# Capitolo 1

Le aziende vivono oggi in un mondo competitivo. Hanno bisogno di applicazioni per soddisfare i loro interessi che diventano sempre più complessi. Le compagnie hanno diversi centri dati e sistemi internazionali che devono trattare problemi complessi il tutto riducendo i costi e abbassando i tempi di risposta dei loro servizi, memorizzando dati su piattaforme sicure. Le applicazioni enterprise devono affrontare cambiamenti e complessità ed essere robuste. Ecco i motivi per cui Java EE è stato creato.

**Understanding Java EE:** Java EE provvede a delle API per gestire collezioni, transazioni (JTA), messaggi (JMS) o persistenza (JPA).

**Architettura**: Java EE è un insieme di specifiche implementate da differenti container. I container sono ambienti java che provvedono a specifici servizi ai componenti che ospitano, come la  gestione del ciclo di vita, dependency injection, concorrenza. Questi componenti usano contratti ben definiti per comunicare con l'infrastruttura di Java EE e con gli altri componenti.

**Componenti**: Definisce 4 tipi di componenti che un implementazione deve supportare:

- **Applets:** Sono GUI eseguite nel browser. Usano Swing API.
- **Application:** Programmi eseguiti sul client.
- **Web Application:** sono eseguite nel contenitore web e rispondono a richieste HTTP da client web.
- **Appplication Enterprise:** Sono eseguite in un EJB (Enterprise Java Bean). Gli EJB sono contenitori che gestiscono le componenti per processare la  logica di business. Si può accedere ad essi o localmente o remotamente.

**Container:** L'infrastruttura di Java EE è partizionata in domini chiamati container. Ogni container ha un suo ruolo specifico e supporta un insieme di API e offre servizi ai componenti. I container nascondono le complessità tecniche e aumentano la portabilità. Java EE ha 4 diversi container:

- **Applet container:** supportate dai browser per eseguire componenti applet. L'applet container usa una sandbox security model dove il codice è eseguito nella “sandbox”.
- **Application client container(ACC)**: Include un insieme di classi Java, librerie e altri file richieste per l'injection, gestione della sicurezza e servizi di naming. ACC comunica con EJB tramite RMI-IIOP e con il web container tramite HTTP.
- **Web container:** Provvede a servizi per gestire ed eseguire componenti web. È responsabile per istanziare e inizializzare e invocare servlets.
- **EJB Container:** Responsabile per gestire l'esecuzione dei “bean” enterprise. Contengono la logica di business dell'applicazione Java EE.

**Servizi:** Java EE fornisce i seguenti servizi:

- **Java Transacion API:** Offre API per le transazioni
- **Java Persistence API:** Standard API per database relazionali.
- **Validation:** provvede a metodi e classi per la validazione dei dati
- **Java Message Service:** Permette ai componenti di comunicare

asincronamente attraverso messaggi. Supporta P2P e Observer
- **Java Naming e Directory:** API usate per accedere ai nomi e directory di sistema.
- **Java Mail:** API per gestire email.
- **Java Bean Activation Framework:** framework per gestire diversi MIME type.
- **XML Processing:** API per processare XML e codificarli.
- **JSON Processing:** API per generare e parsare JSON.
- **Java EE Connector Achitecture:** I Connettori permettono di accedere al sistema enterprise da un componente Java EE.
- **Security Services:** JACC definisce un contratto tra un server Java EE e un fornitore di servizi di autenticazione, JASPIC definisce un intefaccia standard con la quale i moduli di autenticazione possono essere integrati con i container.
- **Web Services:** JAX serve per servizi web usando il protocollo SOAP/HTTP, JAX-RS per servizi web Restful.
- **Dependency Injection:** Serve per iniettare le risorse in componenti gestite.
- **Management:** Java EE definisce API per gestire i contenitori e i server.
- **Deploy:** Definisce un contratto tra i tool distribuiti e i prodotti Java EE.

Protocolli di rete:
1. **HTTP:** Server-side(Servlet, JSP, JSF)
2. **HTTPS:** combinazione di HTTP e Secure Socket Layer
3. **RMI-IIOP:** permette di invocare oggetti remoti indipendentemente dal protocollo sottostante.

**Packaging:** Per essere distribuiti in un container, i componenti devono essere prima impacchettati in archivi formattati. Java EE definisce differenti tipi di moduli che hanno i loro formati di packaging.
- **Application client:** Jar + META_INF(usato per definire estensioni e pacchetti relati)
- **EJB:** contiene una o più sessioni + META_INF.
- **Web Application:** contiene JSP o servlet o JSF impacchettati in un file .war + META_INF.
- **Enterprise:** EJB impacchettati in file .ear + META_INF.

**Annotations e Deployment:** In Java EE la programmazione dichiarativa è fatta usando metadati che sono le annotazioni (@) e i deployment descriptor(XML). Questi ultimi vanno inseriti nel file META
**JCP:** è un ' organizzazione aperta che è coinvolta nella definizione delle future versioni di Java EE. I passi per standardizzare una funzione o altro sono i seguenti:
- Il creatore della funzionalità manda un form ad un gruppo di esperti.
- Il gruppo deve rilasciare una o più specifiche sulla richiesta, implementazione della richiesta e test di compatibilità

## Capitolo 2

**Understanding Beans:** I managed Bean sono oggetti contenitori che

supportano solo un piccolo insieme di servizi base: risorse, ciclo di vita e interazione. I beans sono oggetti CDI che sono costruiti sul modello del Managed Bean.
**Dependency Injection:** é un design pattern che disaccoppia le componenti dipendenti. Invece di un oggetto che cerca altri oggetti, è il contenitore che inietta questi oggetti dipendenti.
**Gestione del ciclo di vita:** Quando si crea un'istanza di un CDI Bean all'interno di un contenitore bisogna iniettarlo e sarà il contenitore a fare il resto. Questo significa che il contenitore è responsabile del ciclo di vita di un bean.
**Visibilità e contesto:** CDI Bean possono avere uno stato e in più sono contestuali, vale a dire che vivono in uno scope ben definito.
**Interception:** Sono usati per frapporsi sull'invocazione di un metodo simile alla programmazione orientata agli aspetti. Quest'ultima separa i concetti cross-cutting (trasversali) dal codice di business. I Bean supportano questa programmazione e gli interceptor. Sono automaticamente innescati dal container quando un metodo del bean è invocato.
**Basso accoppiamento e forte tipizzazione:** Oltre a disaccoppiare la gestione del ciclo di vita e l'interazione con altri bean, un CDI può anche disaccoppiare tramite decoratori concetti di business in modo typesafe, usando annotazioni tipizzate per legare i bean assieme.
**Deployment Descriptor:** Con CDI il deployment descriptor è chiamato “beans.xml”. Può essere usato per configurare alcune funzionalità ma è essenziale per abilitare CDI, questo perché CDI ha bisogno di identificare i bean nella classpath(bean discovery). È durante la bean discovery che CDI trasformaun POJO(semplice classe Java) in un CDI Bean. A tempo di deploy CDI controlla tutti i file nei .jar  e .war e ogni volta che trova un “beans.xml” traformaun POJO in un Bean.
**Scrivere un CDI Bean:** Un CDI Bean può essere qualunque tipo di classe Java che contiene logica di business chiamato o tramite injection o invocato tramite EL in una pagina JSF.
Anatomia di un CDI Bean:

- Non è una classe interna non statica.
- È una classe concreta o annotata.
- ha un costruttore di default senza parametri o dichiara un costruttore annotato con @Inject

**@Inject:** Java EE fa sì che le dipendenze non sono costruite a mano ma lasciate al container. Questo lo si fa con l'annotazione @Inject. La injection la si può fare tramite 3 meccanismi:

```
1)@Inject (injection tramite proprietà)
        private NumberGenerator numbergenerator
2)@Inject (injection tramite costruttore)
        public BookService(NumberGenerator numbergenerator)
                this.numbergenerator = numbergenerator
3)@Inject (injection tramite setter)
public void setnumberGenerator(numberGenerator numberGenerator)
this.numberGenerator = numberGenerator
```

**Default Injection:** fa un injection di default

```
@Default
public class IsbnGenerator imlements NumberGenerator
        public String generateNumber()
                return “some string”

@Inject @Default (chiamerà isbn generator)
private NumberGenerator numberGenerator
```

**Qualificatori:** CDI usa i qualificatori che sono annotazioni per individuare quali metodo applicare al bean.

Esempio:

```
@ThirteenDigits
public class IsbnGenerator implements NumberGenerator

@EightDigits
public class IssnGenerator implements NumberGenerator

(Per passare IsbnGenerator)
@Inject @ThirteenDigits
private NumberGenerator numberGenerator
```

**Qualificatori con membri:**Possiamo aggiungere a un qualificatore dei membri per dettagliarlo.

Esempio:

```
public @interface NumberOfDigits{
Digits value();
boolean odd();
}

public enum Digits{TWO, EIGHT, THRTEEN}
@Inject @NumberOfDigits(value = Digits.THIRTEEN, odd=false)
private NumberGenerator numberGenerator;
```

**Qualificatori multipli:** Si usano più qualificatori per definire caratteristiche.

Esempio

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

//quandò chiamerà il metodo generate number userà quello di isbn
@Inject @ThirteenDigits @Even
private NumberGenerator numberGenerator
```

**Alternatives:** Sono bean annotati con il qualificatore @Alternative. Di default sono disabilitati e per abilitarli, è necessario modificare il file “beans.xml”. Sono utili quando vogliamo iniettare un implementazione su un particolare scenario di deploy.

Esempio: Un generatore di codici da usare in ambiente di testing

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

Ad @Alternative si possono affiancare @Default e Qualificatori.

**Producers:** permette a qualsiasi classe che non sia un bean di essere iniettata in un CDI bean.

Esempio
public class NumberProducer{
@Produces @ThirteenDigits
private String prefix = “13”;
@Produces
private int editorNumber = 84356;
@Produces @Random
public double rabdom(){return new Math.abs(new Random().nextInt())
}

public class IsbnGenerator implements NumberGenerator{
@Inject @ThirteenDigits
private String prefix;
@Inject @ThirteenDigits
private int editorNumber;
@Inject @Random
private double postfix;
//questo metodo ora genererà un numero composto da 13+84356+numero
//casuale
public String generateNumber() return prefix+editorNumber+postfix
**Disposers:** Utilizzati per la distruzione di un oggetto. Un metodo disposer permette all'applicazione di eseguire un pulitura personalizzata di un oggetto restituito da un metodo producer. Ogni metodo disposer deve avere esattamente un parametro dello stesso tipo e qualificatori del producer.

Esempio: Eliminazione di una connessione ad un database

@Produces
private Connection createConnection(){...}

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

**Scope:** Ogni oggetto gestito da CDI ha uno scope ben definito e un cliclo di vita legato ad uno specifico contesto. Un beans è legato dal contesto e ci rimane finché non è distrutto dal contenitore. Mentre il web tier ha uno scope ben definito, il service tier non lo ha perché un POJO o un session bean è usato all'interno di un applicazione web, non è consapevole dei contesti delle applicazioni web. CDI unisce i tier web e services insieme legandoli a scope significativi.

@ApplicationScope: Per l'intera durata dell'applicazione.

@SessionScope: Per tutta la richiesta HTTP.
@ConversationScope: Tra invocazioni multiple all'interno dei confini di sessione con punti di inizio e fine determinato dall'applicazione.
@Dependent: Il ciclo di vita è lo stesso del client (assegnato di default)

**Interceptors:** Permettono di aggiungere concetti di cross-cutting nei bean. Si dividono in 4 tipi:

- Constructor-level: Associati con un costruttore o la classe bersaglio(@AroundConstruct)
- Method-level: associati ad un metodo di business(@AroundInvoke)
- Timeout-method: che si interpongono sui metodi di timeout(@AroundTimeout)
- Life-cycle callback: Si interpongono sul bersaglio istanziando una callback.

Un metodo annotato con un interceptor deve avere il seguente pattern:

Object<METHOD>(InvocationContext ic)throws Exception

- Il metodo può essere public, protected, private, package ma non static o final.
- Deve avere come argomento InvocationContext e deve restituire Object.
- Deve lanciare un'eccezione.

**Class Interceptor:** Per separare gli inteceptor bisogna creare una classe separata e istruire il container su quale bean applicarlo.

Esempio: Una classe che fa da logger

```
public class LogInterceptor{
@Inject Logger

@AroundConstruct
nome-metodo(InvocationContext ic) throws Exception{//logica del metodo}

@AroundInvoke
nome-metodo(InvocationContext ic) throws Exception{//logica del metodo}
}

public class CustomerService{
@Interceptors(LogInterceptor.class)//chiama @AroundConstruct
public CustomerServices(){...}
@Interceptors(LogInterceptor.class)//chiama @AroundInvoke
public createCustomer()
}
```

**Life Cycle Interceptor:** Con delle annotazioni callback è possibile invocare certi metodi, da parte del contenitore, ad una certa fase del ciclo di vita. Queste annotazioni sono @PostConstruct e @PreDestroy.

**Concatenare ed escludere Interceptor:** Le interceptor si possono concatenare in una lista separata da virgola. L'ordine in cui vengono invocati è

determinato dall'ordine in cui sono specificati.

Esempio: @Interceptors({I1.class, I2.class})

possiamo far sì che un metodo non invochi intercettori usando @ExcludeClassInterceptor

**Interception Binding:**Gli interceptor possono essere usati in qualunque Beran gestito ma il CDI lo ha esteso aggiungendo l'interceptor bindong ovvero può essere utilizzato solo se CDI è abilitato. Un interceptor binding è un'annotazione definita dall'utente che è a sua volta annotata con @InterceptorBinding

Esempio: Una classe che fa da logger

@Interceptor
@Loggable
public class LoggingInterception{...}
@Loggable //senza dichiarare l'interceptor class
public class CustomerService{...}

Gli interceptor per essere utilizzati devono essere abilitati nel file “beans.xml”

**Prioritizzare gli Interceptor Binding:** Gli interceptor binding non hanno la possibilità di essere priorotizzati come quelli normali ma per ovviare al problema si può usare l'annotazione @Priority seguita da un intero. La regola è che l'interceptor con valore minore viene chiamato per prima. Si possono usare delle costanti di riferimeto:

- PLATFORM_BEFORE = 0
- LIBRARY_BEFORE = 1000
- APPLICATION = 2000
- LIBRARY_AFTER = 3000
- PLATFOMR_AFTER = 4000

**Decorators:** L'idea è quella di prendere una classe e di incapsularla in un altra. In questo modo quando si chiama una classe decorata passerà attraverso il decoratore prima di raggiungere la classe vera e propria. I decoratori aggiugono logica addizionale ai metodi di business.

Esempio:
@Decorator
public class FromEighttoThirteenDigitsDecorator implements
NumberGenerator{

@Inject @Delegate
private NumberGenerator numberGenerator;

public String generateNumber(){
String issn = numberGenerator.generateNumber();
String isbn = “13-84356”+issn.substring(1);
return isbn;

```
}
}
```

I decoratori possono avere un punto di injection delegato (annotato con @Delegate) con lo stesso tipo del bean che decorano. I decorator devono essere abilitati nel file “beans.xml”.

**Events:** Permettono ai bean di interagire cin nessun tempo di compilazione di dipendenze. Un bean può definire un evento, un altro può chiamarlo e un altro può gestirlo. Questo schema segue l'bserver pattern. Gli eventi sono attivati dal metodo fire() e non è dipendente dall'observer.

Esempio:

```
public class BookService{
@Inject
private NumberGenerator numberGenerator;
@Inject
private Event<Book> bookAddedEvent;

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

Ogni volta che un evento è attivato da un bean, CDI ferma l'esecuzione e la passa all'observer che fa le sue cose e al termine ritorna al punto in cui si era fermato il bean. CDI quindi  non tratta gli eventi asincronamente.

```
public class Inventory{
@Inject
private Logger logger;
List<Book> inventory = new Arraylist<>();
public void addBook(@Observes Book book){
logger.info(“Adding book”);
inventory.add(book);
}
}
```

## Capitolo 3

La validazione dei dati è un'operazione comune che si espande su tutti gli strati di un'applicazione. Essendo cruciale per un'applicazione la validazione viene definita in ogni strato con delle proprie regole e spesso la stessa logica validazione è implementata in ogni strato provvedendo a sprecare tempo, difficile da manutenere. Bean Validation risolve il problema della duplicazione del codice permettendo agli sviluppatori di scrivere una restrizione un'unica volta, usarla e validarla in ogni layer.

**Understanding Constrains and Validation:** Nelle applicazioni etereogenee, gli sviluppatori devono trattare con diverse tecnologie e linguaggi e quindi anche semplici regole di validazione diventano complesse.

**Application**: La validazione a livello applicativo può risiedere in posti multipli per assicurare che i dati siano corretti.
Presentation: In questo strato si validano i dati perché sono spediti da diversi client.
**Business logic:** Questo strato gestisce le chiamate ai servizi interni e esterni...quindi i dati devono essere validati.

**Business model:** Questo strato mappa il modello del dominio nel database quindi bisogna validare i dati prima di memorizzarli.

**Database:** anche se il database gestisce la validazione dei dati, ha vari effetti collaterali: ha costi di errori e performance alti. A livello di database le restrizioni sono solo consapevoli dei dati, non cosa l'utente sta facendo. Quindi i messaggi di errore non sono consapevoli del contesto e non può essere molto esplicito.

**Client:** Dal lato client è importante validare i dati cosicchè l'utente sia velocemente informato che ha inserito dati errati.

**Interoperabilità:** Spesso le applicazioni enterprise hanno bisogno di scambiare dati con sistemi e partner esterni. Queste applicazioni ricevono dati di qualunque tipo di formato. Oggigiorno XML è il linguaggio preferito per scambiare dati tra sistemi eterogenei.

**Bean Validation Specification:** Bean Validation permette di scrivere una restrizione un'unica volta e usarla in qualunque strato dell'applicazione. È disponibile sia lato client che lato server.

Esempio: public class Book{
@NotNull
private String title;
@Size(max=2000)
private String description;
}

**Anatomia di una restrizione:** Sono definite da una combinazione di annotazioni e una lista di implementazione di una validazione. La validazione può essere applicata su metodi, campi, tipi. Bean Validation permette di scriver anche le proprie validazioni. Una restrizione è formata da: un annotazione definite dalle restrizione, una lista di classi che implementano l'algoritmo della restrizione.

**Constraint Annotation:** La restrizione in un Java Bean è espressa attraverso una o più annotazioni. Un annotazione è considerata una restrizione se la sua politica di Retention contiene RUNTIME e se l'annotazione stessa è annonata con javax.validation.Constraint. Constraint Annotation sono solo regolari annotazioni e devono definire alcuni metatag:

- @Target({METHOD, FIELD,...}) Specifica il target con il qualel'annotazione può essere usato.
- @Retention(RUNTIME) Specifica come l'annotazione sarà operativa.
- @Constraint(valitatedBy=Class.class) Specifica la classe che incapsula l'algoritmo di validazione.
- @Documented: Questa meta-annotation specifica che questa annotazione sarà inclusa nella javadoc.

Inoltre definisce tre attributi extra:
- Message: Provvede per una restrizione di restituire un messaggio internazionalizzato se la restrizione non è valida.
- Groups: I gruppi sono tipicamente usati per controllare l'ordine con il quale le restrizioni sono valutate.
- Payload: È usato per associare metadati con una restrizione.

**Constraint Implementation:** Le restrizioni sono definite dalla combinazione di un' annotazione e zero o più classi implementatrici. Una classe deve implementare ConstraintValidation che definisce 2 metodi:
- inizialize: Questo metodo è chiamato dal fornitore del Bean Validation prima di qualunque uso della restrizione. Qui è dove di solito si inizializzano i parametri della restrizione se ce ne sono.
- IsValid: Qui è dove è implementato l'algoritmo di validazione.

**Applicare una restrizione:** Una volta che si ha l'annotazione e la sua implementazione, la si può applicare a un dato tipo di elemento. Questa è una decisione che gli sviluppatori devono fare e implementare usando @Target meta-annotation.
- FIELD: per attributi.
- METHOD: per metodi.
- CONSTRUCTOR: per costruttore.
- PARAMETER: per imparametri di metodi e costruttori.
- TYPE: Per i bean, classi e interfacce.
- ANNOTATION TYPE: per restrizioni composte da altre restrizioni.

**Buil-in Constraints:**
1. **assertFalse/True:** (l'elemento deve essere vero o falso)
2. **decimalMin/Max:** (l'elemento deve essere più grande o più piccolo di uno specifico valore)
3. **future/past:**(l'elemento deve essere una data futura o presente)
4. **Max/Min:** (l'elemento deve essere più grande o più piccolo di uno specifico valore)
5. **NotNull/Null:** (l'elemento deve essere nullo o non nullo)
6. **Pattern:** (l'elemento deve seguire un espressione regolare)
7. **Digits:**(l'elemento deve essere un numero all'interno dell'intervallo)
8. **Size:**(la grandezza dell'elemento non deve essere in uno specifico intervallo)

**Composition Costraints:** Un modo per creare nuove restrizioni è quello di aggregare quelle già esistenti per crearne una di più alto livello.

**Generic Constraints:** Spesso si ha bisogno di algoritmi di validazione più

complessi. In questo caso si ha bisogno di aggiungere una classe alle annotazioni.

Esempio: Classe che usa URL annotation
public class ItemServerConnection{
@URL
private String resourceURL;
@NotNull @URL(protocol=”http”, host=”www.cdbookstore.com”)
private String itemURL;
@URL(protocol=”ftp”, port=”21”)
private String ftpServerURL;
}

custom URL annotation

@Costraint(validatedBy={URLValidator.class})
@Target({METHOD, FIELD, ANNOTATION:TYPE, CONSTRUCTOR, PARAMETER})
@Retention(RUNTIME)
public interface URL{
String message() default “Malformed URL”;
Class<?>[] groups() default {}
Class<? Extends Payload>[] payload default{};
String protocol() default””;
String host() default””;
int port() default -1;
}

Constraint Implementation

class URLValidator implements ConstraintValidator<URL, String>{
private String protocol;
private int host;
private inr port;

public void inizialize(URL url){
this.protocol = url.protocol();
this.host = url.host();
this.port = url.port();
}

public boolean isValid(String value, ConstraintValidatorContext context)
if(value==null || value-length()==0){return true}
try{
url=new URL(value);
}catch(MalformedURLException e){
return false;
}
if(protocol !=null && protocol.length()>0 && !url.getProtocol().equals(protocol))
{return false;}
if(host !=null && host.length()>0 && !url.gethost().startswith(host)){return
false;}
if(port != -1 && url.getPort()!=port){return false;}

return true;
}

**Class Level Constraints:** Si possono creare restrizioni su di un intera classe.

Esempio:
@ChronologicalDates
public class Order{
private String orderId;
private Double totalAmount;
private Date creationDate, paymentDay, deliveryDate;
}

public class ChronologicalDatesValidator implements
ConstraintValidator<ChronologicalDates, Order>{
boolena isValid(Order order, ConstraintValidatorContext context){
return order.getCreationDate().getTime() < order.getPaymentDate().getTime()
< order.getDeliveryDate().getTime();
}
}

**Method level Constraint:** Sono dichiarate sui metodi così come sui costruttori. Queste restrizioni possono essere aggiunte ai parametri o ai metodi stessi. Questo permette la ben-nota Programmazione per contratto(Post-Condizioni,Pre-Condizioni).Anche la bean validation si può ottenere tramite annotazionio XML.

**Validation API:** L'API principale è l'interfaccia Validator. Mantine il contratto per validare oggetti indipendentemente dallo strato nel quale è implementato. Una volta che la validazione fallisce un insieme d interfacce di ConstraintViolation è restituito. Queste interfacce espongono il contesto di violazione della restrizione così come il messaggio che descrive la violazione.

**Validator:** Le API dell'interfaccia Validator sono in grado di validare le istanze di un bean usando solo pochi metodi. Tutti questi metodi hanno la seguente routine per ogni restrizione dichiarata:

- Determina l'appropriata implementazione di ConstraintValidator da usare per la restrizione.
- Esegue il metodo isValid.
- Se isValid è falso la BeanValidation provvede a daggiungere ina ConstraintViolation alla lista delle restrizioni violate.

Se accadono failure durante la routine di validazione viene lanciata una ValidationException. I metodi validate, validateProperty e validateValue sono usati per validare un intero bean. Il metodo forExecutables provvede all'accesso di un ExecutableValidator.

**ConstraintViolation:** Tutti i metodi di validazione restituiscono un insieme di ConstraintViolation che sono iterate in modo da vedere quali errori di validazione occorrono. Se l'insieme è vuoto la validazione ha avuto successo altrimenti un'istanza di ConstraintViolation è aggiunta all'insieme per ogni

restrizione violata.

**Obtaining a Validator:** Il primo passo attraverso la validazione di un bean è tenere un istanza di Validator.

Java:
ValidatorFactory factory = Validation.builddefaultValidationFactory();
Validator validator = factory.getValidator();

JNDI:
@Resource validatorFactory validatorFactory;
@Resource Validator validator;

CDI:
@Inject ValidatorFactory;
@Inject Validator;

**Validare un bean:** Una volta creato un Bean con le restrizioni se ne crea un istanza e si chiama Validator.validate();

CD cd = new CD(“titolo”, 12.5f);
Set<ConstraintViolation<CD>> violations = validator.validate(cd);
assertEquals(0, violations.size());

**Validare proprietà:** Con il metodo ValidateProperty() possiamo validare una singola proprietà di uno specifico oggetto.
Esempio:
CD cd = new CD();
cd.setNumberOfCDs(2);
Set<ConstraintViolation<CD>> violations = validator.validateProperty(cd, “numberOfCDs”);
assertEquals(0, violations.size());

**Validare valori:** Il metodo Validator.validateValue() controlla se una singola proprietà di una data classe può essere valutata con successo.

Esempio:
Set<ConstraintViolation<CD>> constr = validator.validateValue(CD.class, “numberOfCDs”, 2);
assertEquals(0, constr.size());

**Validare Metodi:** I metodi per la validazione dei metodi si trovano nell'interfaccia ExecutableValidator. Il metodo forExecutables(), restitusce questo ExecutableValidator sul quale puoi invocare i suoi metodi.

Esempio:
CD cd = new CD(“Pd”, 13.6f);
Method method = Cd.class.getMethod(“calculatePrice”, Float.class);
ExecutableValidator methodValidator = validator.forExecutable();
Set<ConstranintViolation<CD>> violations = methodValidator.validateParameters(cd ,method, new Object[]{new Float(1.2)});

assertEquals(1, violations.size());

**Validare Gruppi:** Un gruppo definisce un sottoinsieme di restrizioni. Invece di validare tutte le restrizioni per un dato bean, solo un sottoinsieme è validato. Ogni restrizione definisce la lista dei gruppi a cui appartiene. Se nessun gruppo è esplicitamente dichiarato una restrizione appartiene al Default Group.

Esempio:
CD cd = new CD();
cd.setDescription("cd di pd") //la sua restrizione appartiene ad una altro gruppo.
Set<ConstriaintViolation<CD>> violations = validator.validate(cd, Default.class);
asserEquals(2, violation.size()) //verranno validate tutte le restrizioni appartenenti a Default.class quindi descrizione no.

# Capitolo 4

Le applicazioni sono formate da lofica di business, interazione con gli altri sistemi, interfacce utente...e dati. I dati persistenti sono dappertutto e molte volte si usano i database relazionali. JPA è la tecnologia che usa Java Enterpise per unire database ed oggetti.
**Understandind Entities:** Le entità sono oggetti che vivono brevemente in memoria e permanentemente nei database. Si possono manipolare le entità usando JPQL(Java Persistence Query Language). In JPA un'entità è un POJO (Plain Old Java Object)

**Anatomia di un'entità:** Per una classe essere un'entità, deve essere annotata con @Entity. L'annotazione @Id definisce la chiave primaria. Per essere un entità una classe deve seguire le segueti regole:

- L'intera classe deve essere annotata con @Entity.
- @Id deve essere usata per assegnare una chiave primaria.
- La classe non deve avere argomenti al costruttore che deve essere pubblico o protetto.
- Un'interfaccia o un enumerazione non possono essere entità.
- La classe non può essere final e nemmeno metodi o variabili.
- Se un'istanza di un'entità deve essere passata ad un oggetto deve implementare Serializable.

**Object-Relational Mapping:** Il principio ORM è di delegare a un tool esterno, l'obbligo di creare una corrispondenza tra oggetti e tabelle. JPA mappa gli oggetti nel database attraverso i metadati. Il metadato può essere scrutto in due dirrefenti modi: annotazioni o XML.

**Configuration over Exception:** Java EE introduce l'idea di configurazione attraverso l'eccezione. Questo significa che il contenitore deve applicare le regole di default. In altre parole dover inserire una configuarazione è l'eccezione alla regola.

**Regole di default:**

1. Il nome dell'entità è lo stesso di quello della tabella nel database.
2. I nomi degli attributi sono gli stessi delle colonne nel database.
3. Regole sui primitivi: String = VARCHAR, Long = BIGINT.

**Querying Entities:** Il pezzo centrale delle API responsabile di orchestrare le entità è EntityManager. Il suo ruolo è gestire le entità e permettere operazioni fondamentali(creare, leggere, aggiornare e cancellare).

Esempio:
EntityManagerFactory em =
Persistence.createEntityManagerFactory(“chapter04PU”)//persistence unit
em.persist(book)//inserisce un libro nel database;

L'entity manager permette anhe di gestire le query

Esempio:
“SELECT b FROM Book b WHERE b.title='H2G2'”;
/*NOTA: usa b come alias per accedere alla tabella
/*NOTA: b.title è l'attributo dell'oggetto e non campo tabella

Le query statiche sono definite con l'annotazione @NamedQuery

Esempio:
@NamedQuery(name=”find-books”, query=”SELECT b FROM Book b”);

**PersistenceUnit:** EntityManagerFactory passa il nome di un persistent unit come parametro. Persistent unit indica all'entity manager il tipo di database da usare e i parametri di connessione che sono definiti nel file persistence.xml.

**EntityLifeCyle and Callbacks:** Le entity hanno un ciclo di vita e 4 categorie di operazioni(persistenting, updating, removing e loading). Ogni operazione ha pre e post eventi che possono essere intercettati dall'entity manager per invocare un metodo di business.si può ache integrare la BeanValidation con JPA

Esempio:
@Entity
public class Book implements Serializable{
@Id
private Integer id;
@NotNull
private String title;
}

## Capitolo 6

**Entity Manager:** Quando un entity manager ottiene una referenza a un'entità è detta gestita. La forza di JPA è che l'entità può essere usata regolarmente da differenti strati dell'applicazione e poi gestita dall'entity manager per effettuare operazioni col database.

**Obtaining an Entity Manager:** A seconda se un container-gestito o

un'applicazione gestita. Nel primo caso, le transazioni sono gestite dal contenitore e non bisogna esplicitamente  scrivere il commit o il rollback, nel secondo caso lo si deve fare. Il termine applicazione gestita vuol dire che un applicazione è responsabile di ottenere esplicitamente un istanza di entity manager e gestendone il ciclo di vita.

**Persitence Context:** Il contesto di persistenza è un insieme di istanze di entità gestite in un dato tempo per una data transazione dell'utente: solo un'istanza di un'entità con la stessa identità di persistenza, può esistere in un contesto persistente. Esempio(Se ho un'istanza della classe Libro con id 12, nel contesto di persistenza nessun altra istanza nel medesimo contesto può avere lo stesso Id). L'entity manager aggiorna o consulta il contesto di persistenza ogni qualvolta che un metodo di entity manager viene chiamato. Il contesto di persistenza può essere visto come una sorta di cache di primo livello: è un piccolo spazio di vita dove l'entity manager memorizza le entità prima di scaricare il contenuto nel database. Il file persistence.xml definisce le impostazioni per connettersi al database e la lista di entità da essere gestite nel contesto di persistenza.

**Manipolare Entità:** Usiamo i metodi di Entity Manager

- void persist(Object Entity)//crea un'istanza gestita e persistente
- <T> T find(Class<T> entityClass, Object primaryKey)//Cerca un'entità con una specifica chiave primaria.
- <T> getReference(Class<T> entityClass, Object primaryKey)//Ottiene un'istanza il cui stato può essere eliminato.
- Void remove(Object entity)//rimuove l'entità dal contesto e dal database.
- <T> T merge(T entity) //Ordina lo stato di una data entità nel corrente contesto di persistenza.
- Void refresh(Object entity) //Aggiorna lo stato di un'istanza dal database sovrascrivendo i cambiamenti fatti alle entità.
- Void flush() //Sincronizza il contesto di persistenza col database sottostante.
- Void clear()//Pulisce il contesto di persistenza.
- Void detach(Object entity) //Rimuove un'entità dal contesto di persistenza.
- Boolean contains(Object entity) //Controlla se l'istanza è un entità appartenente alcorrente contesto di persistenza.

**Persisting Entity:** Significa inserire un dato nel database che non  esiste ancora. Per farlo, istanziare l'entità, settare gli attributi e legare le entità all'Entity Manager con persist().

Esempio:
Customer customer = new Customer(“Antony”,“Balla”,“tballa@mail.com”);
tx.begin();//Metodo di Entity Transaction
em.persist(customer);
tx.commit(); //Metodo di Entity Transaction.

**Importante:** l'ordine dell'inserimento delle entità legate da chiave esterna non è importante.

**Finding by ID:**

- usare find(): Customer customer = em.find(Customer.class, 12)
- usare getReference(): diverso da find perché non recupera i dati.

**Removing an Entity:** usare remove(): Viene rimosso dal database e dal contesto di persistenza.

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

**Remove Orphans:** Con JPA possiamo informare il fornitore di persistenza(il database) di rimuovere automaticamante gli orfani o tramite rimozione a cascata. Le associazioni uno a molti includono questa opzione aggiungendo orphanRemoval = true all'annotazione.

Esempio:
@OneToOne(fetch = FetchType.LAZY, orphanRemoval = true)
private Address address;

**Flushing an Entity:** col metodo flush() il fornitore di persistenza può essere forzato a scaricare i dati nel database ma non fa commit alla transazione.

Esempio:
tx.begin();
em.persist(customer);
em.flush();
tx.commit();

**Refreshing an Entity:** col metodo refresh() sincronizza i dati sovrascrivendo i dati dell'entità gestita con i dati persistenti nel database. Un tipico caso d'uso è quando si vuole annullare i cambiamenti effettuati all'entità solo in memoria.

Esempio:
Customer customer = en.find(Customer.class, 1234L);
assertEquals(customer.getFirstName(), “Antony”);
customer.setFirstName(“William”);
em.refresh(customer);
assertEquals(customer.getFirstName(), “Antony”);

**Content of the Persistence Context:** Il contesto di persistenza mantiene le entità gestite.

**Contains:** Il metodo contains() permette di controllare se una particolare istanza di un'entità è gestita dall'entity manager all'interno del contesto.

**Clear and Detach:** Il metodo clear() svuota il contesto di persistenza. Il metodo detach() rimuove una data entità dal contesto di persistenza.

**Merging an Entity:** Il metodo merge() riattacca un'entità al contesto di

persistenza. Il metodo clear() forza il detach dell'entità ma continua a vivere fuori dal contesto di persistenza.

**Update an Entity:** se l'entità è gestita, i cambiamenti saranno riflessi nel database automaticamente altrimenti bisognerà esplicitamente chiamare merge().

**Cascading Events:** per gli eventi a cascata basta aggiungere l'attributo **cascade** alle annotazioni @OneToOne, @OneToMany, @ManyToOne, @ManyToMany.

Esempio:
@OneToOne(fetch=FetchType.LAZY, cascade={CascadeType.PERSIST, CascadeType.REMOVE})

## Capitolo 5

Possiamo mappare un'entità tramite xml descrivendo  le caratteristiche dell'entità. Gli xml prendono la precedenza sulle annotazioni. Ricordarsi che nel persistence.xml bisogna aggiungere  la referenza dell'entità.

**Relationship Mapping:** Associazioni fra classi:

- Unidirezionali: Una classe può navigare nell'altra ma non il contrario. La prima classe ha un'istanza dell'altra.
- Bidirezionali: Una classe può navigare nell'altra e viceversa. In java entrambe le classi hanno un'istanza dell'altro.
- Cardinalità: numero di oggetti coinvolti nell'associazione.

Nei database le relazioni si ottengono o tramite colonne(chiave esterna) o tramite tabelle che contengono òe chiavi delle entità. In JPA esistono le annotazioni standard che specificano le relazioni

- @OneToOne: //chiave nell'entità che porta la relazione
- @OneToMany: //chiave nell'entità a molti
- @ManyToMany: //chiave in una tabella esterna

Nelle relazioni bidirezionali bisogna aggiungere l'elemento mappedBy per stabilire chi è il portatore della relazione.

**@JoinColumn:** serve per indicare quale colonna sarà usata come chiave esterna. //Relazioni Uno ad Uno
**@JoinTable:** serve per indicare quale tabella farà da join alle due entità. //Relazione Molti a Molti

Esempio:
public class Order{
@Id @GeneratedValue
private Long id;
@Temporal(TemporalType.TIMESTAMP)
private Date creationType;
@OneToMany
@JoinTable(name=”jnd_ord_line”, joinColumns =

@JoinColumn(name=”order_fk”), inverseJoinColumns=
@JoinColumn(name=”order_line_fk”))//indica colonna dell'altro capo della relazione
private List<OrderLine> orderLines;
}

**JPQL:** Permette di manipolare le entità attraverso un linguaggio la cui sintassi è simile a SQL con la differenza che SQL restituisce un risultato sottoforma di righe-colonne mentre JPQL usa un Entità  o una collezione di entità. JPQL usa il meccanismo di mapping per trasformare una query JPQL in un linguaggio comprensibile a un database SQL.

Esempio_1: SELECT b FROM Book b

//Invece di selezionare da una tabella, JPQL seleziona entità. FROM è usato per dare ad un alias all'entità: b è un alias per Book. SELECT indica che il tipo di risultato della query è un entità Book. L'esecuzione di questo statement risulterà in una lista di zero o più istanze di Libro.

Esempio_2: SELECT b FROM Book b WHERE b.title = 'H2G2'

//nota che l'alias lo si usa per navigare tra gli attributi delle entità e non sulle colonne del database.

**SELECT:** retituisce una delle seguenti form: un'entità, attributi di entità, un espressione costruttore.

Esempi:
1. SELECT c FROM Customer c
2. SELECT c.firstName FROM Customer c
3. SELECT c.firstName, c.lastName FROM Customer c

Da JPA 2.0 un attributo può essere restituito a seconda di una condizione(Usando CASE WHEN...THEN...ELSE...END)

Esempio:
SELECT CASE b.editor WHEN 'Apress' THEN b.price *0,5 ELSE b.price *0,8 END FROM Book b

//Restituisce il prezzo dei libri ese sono di casa editrice Apress sono scontati del 50% altrimenti dell'80%.

SELECT NEW org.agonical.javaee7,CustomerDTO(c.firstName, c.lastName, c.address.street)

//Restituisce una lista di CustomerDTO che sono stati istanziati con l'operatore new e inizializzato con c.firstName, c.lastName e c.address.street

SELECT DISTINCT c FROM Customer c

//restituisce una lista di customer rimuovendo i duplicati

Con select si possono usare le clausole AVG, COUNT, MAX, MIN, SUM e possono essere raggrupati con GROUP BY e filtrati con HAVING.

**FROM:** definisce entità dalla dichiarazione di variabili identificatrici(o alias). La sintassi di From consiste di un entità e di un alias

**WHERE:** Espressione condizionale usata per restringere i risultati di SELECT, UPDATE e DELETE. Si possono anche usare gli operatori logici AND e OR per restringere ancora di più il risultato della query. La clausola where può essere accompagnata da =, <, >, <=, >=, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF].

**Binding Parameters:** JPQL supporta due tipi di parametri, quelli positional e quelli named.

**PositionalParameters:** Sono strutturati da {?} seguiti da un intero. Quando laquery è eseguita, il parametro numerico deve essere sostituito e bisogna specificarlo.

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

**Named Parameters:** Sono strutturati da una stinga con prefisso (:). Quando la query è eseguita i parametri devono essere specificati e sotituiti.

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

**Subqueries:** Un sottoquery è una query all'interno di una condizione WHERE o HAVING

Esempio: SELECT c FROM Customer c WHERE c.age = (SELECT MIN(cust.age) FROM Customer cust)

**ORDER BY:** Ordina i valori di una query SELECT. Può essere ASC o Desc (crescente/decrescente)

Esempio: SELECT c FROM Customer c WHERE c.age >18 ORDER BY c.age DESC.

**HAVING BY:** definisce un filtro dopo che i risultati con le query sono stati raggruppati.

Esempio: SELECT c.address.country COUNT(c) FROM Customer c GROUP BY c.address.country HAVING c.address.country<>'UK'

**DELETE:** Per eliminare una lista di un entità usiamo delete

Esempio: DELETE FROM Customer c WHERE c.age < 18

**UPDATE:** Aggiorna le entità

Esempio: UPDATE Customer c SET c.firstName = 'TOO YOUNG' WHERE c.age <

**Query Dinamiche:** Per creare query dinamiche usando il metodo EntityManager.createQuery() prende una stringa come parametro che rappresenta una query JPQL. Il risultato della query è una lista che si ottinene tramite il metodo getResultList().

Esempio:
TypedQuery<Customer> query = em.createQuery(“SELECT c FROM Customer c”, Customer.class);
List<Customer> customers = query.getResultList();

**Con Prepared Statement:** query = em.createQuery(“SELECT c FROM Customer c WHERE c.firstName=:fname”);
query.setParameter(“fname”, “Betty”);
List<Customer> customers = query.getResultList();

oppure
query = em.createQuery(“SELECT c FROM Customer c WHERE c.firstName=?1”);
query.setParameter(1, “Betty”);
List<Customer> customers = query.getResultList();

**NamedQueries:** Sono query immutabili. Sono più effivienti di quelle dinamiche perché vengono tradotte direttamente da JPQL a SQL.

Esempio:
@NamedQuery(name=”findAll”, query=”SELECT c FROM Customer c”)
TypedQuery<Customer> query = em.createNamedQuery(“find-all”);

Native Queries: JPA permette di utilizzare caratteristiche specifiche di un database usando query native. Le query native permettono uno statement SQL come parametri e restituisce un istanza di Query per eseguire quello statement. Tuttavia non sono portabili.

Esempio:
@Entity
@NamedNativeQuery(name=”find-all”, query=”select * from t_customer”)
@Table(name=”t_customer”)
public class Customer{…}

**StoredProcedureQuery:** Le stored procedure sono differenti nel senso che sono attualmente memorizzate nel database stesso ed eseguite all'interno di esso. I vantaggi:

- Migliori performance a causa della precompilazione della stored-procedure.
- Mantiene le statistiche sul codice in modo da ottimizzarlo.
- Ridurre l'ammontare di dati passati sulla rete mantenendo il codice sul server.
- Alterare il codice senza replicarlo.
- Nascondere i dati grezzi permettendo solo alle stored-procedure.

In JPA usiamo l'annotazione @NamedStoredProcedureQuery.

Esempio:
```
@Entity
@NamedStoredProcedureQuery(name=”archieveOldbook”,
procedureName=”sp_archive_books”, parameters=
{
@StoredProcedureParameters(name=”archiveDate”, mode=IN,
type=Date.class),
@StoredProcedureParameters(name=”warehouse”, mode=IN,
type=String.class)
})

StoredProcedureQuery query =
em.createNamedStoredProcedureQuery(“archiveOldbook”);
query.setParameters(“archiveDate”, new Date());
query.setParameter(“maxBookArchived”, 1000);
query.execute();
```

**Entity LifeCicle:** Quando un'entità è istanziata è visto come un POJO regolare della JVM. Quando l'entità è persistita dall'entity manager sincronizzerà i valori dei suoi attributi con il database sottostante (tramite il metodo persist()). Nel momento in cui viene chiamato il metodo remove(), l'entità viene rimossa dal database ma continua a vivere nella JVM finché il garbager collector non lo rimuove. Quando viaggia attraverso la rete, l'entità deserializzata viene vista come detached  e deve essere reinserito tramite il metodo merge().

**Callback:** Ogni ciclo di vita dell'entità ha dei “pre” e “post” eventi che possono essere intercettati dall'entity manager per invocare metodi di business. Tutto questo tramite annotazioni.

- @Pre/PostPersist
- @Pre/PostUpdate
- @Pre/PostRemove
- @PostLoad

Esempio:
```
@PrePersist
@PreUpdate
private void validate(){
if(firstName==null) throw new IllegalArgumentException();
}
```

Le seguenti regole si applicano ai metodi di callback life cycle:

- I metodi possono essere public, protected, private ma non static e final.
- Un metodo può essere annotato con annotazioni multiple ma non dello stesso tipo(non puoi mettere due aanotazioni uguali alla stessa entità)
- Può lanciare eccezione controllate ma non quelle non controllate.
- Un metodo non può invocare operazioni di EntityManager o Query.
- Se un metodo è ereditato da una superclasse, verrà invocato prima quello della classe padre.

- Se eventi a cascata sono usati nelle relazioni, i metodi di callback saranno chiamati a cascata.

**Listeners:** Sono usati per estrarre la logica di business per separare la classe e condividerla con altre entità. Un EntityListener è un POJO nel quale si definiscono uno o più metodi di callback. Per registrare un listener si deve usare l'annotazione @EntityListener.

Esempio:
```
public class DataValidationListener{
@PrePersist
@PreUpdate
private void validate(Customer customer){some stuff}
}

@EntityListener(DataValidationListener.class)
@Entity
public class Customer{//constructor, getter, setter.}
```

Anchei listener possono essere definiti tramite xml nel tag <entity-listener>. Quando un evento è evocato, i listener sono eseguiti nel seguente ordine:

1. @EntityListeners per una data entità o superclasse nell'ordine dell'array.
2. EntityListener per la superclasse (la prima nella gerarchia).
3. EntityListener per l'entità.
4. Callback della superclasse (la prima nella gerarchia).
5. Callback entità.

## Capitolo 7

Il business layer è implementato nel Java Bean Enterprise (EJB). EJB sono dei componenti lato server che incapsulano la logica di business e si prendono cura della transazione e sicurezza. Hanno anche uno stack integrato per messaggiare,dependecy injection, component life-cycle, AOP(Programmazione orientata agli aspetti), interceptors. EJB riducono la complessità portando scalabilità e riusabilità. Il tutto avviene annotando un POJO che sarà distribuito nel container. Un EJB container è un ambiente a runtime che provvede a servizi, come gestione della transazione, controllo della concorrenza, sicurezza.

**Tipi di EJB:** Un session bean può essere di tre tipi:

- Stateless: Il session bean non contiene stato tra i metodi e qualunque istanza può essere usata per qualunque client. Sono usati per la gestione di task che pososno essere conclusi con un singolo metodo.
- Stateful: Il session bean contiene lo stato il quale può essere ottenuto attraverso i metodi per un singolo utente.
- Singleton: Un singolo session bean è condiviso tra i client e supporta accessi concorrenti. Il container si assicurerà che una sola istanza esista per l'intera applicazione.

Anche se i tre bean hanno le loro specifiche feature, hanno molto in comune. Prima di tutto hanno lo stesso modello programmativo. Message-driven beans(MDBS) sono usati per integrarsi con sistemi esterni ricevendo messaggi

asincroni.

**Process and Embedded Container:** Da quando sono stati inventati, gli EJB dovevano essere eseguiti in un container che deve essere eseguito al di sopra della JVM. Questo metodo è approrpiato per un ambiente di produzione dove il server è in esecuzione continuamente. Ma non va bene in fase di sviluppo dove il container deve essere inizializzato, distribuito e debuggato. Per risolvere questi problemi, alcuni application server usamo embedded container. L'idea è quella di eseguire EJB application all'interno dell'ambiente Java permettendo ai client di eseguirlo all'interno della JVM.

**Servizi offerti dal Container:**

- Comunicazione remota ai client: senza scrivere codice complesso, un EJB può invocare metodi remoti attraverso protocolli standard.
- Dependency Injection: Il container può iniettare varie risorse a un EJB.
- State Management: Per i bean stateful, il container gestisce il loro stato trasparentemente.
- Pooling: Per i bean stateless, il container crea un numero di istanze che possono essere condivise dai vari client.
- Component Life-cycle: Il container è responsabile per gestire il ciclo di vita di ogni componente.
- Messaging: Il container permette lo scambio di messaggi.
- Gestione delle Trnasazioni: Un EJB può creare annotazioni per informare il container sulla politica di transazione.
- Sicurezza: l'accesso alle classi o al metodo può essere specificata da EJB per rinforzare l'utente e il suo ruolo di autorizzazione.
- Supporto alla concorrenza: Tutti i tipi di EJB(a eccezione dei singleton sono thread-safe).
- Intercettori: i concetti trasversali possono essere inseriti in intercettori.
- Invocazioni di metodi asincroni: da EJB 3.1 si possono usare le chiamate asincrone.

Quando un client invoca un EJB non lavora direttamente con l'istanza di EJB ma con un Proxy ad essa. Un EJB container interagirà di solito con altri container: Servlet-container, application client, message-broken.

**EJB-Lite:** EJB definisce complesse tecnologie che sono meno usate oggi (ad esempio IIOP). Gli sviluppatori potrebbero non utilizzare mai queste tecnologie e per questa ragione, la specifica definisce un sottoinsieme delle API di EJB conosciuto come EJB lite.

Esempio:

```
@Stateless
public class BookEJB{
@PersitenceContext(unitName=”chapter07PU”)
private EntityManager em;

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

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

**Anatomia di un EJB:** Un EJB è fatto dai seguenti elementi:

- Una classe bean: Continene i metodi di business e può implementare uno o più interfacce. Deve essere o @Stateless, @Stateful, @Singleton.
- Interfacce di business: Contiene la dichiarazione dei metodi di business. Può avere interfacce locali o remote.

**Bean Class:** È una class Java standard che implementa la logica di business.

- Deve essere annotato con @Stateless, @Stateful, @Singleton o con un equivalente in xml.
- Deve implementare i metodi delle sue interfacce.
- Le classe deve essere definito come public e non final o non abstract.
- La classe deve avere un costruttore pubblico con costruttori senza argomenti.
- La classe non deve definire il metodo finalize().
- I metodi di business non possono essere final o static e i nomi non possono cominciare con ejb.
- L'argomento e il valore di ritorno di un metodo remoto deve essere RMI Types.

**Remote, Local and No Interface views:** A seconda di dove il client invoca invoca un session bean, il bean dovrà implementare interfacce locali o remote. Nel caso di metodi remoti, bisogna invocarli tramite RMI. Possiamo usare delle annotazioni per definire se un interfaccia è locale o remota.

@Remote: I parametri dei metodi sono passati per valore e necessitano di essere serializzati.
@Local: I prametri devono essere passati per referenza dal client al bean.

**Portable JNDI Name:** Le API di JNDI hanno un implementazione specifica. Quando un EJB è distribuito, il nome dell'EJB nella directory di servizio è non portabile. Da EJB 3.1 ha dei nomi portabili cosicchè ogni volta che un bean è distribuito al container è automaticamente legato a un JNDI portatile.

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

scope: definisce una serie di standard namespace: global, app, module, comp.
App-name: richiesto solo se il session bean è impacchettato all'interno di un file ear o war.
Module-name: il nome del module nel quale il session bean è impacchettato.
Fully-qualified-interface-name: è il nome qualificato di ogni interfaccia definita.

Esempio:

```
package org.agoncal.book.javaee7
@Stateless
```

```
@Remote(ItemRemote.class)
@LocalBean
public class ItemEJB implements ItemRemote{}
```

JNDI = java:global/cdbookstore/ItemEJB!org.agoncal.book.javaee7.ItemRemote.

**Bean Stateless:** Sono i più popolari session bean. Sono semplici, potenti ed efficienti. Con stateless indichiamo che un task deve essere completato in una singola chiamata a metodo. Gli stateless session bean seguono l'architettura stateless e sono i componenti più efficientiperchè possono essere messi assieme e condivisi da diversi client. Visto che le istanze non hanno uno stato, tutte le istanze sono uguali. Quando un client invoca un metodo su uno stateless bean, il container prende un'istanza e la assegna al client. Quando il client termina la richiesta, l'istanza ritorna e può essere riutilizzata.

**Bean Stateful:** Preservano lo stato. Sono utili per i task che devono essere fatti in diversi step, ognuno dei quali risiede sullo stato mantenuto nei precedenti step. Quando un client invoca un bean stateful, il container EJB necessita di provvedere alla stessa istanza per ogni sottoseguente invocazione al metodo. Il tutto avviene a scapito di memoria perché ad ogni client, viene assegnato un bean stateful. Per evitare uno spreco di memoria, il container pulisce temporaneamente gli stateful bean dalla memoria prima che la prossima richiesta del client li riporti indietro. Questa tecnica è chiamata passivation and attivaction. Passivation è il processo rimuovere un'istanza dalla memoria e salvarlo in locazioni persistenti. Actiovation è il processo inverso di ristorare lo stato e applicarlo all'istanza. Queste cose vengono fatte automaticamente dal container, le uniche cose di cui lo sviluppatore si deve occupare è liberare alcune risorse(database connection, JMS factories).

**Bean Singleton:** È un session bean che è istanziato una volta nell'applicazione. Un uso comune è una cache di sistema. In un ambiente gestito innanzitutto si rende il costruttore privato. In Java EE basta aggiungere @Singleton.

**Packaging:** La maggior parte dei componenti necessita di essere impacchettata prima di essere distribuita in un container.Una volta impacchettati in jar, si possono distribuire direttamente nel container. Un'altra opzione è di mettere il jar file in un ear. Da EJB 3.1 possono essere impacchettati in un war file.

**Deploying an EJB:** I session bean sono delle componenti gestite, e questo è un vantaggio perché il container tratta ogni sorta di servizi lasciando concentrare il programmatore sul codice di business. Gli svantaggi sono che sarà sempre necessario eseguire un EJB nel container. Prima questi container venivano eseguiti in processi separati ma da EJB 3.1, si possono usare gli embedded EJB che vengono eseguiti dall'ambiente Java. Gli EJB embeddable lavorano solo con il sottoinsieme EJB Lite.

Esempio:

```
main(String[] args){
Map<String, Object> properties = new HashMap<>();
properties.put(EJBContainer.MODULES, new File(“target/classes”));
```

```
try(EJBContainer ec = EJBContainer.createEJBContainer(properties)){
Context ctx = ec.getContext();
Book book = new Book();
ItemEJB item = (ItemEJB) ctx.lookup(“java:global/classes/ItemEJB”);
item.create(book);
}
}
```

Invoking Enterprise Java Beans: Per invocarlo nel client usiamo l'annotazione (@EJB o @Inject)

Con EJB:
```
@Stateless
public class ItemEJB{...}

@EJB
ItemEJB item;
```

Con Inject:
```
@Stateless
public class ItemEJB{...}

@Inject
ItemEJB item;
```

Con JNDI: I session bean possono essere anche cercati usando JNDI. Bisogna usare l'InitialContext di JNDI e cercare un EJB usando il nome definito dall'EJB.

```
Context ctx = new InitialContext();
ItemRemote item = (ItemRemote) ctx.lookup(“java:global/cdbookstore/ItemEJB!
org.agonical.book.javaee7.ItemRemote”);
```

## Capitolo 10

**Understanding Web Pages:** Per creare pagine visibili nel browser dobbiamo usare: HTML, CSS, Javascript, XHTML.

**HTML:** usare i tag per strutturare la pagina.

**XHTML:** Deriva da html ma è riformulato in un XML ristretto. Questo significa che un documento XHTML è un documento XML che segue uno schema preciso. In contrasto con HTML ha il vantaggio di provvedere a una validazione dei documenti usando tools XML. Per farlo si deve inserire all'interno della pagina le DTD(Document Type Definition) che specifica con quale tool deve essere codificatoa la pagina. Un DTD può essere di tre tipi: Transitional, Frameset, Strict.

**CSS:** linguaggio di stile usato per descrivere la presentazione dei documenti HTML e XHTML.

**DOM:** È una specifica della W3C per accedere e modificare il contenuto e la

struttura dei documenti HTML e XHTML.

**JAVASCRIPT:** linguaggio di scripting per lo sviluppo lato client.

**Understanding JSF:** Sono applicazioni web che intercettano richieste HTTP attraverso Faces Servlet e produce HTML. L'architettura permette di attaccare qualunque pagina in linguaggio dichiarativo, reindirizzare per servizi differenti e creare pagine usando eventi, listener e componenti.

- Faces Servlet: è la Servlet principale per l'applicazione e può essere opzionalmente configurata nel file-descriptor “faces_config.xml”.
- Pagine e Componenti: JSF permette multiple PDL ma Facelets è quello raccomandato.
- Renderers: Sono responsabili per mostrare un componente e tradurre gli input dell'utente nei valori delle proprietà dei componenti.
- Converters: Converte il valore di un componente da valori di markup.
- Validator: Responsabili di assicurare che il valore inserito dall'utente sia valido.
- Backing Beans and navigation: la logica di business è fatta in backing beans i quali controllano anche la navigazione tra le pagine.
- Ajax support: Supporto alle richieste Ajax.
- Expression language: È usato nelle pagine JSF per legare variabili e azioni tra i componenti e i backin bean.

**FacesServlet:** JSF usa il pattern MVC: disacoppiare la vista(pagine) e il modello(i dati da essere mostrati nella view). Il controller gestisce le azioni che potrebbero portare cambiamenti nel modello e aggiornare la view. In JSF questo controller è una Servlet chiamata FacesServlet. Essa gestisce il ciclo di vita per le richieste web.

**Pages and Component:** Le componenti in JSF sono le Facelets che sono pagine costruite come un albero di componenti che provvedono a specifiche funzionalità per interagire con l'utente.

**Facelets:** Quando JSF è stato creato, l'intenzione era di riusare JSP. JSP è il linguaggio della pagina e JSF è uno strato al di sopra di esso. Facelets provvedono a un'alternativa XHTML per le pagine in un'applicazione JSF.

**Renderers:** JSF supporta due modelli programmativi: la diretta implementazione e l'implementazione per delegazione. Nel primo caso, i componenti devono essere decodificati e codificati in una rappresentazione grafica. Nel secondo caso sono delegate ad un renderer. JSF include supporto a standard renderer kit.

**Converters e Validators:** Una volta che la pagina è reindirizzata l'utente può interagire con essa inserendo dati. Non essendoci alcun tipo di restrizione, un renderer non può conoscere in anticipo come mostrare l'oggetto. Qui è dove i converter entrano in gioco: traducono un oggetto in una stringa e viceversa. Alcune volte i dati devono essere validati prima di essere processati al ritorno. I validatori sono responsabili per assicurare che il valore inserito dall'utente è accettabile.

**Backing Beans and Navigation:** Un backing bean è una classe Java specializzata che sincronizza valori con componenti, processa la logica di business e gestisce la navigazione tra pagine. Si può associare un componente con uno specifico backing bean o azione usando EL (Expression Language).

Esempio:
<h:inputText value=”#{bookController.book.isbn}”/>
<h:commandButton value=”Create”
action=”#{bookController.doCreateBook}”/>

La prima linea di codice aggancia il valore dell'input testuale direttamente alla proprietà book.isbn di un backing bean chiamato BookController

@Named //usato affinché la classe sia usata all'intenro di EL
@RequestScoped
public class BookController{
@Inject
private BookEJB bookEJB;
private Book book = new Book();

public String doCreateBook(){
book = bookEJB.createBook(book);
return “listBooks.xhtml”
}
}

**Expression Language:** Lega i bean alle pagine JSF. Si possono usare EL statement per stampare il valore di variabili, accesso agli attributi di un oggetto in una pagina. La sintassi è:#{expr}. Può usare gli operatori aritmetici, logici, relazionali.

**Ajax Support:** Da JSF 2.0 il supporto ad Ajax è stato aggiunto con il tag <f:ajax>.

## Capitolo 13

Message Oriented Middleware(MOM) è un software che permette lo scambio di messaggi asincroni tra sistemi eterogenei ovvero il produttore e il consumatore non devono essere per forza disponibili allo stesso tempo.

**Understanding Messaging:** In Java EE le API che trattano questi concetti è chiamata JMS. Ha una serie di interfacce e classi che permettono di creare un messaggio, mandarlo e riceverlo. Ad un livello più alto un'architettura a

messaggi consiste dei seguenti componenti:

- Un fornitore: gestisce il buffering e lo smistamento dei messaggi.
- Clients: è qualunque applicazione java chi produce o consuma messaggi.
- Messaggi: Sono gli oggetti che i client mandano o ricevono dal fornitore.
- Object Administrated: Un message broker deve provvedere ad oggetti amministrati al client attraverso JNDI o injection.

Il fornitore di messaggi permette comunicazione asincrona provvedendo a una destinazione dove i messaggi possono essere tenuti finché non possono essere consegnati ai client. Ci sono due tipi di destinazione:

1. **P2P:** In questo modello la destinazione usata per mantenere messaggi è chiamata coda. Il client mette il messaggio nella coda ed una volta ricevuto, viene eliminato dalla coda.
2. **(Pub-sub):** La destinazione è chiamata topic. Un client pubblica un messaggio a un topic  e tutti i sottoscritti ai topic ricevono il messaggio.

**Point to Point:** Nel modello P2P, un messaggio viaggia da un singolo fornitore a un singolo consumatore. Il modello è costruito attorno al concetto di coda, mandanti e riceventi(del messaggio). Una coda contiene i messaggi mandati dal mandante finché non sono consumati e il mandante e il ricevente non hanno dipendenze di tempo. Ogni messaggio è mandato ad uno specifica coda e il ricevente, estrae il messaggio dalla coda. Le code conservano tutti i messaggi mandati fino a quando non sono consumati o fino a che non muoiono. L'ordine non è definito per l'arrivo dei messaggi.

**Publish-Subscribe:** In questo modello un singolo messaggio è mandato da un singolo fornitore a diversi potenziali consumatori. Il modello è costruito sui concetti di topic, publisher, subscriber. I consumatori sono chiamati anche subscribers perché hanno prima necessità di iscriversi al topic. Il topic mantiene i messaggi finché non sono distribuiti a tutti i sottoscritti. A differenza del modello P2P c'è un tempo di dipendenza tra publisher e subscriber: i subscriber non ricevono messaggi mandati prima della loro sottoscrizione, e se questi ultimi sono inattivi per un certo periodo non riceveranno i messaggi precedenti quando ritorneranno attivi. Subscriber multipli possono consumare lo stesso messaggio. Il modello può essere utilizzato per applicazioni broadcast.

**Administrated Object:** Sono oggetti che sono configurati amministrativamente (opposto a programmativamente). Il fornitore permette a questi oggetti, di essere configurati e li rende disponibili nel namespace di JNDI. Gli oggetti amministrati sono creati un'unica volta e sono di due tipi:

- Connection factories: Usato dai client per creare una connessione verso la destinazione.
- Destinations: I punti di distribuzione che riceve, mantiene, e distribuisce messaggi. Le destinazioni possono essere topic o code.

**Message-Driven Beans:** Sono consumers di messaggi asincroni che sono eseguiti all'interno di un EJB container. Gli MDB sono stateless, vale a dire che il container EJB può avere numerose istanze, eseguendole concorrentemente, per processare provenienti da vari producers. Le applicazioni client non possono accedere direttamente a MDB, l'unico modo per comunicare è mandare un

messaggio a destinazione dove MDB è in ascolto. In generale MDB è in ascolto di una destinazione e quando il messaggio arriva, lo consuma e lo processa. Essendo stateless MDB non mantiene lo stato attraverso invocazioni separate da un messaggio ricevuto al prossimo. MDB risponde ai messaggi ricevuti dal container, dove i session bean stateless rispondono alle richieste del client attraverso un'appropriata interfaccia.

**History:** JMS è stato pubblicato nel 1992. È stato creato dai maggiori venditori di middleware per portare la capacità dei messaggi al Java. Prima di questa data, le compagnie non avevano modi semplici per collegare differenti applicazioni e dovevano scrivere diversi adattatori software per i sistemi per tradurre i dati da sorgenti delle applicazioni in un formato che il destinatario potesse capire.

**JMS 2.0:**

1. Gli oggetti col metodo close(Connection, Session) implementano l'interfaccia AutoCloseable permettendogli di essere usata in Java EE 7.
2. API semplificata.
3. Aggiunto metodo getBody() per estrarre il corpo del messaggio.
4. Nuove eccezioni.
5. Nuovi metodi per implementare messagistica asincrona.

**EJB 3.0:** introduce chiamate asincrone all'interno di session bean usando @Asynchronus.

**Java Messaging API:** Permette alle applicazioni di creare, mandare, ricevere e leggere messaggi asincronamente. Definisce un comune insieme di interfacce e classi che permettono ai programmi di comunicare con altri fornitori di messaggi. JMS ha specifiche API a seconda se il modello sia pub-sub o P2P.

**Simplified API:** 3 interfacce fondamentali(JMSContext, JMSProducer, JMSConsumer). JMSRuntime Excepion che è un eccezione controllata, il codice è più semplice da leggere e scrivere.

**JMSContext:** è l'interfaccia principale e crea una connessione a un fornitore JMS un contesto single-thread per mandare e ricevere messaggi.

**JMSProducer:** È usato per mandare messaggi per conto di JMSContext. È creato dal metodo di Context createProducer(). Provvede anche ad opzioni per settare proprietà dei messaggi, header.

**JMSConsumer:** È usato per ricevere messaggi da una queue o topic. È creato dal metodo di Context createConsumer(). Un cliente può ricevere messaggi sincronamente e asincronamente. Per una consegna asincrona il client può registrare un MessageListenerObject con JMSConsumer.

**Writing Message Producers:** A seconda se l'applicazione gira o meno all'interno di un container, bisogna apportare delle modifiche.

Produrre un messaggio fuori dal container:

1. Ottenere una connessione e una cosa usando JNDI lookup.
2. Creare un JMSContext usando la connectio factory.

3. Creare un JMSProducer usando createProducer().
4. Manada un messaggio usando il metodo send().

Esempio: public class Producer{
main(){
try{
Context ctx = new InitialContext();
ConnectionFactory connectionFactory = (ConnectionFactory)
ctx.lookup(“jms/javaee7/ConnectionFactory”);
Destination queue = (Destination) ctx.lookup(“jms/javaee7/Queue”);
try(JMSContext context = connectionFactory.createContext()){
context.creteProducer().send(queue, “Text Message”);
}catch(NamingException e){
e.printStackTrace();
}
}
}
}

**Produrre un messaggio all'interno di un container con CDI:** Quando è eseguito all'interno di un container, si può iniettare con @Inject

Esempio: public class Producer{
@Inject
@JMSConnectionFactory(“jms/javaee7/ConnectionFactory”)
private JMSContext ctx;
@Resource(lookup=”jms/javaee7/Queue”)
private Queue queue;

public void sendMessage(){
ctx.createProducer().send(“TextMessage”);
}
}

**Writing Message Consumers:** Il client può consumare messaggi in due modi:

- Sincrona: Il ricevitore prende il messaggio esplicitamente dalla destinazione tramite il metodo receive().
- Asincronamente: il ricevitore decide di registrare un evento tale che è attivato quando il messaggio arriva. Deve implementare l'interfaccia MessageListener e quando il messaggio arriva, il fornitore lo spedisce chiamando il metodo onMessage().

**Consegna Sincrona:** Necessita di inizializzare un JMSContext un loop che aspetta finché non arrivano nuovi messaggi e richiedere i messaggi che arrivano usando dei suoi metodi di receive(). Ci sono diverse variazioni di receive() che permettono ai client di prendere o aspettare il messaggio successivo.

Esempio: public class Customer{
main(){
//creazione del context, factory e coda come il codice di sopra.

```
try(JMSContext context = connectionFactory.createContext()){
while(true){
String message = context.createConsumer(queue).receiveBody(String.class);
}
}catch(NamingException e){e.printStackTrace();}
}
}
```

**Consegna Asincrona:** È basata sul gestore degli eventi. Un client può registrare un oggetto che implementa l'intefaccia MessageListener. Un messagelistener è un oggetto che agisce come un asincrono gestore di eventi per i messaggi. Come un messaggio arriva, il fornitore lo consegna chiamando il metodo onMessage() che prende come argomento il messaggio.

Esempio: public class Listenerv implements MessageListener{
main(){
//creazione del context, factory e coda come il codice di sopra.
try(JMSContext context = connectionFactory.createContext()){
context.createConsumer(queue).setMessageListener(new Listener());
}catch(NamingException e){e.printStackTrace();}
}

public void onMessage(Message message){
System.out.println(“message arrived:”+message.getBody(String.class));
}
}

**Meccanismi di Realizzabilità:** JMS definisce una serie di livelli di realizzabilità per assicurarsi che i messaggi siano consegnati anche se il fornitore è sotto sforzo o crasha.

**Filtering messages:** Alcune applicazioni necessitano di filtrare i messaggi che ricevono. I messaggi sono composti da tre parti: header, properties, body. L'header continene un fissato numero di campi, le proprietà sono un insieme di coppie chiave-valore che l'applicazione può usare per settare valori. Il filtro può essere fatto su questi due campi ed è il consumatore a specificarlo. I selettori dei messaggi assegnano il lavoro di filtraggio a JMS. Un selettore è una stringa che contiene un'espressione.

Esempio:
context.createConsumer(queue, JMSProperty < 6).receive();
context.createConsumer(queue, JMSProperty < 6 AND orderAmount <200).receive();

Settare valori:
//settare proprietà
context.createTextMessage().setIntProperty(“orderAmount”, 1530);
//settare header
context.createJMSPriority(5);

Settare tempo di vita del messaggio: si usa il metodo setTimeToLine() che

prende un numero in millisecondi.

Esempio:
context.createProducer().setTimeToLive(1000).send(queue, message);

Specify Persistence of messages: JMS supporta due tipi di consegna del messaggio: persistente e non persistente. La prima assicura che il messaggio sia consegnato solo una alla volta al consumatore, mentre il secondo più volte. La prima è più realizzabile ma aumenta i costi di performance. La modalità di consegna può essere specificata usando il metodo setDeliveryMode()

Esempio:
context.createProducer().setDeliveryMode(DeliveryMode.NON_PERSISTENT).send(queue, message);

**Control concurrency:** Serve per ricevere la consapevolezza che il messaggio sia stato ricevuto. Nelle sessioni transazionali, la concorrenza accade automaticamente altrimenti deve essere specificata.

- AUTO_KNOWLEDGE: La sessione automaticamente conosce se il messaggio è stato recepito.
- CLIENT_KNOWLEDGE: Un client conosce un messaggio esplicitamente chiamando il metodo ackowledge().
- DUPS_OK_ACKNOWLEDGE: Conoscenza pigra della consegna del messaggio. È simile alla consegna di messaggi duplicati.

Esempio:
@Inject
@JMSessionMode(JMSContext.AUTO_KNOWLEDGE)
private  JMSContext context;
context.createProducer().send(queue, message);
message.ackwoledge();

**Creating durable consumers:** Lo svantaggio di usare il metodo pub-sub, è che il consumer del messaggio deve essere in esecuzione quando i messaggi sono mandati al topic. Usando i durable consumer JMS provvede ad un modo di mantenere i messaggi nel topic finché non tutti i subscriber lo ricevono. Con questa tecnica un consumer può essere offline per un certo periodo, quando si riconnette, riceverà il messaggio che è arrivato durante la sua disconnessione.

Esempio:
context.createDurableConsumer(topic,
“javaee7DurableSubscription”).receive();

**Set priorities:** Si possono usare le priorità dei messaggi per istruire JMS fornitore,  a consegnaretry(JMSContext context = connectionFactory.createContext()) messaggi urgenti prima. Il range della priorità va da 0 a 9.

Esempio:

context.createProducer().setPriority(2).send(queue, message);

**Writing Message-Driven Beans:** MDB provvede a un modello standard per messagiare asincronamente per applicazioni enterprise in esecuzione su EJB container. Un MDB è un consumer asincrono che è invocato dal container come risultato dell'arrivo di un messaggio.  Il container è in ascolto di una destinazione e delega la chiamata a MDB finché non arriva il messaggio. Conviene usarlo perché essendo un container gestisce tutto lui, multithreading, sicurezza, transazioni.

Esempio:
@MessageDriver(mappedName=”jms/javaee7/Topic”)
public class BillingMDB implements MessageListener{

public void onMessage(Message message){
System.out.println(“Message received: ”+message.getBody(String.class));
}
}

**Anatomia di un MDB:** Sono differenti dai session bean perché non implementano interfacce remote o locali ma l'interfaccia MessageListener. Mantiene tutte le caratteristiche di un session bean (ciclo di vita, annotazioni, callback, interceptor, injection). I requisiti sono:

- La classe deve essere annotata con @MessageDriver.
- Deve implementare l'interfaccia MessageListener.
- La classe deve essere public e non abstract o final.
- La classe deve avere un costruttore senza argomenti.
- La classe non deve definire il metodo finalize().

@MessageDriven: @Target(TYPE)@Retention(RUNTIME)
public @interface MessageDriven{
String name()//nome dell'MDB
ClassMessageListenerInterface default.Object.class//specifica quale message listener MDB implementa.
ActivationConfigProperty[] activationConfig() default{}//specifica una configurazione e prende un array di annotazione @ActivationConfigProperty
String mappedName()//indica il nome JNDI della destinazione su cui mettersi in ascolto
String description();//descrizione
}

**@ActivationConfigProperty:** JMS permette la configurazione di certe proprietà(selector, ackwoledge mode, durable subscribers e così via). Queste proprietà possono essere settate, usando questa annotazione. Si passano come parametri all'annotazione @MessageDriven e consiste in una coppia chiave-valore.

Esempio:@MessageDriven(mappedName=”jms/javaee7/topic”, activationConfig={@ActivationConfigProperty(propertyName=”ackowledgeMode”, propertyValue=”Auto-knowledge”)})

**Dependencies Injection:** MDB può usare la dependency injection. MDB può essere anche iniettato usando @Resource.

Esempio: @Resource private MessageDrivenContext context;

**MDB Context:** MessageDrivenContext provvede accesso al contesto di esecuzione che il container provvede per un'istanza di MDB. Il container passa l'interfaccia MessageDrivenContext a questa istanza che rimane associata per il tempo di vita dell'MDB. Questo  dà a MDB la possibilità di annullare una transazione e ottenere l'utente principale ed altro.

**LifeCycle and Callback Annotation:** Il ciclo di vita di un MDB è identico a quello di un bean stateless. Il container crea prima un'istanza di MDB e se applicabile, inietta le necessare risorse tramite annoazioni o deployment descriptor. Il container chiama poi il metodo di callback @PostConstruct. Dopo do questo MDB è pronto e aspetta di consumare qualunque messaggio incombente.

**MDB as Consumer:** MDB è strutturato per essere asincrono. Può essere anche sincrono ma non è raccomandato perché consuma molte risorse del server.

**MDB as Producer:** un MDB può essere producer ma si devono utilizzare alcune API di JMS

Esempio:
```
public class BillingMDB implements MessageListener{
@Inject
@JMSConnectionFactory(“jms/javaee7/ConnectionFactory”)
@JMSSessionMode(JMSContext.AUTO_ACKNOWLEDGE)
private JMSContext contex;
@Resource(lookup=”jms/javaee7/Queue”)
private Destination queue;

public void onMessage(Message message){
System.out.println(“Message received”+message.getBody(String.class));
sendPrintingMessage();
}

private void sendPrintingMessage()throws JMSException{
context.createProducer().send(queue, “Message has been received and
resent”);
}
}
```

**Transactions:** MDB può essere BMT(beans messaged transaction) o CMT (container, managed transaction). La transaction vale anche per i messaggi. Il container inizierà la transazione prima del metodo onMessage() e farà il commit della transazione quando il metodo restituisce.

MDB può usare le seguenti annotazioni sui metodi di business:

- REQUIRED: Se MDB invoca altri bean, il container passa il contesto di

transazione con l'invocazione.
- NON REQUIRED: Se MDB invoca aktri bean, il container non passa il contesto di transazione con l'invocazione.

**Handling Exceptions:** JMS definisce delle eccezioni che derivano da JMSException. JMS definisce due tipi di eccezioni:
- Application Exception: Eccezione controllata che estendono Exception e non causano roll-back.
- System Exception: Eccezione non controllata che estendono RunTimeException e causano roll-back.

## Capitolo 14

**SOAP:** servizi web più architettura orientata ai servizi. Le SOAP sono servizi web debolmente accoppiati perché il client non conosce i dettagli dell'implementazione. Il consumer è capace di invocare un servizio web SOAP usando un'interfaccia che descrive i metodi di business disponibili. Il consumer manda una richiesta in XML al servizio web e riceverà una risposta in XML(opzionalmente).

**Understanding SOAP:** SOAP è costituito da un tipo di logica di business esposta tramite un servizio ad un client. Tuttavia a differenza di EJB provvede ad un interfaccia debolmente accoppiata usando XML. SOAP standard specifica che l'interfaccia debba definire il formato dei messaggi e server registry. Le tecnologie che usa:
- XML: linguaggio di base.
- WSDL: definisce il protocollo, interfacce, tipo di messaggio, e interazione tra il consumer e provider.
- SOAP: codifica dei messaggi basata su XML.
- HTTP: per il trasporto dei messaggi.
- UDDI: è un meccanismo di server registry utilizzando per memorizzare e categorizzare i servizi delle interfacce SOAP(WSDL).

**JAX-WS:** é un insieme di API e annotazione che permette di costruire e consumare servizi web con java. Si occupa lui di generare e parsare un messaggio.

**Web Services:** Definisce il modello di programmazione e il comportamento a run-time dei servizi web.

**WS-Metadata:** Provvede ad annotazioni per facilitare la definizione e la distribuzione di servizi web.

**Writing SOAP Web Services:** Usiamo un approccio “dall'alto verso il basso” conosciuto anche come primo contratto. Questo approccio comincia con il WSDL ovvero definendo operazioni, messaggi, ed altro. Quando il consumer e il provider sono d'accordo. Si può implementare la classe Java basata su quel contratto. Possiamo usare un approccio “bottom-up” ovvero creano prima la classe e poi  il WSDL. Scelto un approccio, usiamo JAX-WS per aggiustare il codice.

Esempio:

```
@WebServices
public class CardValidator{
public boolean validate(CreditCard creditCard){
Character lastDigit =
creditCard.getNumber().charAt(creditCard.getNumber().lenght()-1);
if(Integer.parseInt(lastDigit.toString())%2!=0)
return true;
else
return false;
}
}
```

Essendo che i dati per essere scambiati devono fare uso di XML la classe CreditCard viene annotata con Jax B.

```
@XmlRootElement
public class CreditCard{
@XmlAttribute(required=true)
private String number;
@XmlAttribute(name=”expiry-date”, required=true)
private String expiryDate;
@XmlAttribute(name=”control_number”, required=true)
private Integer controlNumber;
}
```

**Anatomia di Web Service SOAP:**

- La classe deve essere annotata con @Webservice
- La classe deve può implementare zero o più interfacce che devono essere annotate con @WebService
- La classe deve essere pubblica e non abstract o final.
- La classe deve avere un costruttore pubblico di default.
- La classe non deve definire il metodo finalize().
- Per cambiare un SOAP in un EJB la classe deve essere annotata @Stateless, @Singleton.
- Un servizio deve essere stateless e non deve salvare lo stato del client durante chiamate a metodi.

**SOAP Web Services EndPoints:** JAX-WS permette a classi Java o EJB di essere esposti come servizi web. Usando EJB si beneficia delle sue caratteristiche.

**WSDL Mapping:** A livello di servizi, i sistemi sono definiti in termini di oggetti, interfacce, metodi. Una traduzione da oggetti Java a WSDL è necessaria. JAXB usa annotazioni per mappare le classi java in WSDL e determinare come serializzare un'invocazione  di un metodo a una richiesta SOAP e deserializzare una risposta in un istanza del tipo di ritorno di un metodo. JAX-WS definisce due differenti tipi di annotazioni:

- WSDL mapping annotations: Permettono di cambiare il mapping WSDL/Java. @WebMethod, @WebResult, @WebParam, @OneWay sono usate sul servizio web per personaizzare la firma di metodi esposti.

- SOAP binding annotations: Queste annotazioni permettono di personalizzare il SOAP binding.

**@WebServices:** marca una classe Java a un'interfaccia come servizio web.

Esempio:
@WebServices
public class CardValidator{...}

Quando si usa @WebService, tutti i metodi pubblici del servizio web sono esposti ad eccezione di quelli che usano l'annotazione @WebMethd.

**@WebMethod:** Esclude un metodo dal WSDL o personalizza alcuni elementi del mapping dei metodi.

Esempio:
@WebMethod(operationName=”ValidateCreditCard”)
public boolean validate(CreditCard creditCard){//logica}

@WebMethod(exclude=true)
public void validate(Long creditCardNumber){//logica}

**@Webresult:** Controlla il nome generato del valore del messaggio restituito nel WSDL.

Esempio:
@WebResult(name=”IsValid”)
public boolean validate(CreditCard creditCard){//logica}

**@WebParam:** Personalizza i parametri per un metodo web. Le sue API permettono di cambiare il nome dei parametri nel WSDL, il namespace e il tipo. Tipi validi sono IN, OUT o INOUT.

Esempio:
@WebService
public class CardValidator{
public boolean validate(@WebParam(name=”CreditCard”, mode=IN)CreditCard creditCard){//logica}
}

**@OneWay:** è usata sui metodi che non hanno un valore di ritorno. L'annotazione può essere vista come un interfaccia di markup che informa il container che l'invocazione può essere ottimizzata.

**@SOAP Binding:** Descrive come il servizio web è legato a un protocollo di messaggio. Ci sono due stili di SOAP definiti in WSDL: RPC e document.

1. Document: Il messaggio soap contiene il documento. È mandato come un unico documento nel body senza regole di formattazione.
2. RPC: Il messaggio soap continene i parametri e i valori di ritorno. Il body contiene un elemento con il nome del metodo o procedure remote da invocare.

Un SOAP Binding deve poi scegliere due formati di serializzazione/deserializzazione:

- Literal: I dati sono serializzati in accordi ad uno schema XML.
- Encoded: SOAP encodind specifica come gli oggetti, le strutture, gli array devono essere serializzati.

Ci sono 4 stili: Document/Literal, Document/Encoded, RPC/Literal, RPC/Encoded.

Esempio:

```
@WebService
@SOAPBinding(style=RPC, use=LITERAL)
public class CardValidator{}
```

**Handling Exceptions:** JAX-WS automaticamente converte le eccezioni Java in SOAP fault che sono restituite al client. Si usa @WebFault(nome=Eccezione).

Esempio:

```
@WebFault(name="CardValidatorFault")
public class CardValidatorException extends Exception{//logica}

@WebService
public class CardValidator{
public boolean validate(CreditCard creditCard){
//codice che verifica se ci sono errori
SOAPFactory soapFactory = SOAPFactory.newIstance();
SOAPFactory fault = soapFactory.createFault("the credit card number is not
valid"+new Qname("ValidationFault"));
throw new CardValidatorException(fault);
}
}
```

**Life Cycle and Callback:** Il ciclo di vita è simile ad un bean. L'unica differenza è che EJB endpoint non possono usare interceptor.

**Web Service Context:** Si può accedere all'ambiente e si può accedere iniettando una referenza di WebServiceContext con l'annotazione @Resource. All'interno del contesto, il web service può ottenere informazioni a runtime come implementazioni di classi endpoint, contesto del messaggio e informazioni di sicurezza relativi alla richiesta da da servire.

**Deployment Descriptor:** anche i metadati delle SOAP possono essere definiti usando xml nella cartella WEB-INF.

**Packaging:** può essere impacchettato in un EJB o .war file. Nei war file, si può usare servlet o EJBLite. Gli EJB possono essere solo stateless o singleton. Lo sviluppatore è responsabile di impacchettare:

- Il bean che implementa il servizio e le classi dipendenti.
- Le interfacce del servizio.
- WSDL file.

- Artefatti generati per le richieste SOAP.
- Deployment descriptor (opzionale).

**Publishing SOAP Webservice:** Una volta impacchettati in un war/jar file, basta distribuirlo in un container come GlassFish o Jboss. Nel caso si voglia utilizzare il semplice SE di Java, basta usare il metodo publish().

Esempio:
@WebService
public class CardValidator{
boolean validate(){//logica}
}

main(){
EndPoint.publish("http://localhost:8080/cardValidator", new CardValidator());
}

**Invoke a Service:** è simile a invocare un oggetto distribuito in RMI, la differenza è che il servizio web può essere scritto in qualunque linguaggio usando un proxy. Quando un metodo sul proxy è invocato converte i parametri del metodo in un messaggio SOAP e lo manda al servizio remoto usando HTTP. Per ottenere la risposta, la risposta SOAP è convertita in un'istanza del tipo restituito. Prima di completare il consumer client necessita di generare il SEI per ottenere il proxy da chiamare nel codice.

**Anatomia di un SOAP Consumer:** Può essere di qualunque tipo di codice Java eseguito su di una JVM in qualunque componente Java EE eseguito in una container. Se eseguito in un container può ottenere l'istanza del proxy.

**Invocare Programmaticamente:** se il consumer è fuori dal container.

Public class WebServiceConsumer{
main(){
CreditCard creditCard = new CreditCard();
//setter
CardValidator cardValidator = new
CardValidatorService().getCardValidatorPort();
cardValidator.validate()//metodo che chiama il proxy che invocherà //il servizio remoto, creerà la SOAP request ecc...
}
}

**Invocare con Injection:** se il consumer è eseguito nel container si può usare la injection per ottenere le referenze del SOAP service proxy usando @WebServiceRef.

Esempio:
public class WebServiceConsumer{
@WebServiceRef
private static CardValidatorService cardValidatorService;

```
main(){
CreditCard creditCard = new creditCard();
CardValidator cardValidator = cardValidatorService.getCardValidatorPort();
cardValidator.validate(creditCard);
}
}
```

**Invocare con CDI:**

```
public class WebServicesProducer{
@Produces
@WebserviceRef
private CardValidatorService cardValidatorService;
}

@Stateless
public class EJB ConsumerWithCDI{
@Inject
private CardValidatorService cardValidatorService;
public boolean validate(CreditCard creditCard){
CardValidator cardValidator = cardValidatorService.getCardValidatorPort();
return cardValidator.validate(creditCard);
}
}
```