

Java EE Guide

Francesco Abate

Sommario

Capitolo 1: Introduzione a Java EE 7.....	7
1.1: I livelli di Java EE	7
1.2: Componenti	7
1.3: Container	7
1.4: Servizi.....	8
1.5: Packaging.....	9
1.6: Annotazioni.....	9
Capitolo 2: Context and Dependency Injection (CDI).....	10
2.1: Panoramica sui bean	10
2.2: Cos'è la Dependency Injection (DI)	10
2.3: Ciclo di vita di un CDI bean	11
2.4: Scope e contesto	11
2.5: Introduzione agli interceptors	11
2.6: Introduzione al Deployment Descriptor	11
2.7: Introduzione alla pratica sui CDI bean.....	12
2.8: Anatomia di un CDI bean	12
2.9: Dependency Injection con i CDI bean.....	12
2.9.1: Iniezione tramite @Inject	13
2.9.2: Qualificatori - specificare l'implementazione.....	14
2.9.3: Qualificatori con membri.....	16
2.9.4: Producers - iniezione di tipi primitivi e POJO	17
2.10: Contesto (scope) e annotazioni	18
2.11: Interceptor.....	19
2.11.1: Interceptor Binding.....	22
2.12: Eventi	22
2.13: Esercizio riepilogativo	24
2.13.1: Scrivere il POJO Book.....	25
2.13.2: Scrivere NumberGenerator, implementazioni e qualificatori	26
2.13.3: Scrivere il gestore dei libri BookService	27
2.13.4: Rendere iniettabile la classe Logger	27
2.13.5: Creare l'interceptor che utilizza il Logger.....	28
2.13.6: Applicare l'interceptor @LoggableInterceptor	28
Capitolo 3: Java Persistence API (JPA)	29
3.1: Introduzione alle entità	30
3.2: Anatomia di un'entità.....	30

3.3: Riguardo l'Object Relational Mapping (ORM)	31
3.4: Introduzione alle query e all'Entity Manager	31
3.5: Persistence Unit.....	33
3.6: Accenni sul ciclo di vita delle entità e le callbacks	34
Capitolo 4: Gestione degli Oggetti Persistenti.....	34
4.1: Entity Manager	35
4.1.1: Ottenere un Entity Manager	35
4.1.2: Entità managed, detached e relazione con l'appartenenza al database	36
4.1.3: Persistence Context	37
4.1.4: Rendere persistente un'entità tramite persist	37
4.1.5: Ricavare un'entità dal database tramite find e getReference	38
4.1.6: Rimuovere un'entità dal database tramite remove	38
4.1.7: Sincronizzazione con il database tramite flush e refresh	38
4.1.8: Contenimento di un'istanza tramite contains.....	39
4.1.9: Rendere detach un'entità.....	40
4.1.10: Aggiornare un'entità nel database tramite merge.....	40
4.2: Ciclo di vita delle entità	40
4.2.1: Callbacks	41
4.3: Query	42
4.3.1: Dynamic Query	43
4.3.2: Named Query	43
4.3.3: Native Query.....	45
Capitolo 5: Enterprise JavaBeans	45
5.1: Servizi offerti dal container	46
5.2: Anatomia di un EJB	46
5.2.1: La classe Bean dell'EJB.....	47
5.2.2: Local e Remote, interfacce riguardanti gli EJB	47
5.3: Servizio di naming JNDI	48
5.4: Session Bean.....	50
5.4.1: Stateless Bean.....	50
5.4.2: Stateful Bean	51
5.4.3: Singleton Bean.....	52
5.5: Dependency Injection.....	54
5.6: Ciclo di vita di un Session Bean	54
5.6.1: Ciclo di vita per Stateless e Singleton Bean.....	54
5.6.2: Ciclo di vita per Stateful Bean.....	55

5.6.3: Callbacks	56
5.7: Esercizio riepilogativo	56
5.7.1: Scrivere l'entità Book.....	57
5.7.2: Scrivere lo Stateless Bean BookEJB	58
5.7.3: Scrivere il Database Producer.....	59
5.7.4: Persistence Unit.....	60
5.7.5: Scrivere il DatabasePopulator e definire il Data Source.....	60
5.7.6: Scrivere la classe con il main	62
5.7.7: La situazione per il client e per il server	62
5.7.8: Testare	62
Capitolo 6: Transazioni	63
6.1: Tipi di lettura	63
6.2: Livelli di isolamento	63
6.3: Java Transaction API	64
6.3: Transazioni negli EJB.....	64
Capitolo 7: Messaging tra componenti	65
7.1: Introduzione a Java Message Service	65
7.1.1: Modello Punto a Punto.....	66
7.1.2: Modello Publish-Subscribe	67
7.1.3: Oggetti Amministrati	67
7.1.4: Introduzione ai Message-Driven Bean	68
7.2: Java Message Service API	68
7.2.1: Accenni sulla Classic API	68
7.2.2: Simplified API.....	71
7.3: Scrivere un Producer di messaggi.....	72
7.3.1: Produrre un messaggio fuori dal Container	72
7.3.2: Produrre un messaggio dentro al Container	72
7.3.3: Produrre un messaggio dentro al Container CDI.....	73
7.4: Scrivere un Consumer di messaggi.....	73
7.4.1: Consegna Sincrona	73
7.4.2: Consegna Asincrona	74
7.5: Filtro dei messaggi	75
7.6: Time to Live dei messaggi.....	75
7.7: Priorità dei messaggi	76
7.8: Message-Driven Bean.....	76
7.8.1: Annotazione @MessageDriven	77

7.8.2: Dependency Injection.....	77
7.8.3: MDB Context	77
7.9: Ciclo di vita di un MDB e i Callback.....	78
7.10: Esercizio riepilogativo	78
7.10.1: Scrivere il POJO OrderBean	78
7.10.2: Scrivere il producer OrderProducer	79
7.10.3: Scrivere il consumer OrderMDB.....	80
7.11: Esercizio completo (persistenza e messaging)	80
7.11.1: Scrivere il POJO Museum.....	81
7.11.2: Scrivere MuseumEJB e la sua interfaccia remota.....	81
7.11.3: DatabaseProducer e DatabasePopulator	82
7.11.4: Interceptor Binding.....	83
7.11.5: Client per le regioni	84
7.11.6: Scrivere il MDB MessagingController	85
7.11.7: Scrivere il gestore di eventi Observer	85
7.11.8: Client per il messaging.....	85
Capitolo 8: SOAP Web Services	86
8.1: Tecnologie e protocolli	86
8.1.1: WSDL	86
8.1.2: SOAP	87
8.1.3: UDDI.....	87
8.1.4: Protocollo di Trasporto.....	87
8.2: Definire un SOAP WS.....	87
8.3: Anatomia di un SOAP WS	88
8.4: Mapping WSDL	89
8.4.1: L'annotazione @WebService.....	89
8.4.2: L'annotazione @WebMethod	89
8.4.3: L'annotazione @WebResult	90
8.5: Ciclo di vita di un SOAP WS	90
8.6: Utilizzare un SOAP WS.....	91
8.6.1: Invocazione di un SOAP WS fuori dal container	91
8.7: Esercizio riepilogativo.....	92
8.7.1: Scrivere i POJO Account e Transfer	92
8.7.2: Scrivere il SOAP WS	92
8.7.3: Scrivere il client	93
8.7.4: Situazione attuale.....	94

8.7.5: Funzionamento.....	95
Riferimenti	96
Extra: Setup Progetto con EJB	97
Creare un progetto per l'EJB	97
Creare un progetto per il client	98
Includere le librerie	99
Includere le librerie del client.....	99
Includere le librerie del server.....	101
Librerie per i SOAP WS	101
Codice e deployment.....	103
Extra: Setup Oggetti Amministrati.....	105
Impostare i parametri di avvio	106
Extra: Setup SOAP WS	108
Aggiornamento del SOAP WS	109
Struttura Client – Web Service	109

Capitolo 1: Introduzione a Java EE 7

Oggigiorno le aziende operano a livello globale, distribuendo i propri sistemi in tutto il mondo e permettendone il funzionamento ogni giorno ad ogni orario. Si cerca, quindi, di offrire un sistema distribuito, portabile, affidabile, sicuro ed efficiente spendendo il meno possibile, garantendo le caratteristiche appena elencate. Java Enterprise Edition cerca di rispondere a queste esigenze.

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.

Java EE introduce una serie di nuove funzioni, tra le quali la Context and Dependency Injection (CDI), la quale permette, in poche parole, l'iniezione automatica di risorse. Vengono inoltre 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: I 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 servono 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 accessi sia localmente che in remoto tramite RMI.

1.3: Container

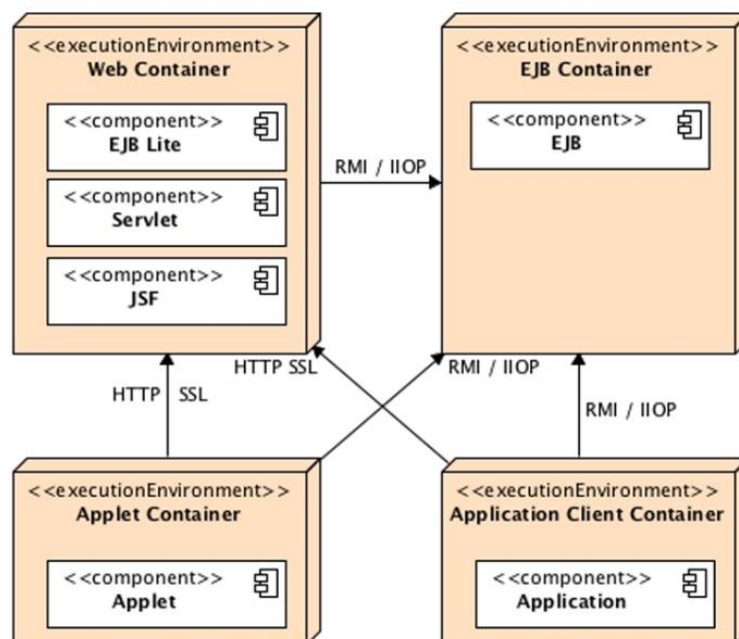
L'infrastruttura di Java EE è partizionata in diversi domini denominati come container, i quali offrono servizi alle componenti supportate (sicurezza, accesso al database, transazioni, servizio di

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:

- Applet container, fornisce un ambiente in cui vengono delineati i limiti nel quale deve essere eseguito il codice;
- Application 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;
- Web container, adatto al managing e all'esecuzione di componenti web come servlets, JSPs, filtri, listeners, ecc. Tale container è responsabile riguardo l'istanziamento, l'inizializzazione e l'invocazione delle servlets, utilizzando HTTP;
- 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.

La seguente immagine mostra i diversi container che Java EE introduce: nei container vengono mostrati i componenti supportati, mentre le frecce che collegano i container mostrano i protocolli che i container utilizzano per comunicare tra loro.



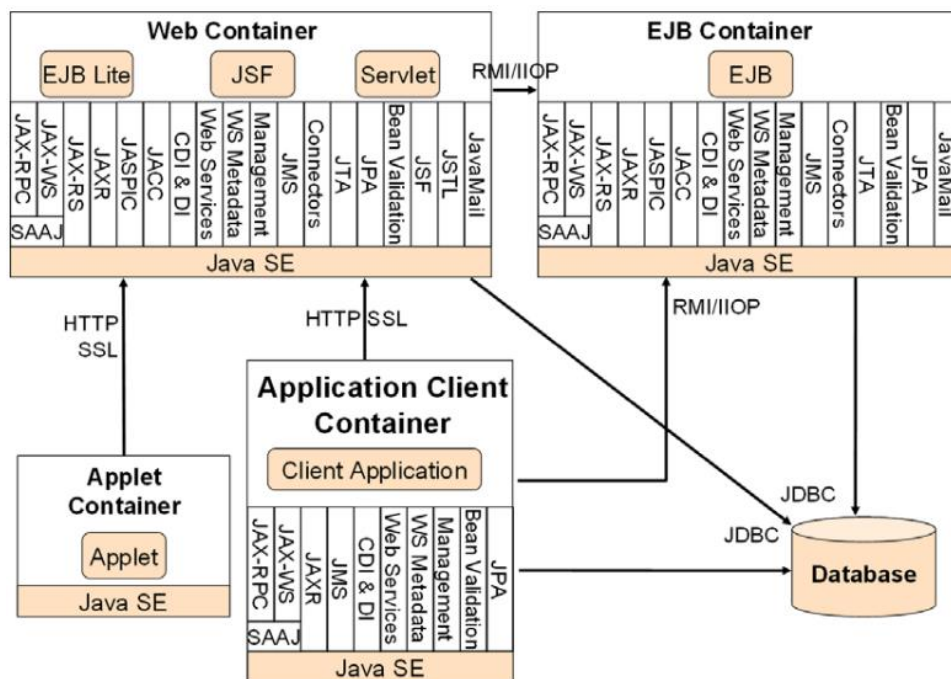
1.4: Servizi

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, in generale, 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: questo servizio permette l'utilizzo delle transazioni, cioè serie di istruzioni le quali vengono eseguite tutte o nessuna;

- JPA – Java Persistence API: questo servizio 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: questo servizio permette alle componenti di comunicare tra loro asincronamente tramite messaggi;
- Naming: questo servizio permette di associare e trovare oggetti tramite operazione di lookup. Tale servizio viene utilizzato anche tramite iniezione, ma in maniera trasparente;
- JavaMail: questo servizio permette all'applicazione di mandare email;
- XML processing: questo servizio effettua la manipolazione di codice XML;
- JSON processing: questo servizio effettua la manipolazione di codice JSON;
- Servizi di sicurezza;
- Web services;
- Dependency Injection (DI): alcune risorse possono essere iniettate in determinati componenti (managed components);
- Deployment.

Nella seguente immagine vengono mostrati i servizi categorizzati in base ai container.



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 e l'approccio dichiarativo: l'imperativo tende a specificare l'algoritmo (cosa si usa per risolvere) che risolve un determinato problema, mentre il dichiarativo tende a specificare come (come si risolve) deve essere risolto un determinato problema. In Java EE, l'approccio dichiarativo utilizza metadati,

annotazioni e deployment descriptors. Vedremo in particolare le annotazioni, le quali seguono tale formato: @annotation.

Capitolo 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 preso controllo del codice di business e che avesse provveduto 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). In questo capitolo approfondiremo la dependency injection (DI), lo scoping, il basso accoppiamento e concetti che hanno a che fare con CDI.

2.1: Panoramica sui bean

Mentre Java SE possiede solamente i Java Bean, Java EE possiede gli Enterprise Java Bean (EJB). Gli EJB 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.

I managed bean sono bean gestiti dal container che supportano solo determinati servizi, quali sono 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, come una servlet può esser vista come un managed bean con altri servizi aggiuntivi.

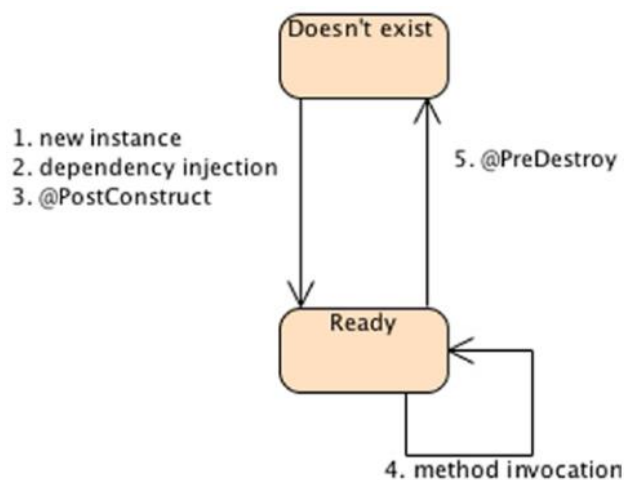
I CDI bean sono una categoria di managed bean con un ciclo di vita migliorato per oggetti con uno stato (stateful objects); sono legati ad un contesto ben definito; permettono la DI ben tipizzata, l'intercezione e l'uso delle annotazioni. Precisamente, tranne in alcune eccezioni, ogni classe Java con un costruttore di default ed eseguita in un container viene considerata come bean.

2.2: Cos'è la 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, quindi, il container effettua in automatico il lookup dell'oggetto per poi iniettarlo. È bene sapere che 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 permesso ai programmatori di iniettare determinate risorse in determinati componenti.

2.3: Ciclo di vita di un CDI bean

Il ciclo di vita di un normalissimo POJO è semplicissimo: 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 decisamente 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. A questo punto sorge spontanea una domanda: come si inizializza un bean se non si richiama il suo costruttore? Ebbene, 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.



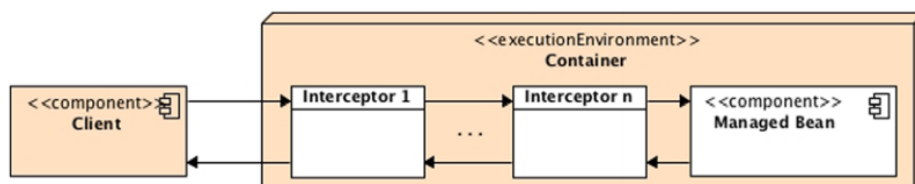
2.4: Scope e contesto

I CDI bean sono contestuali, quindi hanno uno scope ben definito: sostanzialmente, 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: Introduzione 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: Introduzione 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: Introduzione alla pratica sui CDI bean

Un CDI bean non è altro che una classe Java contenente logica di business. Come già detto, i CDI bean non possono essere istanziati tramite keyword “new”, quindi è necessario utilizzare l’iniezione. L’iniezione viene eseguita tramite apposita annotazione @Inject, ma vedremo meglio, tra poco, 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 con i 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.

```
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;  
    }  
}
```

Come possiamo ben vedere, 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. 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: se il generatore di ISBN ed il generatore di ISSN implementano entrambi NumberGenerator, il problema è risolto ma c’è comunque dipendenza a causa dell’interfaccia. A questo punto di 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. In poche parole, 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, mostrato nella successiva pagina, 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'istanziamento del bean.

L'iniezione può avvenire in tre modi differenti: tramite proprietà, tramite metodo setter e tramite costruttore. Nell'esempio utilizzando la @Inject viene mostrata l'iniezione tramite proprietà, quindi nei successivi due esempi vengono mostrati in ordine l'iniezione tramite setter e l'iniezione tramite costruttore.

```
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. Sostanzialmente, quindi, 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, in tal caso, viene anche detta default injection (iniezione di default).

È bene precisare, prima di continuare, il fatto che si possono avere più implementazioni della stessa interfaccia: cambierà il modo con cui viene effettuata l'iniezione (lo vedremo tra poco), ma tra le molteplici implementazioni è possibile specificarne una di default.

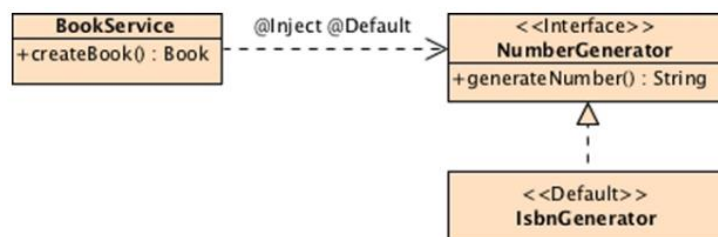
Ricapitolando: se esiste un'unica implementazione dell'interfaccia, allora quell'unica implementazione viene detta di default; se esistono molteplici implementazioni dell'interfaccia, una di queste può essere specificata come default. Ovviamente, nel caso si voglia iniettare un'implementazione di default, è necessario specificarlo tramite apposita annotazione. Mostriamo, di seguito, la specifica dell'implementazione di default e come viene effettuata l'iniezione di default.

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;
```

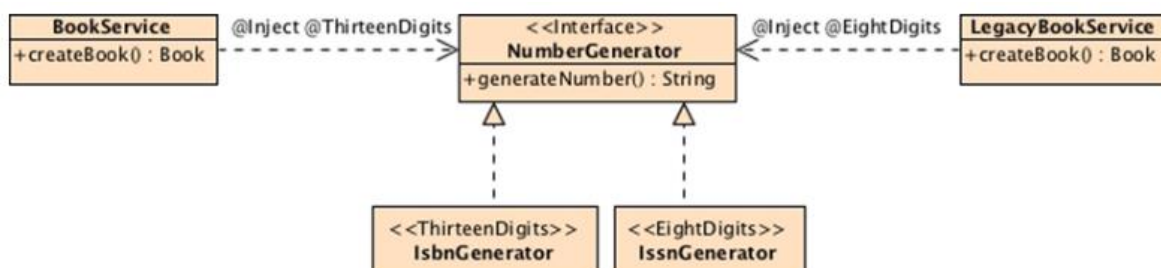


2.9.2: Qualificatori - specificare l'implementazione

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.

Discutiamo, ora, 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, in maniera generale, come funzionano i qualificatori:



Il qualificatore è un'annotazione definita dal programmatore e specifica un tipo associato ad un'implementazione. È utile, quindi, 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, quindi, 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 visto nel capitolo precedente, nel caso si vogliano distinguere molteplici implementazioni sarebbe necessario creare annotazioni extra (ad esempio @TwoDigits, @EightDigits, @TenDigits, @ThirteenDigits), rendendo il codice decisamente 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 vogliono.

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;
    }
}
```

```
@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

Nei capitoli precedenti 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, innanzitutto, 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 (come solito 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, quindi quale verrà iniettata? Ebbene, 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'istanziamento: 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 finchè 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), i quali vediamo di seguito:

- 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, quindi, 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 anche 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

Nel capitolo precedente 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, quindi, 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, però, che tale definizione non regola quale interceptor viene eseguito prima dell'altro, quindi non definisce in qualche modo una priorità tra interceptor: è possibile, però, 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 definisce e lancia eventi, l'observer 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, quindi, 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. Dopo l'esempio riguardante la dichiarazione degli eventi e il loro lancio, vediamo anche l'esempio in cui essi vengono gestiti tramite qualificatori.

```

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);
    }

}

```

```

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);
    }

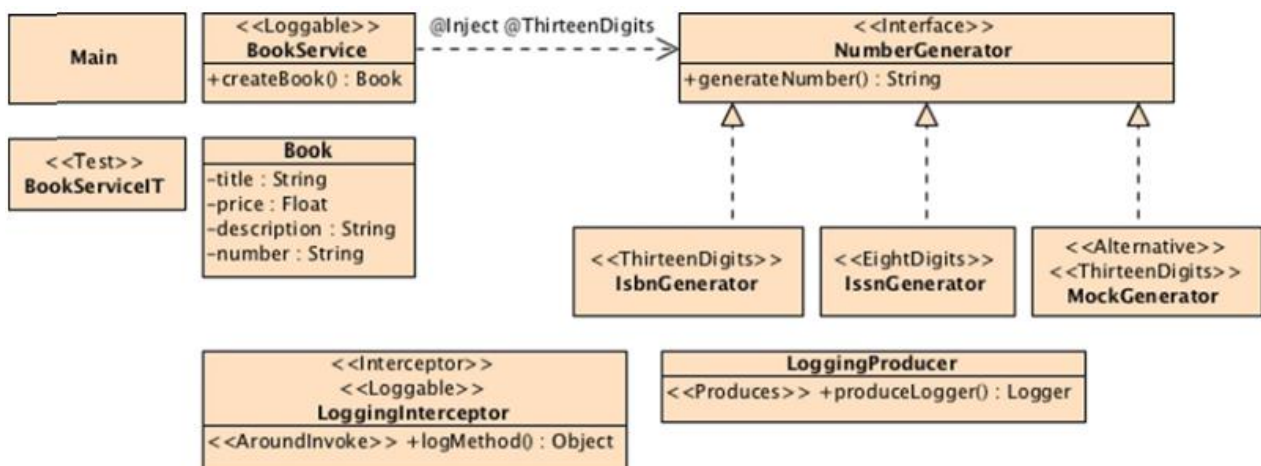
}

```

2.13: Esercizio riepilogativo

Abbiamo visto tanti concetti insieme affiancati da esempi con codice, come lo scrivere i bean, producers, l'utilizzo dell'iniezione, qualificatori ed interceptor binding.

Vediamo ora un grande esempio in cui raccogliamo tutti i concetti visti finora. Prima, però, è bene avere una panoramica di quel che faremo tramite un semplice class diagram.



- La classe BookService contiene i metodi per la gestione dei libri, in tal caso solo riguardanti l'aggiunta;
- L'interfaccia NumberGenerator possiede due implementazioni, le quali permetteranno di generare codici ISBN e ISSN. La terza implementazione è di testing, ma non ci interessa. Le due implementazioni sono realizzate utilizzando i qualificatori;
- La classe LoggingProducer farà uso di producers per rendere iniettabile il logger, il quale terrà conto di ogni operazione che si effettuerà nel sistema;
- La classe interceptor binding LoggingInterceptor, distinto dall'interfaccia InterceptorLogger (annotazione personalizzata necessaria per l'interceptor binding), permetterà al logger di registrare le attività nel sistema in maniera del tutto automatica.

L'esempio che vedremo non sarà totalmente completo siccome non vedremo la classe Main, siccome non abbiamo ancora confidenza con i container.

2.13.1: Scrivere il POJO Book

Il nostro sistema utilizzerà la classe BookService per gestire libri, quindi necessitiamo di una classe Book che rappresenterà i nostri libri. Tale classe descriverà un normalissimo libro fornendo le sue caratteristiche senza far uso di annotazioni.

```
public class Book {
    private String title;
    private String description;
    private String number;
    private Float price;

    public Book(String title, String description, Float price) {
        this.title = title;
        this.description = description;
        this.price = price;
    }

    public String getTitle() {
        return title;
    }
    public String getDescription() {
        return description;
    }
    public String getNumber() {
        return number;
    }
    public Float getPrice() {
        return price;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public void setNumber(String number) {
        this.number = number;
    }
    public void setPrice(Float price) {
        this.price = price;
    }
}
```

2.13.2: Scrivere NumberGenerator, implementazioni e qualificatori

Sappiamo che ogni libro possiede un seriale che lo distingue dagli altri: tale seriale viene creato dal BookService tramite apposito oggetto generatore. Tale generatore di seriali viene iniettato grazie ad @Inject e un qualificatore, il quale specifica quale generatore iniettare. Sappiamo, quindi, di dover creare l'interfaccia NumberGenerator e le sue implementazioni con tanto di qualificatori.

Iniziamo dall'interfaccia NumberGenerator.

```
public interface NumberGenerator {  
    public String generateNumber();  
}
```

Vediamo i qualificatori che distingueranno le implementazioni.

```
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 {}
```

```
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 EightDigits {}
```

Vediamo le implementazioni dell'interfaccia NumberGenerator, le quali saranno collegate ai qualificatori che abbiamo appena stabilito.

Di seguito, vediamo l'implementazione di NumberGenerator qualificata dall'annotazione @ThirteenDigits.

```
@ThirteenDigits  
public class GeneratorThirteen implements NumberGenerator {  
  
    @Inject  
    private Logger logger;  
  
    @Override  
    public String generateNumber() {  
        String serial = "13-84356-" + Math.abs(new Random().nextInt());  
        logger.log(Level.INFO, "Generated ISBN : {0}", serial);  
        return serial;  
    }  
}
```

Di seguito, invece, vediamo l'implementazione di NumberGenerator qualificata dall'annotazione @EightDigits.

```

@EightDigits
public class GeneratorEight implements NumberGenerator{

    @Inject
    private Logger logger;

    @Override
    public String generateNumber() {
        String issn = "8-" + Math.abs(new Random().nextInt());
        logger.log(Level.INFO, "Generated ISBN : {0}", issn);
        return issn;
    }
}

```

2.13.3: Scrivere il gestore dei libri BookService

Avendo creato il POJO Book ed il NumberGenerator, il quale permette di creare identificativi per i libri, possiamo creare BookService, il quale è il gestore che permetterà l'effettiva creazione dei libri. Si vuole che la creazione del libro comporti la generazione di un seriale, il quale viene creato dal NumberGenerator.

```

public class BookService {

    @Inject @ThirteenDigits
    private NumberGenerator generator;

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

```

In tal caso viene iniettata l'implementazione annotata con @ThirteenDigits, quindi nel caso si volesse utilizzare GeneratorEight al posto di GeneratorThirteen basterebbe cambiare l'annotazione con cui si inietta il NumberGenerator.

2.13.4: Rendere iniettabile la classe Logger

Nell'applicazione utilizziamo la classe Logger per stampare in output informazioni riguardo le istruzioni che si stanno eseguendo. La classe Logger, però, non è iniettabile siccome appartiene alla JDK e si necessita dei producers per renderla tale. Il producer è dichiarato nella classe LoggingProducer.

```

public class LoggingProducer {

    @Produces
    public Logger produceLogger(InjectionPoint injectionPoint){
        return Logger.getLogger(injectionPoint.getMember().
            getDeclaringClass().getName());
    }
}

```

In tal modo, la classe Logger è resa iniettabile grazie ad un metodo annotato con @Produces.

Quando verrà richiesta l'iniezione di un Logger, verrà eseguito il metodo produceLogger che restituirà il logger da utilizzare.

2.13.5: Creare l'interceptor che utilizza il Logger

Abbiamo visto l'iniezione di un Logger in molteplici classi, resa ora possibile grazie al producers. Probabilmente abbiamo fatto caso al fatto che tali Logger vengono iniettati ma non utilizzati: come mai? Ebbene, il Logger interverrà da solo all'utilizzo di determinati metodi, stampando in output informazioni riguardo il metodo che si sta utilizzando. Il fatto che il Logger intervenga da solo è reso possibile dall'interceptor.

Si vuole, quindi, creare una classe interceptor binding che contenga i metodi invocati automaticamente dall'esecuzione di determinati metodi: ciò implica che i metodi contenuti verranno annotati con `@AroundInvoke`, siccome è l'annotazione che stabilisce che quel determinato metodo verrà eseguito prima dell'esecuzione di un altro metodo (vedi l'apposito capitolo). Sappiamo, però, che la classe interceptor binding necessita di un'annotazione personalizzata, quindi ne vediamo la sua dichiarazione.

```
import javax.interceptor.InterceptorBinding;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

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

Creata l'annotazione personalizzata, vediamo l'interceptor binding vero e proprio.

```
@Interceptor
@InterceptorLogger
public class LoggingInterceptor {

    @Inject
    private Logger logger;

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

Da questo momento, quindi, sarà possibile utilizzare l'annotazione `@LoggableInterceptor` per far in modo che l'invocazione di un metodo venga preceduta dall'esecuzione del metodo annotato con `@AroundInvoke`.

2.13.6: Applicare l'interceptor `@LoggableInterceptor`

Facendo un punto della situazione, abbiamo classi che ottengono tramite iniezione il Logger ma non che lo utilizzano: per permettere ciò, è necessario annotare i metodi con

@LoggerInterceptor, in modo da far sapere che la loro invocazione sarà preceduta dall'esecuzione dell'interceptor.

Refactorizziamo, quindi, le classi in cui si voglia che il Logger faccia il suo compito, aggiungendo l'annotazione per eseguire l'interceptor.

```
@LoggerInterceptor
public class BookService {

    @Inject @ThirteenDigits
    private NumberGenerator generator;

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

```
@ThirteenDigits
public class GeneratorThirteen implements NumberGenerator{

    @Inject
    private Logger logger;

    @Override
    @LoggerInterceptor
    public String generateNumber() {
        String serial = "13-84356-" + Math.abs(new Random().nextInt());
        logger.log(Level.INFO, "Generated ISBN : {0}", serial);
        return serial;
    }
}
```

Non bisogna, però, dimenticare di abilitare l'interceptor creato. Vediamo il file beans.xml.

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

Capitolo 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, quindi, 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, quindi, i database relazionali non possono trattare determinate cose come istanze di oggetti, interfacce, classi astratte, annotazioni, metodi, attributi, ecc.

Quel che abbiamo visto, però, non implica il fatto che non esista alcun modo per rendere tali dati persistenti. Innanzitutto, cosa si intende per dati persistenti? 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: Introduzione 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, è possibile 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: l'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 numPages;
    private Boolean illustrations;

    public Book() {}

    // Getters and setters
}
```

In sostanza, quindi, 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 senza alcun argomento, quindi un costruttore vuoto. Ciò non implica il fatto che non possano esserci altri costruttori oltre a quello vuoto;
- 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 (vedremo più avanti cosa significa), la classe dell'entità deve implementare Serializable.

3.3: Riguardo l'Object Relational Mapping (ORM)

Il principio di ORM è quello di delegare a tools o framework esterni (nel nostro caso, 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? Come fa JPA a mappare 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, quindi, adatta ad un database.

Come possiamo ben vedere, una semplice annotazione stravolge il comportamento che avrebbe il gestore della persistenza verso tale oggetto. Altro esempio è l'annotazione @Id: con tale annotazione, 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, quali sono:

- 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: Introduzione 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, quindi, 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, principalmente, possono essere create dinamicamente a runtime o staticamente a tempo di compilazione. Il seguente esempio mostra la query statica, cioè la named query. Gli altri tipi di query li vedremo più avanti siccome questa è solamente un'introduzione.

```

@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 numPages;
    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 seguente codice.

```

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. Vediamo ora un esempio completo per poi commentarlo.

```

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();

    }

}

```


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 (vedremo le transazioni tra qualche capitolo): tramite il metodo persist, si salva l'oggetto nel database, quindi viene reso persistente (in realtà non è proprio così, approfondiremo tra qualche paragrafo). 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.

3.5: Persistence Unit

Abbiamo finora discusso in generale il funzionamento delle query e dell'entity manager, quindi stiamo iniziando a toccare con mano il database. Abbiamo visto un esempio che mostra come si interagisce con il database, ma non abbiamo visto quale driver viene utilizzato per la comunicazione, come viene effettuata la connessione, ecc. L'esempio completo del precedente paragrafo pecca di numerose informazioni fornite dalla 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>
```

Prima di continuare, è bene fare un piccolo ed importante commento su un paio di cose. È bene sapere che l'ambiente in cui si lavora può essere application-managed e container-managed (vedremo più avanti cosa significa): 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, appunto, 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: Accenni sul ciclo di vita delle entità e le callbacks

Come già anticipato, 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 solamente 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.

Maggiori dettagli nel capitolo 4.2.

Capitolo 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 decisamente 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. Non ne vedremo tutti i metodi, bensì vedremo direttamente i suoi utilizzi e i metodi che usa in determinate occasioni.

4.1.1: Ottenere un Entity Manager

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 pochino 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 (lo vedremo più avanti). Inoltre, come già anticipato, 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: Entità managed, detached e relazione con l'appartenenza al database

Prima di approfondire per bene l'Entity Manager, è necessario comprendere a fondo il concetto di entità managed e detached. È bene premettere che, a differenza di come si sia capito finora, il fatto che un'entità sia managed o detached non c'entra con la sua appartenenza al database. Sia chiaro sin da ora: il fatto che un'entità appartenga al database non implica il fatto 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.

Ora come ora non conosciamo i metodi e non abbiamo idea di cosa sia una transazione, ma ci basta sapere un paio di cose che approfondiremo successivamente: il metodo `persist`, al contrario di come detto precedentemente per semplificare le cose, 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.

Vediamo un esempio:

```

// 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

```

A questo punto si presenta un gran bel dubbio: come viene salvata l'entità `b` nel database? Ebbene, se la transazione va a buon fine effettuerà un flush automatico del persistence context, liberandolo di ogni entità e salvandole nel database. Nel caso la transazione non vada a buon fine, viene effettuato un rollback, quindi si ritorna allo stato precedente all'inizio della transazione.

Comprendiamo, finalmente, che un'entità può essere managed o detached a prescindere dal fatto che si trovi o meno nel database.

4.1.3: Persistence 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: sostanzialmente, le entità sono conservate in tale spazio di memoria per la durata di una transazione (tra non molto vedremo cosa sia, ma sostanzialmente è un insieme di operazioni). 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. Abbiamo già visto tale file in un capitolo ad esso dedicato, quindi non ci soffermeremo a vederlo. Piuttosto, ci soffermiamo sul tag class: 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 il fatto che l'entità venga salvata nel database: ciò accade nel caso venga effettuato un flush sul persistence context.

```
// et is the EntityManager 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.

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

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;
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 finché 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 EntityManager 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 con il 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 e in questo paragrafo ne vedremo un paio. Ribadendo nel caso non si fosse capito, 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, per qualsiasi ragione, 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: stiamo parlando di 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. Leggi il capitolo 4.1.9 per comprendere al meglio.

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); detach permette di rendere un'entità detached.

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. Nel caso ci abbiate fatto caso, negli esempi riportati, 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 EntityManager 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

L'aggiornamento dell'entità nel database è decisamente più semplice di quel che si aspetta. Come già anticipato precedentemente, 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.

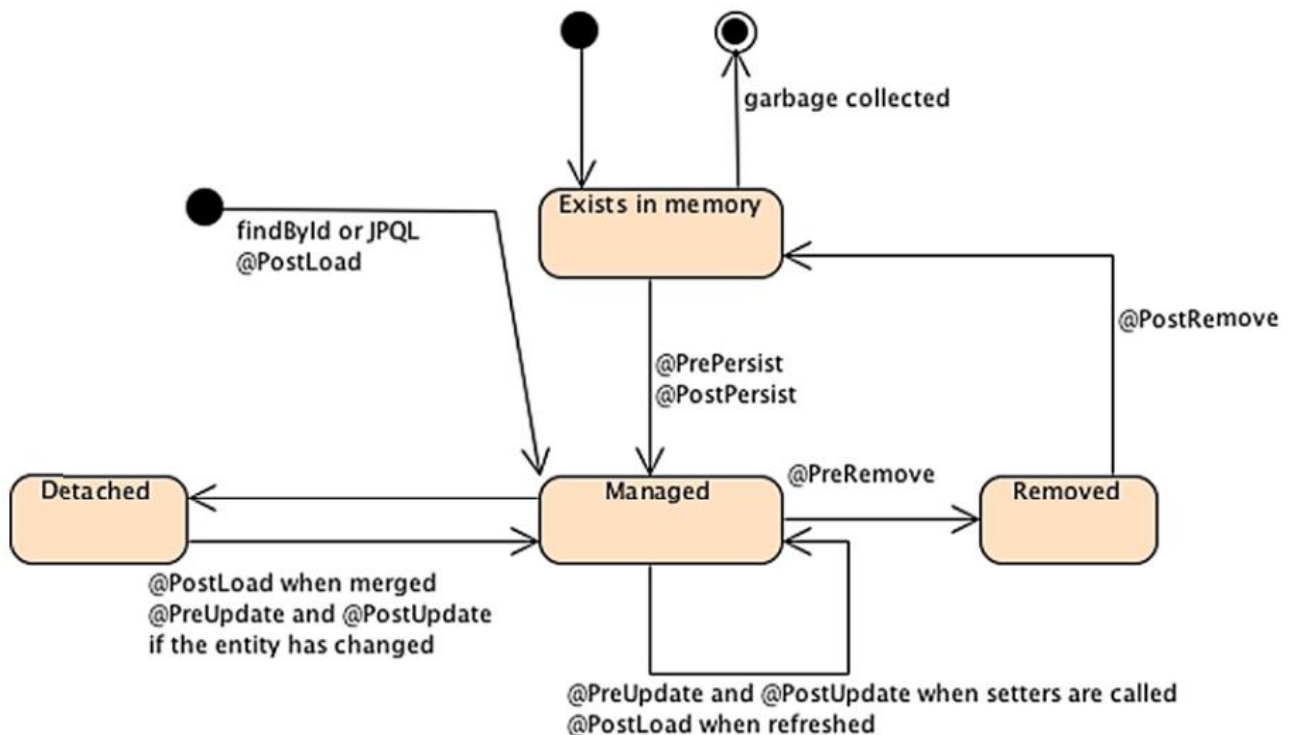
Il metodo merge si utilizza solamente per le entità che non sono più associate al persistence context, quindi solo per le entità detached di cui si ha la necessità di aggiornarla nel database, e serve ad apportare le modifiche apportate sull'entità nel database.

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

4.2: Ciclo di vita delle entità

Di seguito viene mostrato uno schema che raffigura gli stati delle entità in base alle esecuzioni dei metodi dell'Entity Manager. Nello schema non si accennano le 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, in generale, 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, quindi, 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. I 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.

Vediamo un esempio di callback:

```

@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

Come già anticipato, l'Entity Manager è in grado di eseguire query scritte in JPQL: ne esistono ben 5 tipi, ognuna con un funzionamento diverso: nei prossimi paragrafi vedremo le più rilevanti.

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 è davvero molto semplice: basta utilizzare 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 (vedremo tra poco).

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 argomento indica il valore da impostare.

Il primo esempio che vediamo mostra come vengono impostati i parametri posizionali, mentre 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 = ?1 AND b.price = ?2",
        Book.class);
    buildQuery.setParameter(1, title);
    buildQuery.setParameter(2, price);
    return buildQuery.getSingleResult();
}
```

```
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, in generale, 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 si 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 in 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, ad esempio, molteplici query con nome "findAll", mentre ne esisterà sicuramente e solamente 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 eventuali cambi alla query.

```
@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 aggiuntivo. La gestione della query è invariata, cambia solamente, quindi, 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

Per gli appassionati di SQL che non amano le tecnologie diverse, ecco le 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. Vediamo l'esempio:

```
@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`.

Capitolo 5: Enterprise JavaBeans

Nei due capitoli precedenti abbiamo approfondito le entità con il loro layer di persistenza, il quale non è affatto adatto per compiti complessi e per eseguire metodi di business. Per separare il layer di persistenza dal layer di business, quindi, 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, invece, 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 di business senza doversi preoccupare di diversi aspetti del codice che scrive. Andando avanti, approfondiremo meglio questo aspetto.

Nota: prima di continuare, è bene sapere che si parlerà di MDB e di session bean, in particolare di quest'ultimo. MDB e session bean sono due tipologie di EJB, le quali verranno approfondite più avanti.

5.1: Servizi offerti dal container

Come già minimamente anticipato, il container offre molteplici servizi alle applicazioni enterprise. Molti dei servizi che vedremo ci risulteranno sconosciuti o poco comprensibili. Per ora accenniamo solamente cosa offre il container EJB, poi li approfondiremo andando avanti:

- 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 thread-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, quindi, 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, quindi, 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;

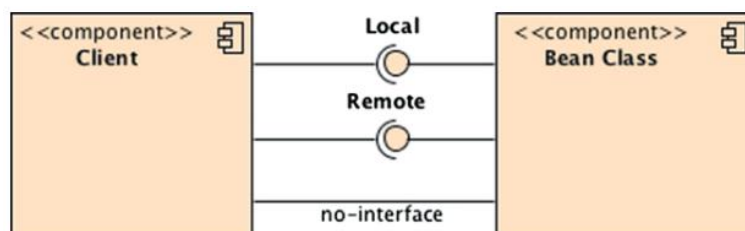
    public Book findBookByID(Long id){
        return em.find(Book.class, id);
    }

    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 decisamente 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: La classe Bean dell'EJB

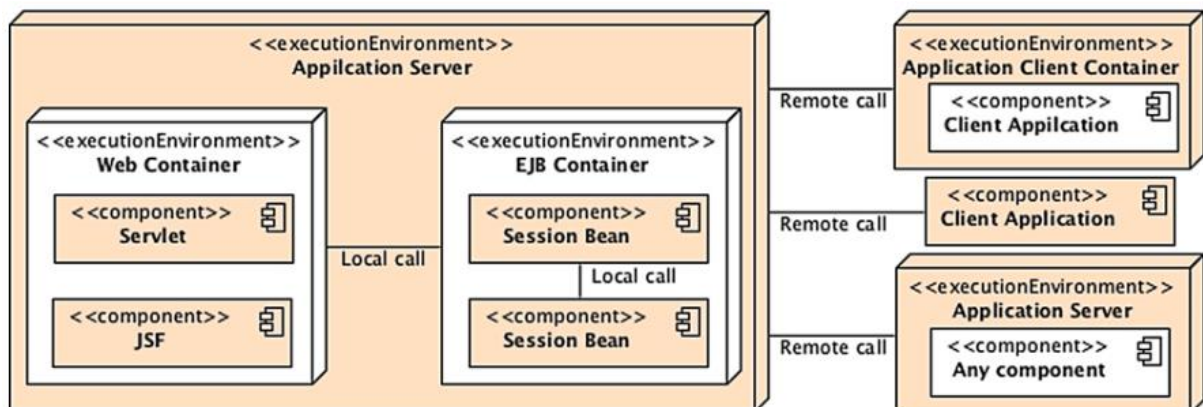
Come già detto, l'EJB è composto da una classe bean e da eventuali interfacce. La classe bean che lo compone deve rispettare alcuni criteri:

- 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 importantissima e da non dimenticare mai 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: in tal caso, 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, in generale, 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. Prima di vedere il perché, è bene premettere una cosa che spesso viene confusa: JNDI non ha assolutamente nulla a che vedere con la @Inject. JNDI non ritorna oggetti, anche perché, appunto, una applicazione Java SE non potrebbe gestire gli EJB! Si fa prima a vedere un esempio per comprenderne il reale funzionamento:

```
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"));  
  
    }  
  
}
```

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.

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, quindi, 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.

Di seguito, vedremo le tre tipologie di session bean esistenti, ognuna delle quali ha specifiche funzionalità.

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

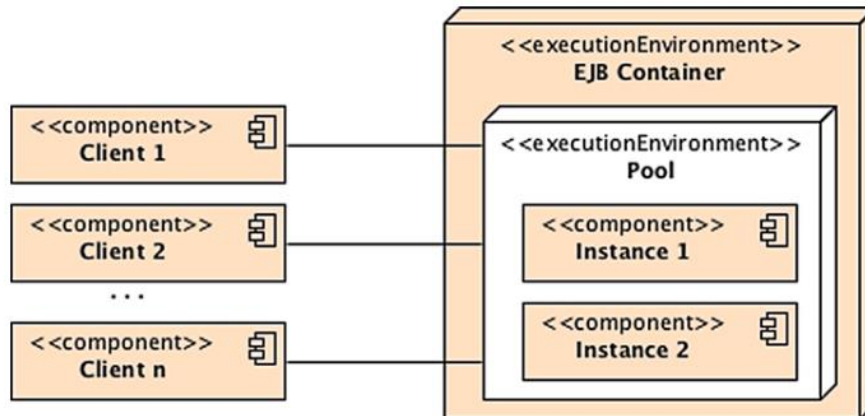
Gli stateless bean sono una tipologia di session bean decisamente efficiente, siccome fanno uso del pooling del container e possono essere condivisi tra molteplici client. Stateless significa senza stato, quindi sono bean che non conservano fondamentalmente alcuna informazione: contengono solamente operazioni 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 tranquillamente gestire un bel po' di clients.



5.4.2: Stateful Bean

Se in una mano abbiamo gli stateless bean che sono EJB senza stato, allora nell'altra mano abbiamo gli stateful bean che sono EJB che conservano lo stato. Gli stateful bean sono una tipologia di session bean in grado, quindi, di mantenere lo stato; sono estremamente 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<>();

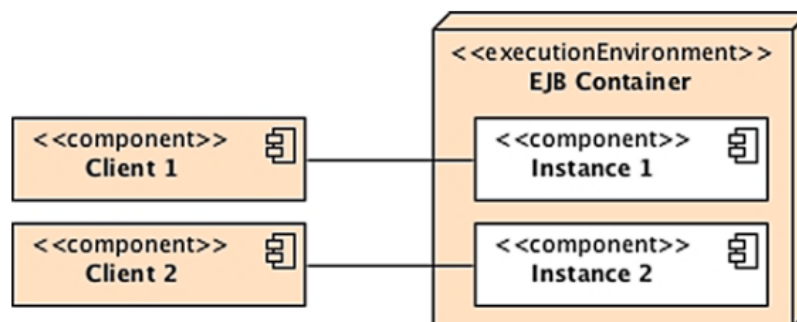
    @Override
    public void addBook(Book b) {
        if (!cartBook.contains(b))
            cartBook.add(b);
    }

    @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` e `@Remove`. L'annotazione `@StatefulTimeout` 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`; l'annotazione `@Remove` va posta su un metodo e permette di rimuovere permanentemente il bean una volta conclusa l'esecuzione del metodo. Ribadendo, sono annotazioni alternative: ciò viene già gestito dal container, quindi già quest'ultimo provvederebbe alla rimozione di tali bean se necessario. Risulta, però, conveniente utilizzare comunque tali annotazioni: in sistemi di grande calibro, lasciare completa gestione al container potrebbe portare problemi di performance.

Inoltre, come magari già intuito, per gli stateful bean non è presente un pool: il riutilizzo di tali bean per altri clients sarebbe in possibile siccome mantengono uno stato. Di conseguenza, quindi, 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 {

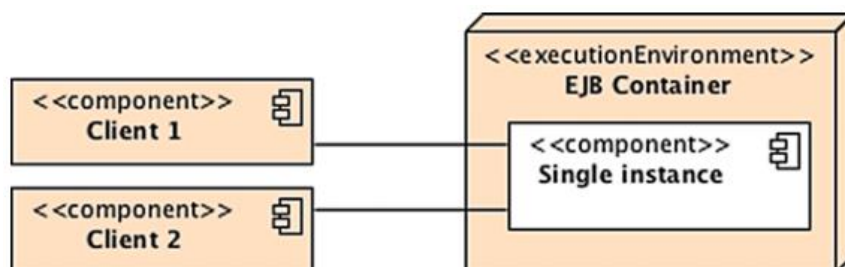
    @Inject
    private BookEJB manager;
    private Book first, second;

    @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);
    }
}
```

Sia chiaro: 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 (vedi capitolo 2.3).

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

Abbiamo già speso tante parole per la DI, ma la riprendiamo qui siccome è bene sapere che 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

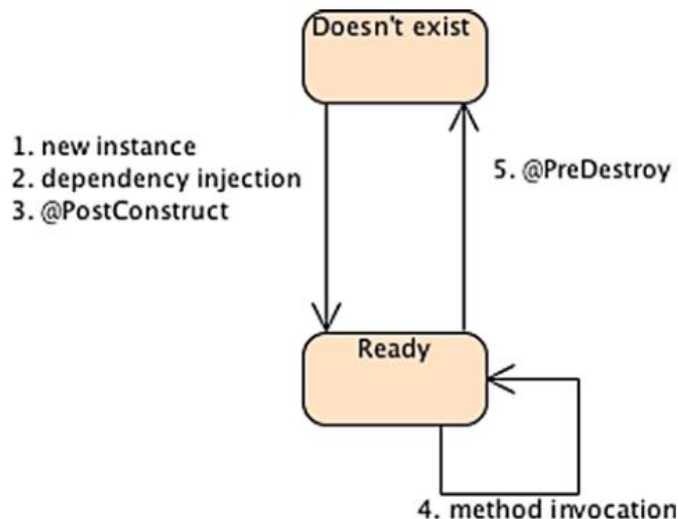
I session bean sono componenti gestiti dal container, quindi vivono in un container EJB che offre determinati servizi. Uno dei tanti servizi che offre è la gestione del ciclo di vita dei bean. In base al tipo di bean che si sta trattando (stateless, stateful, singleton), il ciclo di vita consisterà di diversi stati. Ogni volta che il container cambia il ciclo di vita di un bean, è possibile invocare metodi annotati detti callback.

Negli esempi dei precedenti capitoli 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, inoltre, condividono lo stesso ciclo di vita, il quale è il seguente:

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 di vita 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 finché non verrà rimosso dal pool del container; il singleton bean resta in attesa finché 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 del bean.



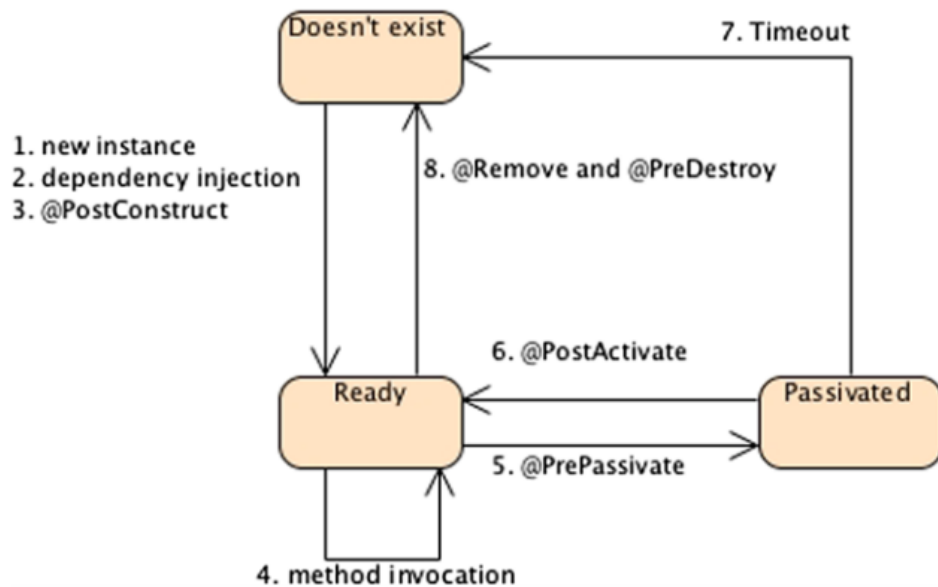
Concludendo, il ciclo di vita dello stateless bean e del singleton bean sono fondamentalmente uguali, con la differenza che lo stateless bean rimane in vita finché il container non lo rimuove dal pool, mentre il singleton bean rimane in vita per tutta la durata della vita del container.

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`.
 - a. Se il client non invoca il bean passivato per il tempo di timeout della sessione, allora il container lo distrugge definitivamente.
 - b. 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

Abbiamo appena visto che 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. Le annotazioni sono mostrate nelle illustrazioni dei precedenti due paragrafi.

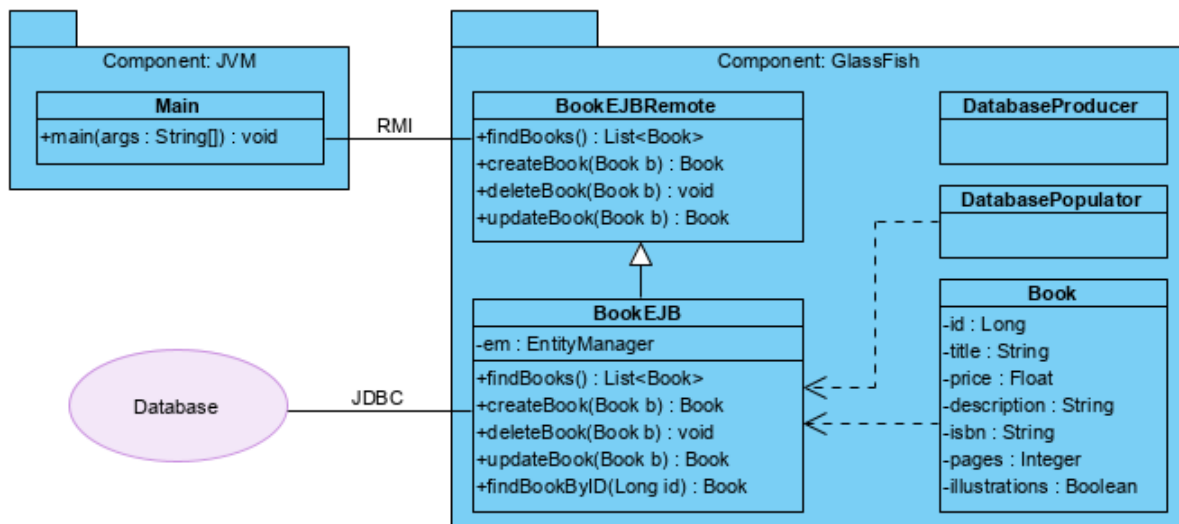
Un metodo, per essere callback, deve rispettare determinate regole:

- 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 prossimo capitolo);
- 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.

5.7: Esercizio riepilogativo

In questo esercizio riepilogativo affronteremo le tematiche viste sin dalla precedente esercitazione: vedremo, quindi, l'interfacciamento al database, l'uso delle query e dei session bean. Prima, però, è bene avere una panoramica di quel che faremo tramite un semplice class diagram.



Notiamo che stavolta abbiamo due componenti: la componente JVM rappresenta il client; la componente GlassFish rappresenta il server. Ergo, per iniziare tale esercizio è necessario seguire la guida “Setup Progetto EJB”, posta negli extra della guida.

- La classe Book è un semplicissimo POJO, salvabile nel database, il quale raffigura un libro;
- La classe BookEJB è un session bean che fornisce metodi per la gestione dei libri;
- L’interfaccia BookEJBRemote viene utilizzata dal client e fornisce tutti i metodi necessari per la gestione dei libri (quindi sono i metodi che BookEJB dovrà implementare);
- La classe DatabaseProducer fornisce un producer per l’Entity Manager, in modo da poterlo iniettare senza dover utilizzare ogni volta @PersistenceContext. Ciò favorisce la modularità, siccome in caso di cambiamenti della persistence unit basterà modificare solamente la classe DatabaseProducer;
- La classe DatabasePopulator gestisce il database.

5.7.1: Scrivere l’entità Book

La classe Book raffigura un libro, descrivendone le sue caratteristiche. Book è annotata da @Entity siccome è salvabile nel database, quindi è un’entità; è annotata da @NamedQueries siccome vengono dichiarate molteplici named query, le quali permettono di ricavare libri.

```

@Entity
@NamedQueries({
    @NamedQuery(name = Book.FIND_ALL, query = "SELECT b FROM Book b"),
    @NamedQuery(name = Book.FIND_BY_ID,
        query = "SELECT b FROM Book b WHERE b.id = :bookid")
})
public class Book implements Serializable{

    public static final String FIND_ALL = "Book.FINDALL";
    public static final String FIND_BY_ID = "Book.FINDBYID";

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

    public Book() {}

    public Book(String title, Float price, String description,
        String isbn, Integer pages, Boolean illustrations) {
        this.title = title;
        this.price = price;
        this.description = description;
        this.isbn = isbn;
        this.pages = pages;
        this.illustrations = illustrations;
    }

    //Getter and setters

```

5.7.2: Scrivere lo Stateless Bean BookEJB

BookEJB è uno stateless session bean che implementa le operazioni CRUD (create, read, update, delete) sull'entità Book. Tale EJB implementa un'interfaccia remota, BookEJBRemote, la quale fornisce al client i metodi richiamabili sull'EJB presente sul server.

```

@Remote
public interface BookEJBRemote {
    public List<Book> findBooks();
    public Book createBook(Book b);
    public void deleteBook(Book b);
    public Book updateBook(Book b);
}

```

L'EJB possiede un riferimento all'EntityManager grazie alla @Inject. Di norma non sarebbe possibile fare ciò, difatti è presente un producer che lo permette (lo vediamo tra poco). Il perché è stato spiegato sia nel paragrafo 5.7 che nel prossimo.

```

@Stateless
@LocalBean
public class BookEJB implements BookEJBRemote {

    @Inject
    EntityManager em;

    @Override
    public List<Book> findBooks() {
        TypedQuery<Book> objQuery =
            em.createNamedQuery(Book.FIND_ALL, Book.class);
        return objQuery.getResultList();
    }

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

    @Override
    public void deleteBook(Book b) {
        em.remove(em.merge(b));
    }

    @Override
    public Book updateBook(Book b) {
        Book updated = em.merge(b);
        return updated;
    }

    public Book findBookByID(Long id) {
        TypedQuery<Book> objQuery =
            em.createNamedQuery(Book.FIND_BY_ID, Book.class);
        objQuery.setParameter("bookid", id);
        return objQuery.getSingleResult();
    }
}

```

5.7.3: Scrivere il Database Producer

Come visto nel precedente paragrafo, l'EJB possiede un EntityManager iniettato tramite l'annotazione @Inject. Come ben sappiamo, per poter iniettare un Entity Manager in ambienti container managed, in teoria sarebbe stato necessario @PersistenceContext(unitName = ...). L'annotazione @Inject non può iniettare un Entity Manager, non potendo specificare il parametro. Abbiamo, però, imparato dal capitolo 2.9.4 che determinati oggetti possono essere resi iniettabili tramite i producers.

```

public class DatabaseProducer {
    @Produces
    @PersistenceContext(unitName = "bookPU")
    EntityManager em;
}

```

5.7.4: Persistence Unit

```
<?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="bookPU" transaction-type="JTA">
    <jta-data-source>jdbc/__default</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
      <property name="eclipselink.target-database" value="DERBY"/>
      <property name="eclipselink.ddl-generation"
        value="drop-and-create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

5.7.5: Scrivere il DatabasePopulator e definire il Data Source

Il data source mostrato nel paragrafo 5.7.4 deve essere creato in un EJB container. Per fare ciò, è possibile utilizzare l'annotazione `@DataSourceDefinition`: il container effettuerà il deploy del bean e creerà il data source.

```

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

    @Inject
    BookEJB controller;
    private Book first, second, third;

    private void initializeBooks() {
        first = new Book("Guide to Java EE", 20F, "Written by me",
            "1-8763-9125-7", 180, true);
        second = new Book("Guide to Python", 15F, "Another guide written by me",
            "1-8763-9482-8", 125, true);
        third = new Book("The world of SQL", 30.50F, "A bad guide of SQL",
            "1-8763-1241-7", 40, false);
    }

    @PostConstruct
    private void populate() {
        initializeBooks();
        controller.createBook(first);
        controller.createBook(second);
        controller.createBook(third);
    }

    @PreDestroy
    private void remove() {
        controller.deleteBook(first);
        controller.deleteBook(second);
        controller.deleteBook(third);
    }
}

```

5.7.6: Scrivere la classe con il main

```
public class ClientBook {

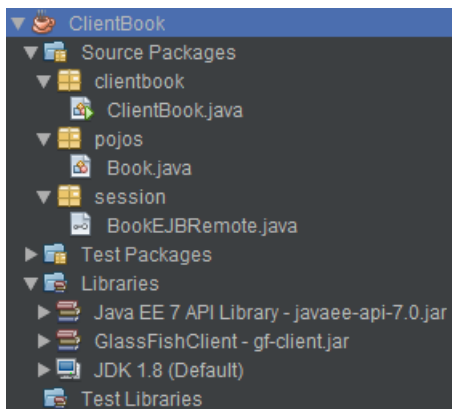
    public static void main(String[] args) throws NamingException {

        Context ctx = new InitialContext();
        BookEJBRemote controller = (BookEJBRemote)
            ctx.lookup("java:global/ServerBook/BookEJB!session.BookEJBRemote");
        List<Book> allBooks = controller.findBooks();
        allBooks.forEach((b) -> {
            System.out.println(b.getTitle());
        });
        Book newBook = new Book("Guide to GML", 60F, "Game Maker Language Guide",
            "1-8763-2847-1", 300, true);
        newBook = controller.createBook(newBook);
        newBook.setPrice(40F);
        controller.updateBook(newBook);
        allBooks = controller.findBooks();
        System.out.println(" >>> Updated books <<< ");
        allBooks.forEach((b) -> {
            System.out.println(b.getTitle());
        });
    }

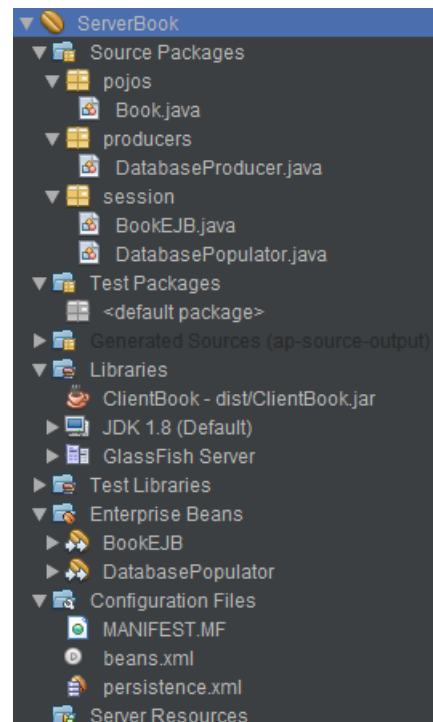
}
```

5.7.7: La situazione per il client e per il server

Progetto client



Progetto server



5.7.8: Testare

Deployare prima il server per poi mandare in run l'applicazione.

```
The world of SQL
Guide to Python
Guide to Java EE
>>> Updated books <<<
The world of SQL
Guide to Python
Guide to Java EE
Guide to GML
BUILD SUCCESSFUL (total time: 12 seconds)
```

Capitolo 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 non va a buon fine, 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, consistente, isolata e durabile. 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 una risorsa, 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

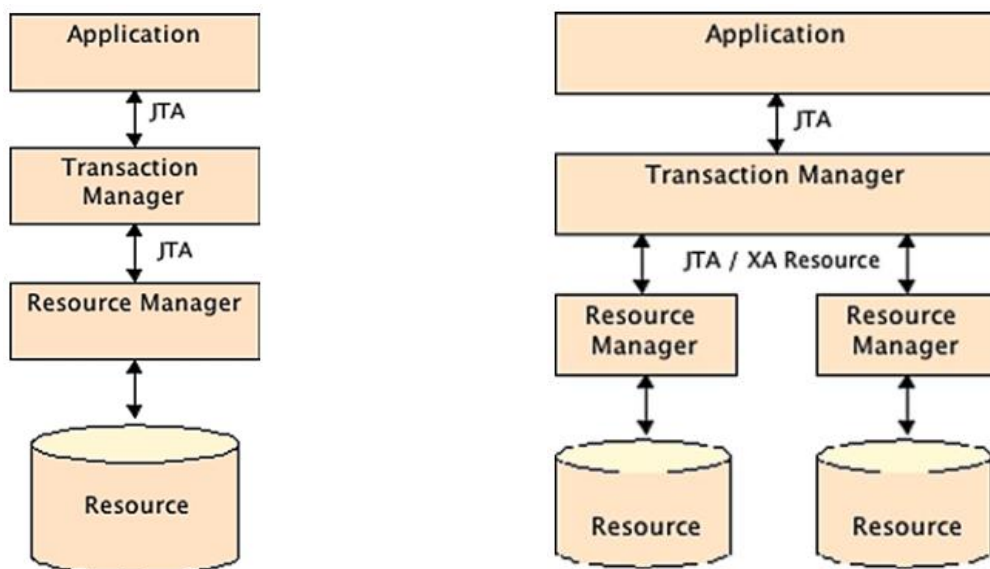
I database utilizzano diverse tecniche per regolare l'accesso concorrente: tali tecniche impattano sul livello di isolamento della transazione. I diversi livelli di isolamento sono i seguenti:

- 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.



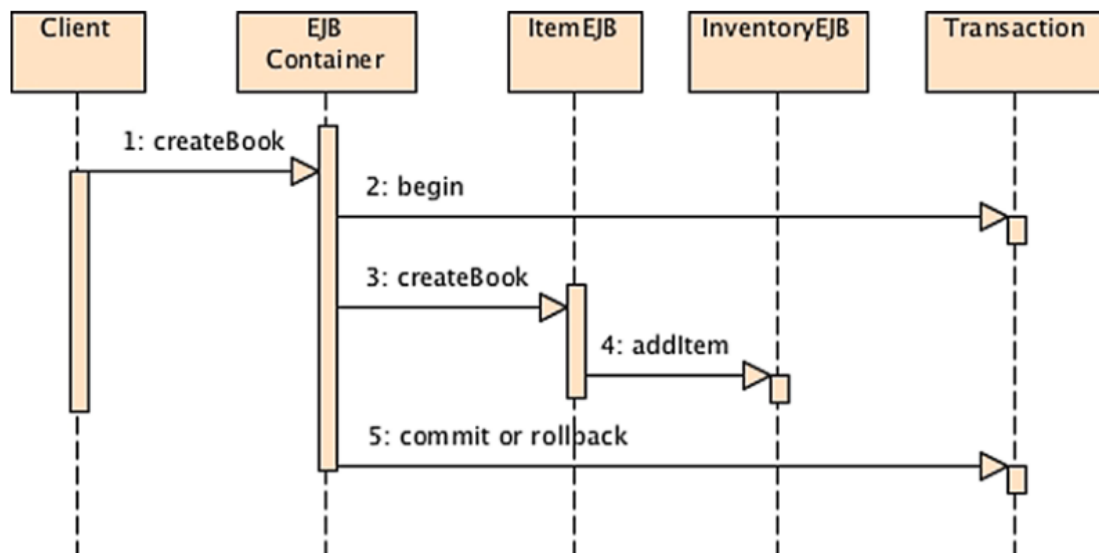
6.3: Transazioni negli EJB

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 generalmente scriveremmo.

```
@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);
    }
}
```

Capitolo 7: Messaging tra componenti

Per quel che abbiamo visto finora, 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.

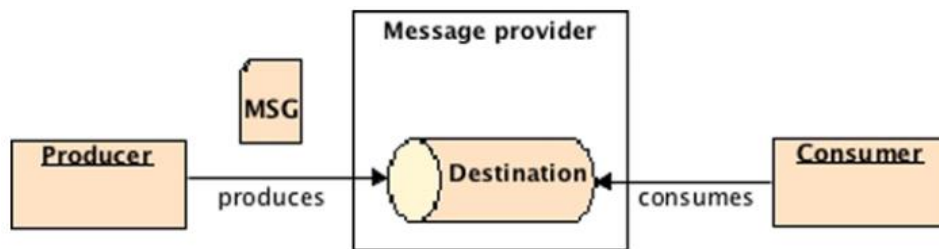
MOM (Message-Oriented Middleware) è un'infrastruttura che permette la comunicazione asincrona tra sistemi eterogenei, cioè diversi tra loro. Tale infrastruttura fornisce il provider, il quale può esser visto come un buffer che gestisce tali messaggi, dove sia il producer (colui che produce i messaggi) 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, quindi, una buona soluzione per l'integrazione di applicazioni esistenti e nuove in un modo a basso accoppiamento.

7.1: Introduzione a Java Message Service

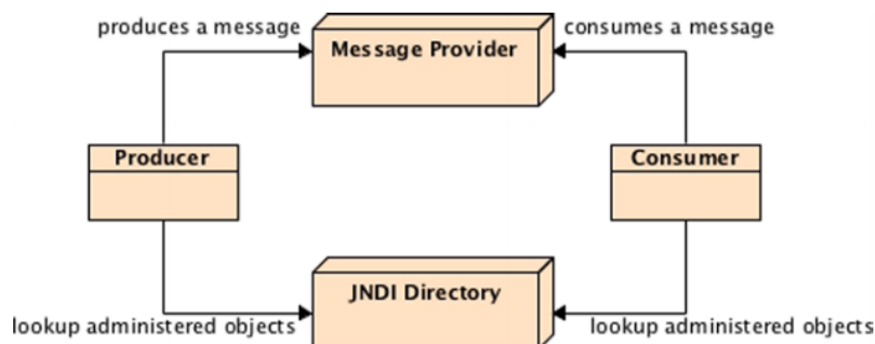
MOM è composto da diverse componenti, le quali entrano in azione quando un messaggio viene inviato. Il software che rispedisce i messaggi viene detto provider (detto anche broker), il quale può essere visto come una sorta di wrapper per una locazione che conserva messaggi, detta destinazione. La componente 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 di cui discuteremo tra poco, i Message-Driven Bean (MDB).

Riprendendo 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 (li vedremo tra poco).

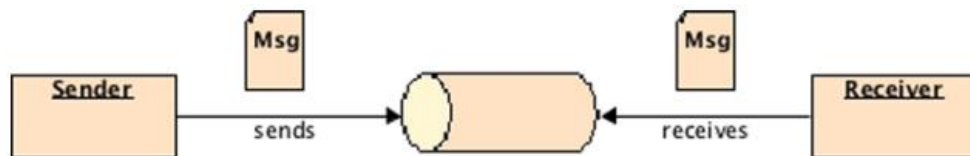


Il provider provvede una destinazione al suo interno, nella quale i messaggi possono essere conservati finchè non verranno consegnati al consumer. Esistono due tipi di destinazione:

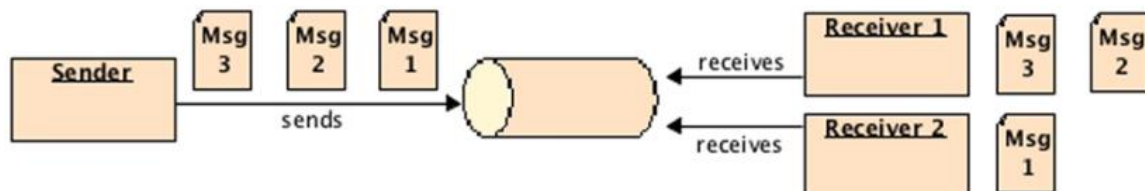
- Nel modello punto a punto (P2P), la destinazione che conserva i messaggi non è altro che una queue. In tale modello, un client inserisce un messaggio nella queue e un altro client lo riceve. Ricevuto il messaggio, il provider rimuove il messaggio dalla queue;
- Nel modello publish-subscribe (pub-sub), 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

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 finchè 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.

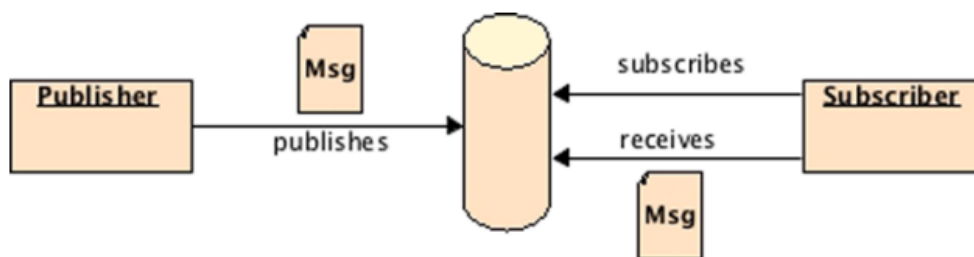


Il modello punto a punto può essere utilizzato solo se esiste un unico producer: ciò non implica che non possano esistere molteplici consumer. In tal caso, ogni messaggio può essere ricevuto da un unico consumer e non da molteplici contemporaneamente.

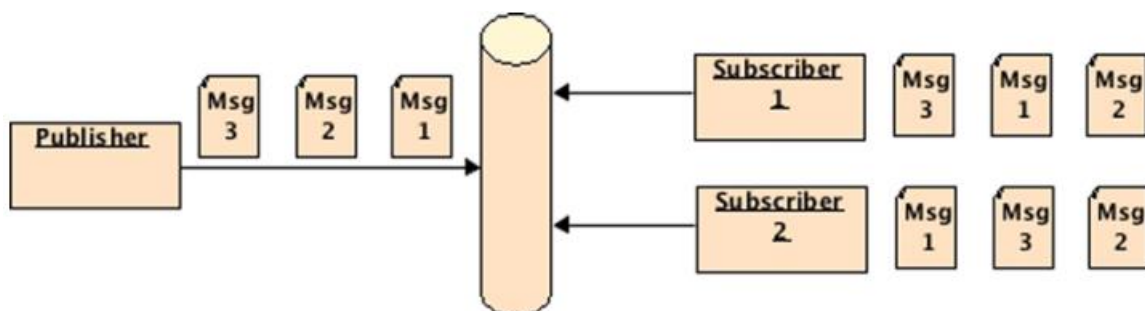


7.1.2: Modello Publish-Subscribe

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 finchè 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.

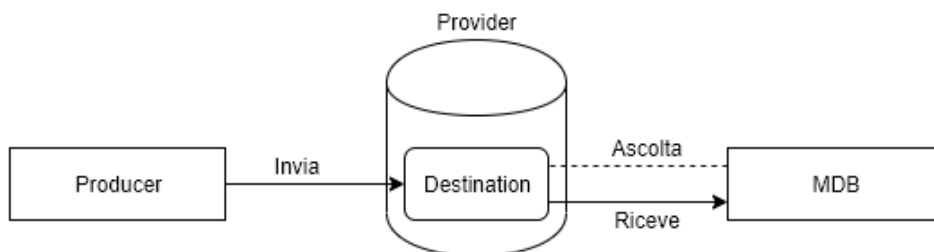


7.1.3: Oggetti Amministrati

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: Introduzione 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, quindi, 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.

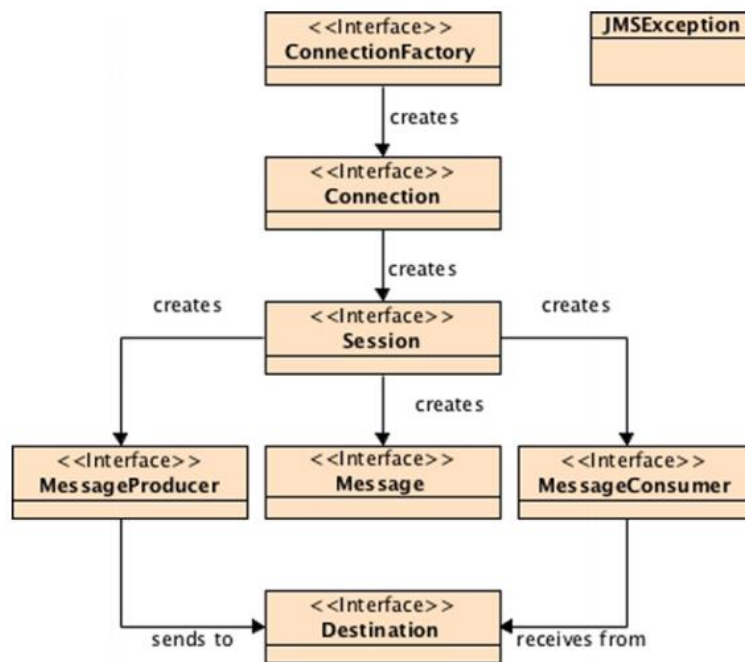
Riprenderemo i MDB nel capitolo 7.8, siccome nei prossimi paragrafi tratteremo i client producer e consumer senza uso dei MDB. Faremo ciò per comprendere a pieno il funzionamento dei producer e dei consumer, per poi passare ai MDB che sono consumer gestiti dal container.

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, per ragioni storiche, attualmente ci sono in giro diversi tipi di API, quali sono Legacy API, Classic API e Simplified API. Noi accenneremo la Classic API, ma ci concentreremo decisamente sulla Simplified 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 dispone 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 decisamente 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.

```

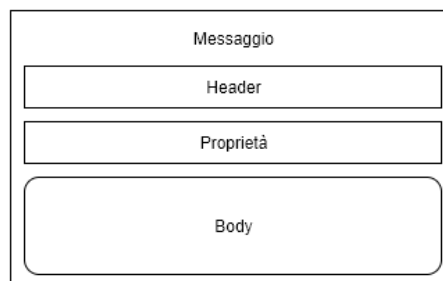
Session s = conn.createSession(true, Session.AUTO_ACKNOWLEDGE);

```

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("orderAmount") + 5F));

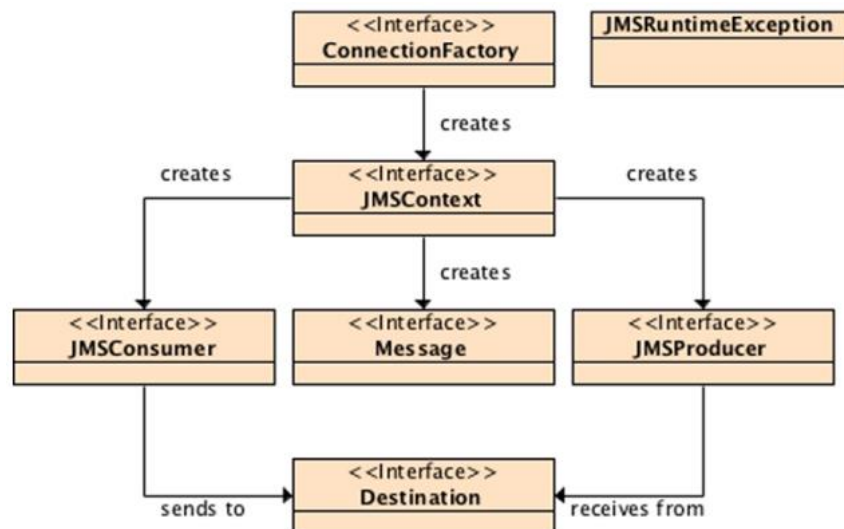
```

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, invece, 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, invece, 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 (lo vedremo tra poco).

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, come detto, 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 Container 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 creato 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 (JMSEException 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 (JMSEException 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. 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.

```
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);
```

7.7: Priorità dei messaggi

È possibile utilizzare le priorità nei messaggi in modo tale che il provider consegna 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.

Sorge una domanda: perché si dovrebbero utilizzare i MDB al posto dei client JMS visti poco fa? 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.

Sia chiaro, 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, come anticipato precedentemente, sarà possibile inviare messaggi per risolvere tale problema, ma lo vedremo più avanti.

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 (JMSEException 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;

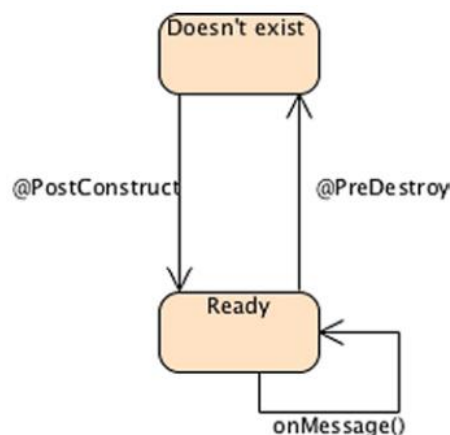
    @Override
    public void onMessage(Message message) {
        try{
            System.out.println("Received " +
                message.getBody(String.class));
        } catch (JMSEException e) {}
    }
}
```

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.

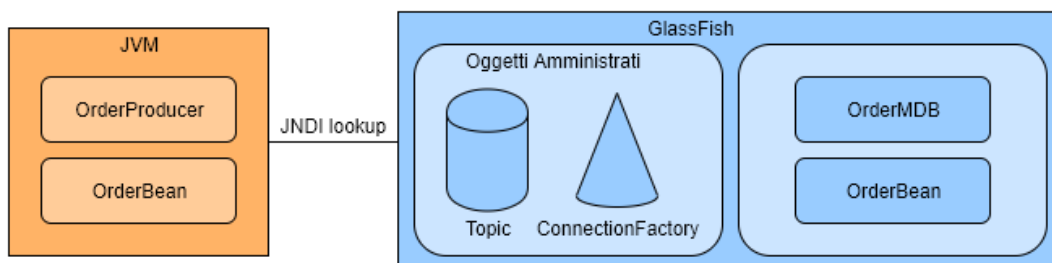
7.9: Ciclo di vita di un MDB e i Callback

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.



7.10: Esercizio riepilogativo

In questo esercizio riepilogativo affronteremo i concetti visti in tale capitolo, quindi i modelli P2P e pub-sub, oggetti amministrati, come connettersi ad un provider, come produrre e consumare messaggi e come utilizzare i MDB per ascoltare le destinazioni.



Anche stavolta abbiamo due componenti: la componente JVM rappresenta il client; la componente GlassFish rappresenta il server. Ergo, per iniziare tale esercizio è necessario seguire la guida "Setup Progetto EJB", posta negli extra della guida. Inoltre, è necessario seguire la guida "Setup Oggetti Amministrati", anch'essa posta negli extra della guida.

- `OrderBean` è un semplice POJO con le caratteristiche di un ordine online;
- `OrderProducer` è il client che produce messaggi;
- `OrderMDB` è il MDB che riceve e gestisce i messaggi;
- `Topic` è la destinazione che riceve i messaggi;
- `ConnectionFactory` permette di creare connessioni al provider.

7.10.1: Scrivere il POJO OrderBean

`OrderBean` sarà il POJO che verrà inviato nei messaggi, quindi necessita di implementare `Serializable`.

```

public class OrderBean implements Serializable{
    private Long orderId;
    private Date creationDate;
    private String customerName;
    private Float totalAmount;

    public Date getCreationDate() {
        return creationDate;
    }

    public void setCreationDate(Date creationDate) {
        this.creationDate = creationDate;
    }

    public String getCustomerName() {
        return customerName;
    }

    public void setCustomerName(String customerName) {
        this.customerName = customerName;
    }

    public Float getTotalAmount() {
        return totalAmount;
    }

    public void setTotalAmount(Float totalAmount) {
        this.totalAmount = totalAmount;
    }

    public Long getOrderId(){
        return orderId;
    }
}

```

7.10.2: Scrivere il producer OrderProducer

OrderProducer è un client creato utilizzando la JMS Simplified API per mandare messaggi contenenti oggetti, quindi utilizzando ObjectMessage.

```

public class OrderProducer {

    public static void main(String[] args)
        throws NamingException, JMSException{
        //Creating the order to send
        Float orderTotal = Float.valueOf(args[0]);
        OrderBean order = new OrderBean
            (18L, new Date(), "Rosario Gagliardi", orderTotal);
        //Preparing the producer
        Context ctx = new InitialContext();
        ConnectionFactory cFactory =
            (ConnectionFactory) ctx.lookup("jms/app/MyConnectionFactory");
        Destination topic = (Destination) ctx.lookup("jms/app/MyTopic");
        try (JMSContext jmsctx = cFactory.createContext()){
            jmsctx.createProducer().send(topic, order);
        }
    }
}

```

Il totale dell'ordine viene passato in input all'avvio del client, come parametro del main: infatti, come possiamo ben vedere, orderTotal viene ottenuto da args[0] (vedi come passare parametri nell'apposito capitolo extra).

7.10.3: Scrivere il consumer OrderMDB

OrderMDB è un consumer MDB annotato con `@MessageDriven` che specifica la destinazione da ascoltare. OrderMDB implementa `MessageListener`, quindi il metodo `onMessage` registrerà i messaggi arrivati alla destinazione.

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

    @Override
    public void onMessage(Message m){
        try {
            OrderBean order = m.getBody(OrderBean.class);
            System.out.println("Order from producer: " + order.getOrderid());
        } catch (JMSEException e) {
            System.err.println("Error while getting message");
            System.err.println(e.getMessage());
        }
    }
}
```

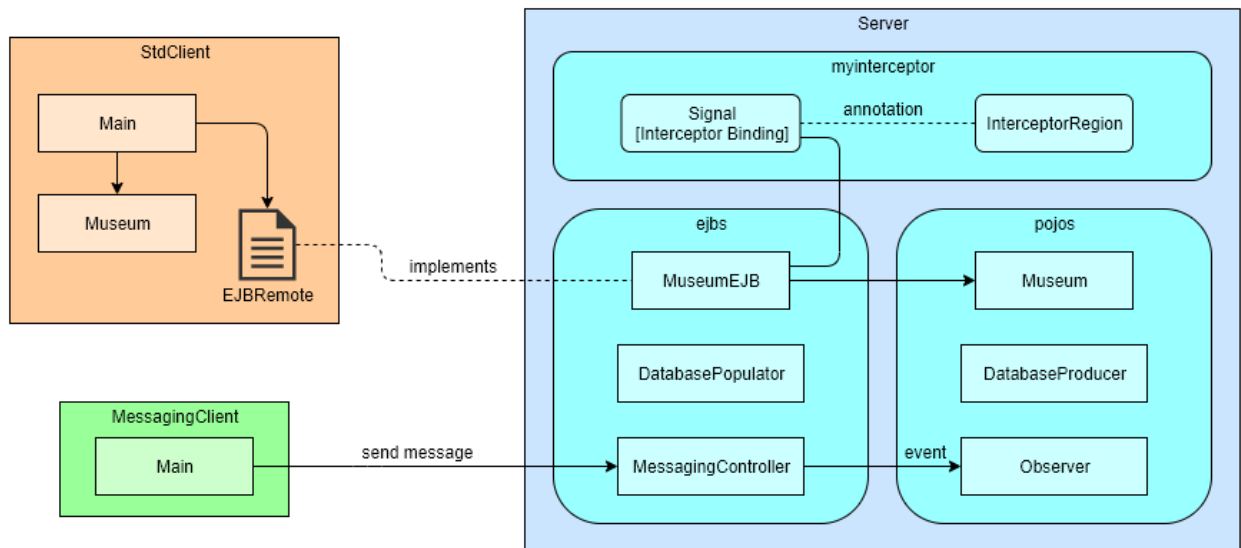
7.11: Esercizio completo (persistenza e messaging)

Concludiamo questo capitolo vedendo un esercizio completo che comprenda tutto ciò che è stato visto finora. Stabiliamo, però, una traccia:

“Scrivere un insieme di EJB e client che rappresentino un archivio di musei, persistente sul database, che contengano le informazioni sul id, nome museo, direttore, numero visitatori per anno, città, provincia e regione. Tramite JPA, si deve gestire l’archivio persistente dei circoli sul database, dove la chiave primaria è id. Devono essere previste query per id, per regione ed una query che restituisce tutti i musei d’Italia. Deve essere previsto un Singleton bean che inizializzi l’archivio. Scrivere un client basato su invocazione di uno Stateless bean che prevede la stampa di tutti i musei di una data regione passata come argomento in input dall’utente. Scrivere un interceptor che provvede a stampare il numero di volte che il metodo che utilizza la query per regione viene invocato. Scrivere un client basato su messaggi che invia un messaggio per aggiornare il numero di visitatori per anno. Tale messaggio deve poi permettere al server di ricavarsi il museo, in modo da poter aggiornare il valore. Quando viene aggiornato il valore, tramite un evento deve essere stampato sul server un messaggio di avviso.”

Dalla traccia comprendiamo subito l’infrastruttura dell’applicazione: si necessita di un server e due client. Il server gestirà il database con le query e i messaggi, mentre un client si dedicherà all’interazione col database e l’altro si dedicherà all’invio di messaggi inerenti alla modifica del museo.

L’infrastruttura dell’applicazione viene mostrata di seguito.



7.11.1: Scrivere il POJO Museum

```
@Entity
@NamedQueries({
    @NamedQuery(name = Museum.FIND_ALL,
        query = "SELECT m FROM Museum m"),
    @NamedQuery(name = Museum.FIND_BY_ID,
        query = "SELECT m FROM Museum m WHERE m.id = :museumid"),
    @NamedQuery(name = Museum.FIND_BY_REGION,
        query = "SELECT m FROM Museum m WHERE m.region = :museumregion")
})
public class Museum implements Serializable{
    public static final String FIND_BY_ID = "Museum.findById";
    public static final String FIND_BY_REGION = "Museum.findByRegion";
    public static final String FIND_ALL = "Museum.findAll";
    @Id @GeneratedValue
    private Long id;
    private String name;
    private String director;
    private Integer numbOfVisitors;
    private String city;
    private String province;
    private String region;
    public Museum() {}
    public Museum(String name, String director, Integer numbOfVisitors,
        String city, String province, String region) {
        this.name = name;
        this.director = director;
        this.numbOfVisitors = numbOfVisitors;
        this.city = city;
        this.province = province;
        this.region = region;
    }
}
```

7.11.2: Scrivere MuseumEJB e la sua interfaccia remota

```
@Remote
public interface MuseumEJBRemote {
    public Museum getMuseumById(Long id);
    public List<Museum> getMuseumByRegion(String region);
    public List<Museum> getMuseumAll();
    public Museum createMuseum(Museum m);
    public void removeMuseum(Museum m);
    public Museum updateMuseum(Museum m);
}
```

```

@Stateless
@LocalBean
public class MuseumEJB implements MuseumEJBRemote {

    @Inject
    EntityManager em;

    @Override
    public Museum getMuseumById(Long id) {
        TypedQuery<Museum> objQuery =
            em.createNamedQuery(Museum.FIND_BY_ID, Museum.class);
        objQuery.setParameter("museumid", id);
        return objQuery.getSingleResult();
    }

    @Override
    @Signal
    public List<Museum> getMuseumByRegion(String region) {
        TypedQuery objQuery =
            em.createNamedQuery(Museum.FIND_BY_REGION, Museum.class);
        objQuery.setParameter("museumregion", region);
        return objQuery.getResultList();
    }

    @Override
    public List<Museum> getMuseumAll() {
        TypedQuery<Museum> objQuery =
            em.createNamedQuery(Museum.FIND_ALL, Museum.class);
        return objQuery.getResultList();
    }

    @Override
    public Museum createMuseum(Museum m) {
        em.persist(m);
        return m;
    }

    @Override
    public void removeMuseum(Museum m) {
        em.remove(em.merge(m));
    }

    @Override
    public Museum updateMuseum(Museum m) {
        return em.merge(m);
    }
}

```

L'annotazione @Signal utilizzata sul metodo getMuseumByRegion indica l'interceptor che vedremo tra poco.

7.11.3: DatabaseProducer e DatabasePopulator

```

public class DatabaseProducer {

    @Produces
    @PersistenceContext(unitName = "museumPU")
    private EntityManager em;

}

```

```

@Singleton
@Startup
@LocalBean
@DataSourceDefinition(
    className = "org.apache.derby.jdbc.ClientDataSource",
    name = "java:global/jdbc/___default",
    user = "APP",
    password = "APP",
    databaseName = "DerbyPool",
    properties = {"connectionAttributes=create=True"}
)
public class DatabasePopulator {

    @Inject
    MuseumEJB controller;
    ArrayList<Museum> listOfMuseums = new ArrayList<>();

    public DatabasePopulator() {}

    @PostConstruct
    public void callbackCreate() {
        listOfMuseums = new ArrayList<>();
        listOfMuseums.add(controller.createMuseum(
            new Museum("Louvre", "Pippo Baudo", 285,
                "Cava de' Tirreni", "SA", "Campania")));
        listOfMuseums.add(controller.createMuseum(
            new Museum("Il museo di Carlo Cracco", "Bastianich",
                12, "Napoli", "NA", "Campania")));
        listOfMuseums.add(controller.createMuseum(
            new Museum("Il museo di Padre Pio", "Pippo Inzaghi",
                148, "Pietralcina", "SA", "Campania")));
    }

    @PreDestroy
    public void callbackDelete() {
        for (Museum m: listOfMuseums) {
            controller.removeMuseum(m);
        }
    }
}

```

7.11.4: Interceptor Binding

```

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

```

```

@Interceptor
@Signal
public class InterceptorRegion {
    private Integer intCounter = 0;
    @AroundInvoke
    public Object counter(InvocationContext ic) throws Exception {
        try {
            intCounter += 1;
            System.out.println("Actual counter: " + intCounter);
            return ic.proceed();
        } finally {
            System.out.println("Ended");
        }
    }
}

```

È necessario dichiarare l'interceptor in beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       bean-discovery-mode="all">
  <interceptors>
    <class>myinterceptor.InterceptorRegion</class>
  </interceptors>
</beans>
```

7.11.5: Client per le regioni

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

    Context ctx = new InitialContext();
    MuseumEJBRemote controller = (MuseumEJBRemote)
        ctx.lookup("java:global/EJBServer/MuseumEJB!ejbs.MuseumEJBRemote");
    String inputRegion;
    System.out.println("*****");
    System.out.print("Insert the region: ");
    try (Scanner keyboard = new Scanner(System.in)) {
        inputRegion = keyboard.nextLine();
    }
    if (inputRegion == null) throw new NullPointerException("Invalid input");
    List<Museum> listOfMuseums = controller.getMuseumByRegion(inputRegion);
    if (listOfMuseums.isEmpty()) System.out.println("No record found");
    else
        listOfMuseums.forEach((m) -> {
            System.out.println(m.toString());
        });
}
```

Client output:

```
*****
Insert the region: Campania
2153 | Il museo di Padre Pio | Pietralcina | Campania
2151 | Louvre | Cava de' Tirreni | Campania
2152 | Il museo di Carlo Cracco | Napoli | Campania
BUILD SUCCESSFUL (total time: 26 seconds)
```

Server output per visualizzare il funzionamento dell'interceptor:

```
Actual counter: 1|#]
Ended|#]
```

7.11.6: Scrivere il MDB MessagingController

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

    @Inject
    private Event<Museum> objEvent;
    @Inject
    private MuseumEJB controller;

    @Override
    public void onMessage(Message message) {
        try{
            Long inputId = message.getBody(Long.class);
            Integer newVisits = message.getIntProperty("newVisits");
            Museum m = controller.getMuseumById(inputId);
            if (m == null) System.out.println("No museum found");
            else {
                if (newVisits != m.getNumOfVisitors()) {
                    m.setNumOfVisitors(newVisits);
                    controller.updateMuseum(m);
                    objEvent.fire(m);
                }
            }
        }catch(JMSEException e){
            System.out.println("Cannot read message");
        }
    }
}
```

7.11.7: Scrivere il gestore di eventi Observer

```
public class Observer {

    public void notify(@Observes Museum m){
        System.out.println("New visitors value: " + m.getNumOfVisitors());
    }
}
```

7.11.8: Client per il messaging

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

    Context ctx = new InitialContext();
    ConnectionFactory cf = (ConnectionFactory)
        ctx.lookup("jms/app/MyConnectionFactory");
    Destination dest = (Destination)
        ctx.lookup("jms/app/MyTopic");
    JMSContext jmsctx = cf.createContext();
    jmsctx.createProducer().
        setProperty("newVisits", Integer.parseInt(args[0])).
        send(dest, Long.parseLong(args[1]));
    //args[0] contains new visits value
    //args[1] contains the id of the museum
}
```

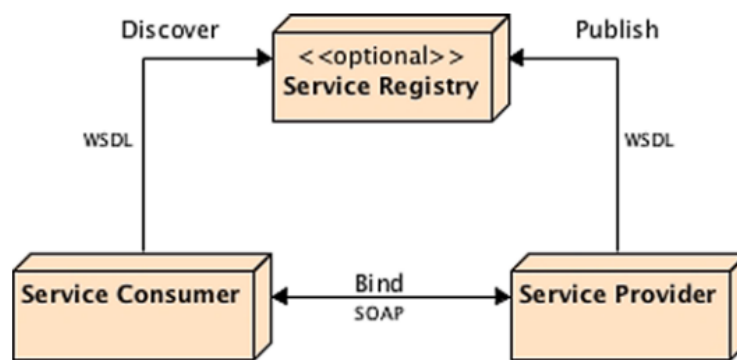
Client output:

```
New visitors value: 1000|#]
```

Capitolo 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 poche parole, i web service provvedono un modo per connettere diversi pezzi di software.

Di seguito 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).

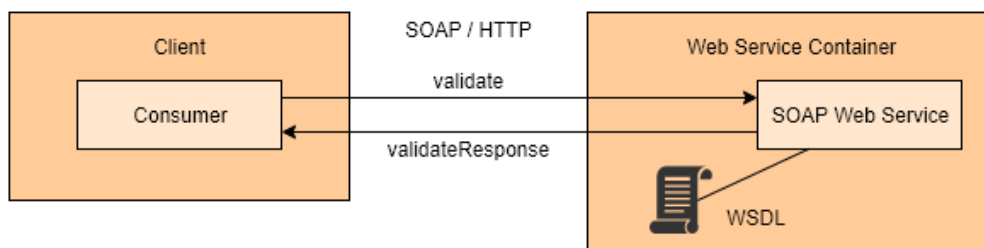
8.1.1: WSDL

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: SOAP

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, quindi, 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

Abbiamo visto i concetti a basso livello riguardanti i SOAP WS, ma come ne scriviamo uno tenendo conto di tutto quel che abbiamo visto poco fa? Ebbene, 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 di 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(  
    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;  
    }  
}
```

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

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`

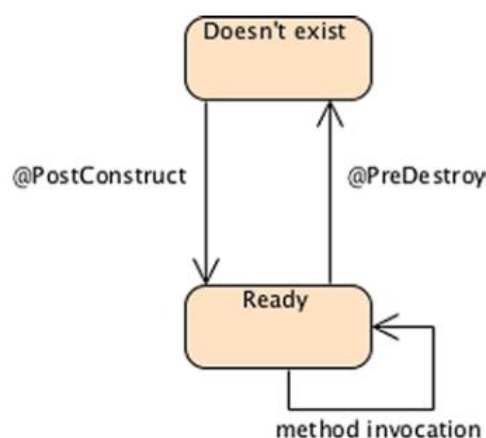
L'annotazione `@WebResult` controlla il nome dell'oggetto di ritorno in WSDL. Nel seguente esempio, il valore ritornato è rinominato con "isValid". Nel caso non venga utilizzata tale annotazione, rimane il valore di default: "return".

```
@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;
    }
}
```

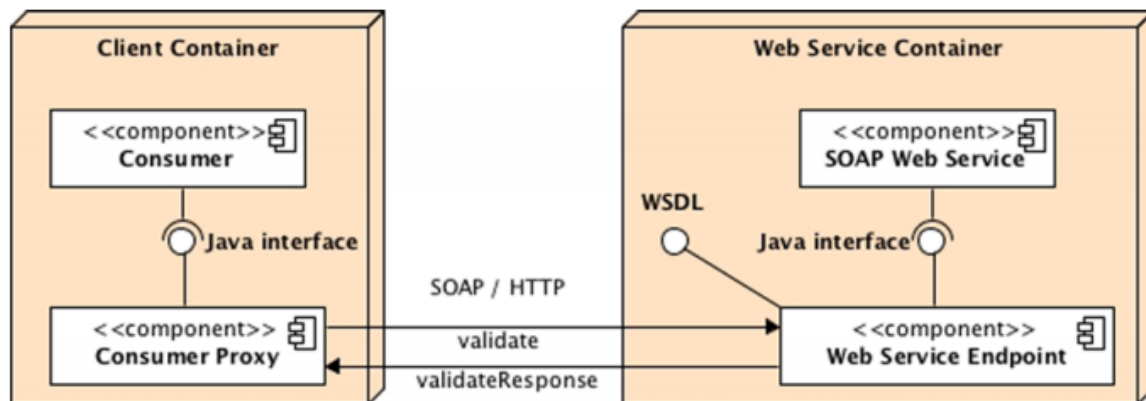
8.5: Ciclo di vita di un SOAP WS

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

(Prima di iniziare tale capitolo, è consigliabile aver letto il capitolo extra "Setup SOAP WS")

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, come nell'esempio, `WSNAME` e `Service()` sono separati da un underscore. Ciò accade quando `WSNAME` si conclude con il carattere "s".

```
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));

}
```

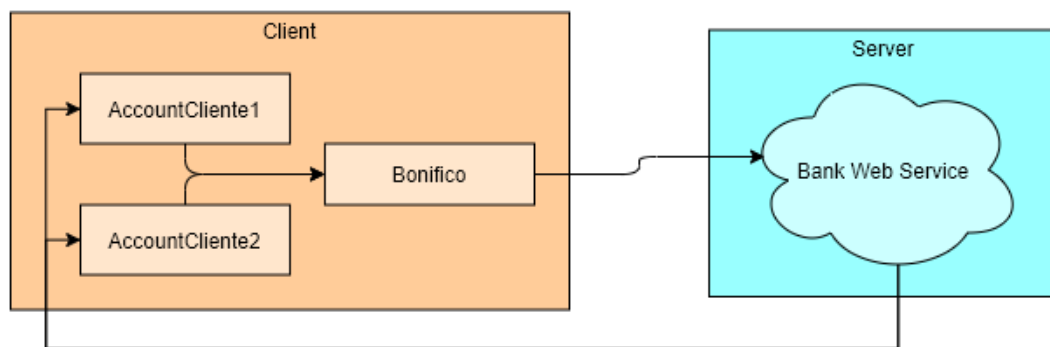
Ricavato il proxy del web service, è possibile effettuare invocazioni di metodi, le quali saranno delegate al web service remoto. Ciò che sta dietro l'invocazione remota fa parte del meccanismo nascosto offerto dal proxy.

8.7: Esercizio riepilogativo

Rivediamo i concetti di quest'ultimo capitolo in un piccolo esempio che mostra in maniera molto semplice il funzionamento di un SOAP WS.

Supponiamo di avere un conto bancario e di dover effettuare un bonifico verso il web service bancario. Il bonifico è caratterizzato dai dati personali, dai dati del beneficiario con il suo codice IBAN, l'importo del bonifico, la data di compilazione, la causale e il proprio IBAN. Il SOAP WS elaborerà il bonifico, intaccando gli account degli interessati.

Possiamo vedere tale situazione con la seguente struttura:



Il client, però, avrà solamente il main: le classi Account e Transfer (che riguarda il bonifico) si troveranno nel server con il web service. Tali classi verranno generate in automatico nel client.

Prima di continuare, è altamente consigliabile seguire il capitolo extra “Setup SOAP WS”.

8.7.1: Scrivere i POJO Account e Transfer

I POJO Account e Transfer contengono informazioni al fine di poter effettuare il bonifico. Account mantiene le informazioni riguardanti l'utente, Transfer mantiene le informazioni riguardanti il bonifico. Entrambi i POJO devono essere annotati da `@XmlRootElement` siccome vengono trattati dal web service.

Dei due POJO vediamo solo le variabili private siccome i restanti metodi sono getter e setter.

```
@XmlRootElement
public class Account {
    private String iban;
    private String name;
    private String surname;
    private Float balance;
```

```
@XmlRootElement
public class Transfer {
    private Account from;
    private Account to;
    private Float amount;
    private String motivation;
```

8.7.2: Scrivere il SOAP WS

Il web service bancario deve poter effettuare il bonifico. Il bonifico deve essere effettuato solo se sul conto del mittente c'è saldo sufficiente. Di conseguenza, vengono stabiliti due metodi: `canTransfer` controlla se il mittente ha abbastanza saldo per poter effettuare il pagamento;

doTransfer esegue il pagamento, restituendo un ArrayList di oggetti Account aggiornati dopo l'operazione.

Nel metodo doTransfer vengono utilizzate molteplici System.out.println(...), in modo da tenere sotto occhio ciò che accade lato server nel web service.

Infine, il web service deve essere, ovviamente, annotato da @WebService, specificando il nome del servizio.

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

    public Boolean canTransfer(Transfer payment) {
        if (payment.getAmount() <= payment.getFrom().getBalance())
            return true;
        else return false;
    }

    public ArrayList<Account> doTransfer(Transfer payment) {
        System.out.println("Transfer from " + payment.getFrom().getName() +
            " " + payment.getFrom().getSurname() + " to " +
            payment.getTo().getName() + " " + payment.getTo().getSurname());
        System.out.println("Motivation: " + payment.getMotivation());
        System.out.println("Amount: " + payment.getAmount());
        payment.getFrom().setBalance(
            payment.getFrom().getBalance() - payment.getAmount());
        System.out.println(payment.getAmount() + " removed from " +
            payment.getFrom().getName());
        payment.getTo().setBalance(
            payment.getTo().getBalance() + payment.getAmount());
        System.out.println(payment.getAmount() + " added to " +
            payment.getTo().getName());
        System.out.println("Transfer done.");
        ArrayList<Account> myList = new ArrayList<>();
        myList.add(payment.getFrom());
        myList.add(payment.getTo());
        return myList;
    }
}
```

8.7.3: Scrivere il client

Nel client verranno istanziati gli oggetti Account, inizializzandoli tramite proprietà setter, e l'oggetto Transfer.

Dopodiché, si ottiene il proxy del servizio. Ricordiamoci come si ottiene il proxy:

» Sia WSNAME il nome del web service, tale proxy si ricava tramite la seguente istruzione:

```
new WSNAMEService().getWSNAMEPort();
```

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

Ottenuto il proxy, si può procedere all'invocazione dei metodi: viene invocato prima il metodo canTransfer per controllare se è possibile effettuare il pagamento e, in caso di esito positivo, viene poi invocato doTransfer per effettuare il pagamento vero e proprio.

A lato client vedremo se l'esito è andato a buon fine o meno.

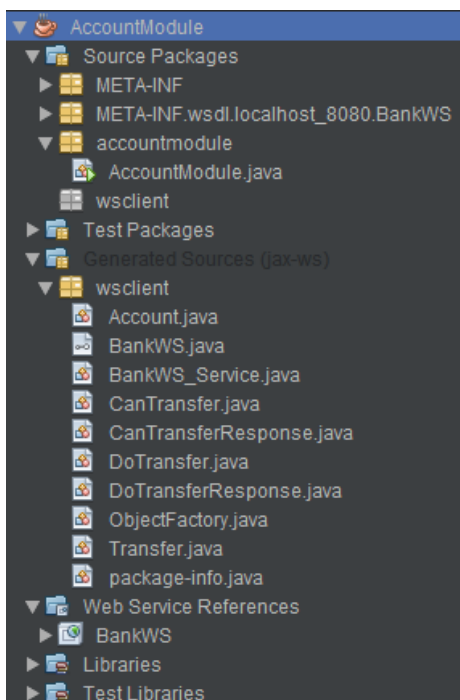
```
public static void main(String[] args) {

    Account first = new Account();
    first.setIban("1234567890");
    first.setName("Francesco");
    first.setSurname("Abate");
    first.setBalance(1200F);
    Account second = new Account();
    second.setIban("0987654321");
    second.setName("Rosario");
    second.setSurname("Gagliardi");
    second.setBalance(1800F);
    Transfer payment = new Transfer();
    payment.setFrom(first);
    payment.setTo(second);
    payment.setAmount(200F);
    payment.setMotivation("A gift from me");

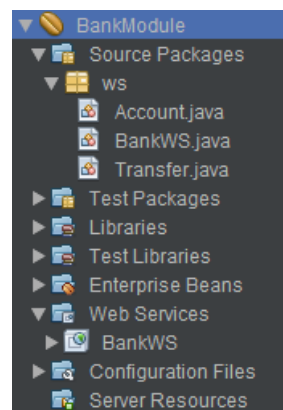
    BankWS service = new BankWS_Service().getBankWSPort();
    Boolean control = service.canTransfer(payment);
    if (control) {
        List<Account> accounts = service.doTransfer(payment);
        first = accounts.get(0);
        second = accounts.get(1);
        System.out.println("Payment done!");
        System.out.println(first.getName() +
            " balance: " + first.getBalance());
        System.out.println(second.getName() +
            " balance: " + second.getBalance());
    }
    else System.out.println("Insufficient balance");
}
```

8.7.4: Situazione attuale

Client



Server



8.7.5: Funzionamento

```
? CLIENT SIDE ?  
Payment done!  
Francesco balance: 1000.0  
Rosario balance: 2000.0  
BUILD SUCCESSFUL (total time: 2 seconds)
```

```
? SERVER SIDE ?|#]  
Transfer from Francesco Abate to Rosario Gagliardi|#]  
Motivation: A gift from me|#]  
Amount: 200.0|#]  
200.0 removed from Francesco|#]  
200.0 added to Rosario|#]  
Transfer done.|#]
```

Riferimenti

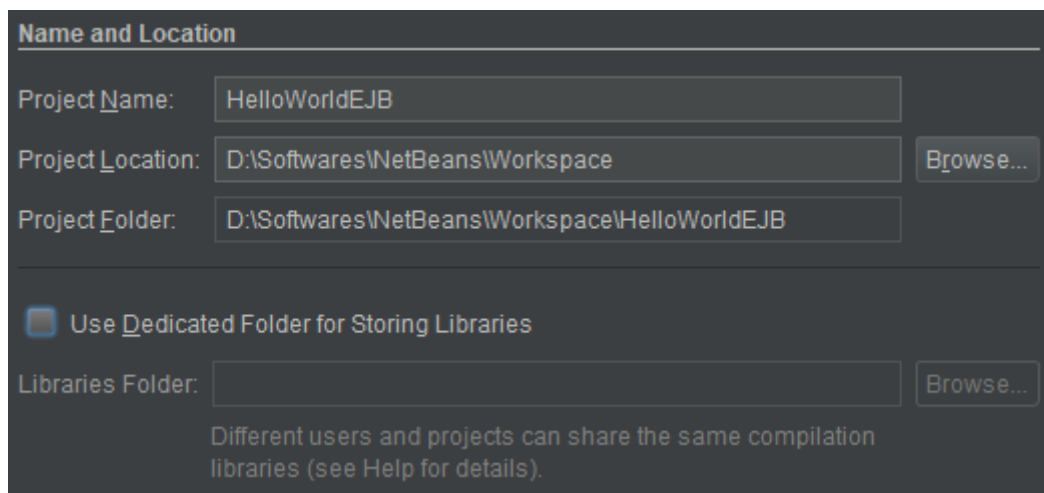
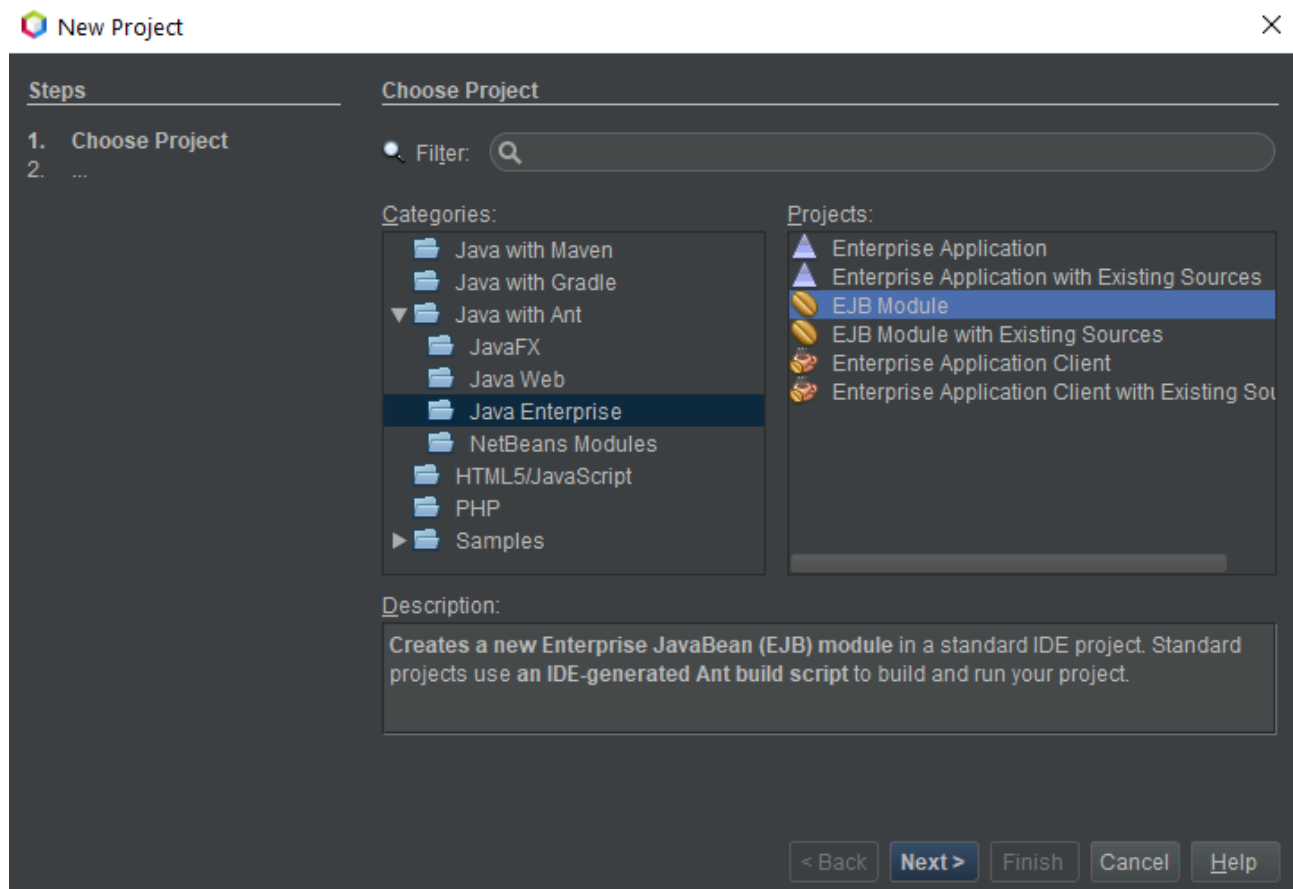
[Facebook](#) | [Linkedin](#) | [GitHub](#) | [GameJolt](#) | [Website](#)

Extra: Setup Progetto con EJB

Passi da seguire:

- 1) Creare un progetto per l'EJB
- 2) Creare un progetto per il client
- 3) Includere le librerie
- 4) Scrivere del codice
- 5) Effettuare il deployment

Creare un progetto per l'EJB



Server and Settings

Add to Enterprise Application: <None>

Server: GlassFish Server Add...

Java EE Version: Java EE 8

Creare un progetto per il client

New Project ×

Steps

1. Choose Project
2. ...

Choose Project

Filter:

Categories:

- Java with Maven
- Java with Gradle
- ▼ Java with Ant
 - JavaFX
 - Java Web
 - Java Enterprise
 - NetBeans Modules
 - HTML5/JavaScript
 - PHP
 - Samples

Projects:

- Java Application
- Java Class Library
- Java Project with Existing Sources
- Java Modular Project
- Java Free-Form Project

Description:

Creates a new Java SE application in a standard IDE project. You can also generate a main class in the project. Standard projects use an IDE-generated Ant build script to build, run, and debug your project.

< Back Next > Finish Cancel Help

Name and Location

Project Name: HelloWorldClient

Project Location: D:\Softwares\NetBeans\Workspace Browse...

Project Folder: D:\Softwares\NetBeans\Workspace\HelloWorldClient

☐ Use Dedicated Folder for Storing Libraries

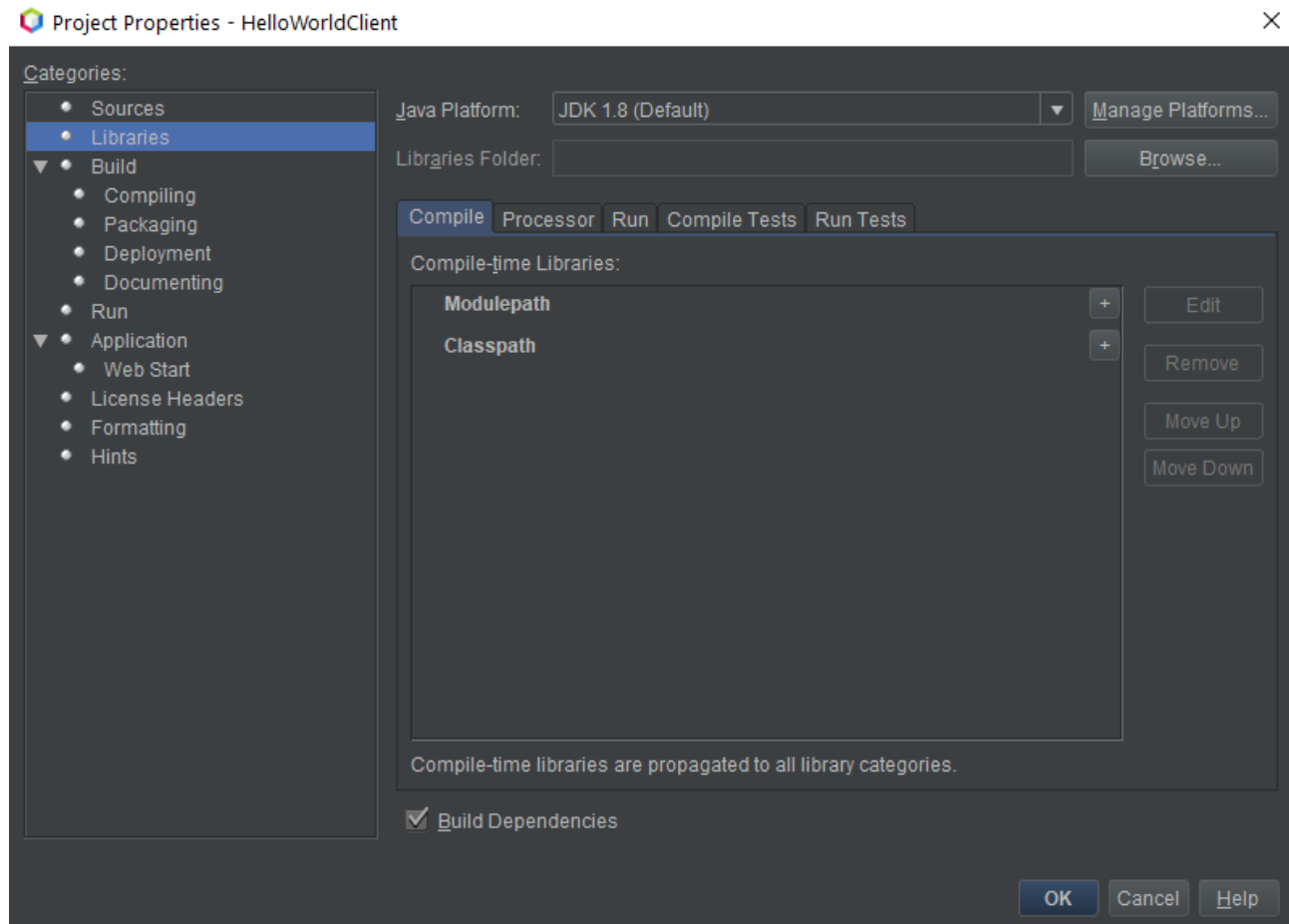
Libraries Folder: Browse...

Different users and projects can share the same compilation libraries (see Help for details).

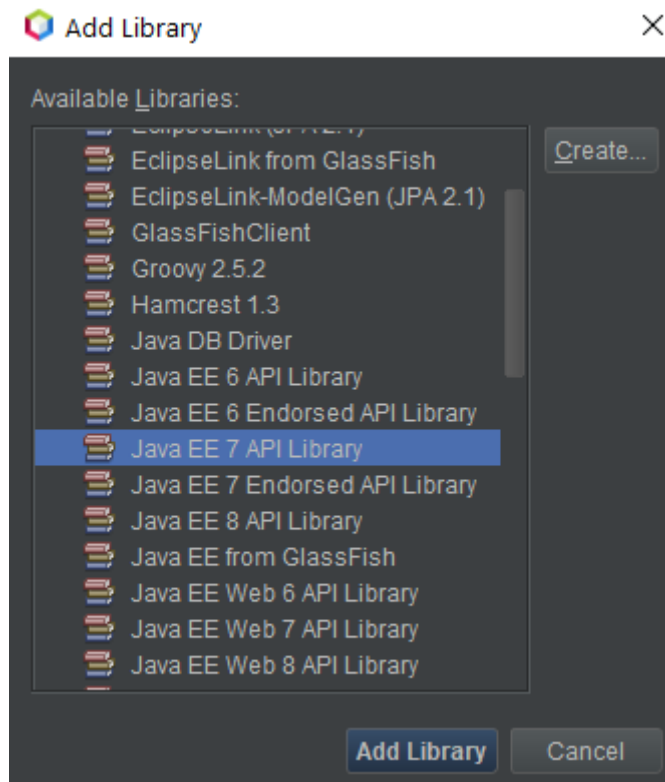
☒ Create Main Class helloworldclient.HelloWorldClient

Includere le librerie

Includere le librerie del client

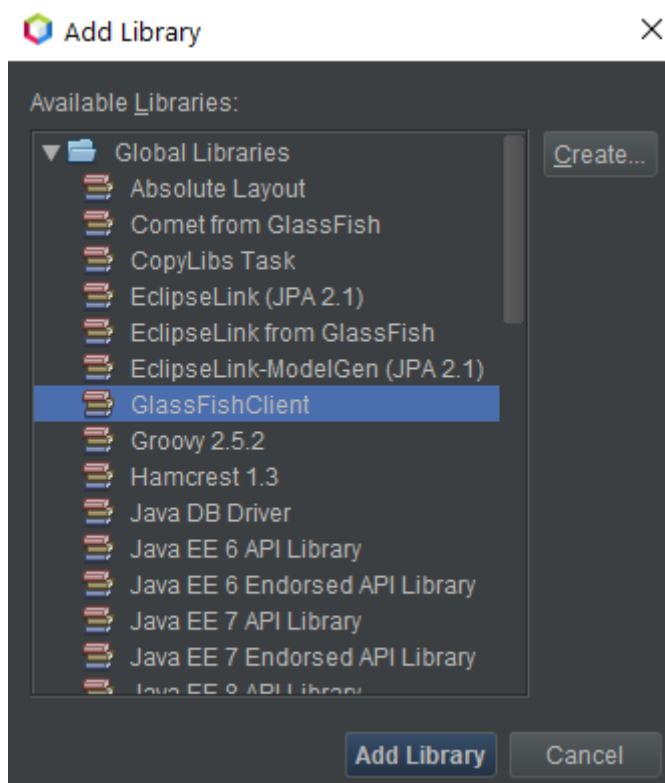


Nel caso si trattino i MDB, utilizzare Java EE 8 API Library



Di seguito è necessario aggiungere la libreria per Glassfish. Tale libreria non è presente, bisogna crearla (nell'immagine precedente risulta già creata). Per crearla, è necessario premere il pulsante Create, dare un nome alla libreria lasciando come tipo "Class Libraries", scegliere un JAR, selezionare gf-client.jar dalla directory del server Glassfish/lib.

Aggiungiamo, quindi, anche la nuova libreria creata.



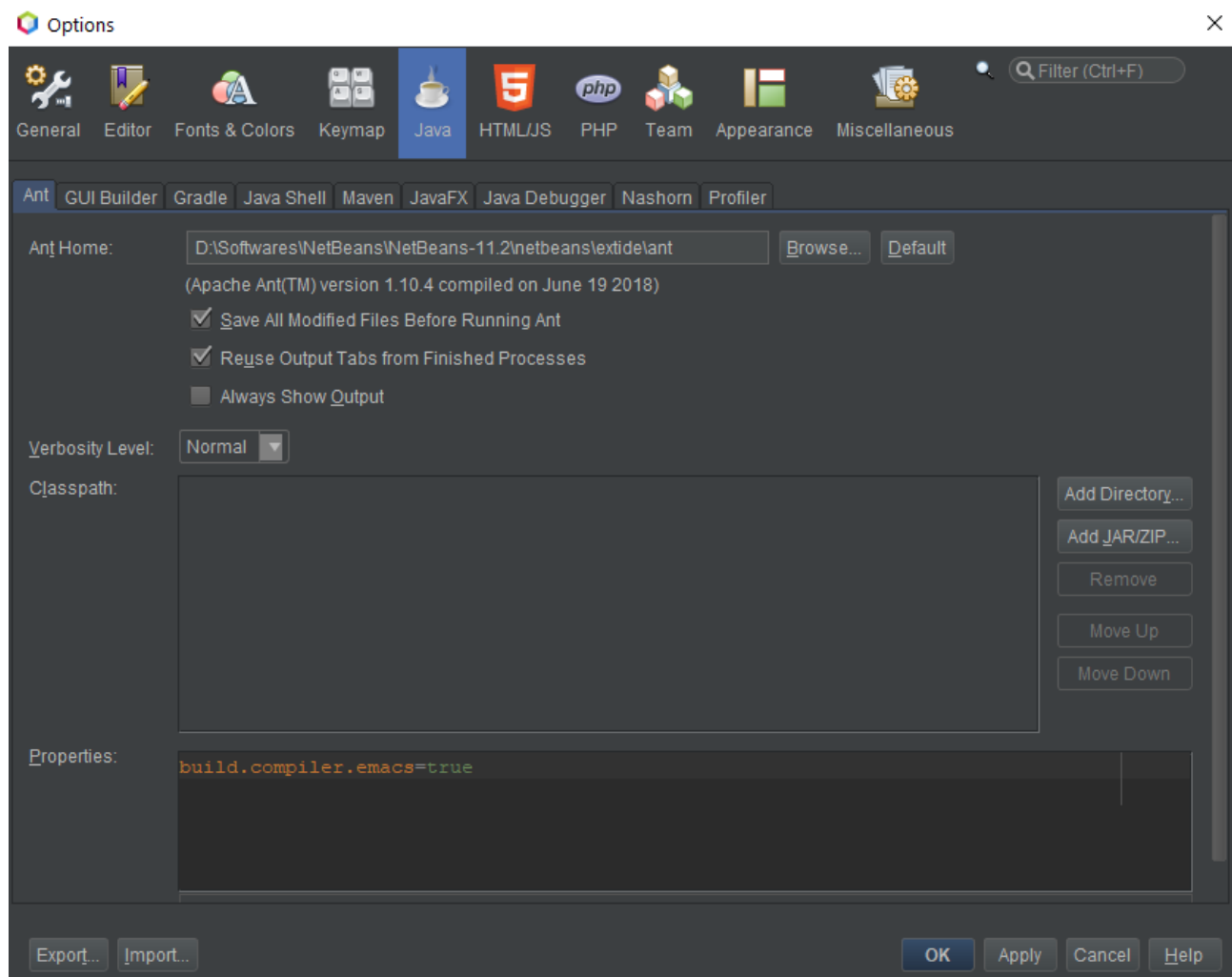
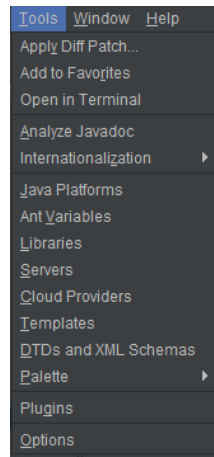
Includere le librerie del server

Non è necessario includere librerie per il server. Nel caso ci fossero problemi nel sorgente di librerie non trovate, includere Java EE 7 API Library come fatto con il client.

Librerie per i SOAP WS

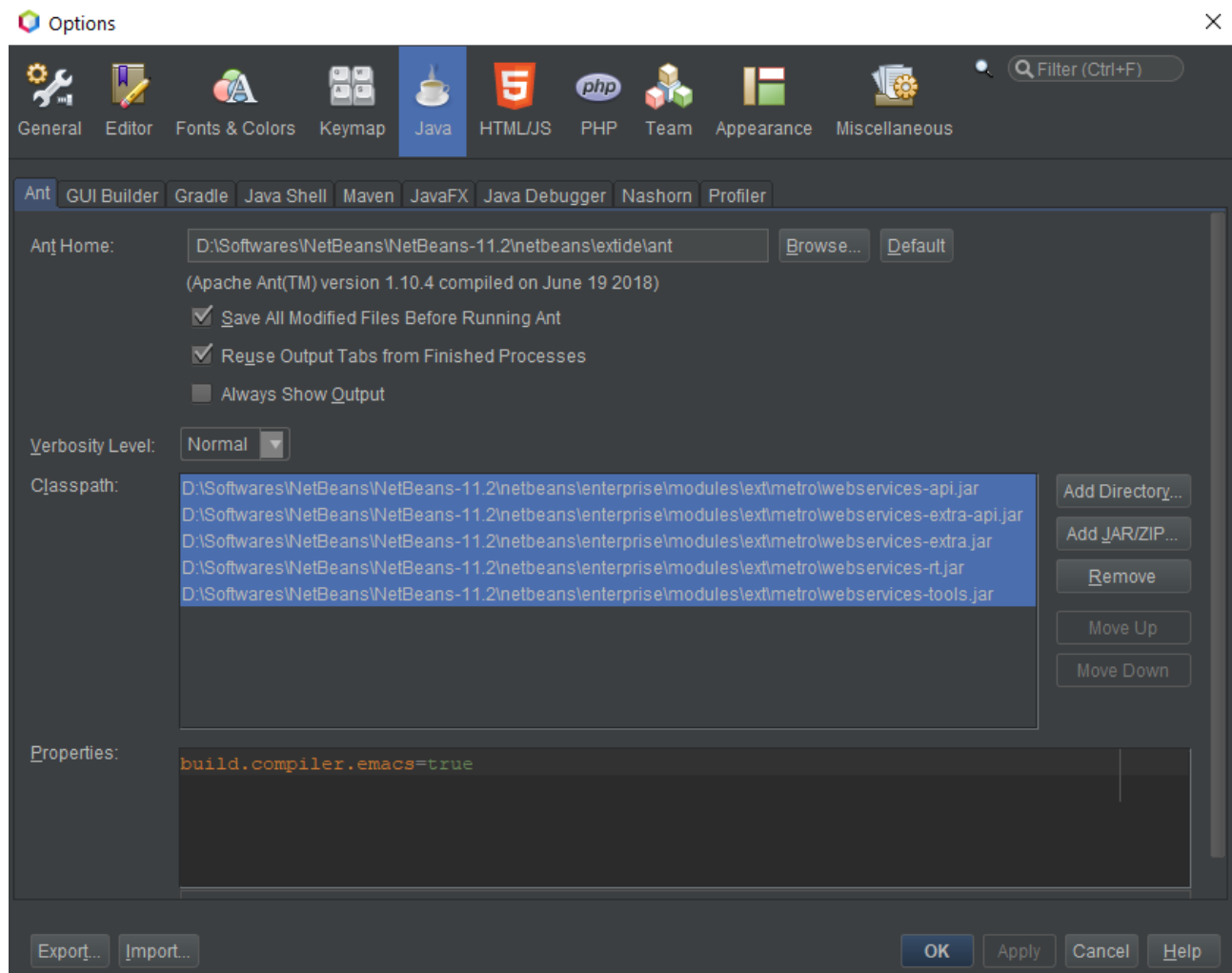
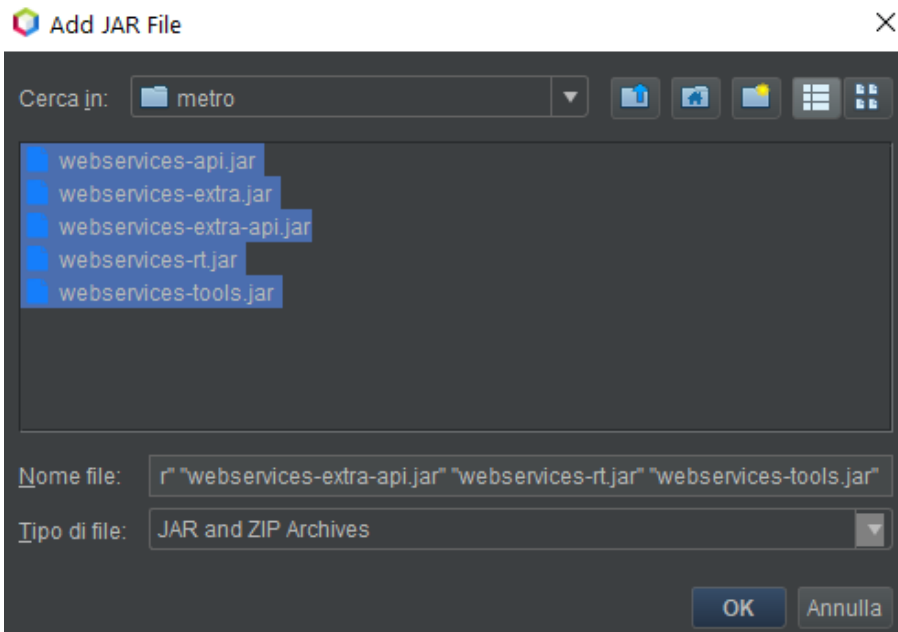
Nel caso si utilizzino i SOAP WS, si necessita di speciali librerie. Tali librerie non vanno incluse nel client o nel server, bisogna importarle globalmente. Per fare ciò, ci dirigiamo nel seguente path:

Tools -> Options -> Java



Procediamo cliccando il pulsante “Add JAR/ZIP” posto sulla destra: aprirà un navigator e selezioneremo tali risorse:

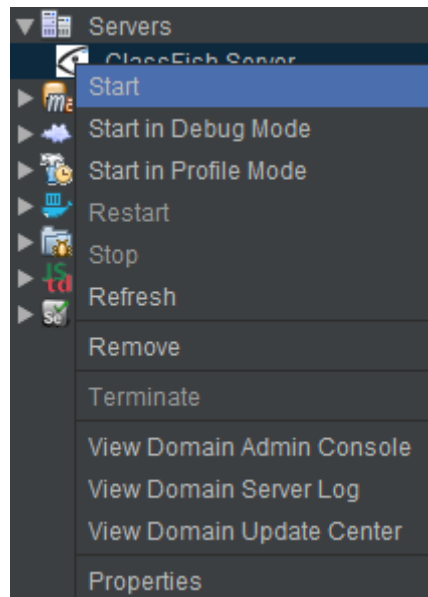
netbeans -> enterprise -> modules -> ext -> metro -> selezionare tutti i JAR e confermare



Applicare e confermare.

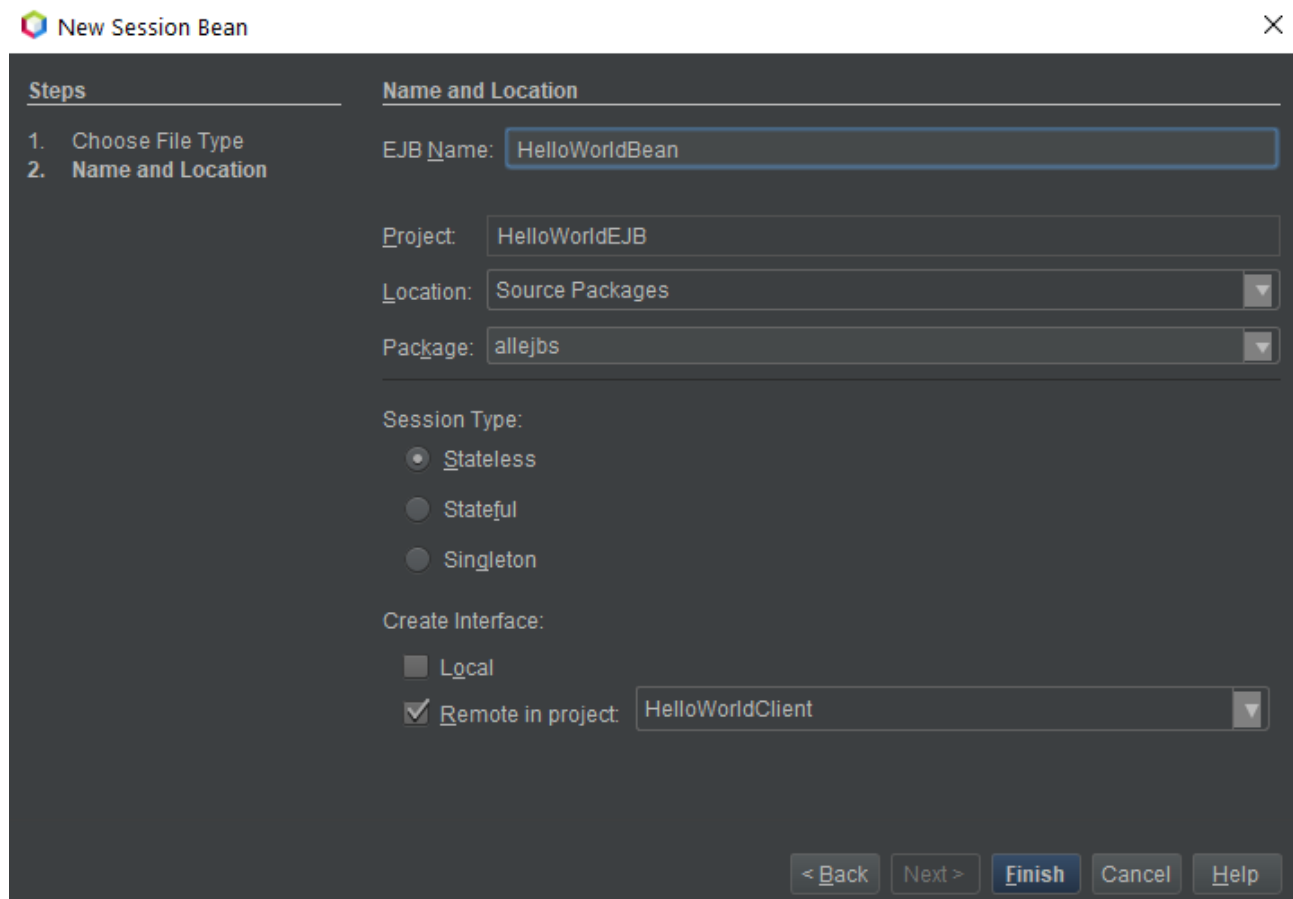
Attenzione: tale procedura va fatta un'unica volta e non ad ogni progetto.

Codice e deployment



Scriviamo del codice per testare il deployment.

Iniziamo creando un EJB stateless che restituisca un semplice saluto.



```
@Stateless
public class HelloWorldBean implements HelloWorldBeanRemote {

    public String greetings(String name){
        return "Hello " + name + "!";
    }

}
```

L'interfaccia remota generata automaticamente nel client deve dichiarare il metodo implementato dal bean.

```
@Remote
public interface HelloWorldBeanRemote {

    public String greetings(String name);

}
```

Fatto ciò, provvediamo con lo scrivere il main nel client. Il main dovrà ricorrere ad un'operazione di lookup per ottenere l'EJB. Siccome nel client non è presente l'EJB, verrà trattata la sua interfaccia remota.

Fatto ciò, creiamo anche il file beans.xml facendo click destro sulla folder dei sorgenti del server, nuovo beans.xml. Cambiamo poi discovery mode in all. Fatto ciò, è possibile effettuare il deployment per poi testare il tutto.

```
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"));

    }

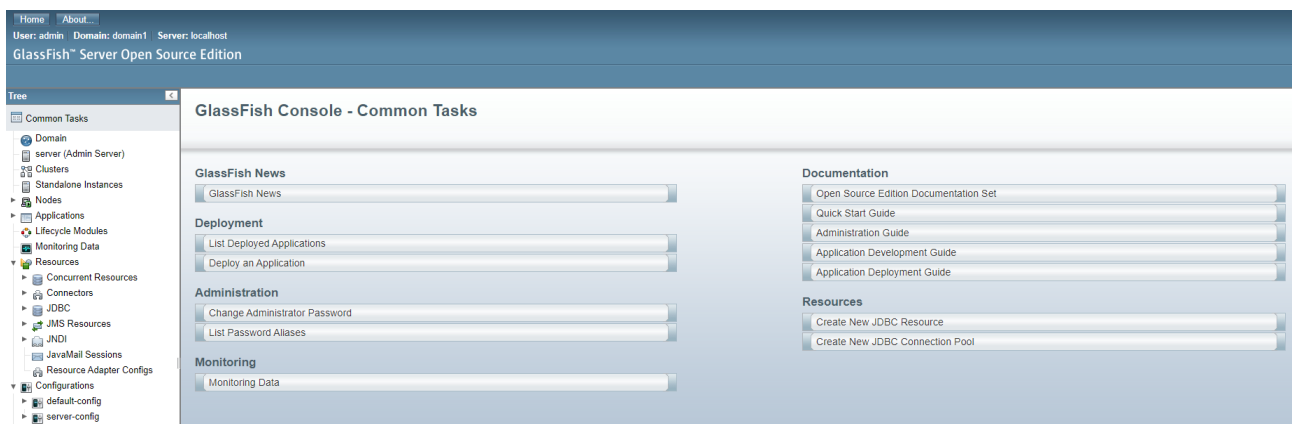
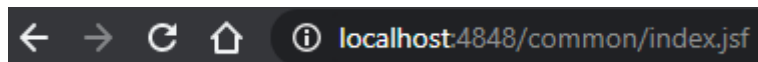
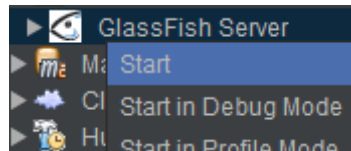
}
```

```
Hello Francesco!
BUILD SUCCESSFUL (total time: 16 seconds)
```

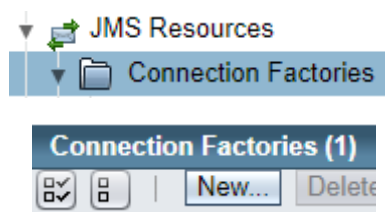

Extra: Setup Oggetti Amministrati

È necessario che il server abbia un Connection Factory e una Destinazione (utilizzeremo il Topic). È possibile configurare tali oggetti dalla Web Console, in tal caso dalla console di Glassfish. Riguardo la creazione del Topic, è necessario creare anche la destinazione fisica.

Iniziamo avviando il server, per poi dirigerci sulla sua Web Console.



Nella barra di navigazione posta a sinistra, apriamo JMS Resources, Connection Factories e ne creiamo una nuova.



New JMS Connection Factory

The creation of a new Java Message Service (JMS) connection factory also creates a connector connection pool for the factory and a connector resource.

General Settings

JNDI Name: *
Resource Type:
Description:
Status: ☒ Enabled

Pool Settings

Initial and Minimum Pool Size: Connections
Minimum and initial number of connections maintained in the pool
Maximum Pool Size: Connections
Maximum number of connections that can be created to satisfy client requests
Pool Resize Quantity: Connections
Number of connections to be removed when pool idle timeout expires
Idle Timeout: Seconds
Maximum time that connection can remain idle in the pool
Max Wait Time: Milliseconds
Amount of time caller waits before connection timeout is sent
On Any Failure: ☐ Close All Connections
Close all connections and reconnect on failure, otherwise reconnect only when used
Transaction Support:
Level of transaction support. Overwrite the transaction support attribute in the Resource Adapter in a downward compatible way.
Connection Validation: ☐ Required
Validate connections, allow server to reconnect in case of failure

General Settings

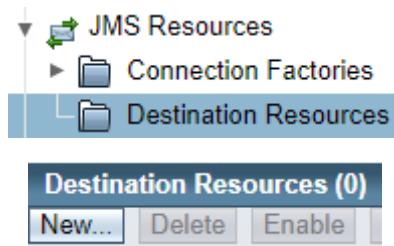
JNDI Name: *

Resource Type:

Description:

Status: ☒ Enabled

Confermare e procedere creando una destinazione.



New JMS Destination Resource

The creation of a new Java Message Service (JMS) destination resource also creates an admin object resource.

JNDI Name: *

Physical Destination Name *
Destination name in the Message Queue broker. If the destination does not exist, it will be created automatically when needed.

Resource Type: *

Description:

Status: ☒ Enabled

JNDI Name: *

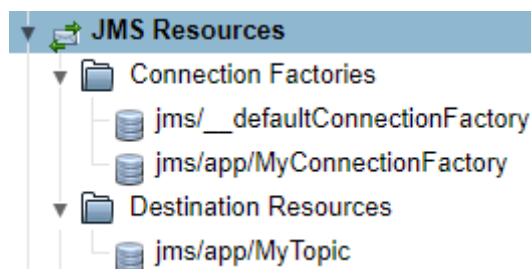
Physical Destination Name *
Destination name in the Message Queue broker. If the destination does not exist, it will be created automatically when needed.

Resource Type: *

Description:

Status: ☒ Enabled

Configurata la destinazione, la gerarchia delle risorse JMS dovrebbe essere la seguente:



Impostare i parametri di avvio

Una applicazione JMS è composta, fondamentalmente, da una classe producer ed una classe consumer: la classe producer è un client che crea e invia messaggi, mentre la classe consumer è un MDB che riceve ed elabora i messaggi. Spesso, il client producer invia messaggi in base ai parametri che riceve in input all'avvio. In questo paragrafo vediamo come passare argomenti al main all'avvio del client.

Iniziamo dirigendoci nelle impostazioni del progetto client per poi andare nella sezione run.

Categories:

- Sources
- Libraries
- ▼ • Build
 - Compiling
 - Packaging
 - Deployment
 - Documenting
 - **Run**
- ▼ • Application
 - Web Start
 - License Headers
 - Formatting
 - Hints

Configuration: <default config> New... Delete

Runtime Platform: Project Platform Manage Platforms...

Main Class: clientmodule.OrderProducer Browse...

Arguments:

Working Directory: Browse...

VM Options:
(e.g. -Xms10m) Customize...

☐ Run with Java Web Start
(To run and debug the application with Java Web Start, first enable Java Web Start)

OK Cancel Help

Categories:

- Sources
- Libraries
- ▼ • Build
 - Compiling
 - Packaging
 - Deployment
 - Documenting
 - **Run**
- ▼ • Application
 - Web Start
 - License Headers
 - Formatting
 - Hints

Configuration: <default config> New... Delete

Runtime Platform: Project Platform Manage Platforms...

Main Class: clientmodule.OrderProducer Browse...

Arguments:

Working Directory: Browse...

VM Options:
(e.g. -Xms10m) Customize...

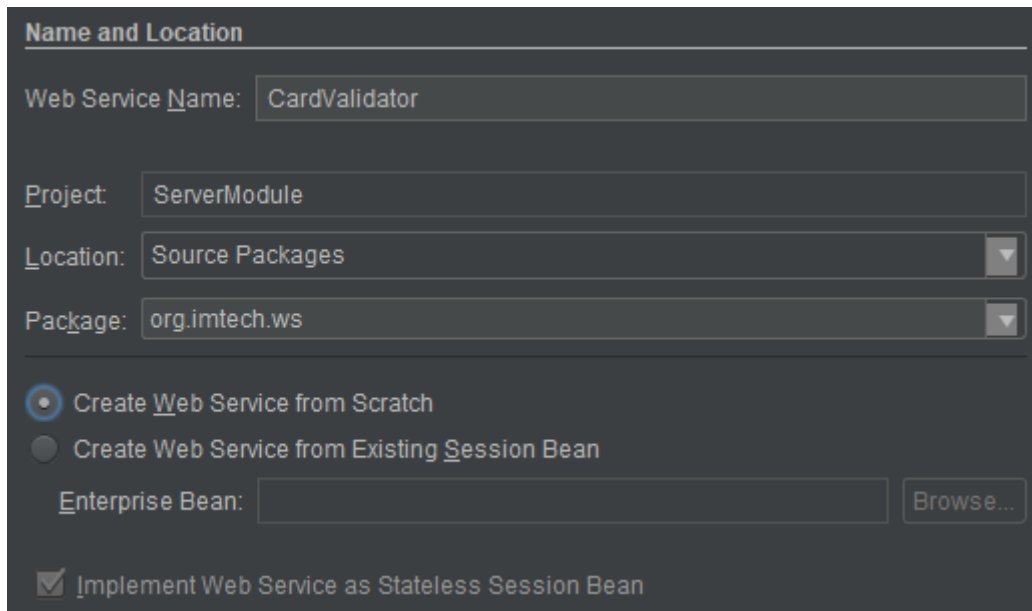
☐ Run with Java Web Start
(To run and debug the application with Java Web Start, first enable Java Web Start)

OK Cancel Help

Extra: Setup SOAP WS

È necessario avere un modulo EJB ed un modulo applicazione Java.

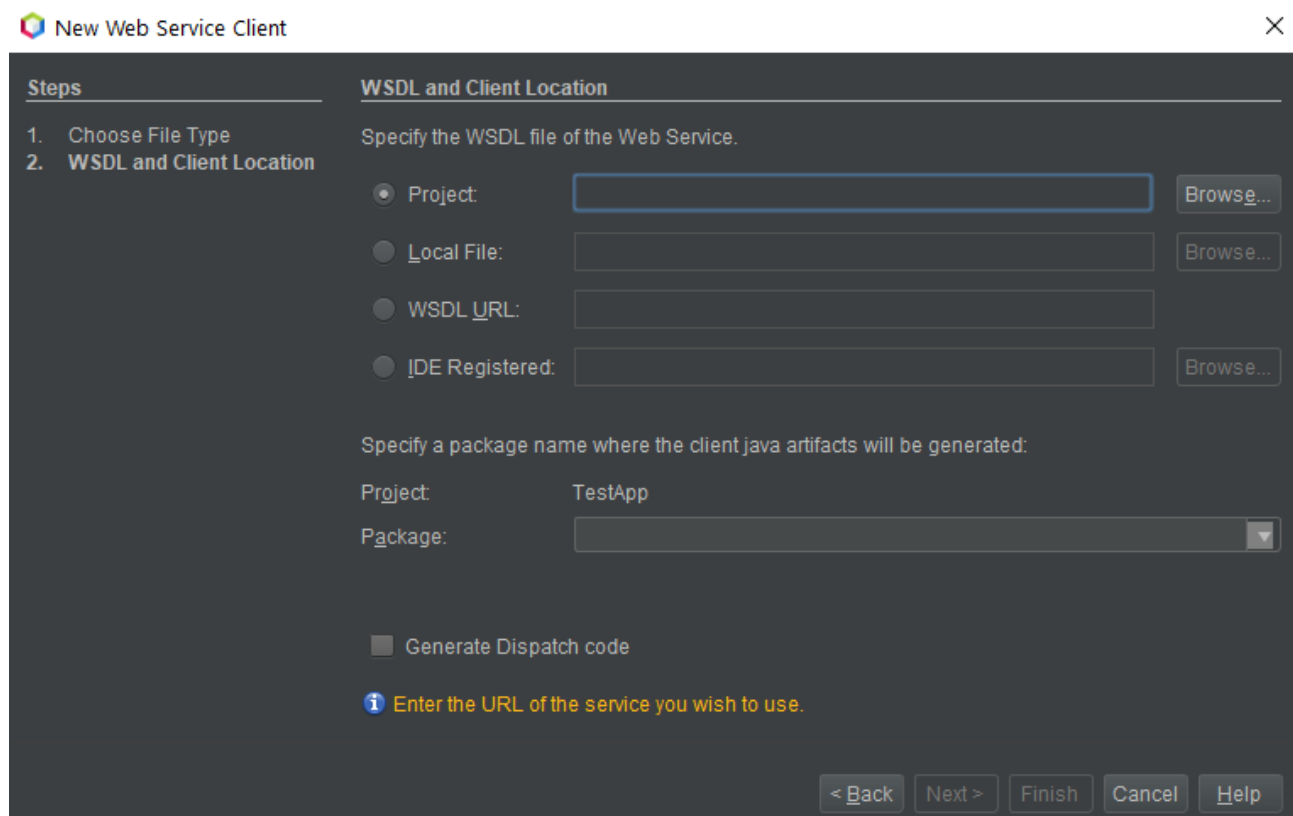
Nel modulo EJB creiamo un nuovo web service.



The screenshot shows the 'New Web Service' dialog box in Eclipse IDE. The 'Name and Location' tab is active. The 'Web Service Name' is 'CardValidator'. The 'Project' is 'ServerModule'. The 'Location' is 'Source Packages'. The 'Package' is 'org.imtech.ws'. There are two radio buttons: 'Create Web Service from Scratch' (selected) and 'Create Web Service from Existing Session Bean'. Below the radio buttons is an 'Enterprise Bean' field and a 'Browse...' button. At the bottom, there is a checked checkbox 'Implement Web Service as Stateless Session Bean'.

Deployare il modulo EJB sul server.

Sull'applicazione client, creare un web service client. La creazione del client viene effettuata solo se il web service è deployato ed esente da problemi che ne vietino il corretto deployment.



The screenshot shows the 'New Web Service Client' dialog box in Eclipse IDE. The 'Steps' tab is active, showing two steps: '1. Choose File Type' and '2. WSDL and Client Location'. The 'WSDL and Client Location' step is selected. The 'Specify the WSDL file of the Web Service.' section has four radio buttons: 'Project' (selected), 'Local File', 'WSDL URL', and 'IDE Registered'. Each radio button has a corresponding text field and a 'Browse...' button. Below this, there is a section 'Specify a package name where the client java artifacts will be generated:' with 'Project' set to 'TestApp' and an empty 'Package' dropdown. At the bottom, there is a checkbox 'Generate Dispatch code' (unchecked) and an information icon with the text 'Enter the URL of the service you wish to use.' The bottom of the dialog has buttons for '< Back', 'Next >', 'Finish', 'Cancel', and 'Help'.

WSDL and Client Location

Specify the WSDL file of the Web Service.

☒ Project:

☐ Local File:

☐ WSDL URL:

☐ IDE Registered:

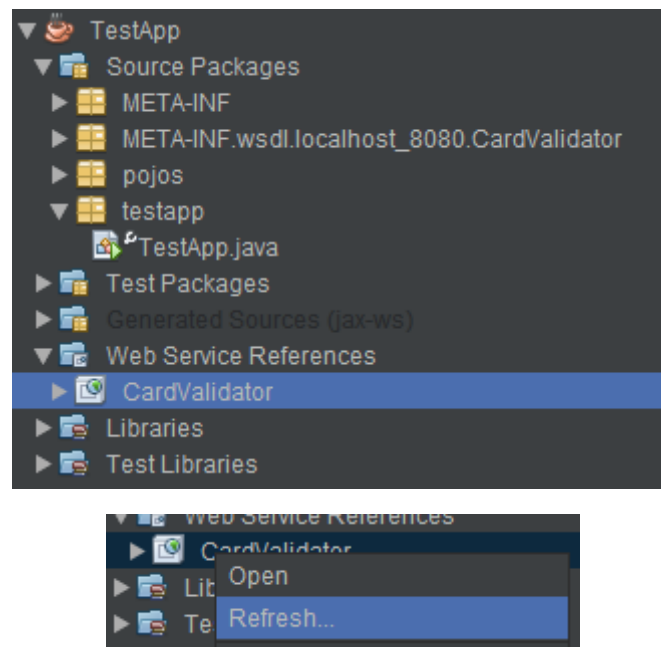
Specify a package name where the client java artifacts will be generated:

Project: TestApp
 Package:

☐ Generate Dispatch code

Aggiornamento del SOAP WS

Nel caso si aggiorni il SOAP WS, sarà necessario rilevare tali modifiche a lato client. Il refresh viene effettuato solo se il web service è deployato ed esente da problemi che ne vietino il corretto deployment.



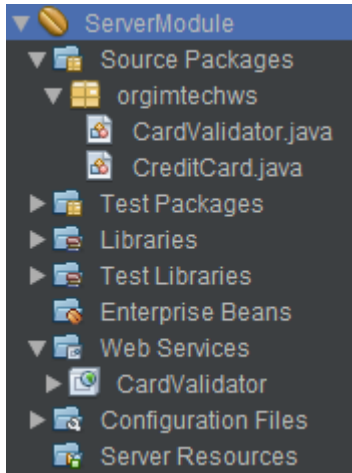
Struttura Client – Web Service

Quando si crea o si aggiorna il client web service nel progetto client vero e proprio, vengono generate diverse classi in quest'ultimo. Le classi generate, sostanzialmente, permettono l'invocazione remota sul web service (sono dei proxy che delegano le chiamate al web service remoto). Sono presenti anche classi che rappresentano i POJO che il web service tratta e classi che

rappresentano i metodi del web service (ebbene, per ogni metodo del web service vengono create due classi).

La situazione, quindi, sarà la seguente:

Progetto Web Service



Progetto Client

