottenere un Entity Manager in ambienti container-managed, specificando la Persistence Unit da utilizzare, definita nel file persistence.xml posto nella cartella META-INF. Nel persistence.xml devono essere specificate tutte le entità che possono essere gestite nel persistence context. Inoltre, se si parla di ambienti container-managed, deve essere specificato anche il tipo di transazione come JTA (transaction-type="JTA").

4.1.4 RENDERE PERSISTENTE UN'ENTITÀ TRAMITE persist

Rendere persistente un'entità significa crearne un'istanza, settarle gli attributi e chiamare il metodo persist dell'Entity Manager. Tramite tale metodo, l'entità entrerà a far parte del persistence context, quindi verrà considerata managed.
Attenzione: l'utilizzo di tale metodo non implica che l'entità venga salvata nel database, ciò accade nel caso venga effettuato un flush sul persistence context.

```
// et is the EntityTransaction instance
et.begin();
// b is detached
em.persist(b);
// b is managed
et.commit(); //flushing persistence context
// b is saved into the database
// b is now detached
```

In tal caso, il metodo persist viene eseguito nel mezzo di una transazione, l'aggiunta al database sarà effettuata solo al commit della transazione siccome genera un flush sul persistence context.

4.1.5 RICAVARE UN'ENTITÀ DAL DATABASE TRAMITE find E getReference

Per ricavare un'entità dal database tramite il suo ID, stabilito tramite l'annotazione @Id, è possibile utilizzare due metodi differenti.
Il primo metodo consiste nell'utilizzo del metodo **find** dell'Entity Manager, il quale accetta due parametri: la classe dell'entità e l'identificativo da cercare.
Il secondo metodo consiste nell'utilizzo del metodo **getReference** dell'Entity Manager, il quale funziona come il metodo find con la differenza che ritorna un riferimento all'oggetto, senza ricavarne i suoi dati, fungendo da proxy.

```
Book returnedBook = em.find(Book.class, "12");
System.out.println(returnedBook.getTitle());
```

```
Book returnedBook;
try {
   returnedBook = em.getReference(Book.class, 12);
} catch (EntityNotFoundException e){
   System.err.println(e.getMessage());
}
```

4.1.6 RIMUOVERE UN'ENTITÀ DAL DATABASE TRAMITE remove

Un'entità può essere rimossa dal database tramite il metodo **remove** dell'Entity Manager. L'utilizzo di tale metodo al di fuori di una transazione rende l'entità passata come argomento detached, se si sta lavorando in una transazione, l'entità rimane managed finchè non viene effettuato il commit, siccome è lì che verrà registrata l'eliminazione e l'operazione di detach. Ovviamente l'entità viene rimossa dal database, quindi sarà comunque accessibile al di fuori di quest'ultimo.

```
//b is in the database
b = em.find(Book.class, 1);
// b is managed
// et is the EntityTransaction instance
et.begin();
em.remove(b);
// b is in the database
// b is managed
et.commit(); //flushing persistence context
// b is now removed from the database
// b is now detached
```

4.1.7 SINCRONIZZAZIONE COL DATABASE TRAMITE flush E refresh

Fino ad ora abbiamo visto della sincronizzazione con il database solamente il salvataggio dell'entità tramite flush automatico o la rimozione. In realtà, ci sono altri metodi che riguardano la sincronizzazione delle entità con il database. La commit di una transazione non fa altro che eseguire un flush di ciò che è conservato nel persistence context, salvando le entità presenti al suo interno.
Nel seguente esempio vengono effettuate due persist, quindi si vogliono salvare due entità, tali entità vengono inserite nel persistence context ma non nel database. Tramite la commit si effettua un flush sul persistence context, quindi le due entità verranno registrate nel database.

```
Book b1 = new Book("Guide to Java", "Java beginning tutorials",
                   "1-84023-742-2", 12.5F, 354, false);
Book b2 = new Book("Guide to Python", "Python beginning tutorials",
                   "1-84023-742-2", 12.5F, 354, false);
// et is the EntityTransaction instance
et.begin();
em.persist(b1);
em.persist(b2);
et.commit();
```

Volendo è possibile esplicitare il flush tramite un apposito metodo dell'Entity Manager chiamato **flush**. Utilizzando questo metodo, il persistence provider sarà esplicitamente forzato a flushare i dati nel database, ma ciò non committerà la transazione.

```
Book b1 = new Book("Guide to Java", "Java beginning tutorials",
                   "1-84023-742-2", 12.5F, 354, false);
Book b2 = new Book("Guide to Python", "Python beginning tutorials",
                   "1-84023-742-2", 12.5F, 354, false);
// et is the EntityTransaction instance
et.begin();
em.persist(b1);
em.flush();
em.persist(b2);
et.commit();
```

Quindi, se la flush registra i cambiamenti dalla normale esecuzione verso il database, allora ci sarà un metodo che registrerà i cambiamenti dal database alla normale esecuzione, cioè **refresh**, altro metodo dell'Entity Manager, il quale sovrascrive l'attuale stato dell'entità con i dati presenti nel database.

```
Book b1 = new Book("Guide to Java", "Java beginning tutorials",
                   "1-84023-742-2", 12.5F, 354, false);
// et is the EntityTransaction instance
et.begin();
em.persist(b1);
et.commit();
// b1 saved into the database
b1.setTitle("Guide to Python"); // New title of the book
em.refresh(b1);
// b1 will contain the old title of the book
System.out.println(b1.getTitle());
```

NOTA: la flush automatica delle transazione rende detached le entità, la flush forzata le rimane managed.

4.1.8 CONTENIMENTO DI UN'ISTANZA TRAMITE contains

Le entità, come ben sappiamo, possono essere gestite o meno dall'Entity Manager. L'Entity Manager fornisce un metodo, contains, il quale permette di controllare se una determinata entità risulta essere tra quelle gestite.

```
Book b1 = new Book("Guide to Java", "Java beginning tutorials",
                   "1-84023-742-2", 12.5F, 354, false);
// et is the EntityTransaction instance
et.begin();
em.persist(b1);
et.commit();
if (!em.contains(b1)) {
   System.err.println("Error while saving instance into DB");
}
```

4.1.9 RENDERE detach UN'ENTITÀ

È possibile manipolare lo stato delle entità portandolo da managed a detached tramite due metodi: **clear** permette di svuotare il persistence context rendendo ogni entità detached (ciò implica non vengano salvate nel database). Un'entità può essere resa detached anche in un altro modo, oltre all'utilizzo dei metodi clear e detach: quando si utilizzano le transazioni e si usa rendere le entità persistenti dentro esse (transaction scoped persistence context), il commit svuota il persistence context salvando le entità nel database e rendendole detached. Le entità diventano detached subito dopo il commit. Ciò non avviene con il flush forzato, tramite esecuzione dell'apposito metodo.

Nel seguente esempio viene riportato l'utilizzo del metodo detach.

```
Book b1 = new Book("Guide to Java", "Java beginning tutorials",
                   "1-84023-742-2", 12.5F, 354, false);
Book b2 = new Book("Guide to Python", "Python beginning tutorials",
                   "1-84023-742-2", 12.5F, 354, false);
// et is the EntityTransaction instance
et.begin();
em.persist(b1);
em.persist(b2);
em.detach(b2);
et.commit();
// b2 is not saved into the database
```

4.1.10 AGGIORNARE UN'ENTITÀ NEL DATABASE TRAMITE merge

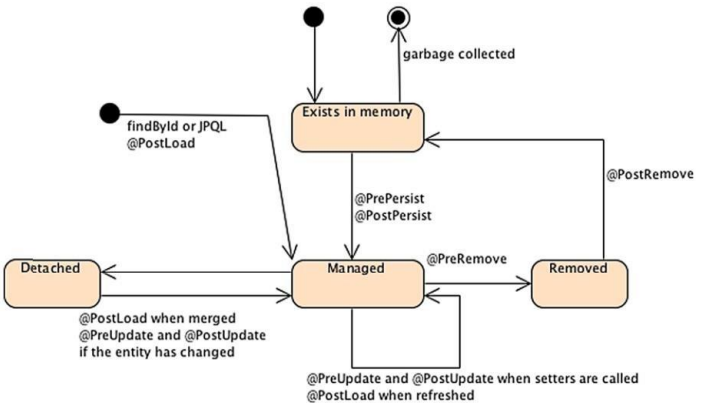
Se l'entità è managed allora gli aggiornamenti al database saranno automatici. Il problema si presenta se si ha a che fare con un'entità detached: in tal caso, viene utilizzato il metodo merge.

Il metodo **merge** si utilizza solamente per le entità che non sono più associate al persistence context, quindi solo per le entità detached di cui si ha la necessità di aggiornarla nel database, e serve ad apportare le modifiche apportate sull'entità nel database.

```
// et is the EntityTransaction instance
et.begin();
em.persist(b);
em.clear();
b.setTitle("MyBook");
em.merge(b);
et.commit();
```

4.2 CICLO DI VITA DELLE ENTITÀ

Viene mostrato uno schema che raffigura gli stati delle entità in base alle esecuzioni dei metodi dell'Entity Manager. Nello schema non si accennano la transazioni: è bene sapere che per la gestione delle entità non sono necessarie le transazioni. Il loro utilizzo dipende dal tipo di ambiente che si sta utilizzando. Riprendendo il capitolo 4.1.1, notiamo che nell'ambiente container-managed non vengono usate le transazioni per effettuare la commit, quindi cambiano i modi di liberare il persistence context. Sostanzialmente vale lo schema riportato; se si parla di transazioni, è bene sapere che il commit libera il persistence context, rendendo tutte le sue entità detached.



4.2.1 CALLBACKS

Il ciclo di vita delle entità si dirama in quattro categorie: persistenza, aggiornamento, rimozione e caricamento, le quali corrispondono alle operazioni di database di inserimento, aggiornamento, rimozione e selezione. Ogni diramazione del ciclo di vita ha degli eventi "pre" e "post" che possono essere intercettati dall'Entity Manager per invocare metodi di business che si desidera invocare in base agli eventi, devono essere annotati da determinate annotazioni, le quali dipendono dal tipo di evento che si sta verificando. Nello schema soprastante, vengono mostrate tutte le annotazioni utilizzabili in base all'evento verificato. I metodi callback vanno posti nella classe su cui si sta operando.

```
@Entity
public class Book implements Serializable {

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

   @PrePersist
   @PreUpdate
   private void checkTitle(){
      if (this.title == null || this.title.equals(""))
         throw new IllegalArgumentException("Invalid title");
   }

   // Here follows constructors, getter and setters

}
```

4.3 QUERY

L'Entity Manager è in grado di eseguire query scritte in JPQL: ne esistono ben 5 tipi, ognuna con un funzionamento diverso.

Per eseguire query di selezione, quindi che utilizzino SELECT, è possibile scegliere tra due metodi in base al result che si vuole ottenere, il metodo **getResultList** esegue la query e ritorna una lista di risultati, il metodo **getSingleResult** esegue la query e ritorna un singolo risultato (lancia un'eccezione se trova molteplici risultati).

Per eseguire query di aggiornamento, quindi che utilizzino UPDATE o DELETE, si utilizza il metodo **executeUpdate**, il quale ritorna il numero di entità affette dall'aggiornamento.

La creazione di una query si utilizza il metodo **createQuery** dell'Entity Manager, il quale accetta come parametro la query in JPQL. L'esecuzione di tale metodo ritorna un oggetto di tipo Query o TypedQuery, in base al numero di argomenti passati.

Le query caratterizzate dal WHERE possono essere caratterizzate da parametri variabili, i quali possono essere specificati in due modi: tramite i parametri posizionali ed i parametri con nome. I parametri posizionali sono caratterizzati da un punto di domanda seguito da un intero, il quale parte da 1 e continua a crescere andando avanti nella query.

```
SELECT b
FROM Book b
WHERE c.title = ?1
   AND c.price = ?2
```

I parametri con nome sono caratterizzati da un carattere di due punti seguito dal nome del parametro, il quale verrà utilizzato per impostare un valore.

```
SELECT b
FROM Book b
WHERE c.title = :booktitle
   AND c.price = :bookprice
```

Per impostare valori ai parametri che abbiamo appena visto, basta utilizzare il metodo **setParameter**, caratterizzato da due argomenti: il primo argomento indica il parametro su cui impostare il valore, mentre il secondo indica il valore da impostare.

Il primo esempio che vediamo mostra come vengono impostati i parametri posizionali:

```
public Book getBookByTitlePrice(
        EntityManager em, String title, Float price){
   TypedQuery<Book> buildQuery = em.createQuery(
           "SELECT b FROM Book b WHERE b.title = ?1 AND b.price = ?2",
           Book.class);
   buildQuery.setParameter(1, title);
   buildQuery.setParameter(2, price);
   return buildQuery.getSingleResult();
}
```

Il secondo mostra come vengono impostati i parametri con nome.

```
public Book getBookByTitlePrice(
        EntityManager em, String title, Float price){
    TypedQuery<Book> buildQuery = em.createQuery(
            "SELECT b FROM Book b "
                + "WHERE b.title = :booktitle AND b.price = :bookprice",
            Book.class);
    buildQuery.setParameter("booktitle", title);
    buildQuery.setParameter("bookprice", price);
    return buildQuery.getSingleResult();
}
```

Inoltre, è bene sapere che è possibile gestire le query in modo tipizzato e non. Quando si crea una query essa viene conservata in una variabile Query o TypedQuery: se si esegue la query, essa ritornerà un tipo di oggetto in dipendenza dalla variabile scelta per contenerla. Se utilizza Query, l'esecuzione ritornerà una lista di oggetti non tipizzati; se si utilizza TypedQuery<TYPE>, l'esecuzione ritornerà una lista di oggetti con tipo specificato TYPE. L'utilizzo di TypedQuery richiede un argomento aggiuntivo nel metodo **createQuery**: sarà necessario passare come secondo parametro la classe del tipo TYPE.

4.3.1 DYNAMIC QUERY

Le **dynamic query** vengono definite e tradotte in SQL al momento durante l'esecuzione dell'applicazione. Sono dette dinamiche proprio perché possono essere costruite tramite concatenazione a runtime, anche in base a determinati controlli e condizioni.

```
public Object dynamicQuery(
        EntityManager em, String title, Float price){
    String q = "SELECT b FROM Book b WHERE b.title = :booktitle";
    if (price > 20) q += " AND c.price = :bookprice";
    Query objQuery = em.createQuery(q);
    objQuery.setParameter("booktitle", title);
    if (price > 20) objQuery.setParameter("bookprice", price);
    return objQuery.getSingleResult();
}
```

4.3.2 NAMED QUERY

Le **named query** sono statiche e non possono cambiare per alcun motivo, quindi non sono per nulla flessibili come le query dinamiche. Il loro essere statiche porta ad essere decisamente efficienti siccome il loro JPQL viene tradotto in SQL non appena l'applicazione parte, invece di tradurre a runtime.

Le **named query** vengono specificate nell'annotazione **@NamedQuery** posta davanti alla definizione della classe di ritorno. Se la classe di ritorno possiede molteplici query, allora le annotazioni vengono raccolte in un'unica annotazione **@NamedQueries**.
L'annotazione **@NamedQuery** richiede due parametri: il primo è il nome della query, il secondo è la query stessa. L'annotazione **@NamedQueries**, invece, richiede come parametro tutte le **@NamedQuery**, raccolte tra le parentesi { }.

```
@Entity
@NamedQueries({
    //First query
    @NamedQuery(name = "Book.findAll",
                query = "select b from Book b"),
    //Second query
    @NamedQuery(name = "Book.findByTitle",
                query = "select b from Book b "
                    + "where b.title = :booktitle"),
    //Third query
    @NamedQuery(name = "Book.findByLowerPrice",
                query = "select b from Book b "
                    + "where b.price < ?1")
})
public class Book implements Serializable {
```

Per convenzione si usa adottare la seguente sintassi per il nome completo della query: ClassName.QueryName, dove ClassName è il nome della classe a cui si fa riferimento e QueryName è il nome reale della query. È una convenzione abbastanza obbligata, siccome potrebbero esistere molteplici query con nome "findAll", mentre ne esisterà una con nome "Book.findAll". Altra convenzione è quella di stabilire il nome della query come costante della classe, in modo da favorire modularità per cambi.

```
@Entity
@NamedQuery(name = Book.FIND_BY_TITLE,
            query = "select b from Book b "
                + "where b.title = :booktitle")
public class Book implements Serializable {
    public static final String FIND_BY_TITLE = "Book.findByTitle";
```

Definita la query, si passa alla creazione dell'oggetto che gestisce la query: in generale si sarebbe utilizzato il metodo createQuery con parametro la query in JPQL. Per le named query deve essere utilizzato il metodo **createNamedQuery**, anch'esso dell'Entity Manager, il quale accetta come parametro il nome stabilito della query. Nel caso si voglia gestire un TypedQuery, come già visto, vale anche qui il parametro addizionale. La gestione della query è invariata, cambia solamente la sua creazione.

```
public Book namedQuery(
        EntityManager em, String title){
    TypedQuery<Book> objQuery = em.createNamedQuery(
            "Book.findByTitle", Book.class);
    objQuery.setParameter("booktitle", title);
    return objQuery.getSingleResult();
}
```

4.3.3 NATIVE QUERY

Questo tipo di query opera su sorgente SQL e la sua creazione ritorna sempre un oggetto Query (quindi con le native query non è possibile utilizzare TypedQuery). Per creare una native query deve essere utilizzato il metodo **createNativeQuery**, appartenente all'Entity Manager, il quale richiede due parametri: la query in SQL e la classe a cui la tabella specificata fa riferimento. La gestione della query è invariata.

```
public Object nativeQuery(
        EntityManager em, String title){
    Query objQuery = em.createNativeQuery(
            "SELECT * FROM table_book", Book.class);
    objQuery.setMaxResults(10);
    return objQuery.getResultList();
}
```

Native Named Query:
Variante delle native query sono le native named query, le quali sono un ibrido tra le native query e le named query.
Vengono specificate proprio come le named query, con la differenza che l'annotazione utilizzata è **@NamedNativeQuery**, la quale prende come parametri il nome della query e la query vera e propria in SQL. Inoltre, va specificata tramite l'annotazione @Table il nome della tabella a cui la classe che segue fa riferimento, in modo da mapparla alla tabella trattata nella query in SQL.

```
@Entity
@NamedNativeQuery(name = Book.FIND_ALL,
                  query = "SELECT * FROM table_book")
@Table(name = "table_book")
public class Book implements Serializable {
    public static final String FIND_ALL = "Book.findAll";
```

Ovviamente nel caso si cerchi di fare qualcosa di più elaborato, bisognerà mettere mano ad annotazioni di mapping con database relazionali.
Per raccogliere molteplici annotazioni **@NamedNativeQuery**, proprio come con le named query, si utilizza l'annotazione **@NamedNativeQueries**.

5. ENTERPRISE JAVABEANS

Per separare il layer di persistenza dal layer di business ricorriamo agli **Enterprise JavaBeans** (**EJB**). Gli EJB sono componenti a lato server che incapsulano la logica di business, si prendono cura di transazioni e sicurezza e possiedono molteplici servizi integrati, come uno stack per messaggi, scheduling, accesso remoto, web service endpoints (SOAP e REST, li vedremo più avanti), dependency injection, ciclo di vita dei componenti, interceptors e tanto altro. Il fatto che gli EJB si integrino bene con altre tecnologie, fa sì che essi siano ottimi componenti per il livello di logica business. Un EJB Container è un ambiente che provvede servizi come la gestione delle transazioni, controllo della concorrenza, pooling, autorizzazioni di sicurezza, ecc. Tale container, permette al programmatore di scrivere codice senza doversi preoccupare di diversi aspetti del codice che scrive.

5.1 SERVIZI OFFERTI DAL CONTAINER

Il container offre molteplici servizi alle applicazioni enterprise:

- **Comunicazione con client remoto**: senza scrivere alcun codice, un client EJB può invocare metodi in remoto;
- **Dependency Injection**: il container può iniettare risorse negli EJB, così come con qualsiasi altro POJO grazie a CDI;
- **Gestione dello stato**: il container gestisce in maniera del tutto trasparente gli stati dei session bean;
- **Pooling**: il container crea un pool di istanze di stateless bean e di MDB condiviso tra molteplici client. Una volta invocato, un EJB ritorna nel pool per poter esser riutilizzato;
- **Ciclo di vita delle componenti**: il container è responsabile riguardo la gestione del ciclo di vita di ogni componente;
- **Gestione delle transazioni**: un EJB può utilizzare le annotazioni per informare il container riguardo la policy da utilizzare. Inoltre, il container gestisce i commit e i rollback;
- **Supporto alla concorrenza**: eccetto per i singleton, tutti gli altri tipi di EJB sono thred-safe, quindi è possibile sviluppare applicazioni senza preoccuparsi di eventuali problemi con i thread;
- **Automatizzazione degli interceptor**: è possibile creare interceptor invocabili automaticamente dal container.

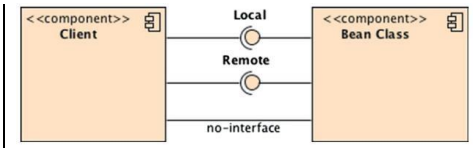
Una volta deployato l'EJB, il container si prenderà cura dei servizi appena descritti, in modo da non far preoccupare il programmatore riguardo aspetti che non siano lo scrivere il codice di business. Gli EJB sono oggetti gestiti, quindi rientrano tra i Managed Bean, che vivono in un container dalla loro creazione alla loro distruzione. Il container fornisce servizi agli EJB, i quali sono comunque limitati verso determinate operazioni, ad esempio, non possono gestire thread, accedere a files utilizzando Java.io, creare ServerSocket e così via.

5.2 ANATOMIA DI UN EJB

Abbiamo visto quanto è potente un EJB, ma non abbiamo visto quanto sia semplice crearlo: basta una classe Java e un'annotazione.

```
@Stateless
public class BookEJB {

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

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

   public Book persistBook(Book b){
      em.persist(b);
      return b;
   }

}
```

È un esempio decisamente banale e semplice, difatti in base ai propri bisogni si può costruire un EJB più ricco che permetta di effettuare chiamate remote, dependency injection e tanto altro. Un EJB è generalmente formato da una classe bean e da un'interfaccia: la classe bean contiene implementazioni di metodi di business e deve essere annotata da una delle tre seguenti annotazioni: **@Stateless**, **@Stateful** o **@Singleton**, in base al tipo di cui si necessita, l'interfaccia contiene la dichiarazione dei metodi di business visibili al client e implementati nella classe bean che compone l'EJB. Un session bean può avere interfacce locali o remote, addirittura nessuna (solo in accesso locale).



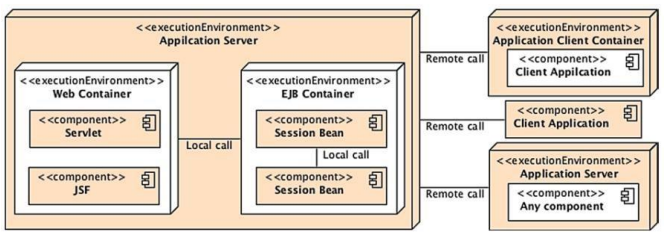
5.2.1 CLASSE BEAN DELL'EJB

L'EJB è composto da una classe bean e da eventuali interfacce. La classe bean che lo compone deve rispettare alcuni criteri:

- Il bean deve essere annotato con @Stateless, @Stateful o @Singleton;
- Se implementa delle interfacce, il bean deve implementarne i metodi;
- Il bean deve essere definito come public;
- Il bean deve possedere un costruttore senza argomenti, il quale verrà utilizzato dal container per crearne delle istanze;
- Il bean non deve definire il metodo finalize;
- I metodi presenti nel bean non devono essere final o static;
- I nomi dei metodi presenti nel bean non devono cominciare con "ejb".

5.2.2 LOCAL E REMOTE, INTERFACCE RIGUARDANTI GLI EJB

La classe bean che compone l'EJB, deve implementare i metodi delle interfacce associate, nel caso ce ne siano. Nel caso il sistema abbia dei client fuori dall'EJB container dell'istanza JVM, allora essi dovranno utilizzare un'interfaccia remota per comunicare. Nel caso, invece, i client e i bean si trovino sulla stessa istanza JVM, allora sarà possibile utilizzare un'interfaccia locale per comunicare.



L'interfaccia che il bean dovrà implementare, definisce molteplici metodi business che l'applicazione client potrà utilizzare. L'interfaccia, in base al fatto se sia locale o remota, deve essere annotata con una delle due annotazioni: **@Remote** denota un'interfaccia remota; **@Local** denota un'interfaccia locale.

```
@Remote
public interface BookEJBRemote {
    public Book persistBook(Book b);
}
```

Vediamo, invece, l'EJB che implementa l'interfaccia:

```
@Stateless
public class BookEJB implements BookEJBRemote {

    @PersistenceContext(name = "testPU")
    EntityManager em;

    @Override
    public Book persistBook(Book b){
        em.persist(b);
        return b;
    }

}
```

Annotazione importante è la **@LocalBean**: è un'annotazione che viene aggiunta in background in automatico e serve a far risultare visibile il bean in locale. Il problema è che tale annotazione viene aggiunta in automatico solo se il bean non implementa interfacce e risulta necessario inserirla.

5.3 SERVIZIO DI NAMING JNDI

JNDI è una API che permette ai client di scoprire e di ottenere oggetti tramite un semplice nome. I nomi di JNDI seguono la seguente sintassi:

java : SCOPE [/APP-NAME] /MODULE-NAME /BEAN-NAME [!INTERFACE-PACKAGE.INTERFACE-NAME]

Analizziamo ogni porzione del nome JNDI:

- **SCOPE** definisce la visibilità della risorsa a cui è associato il nome JNDI. Può assumere uno dei seguenti valori:
  - ***global***, permette alla componente di essere eseguita fuori dall'applicazione, accedendo allo spazio dei nomi globale;
  - ***app***, imposta lo scope all'applicazione;
  - ***module***, imposta lo scope al modulo dell'applicazione;
  - ***comp***, imposta lo scope alla componente, quindi tale nome non sarà accessibile dalle altre componenti;
- **APP-NAME** è facoltativo, è necessario solo se il session bean è compresso in un file ear o in un file war. In tal caso, avrà come valore il nome del file ear o war in cui è contenuto, senza l'estensione .ear o .war;
- **MODULE-NAME** è il nome del modulo nel quale il session bean si trova;
- **BEAN-NAME** è il nome del session bean;
- **INTERFACE-PACKAGE.INTERFACE-NAME** è necessario se il bean implementa delle interfacce. Qui vanno specificate, quindi, tutte le interfacce implementate dal bean specificato in BEAN-NAME.
  - ***INTERFACE-PACKAGE*** indica il nome del package in cui è contenuta l'interfaccia;
  - ***INTERFACE-NAME*** indica l'interfaccia vera e propria.

JNDI è una componente di Java SE ed è fondamentale nel caso i client debbano utilizzare metodi degli EJB. JNDI non ha nulla a che vedere con la @Inject. JNDI non ritorna oggetti, anche perché una applicazione Java SE non potrebbe gestire gli EJB! Si fa prima a vedere un esempio per comprenderne il reale funzionamento:

Nello scope del client abbiamo l'interfaccia remota HelloWorldBeanRemote, in modo da poter richiamare metodi sull'oggetto posto sul server HelloWorldBean, il quale è un EJB. Essendo, quella del client, una applicazione Java SE, essa non può trattare oggetti HelloWorldBean, quindi EJB. Nel main dell'applicazione, cerchiamo un riferimento all'oggetto HelloWorldBean, senza tornarne il vero e proprio oggetto, siccome sarebbe ingestibile. In tal modo, potremo utilizzare l'interfaccia come ponte per invocare metodi sull'EJB posto sul server. È questa la differenza tra iniezione e lookup JNDI.

```
public class HelloWorldClient {

    public static void main(String[] args) throws NamingException {

        Context ctx = new InitialContext();
        // lookup string
        // java:global/EJBProjectName/EJBBean!interfacePackage.interface
        HelloWorldBeanRemote helloBean = (HelloWorldBeanRemote)
                ctx.lookup("java:global/HelloWorldEJB/"
                        + "HelloWorldBean!allejbs.HelloWorldBeanRemote");
        System.out.println(helloBean.greetings("Francesco"));

    }

}
```

5.4 SESSION BEAN

Un **session bean** è un tipo di EJB utilizzato per modellare codice di business. Il termine session si riferisce al ciclo di vita del componente, siccome il container utilizza tali oggetti durante una sessione utente, creandoli e distruggendoli dinamicamente. Non sono bean adatti alla persistenza: difatti, per quel caso, abbiamo gli entity bean (le entità). Il fatto che non siano adatti alla persistenza non significa che non possano utilizzare entità per utilizzare le informazioni persistenti. Gli EJB sono componenti gestiti dal container, quindi devono essere necessariamente eseguiti in un container.
Nota: ricordiamo che gli EJB sono componenti gestite dal container, quindi l'Entity Manager va iniettato tramite l'annotazione @PersistenceContext, specificando il nome della persistence unit.

5.4.1 STATELESS BEAN

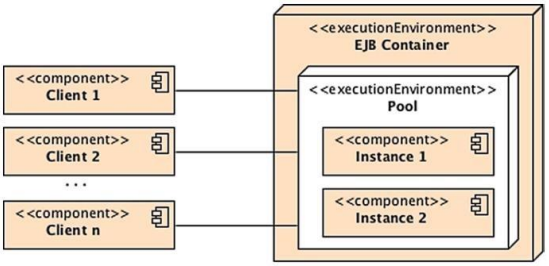
Gli **stateless bean** sono una tipologia di session bean decisamente efficiente, siccome fanno uso del pooling del container e possono essere condivisi tra molteplici client. Stateless significa senza stato, quindi sono bean che non conservano alcuna informazione: contengono solamente operazioni business. Cambiano totalmente la visione di programmazione che avevano fino ad oggi: se prima utilizzavamo un metodo del POJO stesso per salvarlo nel database, ora tale metodo viene invocato su uno stateless bean proprio perché non necessita di intaccare alcuno stato. Nel seguente esempio, BookEJB è uno stateless bean.

```
public static void main(String[] args) throws NamingException {

    Context ct = new InitialContext();
    BookEJBRemote stateless = (BookEJBRemote)
            ct.lookup("java:global/HelloWorldEJB/"
                    + "BookEJB!allejbs.BookEJBRemote");

    Book b = new Book();
    b.setTitle("Java guide");
    b.setDescription("A very good guide to Java");
    b.setPrice(50F);
    b.setIsbn("1-84023-742-2");
    stateless.persistToDatabase(b);

}
```

Di seguito vediamo l'implementazione dello stateless bean, il quale deve essere annotato da **@Stateless**.

```
@Stateless
public class BookEJB implements BookEJBRemote {

   @PersistenceContext(name = "testPU")
   EntityManager em;

   @Override
   public void persistToDatabase(Book b){
      em.persist(b);
   }

}
```

Il fatto che gli stateless bean siano soggetti al pool del container, si intende che per ogni stateless bean ne vengono conservate un certo numero di istanze in memoria e condivise tra i clients. Proprio per il fatto che gli stateless bean non hanno uno stato, ogni istanza è equivalente. Quando un client necessita dell'invocazione di un metodo su uno stateless bean, il container prende un'istanza dal pool e la assegna al client. Conclusa l'invocazione, l'istanza ritorna nel pool in attesa di essere riutilizzata. Ciò implica che poche istanze di stateless bean possono gestire un bel po' di clients.



5.4.2 STATEFUL BEAN

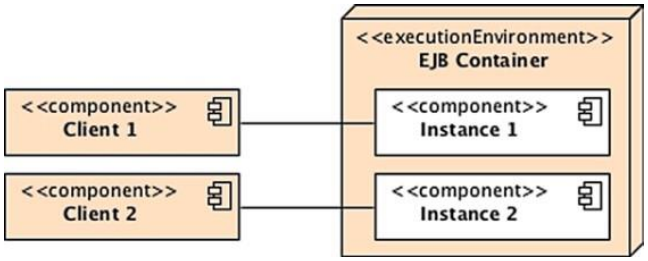
Gli **stateful bean** sono EJB che conservano lo stato e sono una tipologia di session bean in grado di mantenere lo stato, sono utili nel caso si debba eseguire una serie di compiti di cui se ne deve mantenere lo stato. Un esempio perfetto è il carrello in un sistema di acquisto online, l'utente sceglie un oggetto da comprare e quest'ultimo viene salvato nel carrello, il quale è lo stateful bean che memorizza tutti gli oggetti a cui associati. Nel seguente esempio, BookEJB è uno stateful bean.

```
public static void main(String[] args) throws NamingException {

   Context ct = new InitialContext();
   BookEJBRemote stateful = (BookEJBRemote)
         ct.lookup("java:global/HelloWorldEJB/"
               + "BookEJB!allejbs.BookEJBRemote");

   Book b1 = new Book();
   b1.setTitle("Java guide");
   b1.setDescription("A very good guide to Java");
   b1.setPrice(50F);
   b1.setIsbn("1-84023-742-2");
   Book b2 = new Book();
   b2.setTitle("Python guide");
   b2.setDescription("A very good guide to Python");
   b2.setPrice(35.5F);
   b2.setIsbn("1-84023-684-5");

   stateful.addBook(b1);
   stateful.addBook(b2);
   stateful.removeBook(b1);
   stateful.checkout();

}
```

Di seguito vediamo l'implementazione dello stateful bean, il quale deve essere annotato da **@Stateful**.

```
@Stateful
@StatefulTimeout(value = 20, unit = TimeUnit.SECONDS)
public class BookEJB implements BookEJBRemote {

   ArrayList<Book> cartBook = new ArrayList<>();

   @Override
   public void addBook(Book b) {
      if (!cartBook.contains(b))
         cartBook.add(b);
   }

   @Override
   public void removeBook(Book b) {
      if (cartBook.contains(b))
         cartBook.remove(b);
   }

   @Override
   @Remove
   public void checkout() {
      cartBook.clear();
   }

}
```

Oltre all'utilizzo di **@Stateful**, notiamo l'utilizzo di altre due annotazioni, entrambe opzionali: **@StatefulTimeout**, che stabilisce quanto tempo è permesso rimanere in idle al bean, specificando un valore numerico e un valore che indica l'unità di tempo utilizzando java.util.concurrent.TimeUnit, e **@Remove** va posta su un metodo e permette di rimuovere permanentemente il bean una volta conclusa l'esecuzione del metodo.
Queste sono annotazioni alternative: ciò viene già gestito dal container, quindi già quest'ultimo provvederebbe alla rimozione di tali bean se necessario. Inoltre, per gli stateful bean non è presente un pool, il riutilizzo di tali bean per altri clients sarebbe impossibile siccome mantengono uno stato. Di conseguenza ad ogni client corrisponde uno stateful bean.



5.4.3 SINGLETON BEAN

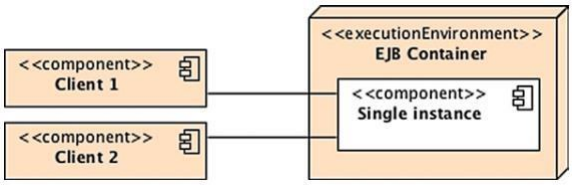
Il **singleton bean** è una tipologia di session bean istanziato un'unica volta per applicazione: sostanzialmente, è un bean che implementa il Singleton Pattern della Gang of Four. Tale bean assicura che esista un'unica istanza di una classe in tutta l'applicazione e provvede l'accesso globale a quest'ultima.

Nel seguente esempio, DatabasePopulator è un singleton bean, di cui vediamo direttamente l'implementazione. Ricordiamo che tale bean deve essere annotato da **@Singleton**.

```
@Singleton
@Startup
@DataSourceDefinition(
   className = "org.apache.derby.jdbc.ClientDataSource",
   name = "java:global/jdbc/sample",
   user = "app",
   password = "app",
   databaseName = "sample",
   properties = {
      "connectionAttributes=;create=true"
   }
)
public class DatabasePopulator {

   @Inject
   private BookEJB manager;
   private Book first, second;

   @PostConstruct
   public void populateDatabase(){
      first = new Book("Java Guide", "Very good guide", "1-8763-9125-7",
                       60F, 580, true);
      second = new Book("Python Guide", "Very bad guide", "1-8763-9143-8",
                       45.8F, 320, false);
      manager.persistToDatabase(first);
      manager.persistToDatabase(second);
   }

   @PreDestroy
   public void destroyDatabase(){
      manager.removeFromDatabase(first);
      manager.removeFromDatabase(second);
   }

}
```

Le annotazioni **@Startup** e **@DataSourceDefinition** non sono obbligatorie: @Startup permette di inizializzare il bean quando si avvia il sistema, @DataSourceDefinition specifica alcuni parametri riguardanti la connessione con il database. Inoltre, tale classe fa uso di annotazioni pre e post che rendono i metodi sottostanti dei callback.

Ritornando al singleton, l'istanza è contesa tra tutti i client e non mantiene alcuno stato tra le invocazioni. Ergo, può necessitare di attenzione sull'aspetto concorrenza.



5.5 DEPENDENCY INJECTION

Nell'ambito dei container EJB esistono molteplici annotazioni che permettono l'iniezione di risorse.

- @EJB inietta un EJB;
- @PersistenceContext inietta un EntityManager (vedi capitolo 4.1.1);
- @WebServiceRef inietta un riferimento ad un web service;
- @Resource inietta diversi tipi di risorse;
- @Inject inietta determinate implementazioni di interfacce (vedi capitolo 2.9.1).

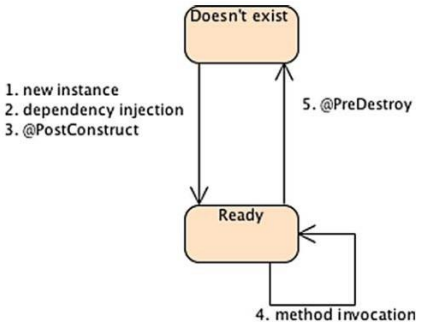
5.6 CICLO DI VITA DI UN SESSION BEAN

I session bean sono componenti gestiti dal container, quindi vivono in un container EJB che offre determinati servizi. Uno dei tanti servizi che offre è la gestione del ciclo di vita dei bean. In base al tipo di bean che si sta trattando (stateless, stateful, singleton), il ciclo di vita consisterà di diversi stati. Ogni volta che il container cambia il ciclo di vita di un bean, è possibile invocare metodi annotati detti callback.
Negli esempi precedenti abbiamo notato che il client non crea mai istanze di session bean utilizzando l'operatore new: ne prende un riferimento tramite JNDI lookup. Sostanzialmente, tramite questa richiesta, il container ne crea un'istanza e ne continua a gestire il ciclo di vita.

5.6.1 CICLO DI VITA PER STATELESS E SINGLETON BEAN

Gli stateless ed i singleton bean hanno in comune il fatto che non mantengono uno stato. Entrambi condividono lo stesso ciclo di vita, il quale è:

1. Il ciclo di vita inizia quando un client richiede un riferimento ad un bean tramite JNDI lookup. Nel caso si tratti di un singleton, il ciclo può iniziare anche tramite l'annotazione @Startup, all'avvio del container.
2. Se i bean creati utilizzano DI o deployment descriptors, il container inietta tutte le risorse richieste.
3. Se l'istanza ha un metodo annotato con @PostConstruct, il container lo invoca.
4. L'istanza del bean processa eventuali chiamate a metodi dal client e resta comunque in attesa per altre chiamate. Lo stateless bean resta in attesa finchè non verrà rimosso dal pool del container; il singleton bean resta in attesa finchè non si chiuderà il container.
5. Al container non serve più il bean: invoca i metodi annotati con @PreDestroy e conclude il ciclo di vita.

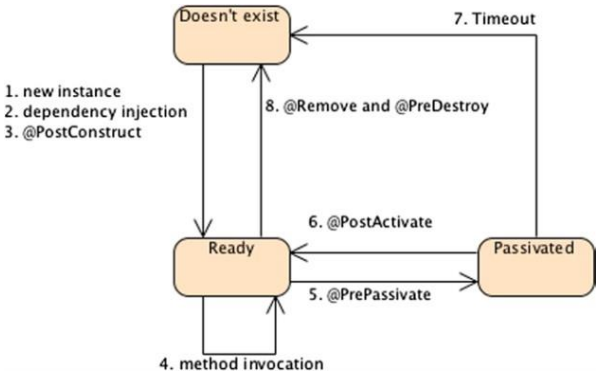


Concludendo, il ciclo di vita dello stateless bean e del singleton bean sono fondamentalmente uguali, con la differenza che lo stateless bean rimane in vita finchè il container non lo rimuove dal pool, mentre il singleton bean rimane in vita per tutta la durata della vita del container.

5.6.2 CICLO DI VITA PER STATEFUL BEAN

Gli stateful bean mantengono un proprio stato, vengono generati e vengono assegnati uno ad uno con i client. Se il client non invoca metodi sul bean per diverso tempo, il container lo rimuoverà prima che la JVM finisca la propria memoria, preservando l'attuale stato in uno storage permanente. Ciò comporta ad avere un ciclo di vita ben diverso dai precedenti bean.
Il fatto che lo stato venga salvato in uno storage comporta alla presenza di due condizioni: passivazione e attivazione. La passivazione è quando il container serializza l'istanza del bean in uno storage permanente, mentre l'attivazione è quando di deserializza il bean, cioè dallo storage si recupera il bean e lo si riporta in memoria principale.

1. Il ciclo di vita inizia quando un client richiede un riferimento ad un bean tramite JNDI lookup. Il container crea un nuovo session bean e lo conserva in memoria.
2. Se i bean creati utilizzano DI o deployment descriptors, il container inietta tutte le risorse richieste.
3. Se l'istanza ha un metodo annotato con @PostConstruct, il container lo invoca.
4. Il bean esegue le chiamate richieste dal client e resta in memoria, in attesa di altre richieste.
5. Se il client rimane in idle per un lungo periodo di tempo, il container invoca il metodo annotato con @PrePassivate e passiva il bean in uno storage.
6. Se il client invoca un bean passivato, il container lo attiva e invoca il metodo annotato con @PostActivate.
   - Se il client non invoca il bean passivato per il tempo di timeout della sessione, allora il container lo distrugge definitivamente.
   - Alternativamente, se il client invoca il metodo annotato da @Remote, il container invoca il metodo annotato con @PreDestroy e conclude il ciclo di vita del bean.



5.6.3 CALLBACKS

In base al tipo di session bean varia il ciclo di vita gestito dal container. Il container permette di poter eseguire del codice in base all'evento che si presenta riguardo il ciclo di vita: un cambio nel ciclo di vita può essere intercettato dal container, il quale invocherà metodi annotati da determinate annotazioni, le quali dipendono dal tipo di evento verificato. Un metodo, per essere **callback**, deve rispettare determinate regole:

- Il metodo non deve avere parametri e deve ritornare void;
- Il metodo non deve lanciare eccezioni checked. Se lancia eccezioni runtime, effettuerà il rollback della transazione, se esiste (parleremo delle transazioni nel prossimocapitolo);
- Il metodo non può essere static o final;
- Un metodo può essere annotato da molteplici annotazioni inerenti agli eventi, ma un'annotazione non può essere applicata a molteplici metodi.

Di solito, i callback vengono utilizzati per allocare o rilasciare risorse dei bean.

6. TRANSAZIONI

Le **transazioni** sono un servizio fornito dal container che assicura lo svolgimento di determinate operazioni in maniera del tutto affidabile. Le transazioni possono essere viste come un gruppo di operazioni eseguite sequenzialmente dove ogni operazione deve andare a buon fine, altrimenti nessuna viene eseguita: se le operazioni vanno a buon fine, la transazione si dice committata, se almeno un'operazione fallisce, la transazione si dice rolled back, siccome annulla l'esecuzione dei metodi andati a buon fine. Una **transazione**, per poter esser definita tale, deve poter essere:

- **Atomica**, perché è composta da una o più operazioni raggruppate in un'unica unità di lavoro, delle quali o vanno tutte a buon fine o falliscono tutte;
- **Consistente**, perché alla conclusione di essa, i dati rimangono consistenti;
- **Isolata**, perché lo stato di una transazione non è visibile da applicazioni esterne;
- **Durabile**, perché eventuali cambiamenti apportati dalla transazione, dopo il commit, saranno visibile anche ad altre applicazioni.

6.1 TIPI DI LETTURA

Può capitare, per qualche motivo, che due o più transazioni leggano gli stessi dati nello stesso momento. L'operazione di lettura si presenta quando una transazione gestisce una risorsa, siccome la legge prima di operarci. In base al livello di isolamento della transazione, si possono avere problemi di accesso concorrente, i quali vengono classificati in base al tipo di lettura che si assume:

- Le ***dirty reads*** sono una tipologia di lettura che indica che una transazione legge cambiamenti su una o più risorse non committati dalla precedente transazione;
- Le ***repeatable reads*** sono una tipologia di lettura che indica la situazione in cui unarisorsa,
- tra una lettura e l'altra nella stessa transazione, non è cambiata;
- Le ***phantom reads*** sono una tipologia di lettura che indica il fatto in cui con una transazione si salvano risorse nel database, visibili anche alle transazioni iniziate prima dell'attuale.

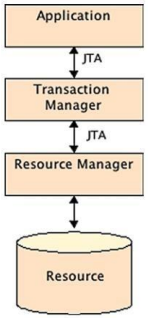
6.2 LIVELLI DI ISOLAMENTO

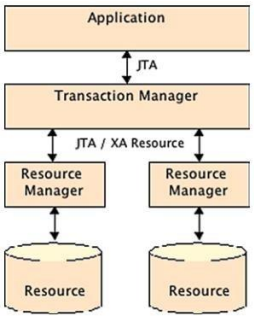
I database utilizzano diverse tecniche per regolare l'accesso concorrente: tali tecniche impattano sul livello di isolamento della transazione. I diversi livelli di isolamento sono i seguenti:

- **Lettura non committata** (isolamento meno restrittivo): la transazione può leggere dati non committati, provocando dirty, nonrepeatable e phantom reads;
- **Lettura committata**: la transazione non può leggere dati non committati. In tal modo, le dirty reads sono prevenute, ma non le nonrepeatable e le phantom reads;
- **Lettura ripetuta**: la transazione non può modificare dati attualmente in lettura da un'altra transazione. Ciò previene le dirty e le nonrepeatable reads, ma non le phantom;
- **Serializzabile**: la transazione ha lettura esclusiva, quindi le altre transazioni non possono leggere e scrivere gli stessi dati.

6.3 JAVA TRANSACTION API

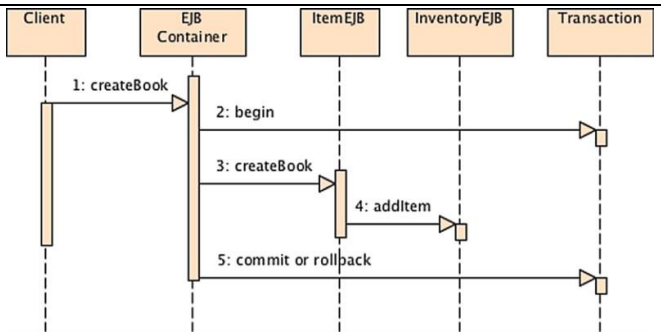
Java Transaction API (**JTA**) è una API che permette l'utilizzo delle transazioni in un ambiente Java. Tale API permette di gestire le transazioni in applicazioni che fanno uso di risorse (ad esempio, i database) tramite un'architettura detta X/Open XA. Tale architettura mette a disposizione un Transaction Manager, il quale coordina le transazioni e conduce i commit ed i rollback tramite il Resource Manager, il quale è responsabile riguardo la gestione delle risorse ad esso associate. Un esempio di Resource Manager è il driver per il database relazionale, come JDBC. Ad ogni risorsa corrisponde un Resource Manager diverso, quindi in caso di applicazioni distribuite l'architettura cambierebbe.





6.3 TRANSAZIONI NEGLI EJB

Quando si sviluppa un EJB con logica di business, non c'è bisogno di preoccuparsi riguardo la struttura interna del Transaction Manager o del Resource Manager siccome JTA nasconde tale complessità. Gli EJB integrano le transazioni sin dalla loro nascita, difatti ogni loro metodo è automaticamente racchiuso in una transazione. Tale comportamento è conosciuto come **container-managed transaction** perché le transazioni sono comunque gestite dal container EJB. Ciò spiega come mai le chiamate a **persist** (metodo dell'Entity Manager) non sono racchiuse dalla transazione che scriveremmo.

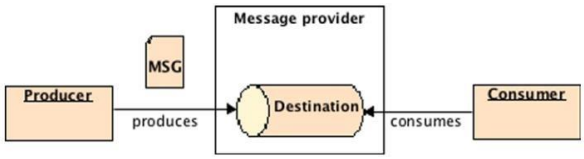


7. MESSAGING TRA COMPONENTI

Le componenti di un sistema possono comunicare tra loro tramite chiamate sincrone e, per permettere ciò, sia la componente mittente che la destinataria devono essere in running. Tale capitolo introduce il **messaging**, il quale per permette la comunicazione asincrona tra componenti.
MOM (**Message-Oriented Middleware**) è un'infrastruttura che permette la comunicazione asincrona tra sistemi eterogenei, cioè diversi tra loro. Tale infrastruttura fornisce il provider, il quale può esser visto come un buffer che gestisce tali messaggi, dove sia il producer (colui che produce i messaggi) che il consumer (colui che elabora i messaggi) non devono essere forzatamente attivi nello stesso tempo per comunicare. Il producer e il consumer non si conoscono, non sanno chi c'è dall'altro lato siccome utilizzano un buffer di mezzo: ciò porta tale tecnica ad essere considerata a basso accoppiamento, siccome non c'è alcuna relazione tra producer e consumer.
Il messaging è una buona soluzione per l'integrazione di applicazioni esistenti e nuove in un modo a basso accoppiamento.

7.1 INTRO A JAVA MESSAGE SERVICE

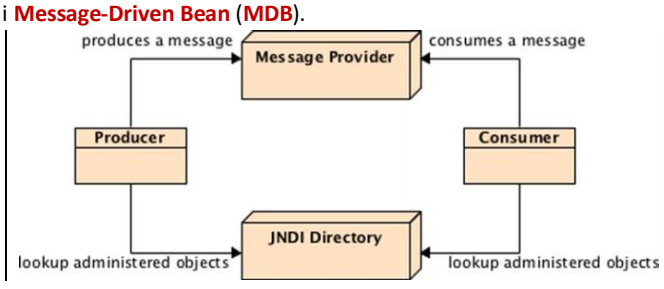
MOM è composto da diverse componenti, le quali entrano in azione quando un messaggio viene inviato. Il software che rispedisce i messaggi viene detto provider (detto anche broker), il quale può essere visto come una sorta di wrapper per una locazione che conserva messaggi, detta destinazione. La componente che invia il messaggio viene detta producer, mentre la componente che lo riceve viene detta consumer. Se la destinazione contiene dei messaggi, qualsiasi consumer interessato può usufruirne.



In Java EE, l'API che ha a che fare con i concetti del precedente paragrafo è chiamata **Java Message Service** (**JMS**). Tale API provvede un set di interfacce e classi che permettono di connettersi ad un provider, creare un messaggio, mandarlo e riceverlo. I messaggi vengono ricevuti da una determinata tipologia di EJB gestiti dal container e messi a disposizione da Java EE, i **Message-Driven Bean** (**MDB**).

MOM, è possibile rivederlo sottoforma di infrastruttura Object Oriented:
- Il provider è un'implementazione che conserva e rispedisce messaggi;
- I client sono componenti che producono e consumano messaggi;
- I messaggi sono oggetti che i client mandano o ricevono tramite il provider;
- Il provider deve provvedere oggetti amministrati ai client.

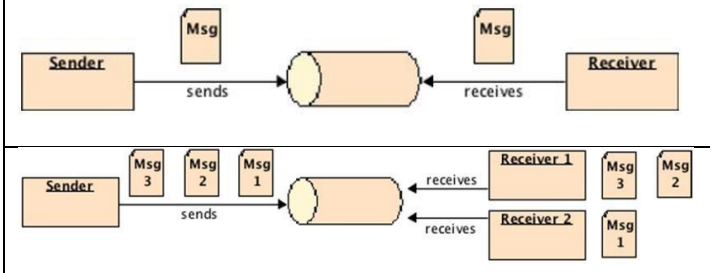
Il provider provvede una destinazione al suo interno, nella quale i messaggi possono essere conservati finchè non verranno consegnati al consumer.



Esistono due tipi di destinazione:
- Nel modello **punto a punto** (**P2P**), la destinazione che conserva i messaggi non è altro che una queue. In tale modello, un client inserisce un messaggio nella queue e un altro client lo riceve. Ricevuto il messaggio, il provider rimuove il messaggio dalla queue;
- Nel modello **publish-subscribe** (**pub-sub**), la destinazione viene chiamata Topic e viene sottoscritta da molteplici client, chiamati "sottoscrittori". Quando un client inserisce un messaggio nella topic, tutti i sottoscrittori lo riceveranno.

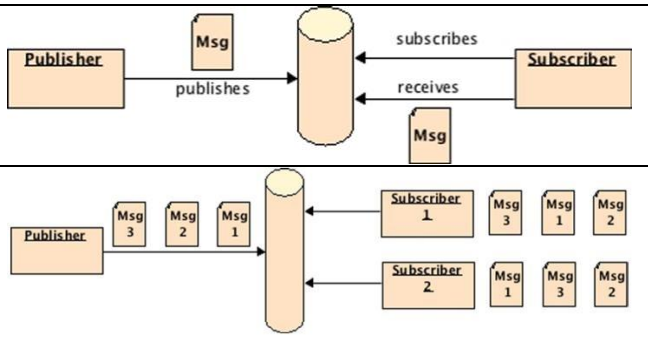
7.1.1 MODELLO PUNTO A PUNTO (P2P)

Nel modello **punto a punto**, un messaggio viaggia da un singolo producer ad un singolo consumer: la queue (destinazione) conserverà il messaggio inviato dal producer finché il consumer non lo riceverà. Ciò significa che il producer può inviare quando gli pare molteplici messaggi nella queue, così come il consumer può ricevere quando gli pare i messaggi.
Il modello punto a punto può essere utilizzato solo se esiste un unico producer: ciò non implica che non possano esistere molteplici consumer. In tal caso, ogni messaggio può essere ricevuto da un unico consumer e non da molteplici contemporaneamente.



7.1.2 MODELLO PUBLISH-SUBSCRIBE

Nel modello **pub-sub**, un messaggio viaggia da un singolo producer a molteplici consumer contemporaneamente. In tale modello, il provider gestisce le sottoscrizioni effettuate dai consumer, i quali vengono visti come dei sottoscrittori siccome, per ricevere i messaggi, devono potersi sottoscrivere al topic (destinazione).
Il topic conserva i messaggi finché non saranno distribuiti ai sottoscrittori. Molteplici sottoscrittori possono consumare lo stesso messaggio, rendendolo tale modello un'architettura broadcast, dove ogni messaggio viene inviato a molteplici destinatari.

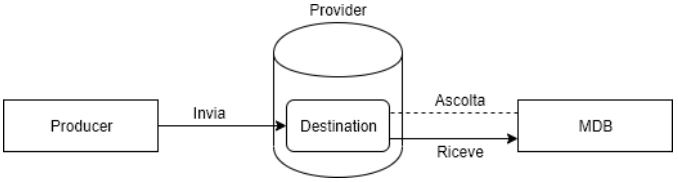


7.1.3 OGGETTI AMMINISTRATI

Gli **oggetti amministrati** sono oggetti configurati amministrativamente e non tramite programmazione che possono essere creati una sola volta. Sono oggetti di cui il provider ne permette la configurazione e li rende disponibili tramite JNDI. Due tipi di oggetti amministrati sono le connection factories, utilizzate dai client per creare la connessione con la destinazione, e le destinazioni stesse. I client possono accedere a tali oggetti tramite specifiche interfacce, ricavando il riferimento tramite JNDI.

7.1.4 INTRO AI MESSAGE-DRIVEN BEAN

I **Message-Driven Bean** (**MDB**) sono consumer asincroni di messaggi eseguiti in un container EJB. Sono una tipologia di EJB, non conservano alcuno stato e se ne possono avere molteplici istanze per processare messaggi provenienti da diversi producer. Nonostante siano molto simili agli stateless bean, i MDB non sono direttamente accessibili dai client: l'unico modo per comunicare con essi è mandare un messaggio alla destinazione su cui il MDB è in ascolto.



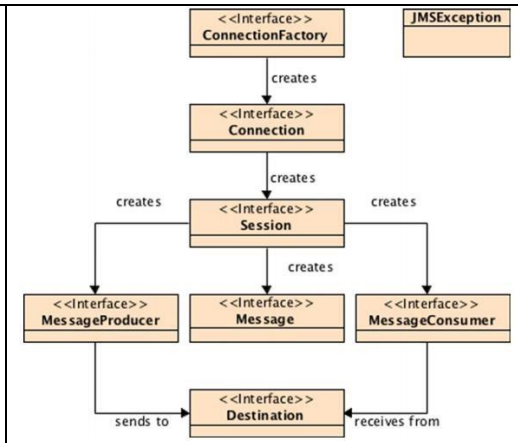
In generale i **MDB** attendono messaggi su una destinazione, che sia una queue o un topic, li ricevono e li elaborano. Essendo stateless, i MDB non possono delegare la logica di business ad altre componenti in modo sicuro.

7.2 JAVA MESSAGE SERVICE API

JMS è una API che permette alle applicazioni di creare, inviare, ricevere e leggere messaggi asincronamente. Definisce un set di interfacce e classi che permettono alle applicazioni di comunicare tra loro tramite dei provider. Con il tempo, JMS si è sempre evoluta e attualmente ci sono in giro diversi tipi di API, quali sono Legacy API, Classic API e Semplified API. Noi accenneremo la Classic API, ma ci concentreremo sulla Semplified API.

7.2.1 ACCENNI SULLA CLASSIC API

La **JMS Classic API** provvede determinate classi ed interfacce per applicazioni che richiedano un sistema di messaggistica, per far comunicare le componenti tra loro.
Questa API provvede la comunicazione asincrona tra i client fornendo una connessione al provider e una sessione nella quale i messaggi possono essere creati, inviati o ricevuti.
Tali messaggi possono contenere del testo o degli oggetti. La struttura della Classic API è la seguente:



Connection Factory:
Il **Connection Factory** è un oggetto amministrato che permette di creare connessioni al provider programmaticamente. Di tali oggetti, si dispone di un'interfaccia ConnectionFactory, la quale incapsula la configurazione dell'amministratore. Per poter utilizzare il Connection Factory, è necessario ricavarne un riferimento tramite JNDI lookup.

Destination:
La **destinazione** è un oggetto amministrato che contiene una configurazione legata al provider, come l'indirizzo di destinazione. Di tali oggetti, si disponde di un'interfaccia Destination, la quale può essere utilizzata ottenendone un riferimento tramite JNDI lookup.

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

Connection:
L'oggetto **Connection** incapsula una connessione al provider, viene restituito tramite il metodo **createConnection** del Connection Factory. L'apertura di una connessione è dispendiosa, quindi è stato valutato essere giusto rendere tale oggetto condiviso e thread-safe.
Ottenuto l'oggetto Connection tramite apposito metodo dal Connection Factory, si invoca il metodo start per poter iniziare a ricevere messaggi; se non si vogliono ricevere più messaggi per un determinato periodo di tempo, si utilizza il metodo stop; se non si vogliono ricevere più messaggi definitivamente, si utilizza il metodo close, il quale chiude anche i producer, i consumer e le sessioni.

```
Connection conn = cfactory.createConnection();
conn.start();
//Connection started
conn.stop();
//Connection is paused
conn.close();
//Connection is closed
```

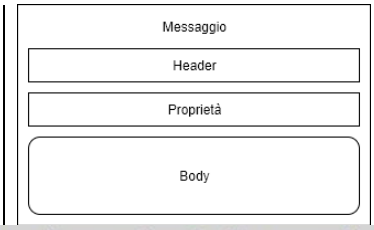
Session:
La sessione è un oggetto single-thread utilizzato per creare, produrre e consumare messaggi. È possibile creare una sessione da una connessione utilizzando il metodo createSession. Una sessione provvede un contesto transazionale dove i messaggi da mandare o ricevuti sono raggruppati in un'unica unità di lavoro, assicurando che vengano inviati tutti i messaggi o nessuno.

```
Session s = conn.createSession(true, Session.AUTO_ACKNOWLEDGE);
```

Il primo parametro specifica se la sessione è transazionale, mentre il secondo parametro indica se la sessione gestisce automaticamente gli ack dei messaggi che sono stati ricevuti con successo.

Messaggi:

Per comunicare, i client scambiano messaggi tra loro: il producer manda il messaggi ad una destinazione ed il consumer lo riceve. I messaggi, sostanzialmente, sono oggetti che incapsulano informazioni e sono composti da tre elementi: l'header contiene informazioni riguardanti il messaggio; le proprietà sono coppie (nome, valore) che l'applicazione può impostare o leggere; il body contiene il vero e proprio contenuto del messaggio, che può essere del testo come può essere un qualsiasi oggetto.



L'header è composto da coppie (nome, valore) che identificano il messaggio, le quali vengono impostate automaticamente dei metodi send e publish. Volendo, il programmatore può manualmente impostare tali coppie. Le proprietà sono anch'esse coppie (nome, valore), ma vengono stabilite solamente dall'applicazione che manda tali messaggi. Tali proprietà possono essere viste come delle coppie personalizzate.

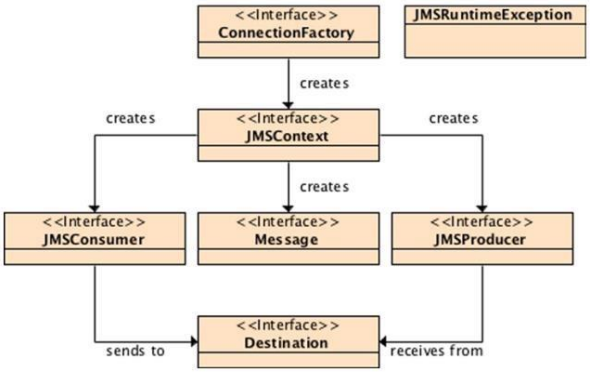
```
m.setStringProperty("accountName", "Francesco");
m.setFloatProperty("orderAmount", 125.6F);
m.setBooleanProperty("freeShipping", false);
if (!m.getBooleanProperty("freeShipping"))
   m.setFloatProperty("orderAmount",
      (m.getFloatProperty("orderAmour") + 5F));
```

Il body del messaggio è opzionale, contiene dati da inviare o ricevere e può contenere diversi formati di dati, anche gli oggetti, in base all'interfaccia che si sceglie di utilizzare.

```
String msgText = textMessage.getText();
System.out.println(msgText);
objectMessage.setObject(book);
```

7.2.2 SIMPLIFIED API

La **Simplified API** provvede le stesse funzionalità della Classic API, ma richiede meno interfacce ed è molto più semplice da utilizzare: introduce tre nuove interfacce rispetto alla Classic API, quali sono JMSContext, JMSProducer e JMSConsumer.
Tali interfacce si affidano al ConnectionFactory visto nel capitolo precedente.
La struttura della Simplified API è la seguente:



Le interfacce introdotte sono, quindi, le seguenti:

- ***JMSContext*** rappresenta la connessione single-threaded attiva verso un provider, nella quale è possibile inviare e ricevere messaggi;
- ***JMSProducer*** è un oggetto creato da JMSContext utilizzato per inviare messaggi ad una queue o un topic;
- ***JMSConsumer*** è un oggetto creato da JMSContext utilizzato per ricevere messaggi inviati da un producer ad una queue o un topic.

JMSContext:

JMSContext è l'interfaccia più importante della Simplified API siccome combina le funzionalità di due oggetti che erano originariamente separati nella Classic API: Connection e Session.
Un **JMSContext** può essere creato in due modi: se l'ambiente è gestito dall'applicazione, allora è necessario utilizzare uno dei metodi **createContext** su un ConnectionFactory, per poi esser chiuso quando non serve più; se l'ambiente è gestito dal container è possibile iniettare JMSContext tramite l'annotazione @Inject.
Quando l'applicazione necessita di mandare un messaggio, è necessario creare un producer: ciò è permesso dal metodo createProducer del JMSContext. Quando l'applicazione necessita di ricevere un messaggio è necessario creare un consumer: ciò è permesso dal metodo createConsumer del JMSContext.

JMSProducer:

Il **JMSProducer** viene utilizzato per mandare messaggi ad una specifica destinazione tramite appropriati metodi. Il JMSProducer viene creato tramite apposito metodo del JMSContext, createProducer, e permette di impostare determinati criteri per l'invio, le proprietà del messaggio e gli headers.

JMSConsumer:

Il JMSConsumer viene utilizzato per ricevere messaggi da una specifica destinazione, che sia una queue o un topic. Il JMSConsumer viene creato tramite apposito metodo del JMSContext, createConsumer, specificando anche un selezionatore di messaggi, in modo da essere più restrittivi riguardo i messaggi ricevuti.
È possibile ricevere messaggi in maniera sincrona o asincrona: se la consegna è asincrona, è necessario utilizzare un MessageListener che faccia da tramite. In tal caso, appena arriva il messaggio, il provider lo consegna al JMSConsumer invocando il metodo onMessage del MessageListener.

7.3 SCRIVERE UN PRODUCER DI MESSAGGI

In generale, il producer, per essere creato, segue una determinata logica: si ottengono tramite JNDI lookup o tramite DI il Connection Factory e la Destination. Il Connection Factory ci consente di ottenere l'oggetto JMSContext, dal quale si ricava il producer. Ricavato il producer, tramite il metodo send si invia il messaggio, specificando la Destination ottenuta. La sua creazione segue i passi appena elencati, ma varia la gestione del Connection Factory e della Destination, in base al dove si scrive il producer.

7.3.1 PRODURRE UN MESSAGGIO FUORI DAL CONTAINER

In questo paragrafo vediamo come creare un producer che mandi messaggi ad una queue fuori dal container, in ambiente Java SE. I Connection Factory e i Destination sono oggetti amministrati che risiedono in un provider e che devono essere dichiarati nel namespace JNDI, in modo da poterli ricavare tramite JNDI lookup. Creando il producer fuori dal container, si disporrà solamente delle interfacce remote e si dovrà necessariamente ricorrere al lookup JNDI.

```
public static void main(String[] args) throws NamingException {

   Context ctx = new InitialContext();
   //JNDI lookup for Connection Factory and Destination
   ConnectionFactory cfactory = (ConnectionFactory)
         ctx.lookup("jms/msg/ConnectionFactory");
   Destination destinationQueue = (Destination)
         ctx.lookup("jms/msg/Queue");
   try (JMSContext jmsctx = cfactory.createContext()){
      jmsctx.createProducer().send(destinationQueue,
            "That's a message");
   }

}
```

7.3.2 PRODURRE UN MESSAGGIO DENTRO AL CONTAINER

Quando il codice del client viene eseguito dentro un container, per tale caso è possibile utilizzare la DI invece del JNDI lookup. È possibile utilizzare la DI perché, essendo a lato server, si dispone delle risorse vere e proprie e non per forza delle interfacce (è per questo che usiamo Queue anziché Destination, la quale è un'interfaccia remota per Queue), quindi è inutile ricorrere a JNDI. Per iniettare un Connection Factory o un Destination, quindi, è necessario utilizzare @Resource, specificando il nome della risorsa nel namespace.

```
@Stateless
public class ProducerEJB {

   @Resource(lookup = "jms/msg/ConnectionFactory")
   private ConnectionFactory cfactory;
   @Resource(lookup = "jms/msg/Queue")
   private Queue queue;

   public void sendMessage(){
      try(JMSContext jmsctx = cfactory.createContext()){
         jmsctx.createProducer().send(queue, "That's a message");
      }
   }

}
```

7.3.3 PRODURRE UN MESSAGGIO DENTRO AL CONTAIRE CDI

Quando il producer è in esecuzione in un container con CDI abilitato, è possibile iniettare direttamente il JMSContext, invece di recuperarsi prima il Connection Factory. Ciò è reso fattibile dalle annotazioni @Inject e @JMSConnectionFactory: quest'ultima serve a specificare il nome dallo spazio di naming del Connection Factory da cui creare il JMSContext. Se tale annotazione viene omessa, allora verrà utilizzato il Connection Factory di default.

```
@Stateless
public class ProducerEJB {

   @Inject
   @JMSConnectionFactory("jms/msg/ConnectionFactory")
   private JMSContext jmsctx;
   @Resource(lookup = "jms/msg/Queue")
   private Queue queue;

   public void sendMessage(){
      jmsctx.createProducer().send(queue, "That's a message");
   }

}
```

7.4 SCRIVERE UN CONSUMER DI MESSAGGI

Per ricevere i messaggi, un client utilizza un JMSConsumer, il quale è un consumer e viene creando passando la destinazione (che sia una queue o un topic) al metodo createConsumer, del JMSContext. Il client può ricevere messaggi in modo sincrono o asincrono: nel modo sincrono, il consumer invoca il metodo receive per ottenere il messaggio dalla destinazione; nel metodo asincrono, il provider consegna i messaggi al consumer tramite il metodo onMessage appartenente al MessageListener, il quale fa da tramite nella consegna.

7.4.1 CONSEGNA SINCRONA

Un consumer sincrono attende sempre l'arrivo di un nuovo messaggio e lo richiede non appena arriva utilizzando il metodo receive. La creazione del consumer è decisamente semplice: si ottiene un Destination ed un Connection Factory tramite JNDI lookup; da quest'ultimo si ottiene un oggetto JMSContext che permette la creazione del JMSConsumer tramite il metodo createConsumer, il quale come argomento richiede la destinazione; JMSConsumer rimane in loop, in attesa di nuovi messaggi. Nel caso non arrivino messaggi, l'esecuzione si blocca nel loop siccome il consumer rimane in attesa di un messaggio.

```
public static void main(String[] args) throws NamingException {

   Context ctx = new InitialContext();
   ConnectionFactory cFactory =
           (ConnectionFactory) ctx.lookup("jms/app/cfactory");
   Destination queue =
           (Destination) ctx.lookup("jms/app/destination");
   try (JMSContext jmsctx = cFactory.createContext()) {
      while (true) {
         String msg =
                 jmsctx.createConsumer(queue).receiveBody(String.class);
         if (msg.equalsIgnoreCase("Close"));
         break;
      }
   }

}
```

Nota: come con i producer, è possibile effettuare l'iniezione con @Resource, @Inject o @JMSConnectionFactory nel caso si stia eseguendo l'applicazione in un container.

7.4.2 CONSEGNA ASINCRONA

Per la consegna asincrona, la questione cambia un pochino. La consegna asincrona è basata sulla gestione degli eventi: un client può registrare un oggetto che implementi l'interfaccia MessageListener, rendendolo così una sorta di tramite. Quando un messaggio arriva, il provider lo consegna invocando il metodo onMessage, il quale appartiene al MessageListener. In tal modo, il consumer non ha il bisogno di effettuare un ciclo e attendere in tal modo sempre nuovi messaggi. Sostanzialmente, l'implementazione del MessageListener non sarà altro che il consumer vero e proprio. Per realizzare ciò, è necessario implementare l'interfaccia stessa, definendo il metodo onMessage; si procede ottenendo un Connection Factory e un Destination tramite JNDI lookup; dal Connection Factory si ricava il JMSContext, dal quale si ricava il JMSConsumer, il quale richiamerà il metodo setMessageListener per associare un nuovo MessageListener.

```
public class ConsumerListener implements MessageListener{

   public static void main(String[] args) throws NamingException {
      Context ctx = new InitialContext();
      ConnectionFactory cFactory =
              (ConnectionFactory) ctx.lookup("jms/app/ConnectionFactory");
      Destination queue =
              (Destination) ctx.lookup("jms/app/Queue");
      try (JMSContext jmsctx = cFactory.createContext()){
         jmsctx.createConsumer(queue).
                 setMessageListener(new ConsumerListener());
      }
   }

   @Override
   public void onMessage(Message message) {
      try {
         System.out.println("Received: " + message.getBody(String.class));
      } catch (JMSException e) {}
   }

}
```

7.5 FILTRO DEI MESSAGGI

Alcune applicazioni potrebbero avere necessità di filtrare i messaggi che ricevono. Quando un messaggio viene consegnato a molteplici client, risulta utile utilizzare dei criteri per ricevere solo determinati tipi di messaggi. Tali criteri, chiamati "**selector**", vanno stabiliti in base alle coppie (nome, valore) presenti nell'header e nelle proprietà del messaggio. Il **selector** non è altro che una stringa che specifica i requisiti che devono avere determinate coppie e va specificato nel metodo createConsumer, come argomento secondario ed opzionale.

```
jmsctx.createConsumer(queue,
        "JMSPriority < 6").receive();
jmsctx.createConsumer(queue,
        "JMSPriority < 6 AND orderAmount < 200").receive();
jmsctx.createConsumer(queue,
        "orderAmount BETWEEN 500 AND 2000").receive();
```

Ovviamente, tali criteri, per poter esser filtrati, devono poter esser stati impostati sul messaggio ricevuto.

```
Message m = jmsctx.createTextMessage();
try {
   m.setIntProperty("orderAmount", 850);
   m.setJMSPriority(3);
} catch (JMSException e) {}
JMSProducer producer = jmsctx.createProducer();
producer.send(queue, m);
```

7.6 TIME TO LIVE DEI MESSAGGI

Il producer invia messaggi al provider, il quale li conserva nella destinazione e li consegna ai consumer che li richiedono. Può capitare, però, che molti di questi messaggi non vengano richiesti da alcun consumer, quindi rimarranno nella destinazione in attesa di un consumer che li richieda: ciò può rappresentare un grande carico per il provider, in caso di grandi numeri di messaggi.

```
TextMessage m = jmsctx.createTextMessage();
try {
   m.setText("Hello everyone!");
} catch (JMSException e) {}
JMSProducer producer = jmsctx.createProducer();
producer.setTimeToLive(1000); // 1 second
producer.send(queue, m);
```

Per ovviare a tale problema, è possibile impostare il Time to Live sul producer, il quale è un parametro in millisecondi che avvia un countdown e, al termine di esso, se il messaggio non è stato richiesto verrà distrutto.

7.7 PRIORITÀ DEI MESSAGGI

È possibile utilizzare le priorità nei messaggi in modo tale che il provider consegni prima i messaggi più urgenti. JMS definisce la priorità come un valore intero che va da 0 a 9 (dove 9 è la priorità più urgente). È possibile specificare la priorità sul producer tramite il metodo setPriority, in modo tale che ogni messaggio creato abbia tale priorità. È ovviamente possibile anche impostare la priorità per il singolo messaggio, siccome è una coppia (nome, valore) nell'header: per farlo, è necessario invocare il metodo setJMSPriority sul messaggio.

```
TextMessage msgWithPriority = jmsctx.createTextMessage();
TextMessage msg = jmsctx.createTextMessage();
try {
   msgWithPriority.setText("Hello everyone!");
   msgWithPriority.setJMSPriority(5);
   msg.setText("No priority here");
} catch (JMSException e) {}
JMSProducer prodWithPriority =
        jmsctx.createProducer().setPriority(5);
JMSProducer producer = jmsctx.createProducer();
prodWithPriority.send(queue, msg);
producer.send(queue, msgWithPriority);
```

7.8 MESSAGE-DRIVEN BEAN

Un **MDB** è un consumer asincrono stateless, nascosto al producer, invocato dal container per gestire i messaggi che arrivano alla destinazione.
Si utilizzano i MDB al posto dei client JMS proprio perché i MDB sono componenti gestiti dal container, il quale gestisce la sicurezza, il multithreading, le transazioni, ecc. Proprio come gli altri EJB, il MDB può accedere alle risorse gestite dal container, quindi può interagire col database, può utilizzare l'Entity Manager, ecc. Il container gestisce i messaggi in arrivo alla destinazione tra le diverse istanze di MDB, i quali saranno disponibili in un pool, proprio come gli EJB stateless. Precisamente, non appena un messaggio arriva nella destinazione, un'istanza di MDB viene presa dal pool per poter gestire il messaggio. I MDB sono totalmente differenti dai session bean: non implementano interfacce locali o remote, bensì implementano l'interfaccia MessageListener e il metodo onMessage associato (vale ciò che è stato detto nel capitolo 7.4.2). Il fatto che non implementino interfacce remote, implica il fatto che i client non possano invocare metodi in maniera diretta sul MDB, bensì, sarà possibile inviare messaggi per risolvere tale problema.

Un MDB, per essere considerato tale, deve rispettare le seguenti condizioni:

- La classe deve essere annotata con @MessageDriven;
- La classe deve implementare l'interfaccia MessageListener;
- La classe deve essere public;
- La classe deve avere un costruttore senza argomenti che il container utilizzerà per creare le istanze del MDB;
- La classe non deve definire il metodo finalize.

```
@MessageDriven(mappedName = "jms/app/Topic")
public class TestMDB implements MessageListener {

   @Override
   public void onMessage(Message message) {
      try{
         System.out.println("Received " +
                  message.getBody(String.class));
      } catch (JMSException e) {}
   }

}
```

7.8.1 ANNOTAZIONE @MessageDriven

L'annotazione **@MessageDriven** è obbligatoria per i MDB siccome serve al container per capire che tale classe è un MDB. Tale annotazione presenta un argomento in particolare, mappedName: specifica la destinazione nella quale il MDB deve ascoltare.

7.8.2 DEPENDENCY INJECTION

Come tutti gli altri EJB, i MDB possono utilizzare la DI per acquisire risorse.

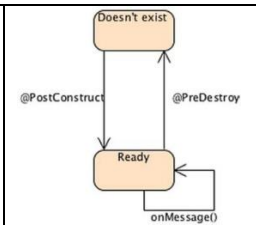
```
@MessageDriven(mappedName = "jms/app/Topic")
public class TestMDB implements MessageListener {

   @PersistenceContext(unitName = "testPU")
   private EntityManager em;
   @Inject
   private BookEJB controller;
   @Resource(lookup = "jms/app/ConnectionFactory")
   private ConnectionFactory cFactory;
   @Resource
   private MessageDrivenContext mdctx;

   @Override
   public void onMessage(Message message) {
      try{
         System.out.println("Received " +
                  message.getBody(String.class));
      } catch (JMSException e) {}
   }

}
```

7.8.3 MDB CONTEXT

L'interfaccia **MessageDrivenContext** provvede l'accesso al contesto provvisto dal container per il MDB: ciò fornisce al MDB metodi per gestire le transazioni, di effettuare lookup, ecc.

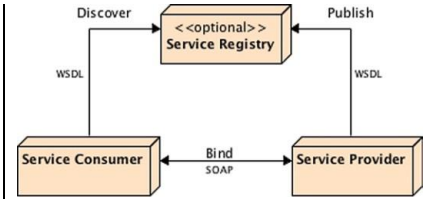
7.9 CICLO DI VITA DI UN MDB E I CALLBACK

Il ciclo di vita di un MDB è praticamente identico a quello di un session bean (capitolo 5.6.1). Il container crea un'istanza di MDB e inietta le risorse necessarie tramite DI, richiama il callback annotato con @PostConstruct e rende disponibile il MDB alla ricezione dei messaggi, i quali verranno ricevuti tramite il metodo onMessage (il perché viene spiegato nei capitoli 7.8 e 7.4.2). Il callback annotato con @PreDestroy viene invocato prima che il MDB venga rimosso dal pool per poi essere distrutto.



8. SOAP WEB SERVICES

Con il termine "web service" si intende qualcosa di accessibile nel web che fornisca dei servizi. Le applicazioni che utilizzano i web service possono essere implementate con diverse tecnologie, tra le quali SOAP. I **SOAP** web service (**SOAP WS**) sono considerati a basso accoppiamento perché il client, cioè il consumer, non conosce i dettagli dell'implementazione (come il linguaggio utilizzato per scrivere tale servizio, i vari metodi, ecc). Il consumer può invocare un SOAP WS utilizzando la sua interfaccia in XML, la quale fornisce tutti i metodi di business che il web service mette a disposizione: tali metodi possono essere implementati in qualsiasi linguaggio. In breve, i web service provvedono un modo per connettere diversi pezzi di software.
Vediamo un'immagine che mostra le interazioni di un SOAP WS. Opzionalmente, il SOAP WS può registrare la propria interfaccia in un registro chiamato UDDI, in modo tale che il consumer possa analizzarla.



8.1 TECNOLOGIE E PROTOCOLLI

I SOAP WS dipendono da diverse tecnologie e protocolli di trasporto, quelli che vedremo sono i seguenti:

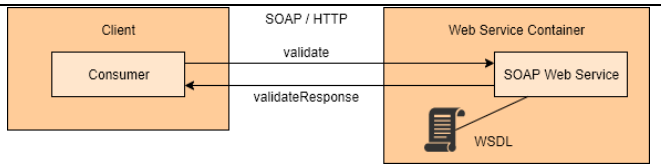
- **XML**: linguaggio di markup sul quale i SOAP WS sono costruiti e definiti;
- **WSDL**: definisce il protocollo, l'interfaccia, tipi di messaggio e interazioni tra consumer e provider;
- **SOAP**: è il protocollo che permette ai componenti di un'applicazione di comunicare tra loro, basato su XML;
- I **messaggi** vengono scambiati utilizzando un protocollo di trasporto, come HTTP o JMS;
- **UDDI**: servizio opzionale del service registry, serve a conservare e categorizzare le interfacce del web service (WSDL).

8.1.1 WSDL

Web Services Description Language (**WSDL**) è un linguaggio di definizione delle interfacce che definisce le interazioni tra consumer e SOAP WS, descrive il tipo del messaggio, la porta, il protocollo di comunicazione, ecc. In un certo senso, è possibile vedere WSDL come un'interfaccia Java ma scritta in XML.

8.1.2 SOUP

Simple Object Access Protocol (**SOAP**) è un protocollo che provvede il meccanismo di comunicazione con i web service, in modo da permettere lo scambio di dati in XML tramite apposito protocollo, generalmente HTTP. SOAP provvede un protocollo indipendente in grado di connettere servizi distribuiti tra loro.



8.1.3 UDDI

I consumer che hanno necessità di interagire con i provider di servizi nel web hanno necessità di trovare informazioni riguardo la connessione con essi. **Universal Description Discovery and Integration** (**UDDI**) provvede un approccio per trovare informazioni riguardanti il web service e le invocazioni su di esso. Sostanzialmente, il service provider pubblica un WSDL in un registro UDDI disponibile nel web, il quale verrà scaricato ed analizzato da potenziali consumer.

8.1.4 PROTOCOLLO DI TRASPORTO

Un consumer, per poter comunicare con un web service, necessita di un modo per inviare messaggi. I messaggi SOAP possono essere trasportati nella rete utilizzando un protocollo che entrambe le parti supportino: di solito viene utilizzato HTTP.

8.2 DEFINIRE UN SOAP WS

È possibile utilizzare un tipo di approccio detto "bottom-up" che consiste nell'implementare la classe generando automaticamente il WSDL e utilizzando annotazioni specifiche per mappare Java verso WSDL.
Il web service definito in Java non è altro che un POJO annotato e deployato in un container per web service. L'annotazione utilizzata è @WebService e permette di generare a runtime l'infrastruttura che gestisce i messaggi in XML tramite chiamate HTTP.
È possibile che tra il consumer ed il SOAP WS vengano trasmessi oggetti Java: in tal caso, ci serve qualcosa che converta l'oggetto Java in XML per la trasmissione: l'annotazione @XmlRootElement.

Attenzione: l'annotazione @WebService deve obbligatoriamente definire serviceName, il quale indica il nome del web service.

```
@WebService(serviceName = "CardValidator")
@Stateless
public class CardValidator {

   public Boolean validate(CreditCard card) {
      if (card.getSerial().length() != 16 ||
            card.getControlNumber() < 100 ||
            card.getControlNumber() > 999)
         return false;
      else return true;
   }

}
```

Il metodo validate del SOAP WS CardValidator utilizza un POJO CreditCard: tale POJO dovrà essere mappato in XML tramite apposita annotazione @XmlRootElement.

```
@XmlRootElement
public class CreditCard {

   private String serial;
   private String expiryDate;
   private Integer controlNumber;

   public String getSerial() {
      return serial;
   }
   public void setSerial(String serial) {
      this.serial = serial;
   }
   public String getExpiryDate() {
      return expiryDate;
   }
   public void setExpiryDate(String expiryDate) {
      this.expiryDate = expiryDate;
   }
   public Integer getControlNumber() {
      return controlNumber;
   }
   public void setControlNumber(Integer controlNumber) {
      this.controlNumber = controlNumber;
   }

}
```

8.3 ANATOMIA SI UN SOAP WS

Un web service, per essere considerato tale, deve rispettare determinati requisiti:

- La classe deve essere annotata da @WebService specificando l'attributo serviceName;
- La classe può implementare zero o più interfacce, le quali devono essere annotate anch'esse con @WebService;
- La classe deve essere pubblica;
- La classe deve avere un costruttore pubblico di default;
- La classe non deve definire il metodo finalize;
- Per rendere un SOAP WS un EJB endpoint, la classe deve essere annotata con @Stateless o @Singleton;
- Un servizio deve essere stateless, quindi non deve salvare alcuno stato tra le diverse chiamate tra i client.

È possibile esporre classi Java ed EJB come web service tramite una semplice annotazione, quale è @WebService. Nel caso, però, si tratti di un EJB, è necessario utilizzare l'annotazione @Stateless. La classe Java e l'EJB sono decisamente simili, hanno lo stesso e identico comportamento, ma l'uso degli EJB endpoint comporta dei benefici, come le transazioni e la sicurezza gestite dal container. Inoltre, è possibile utilizzare anche gli interceptor, i quali sono vietati nei servlet endpoint. Il codice di business può essere, quindi, esposto sia come web service che come EJB, in modo da utilizzarlo tramite SOAP o tramite RMI con interfaccia remota.

8.4 MAPPING WSDL

Come ben sappiamo, i web service sono sistemi definiti in termini di messaggi XML, operazioni WSDL e messaggi SOAP. Riguardo Java, sappiamo che le nostre applicazioni sono definite in termini di oggetti, interfacce e metodi. È necessaria, quindi, una sorta di traduzione da oggetti Java ad operazioni WSDL. Esistono, quindi, delle annotazioni che permettono il mapping semplificato da Java a WSDL e SOAP.
Le annotazioni WSDL permettono di cambiare il mapping Java/WSDL e sono le seguenti: @WebMethod, @WebResult, @WebParam, @OneWay. Le annotazioni SOAP, invece, sono le seguenti: @SOAPBinding e @SOAPMessageHandler. Ne vedremo solo alcune.

8.4.1 L'ANNOTAZIONE @WebService

L'annotazione **@WebService** contrassegna una classe o un'interfaccia come appartenente ad un web service. Se utilizzata direttamente su una classe, come visto in un paio di capitoli fa, il container ne genererà direttamente l'interfaccia.
Nel caso si voglia rendere esplicita l'interfaccia annotata, la classe che la implementerà dovrà specificare nell'annotazione @WebService l'interfaccia a cui fa riferimento.

@WebService è caratterizzata da molteplici attributi, i quali permettono di specificare diverse caratteristiche del web service.

```
@WebService(
        serviceName = "CardValidator",
        endpointInterface = "interfaces.Validation"
)
@Stateless
public class CardValidator implements Validation{

   public Boolean validate(CreditCard card) {
      if (card.getSerial().length() != 16 ||
            card.getControlNumber() < 100 ||
            card.getControlNumber() > 999)
         return false;
      else return true;
   }

}
```

8.4.2 L'ANNOTAZIONE @WebMethod

Utilizzando l'annotazione **@WebService** è possibile modificare attributi riguardanti il web service vero e proprio. È possibile, tramite apposite annotazioni, modificare il mapping di default che si applica per i metodi del web service. Per personalizzare il mapping dei metodi, è possibile utilizzare l'annotazione @WebMethod, la quale permette di rinominare i metodi o di escluderli dal WSDL.

```
@WebService(serviceName = "CardValidator")
@Stateless
public class CardValidator{

   @WebMethod(operationName = "validate",
              exclude = false)
   public Boolean validate(CreditCard card) {
      if (card.getSerial().length() != 16 ||
            card.getControlNumber() < 100 ||
            card.getControlNumber() > 999)
         return false;
      else return true;
   }

}
```

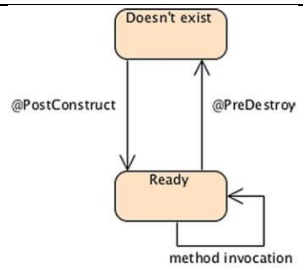
8.4.3 L'ANNOTAZIONE @WebResult

L'annotazione @WebResult controlla il nome dell'oggetto di ritorno in WSDL. Nel seguente esempio, il valore ritornato è rinominato con "isValid". Nel caso non venga utilizzata tale annotazione, rimane il valore di default: "return".

```
@WebService(serviceName = "CardValidator")
@Stateless
public class CardValidator{

   @WebResult(name = "isValid")
   public Boolean validate(CreditCard card) {
      if (card.getSerial().length() != 16 ||
            card.getControlNumber() < 100 ||
            card.getControlNumber() > 999)
         return false;
      else return true;
   }

}
```

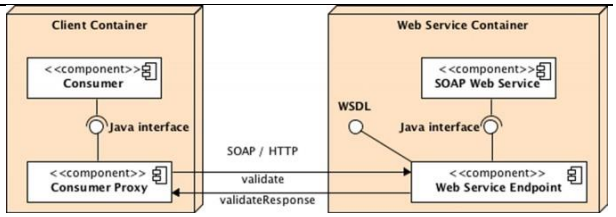
8.5 CICLO DI VITA DI UN SOAP WS

Anche i SOAP WS possiedono un proprio ciclo di vita, decisamente simile a quello dei managed beans. È praticamente lo stesso ciclo di vita delle componenti che non conservano alcuno stato: se esistono, allora possono processare richiesta senza conservare lo stato.



8.6 UTILIZZARE UN SOAP WS

Abbiamo visto come scrivere un SOAP WS, ora vediamo come invocarlo da client come servizio. L'invocazione di un SOAP WS è molto simile all'invocazione di un oggetto distribuito con RMI: si ottiene il riferimento al servizio e si invocano metodi su di esso. Precisamente, ne viene ottenuto un proxy, il quale permette di effettuare chiamate anche su web service non scritti in Java tramite dei tool interni.



Innanzitutto, per invocare un WS SOAP si necessita di un consumer, il che non è altro che un client sulla JVM capace di comunicare con le componenti di un container. Nel caso il consumer si trovi in un container, quest'ultimo può ottenere un'istanza del proxy direttamente tramite iniezione: per iniettare un SOAP WS, è necessario utilizzare l'annotazione @WebServiceRef o un producer dedicato.

8.6.1 INVOCAZIONE DI UN SOAP WS FUORI DAL CONTAINER

Se il consumer è un client posto al di fuori di un container, è necessario invocare il SOAP WS programmaticamente. Non si utilizzerà il web service direttamente, bensì un suo proxy generato. Sia WSNAME il nome del web service, tale proxy si ricava tramite la seguente istruzione:

new WSNAMEService().getWSNAMEPort();

Nota: alcune volte WSNAME e Service() sono separati da un underscore. Ciò accade quando WSNAME si conclude con il carattere "s".

Ricavato il proxy del web service, è possibile effettuare invocazioni di metodi, le quali saranno delegate al web service remoto. Ciò che sta dietro l'invocazione remota fa parte del meccanismo nascosto offerto dal proxy.

```
public static void main(String[] args) {

   CreditCard card = new CreditCard();
   card.setSerial("55566600");

   CardValidator service =
         new CardValidator_Service().getCardValidatorPort();
   System.out.println(service.validate(card));

}
```