



Università di Bologna  
CdS Laurea Magistrale in Ingegneria Informatica  
I Ciclo - A.A. 2013/2014

## 05 – Enterprise Java Beans 3.0 e Servizi di Sistema (Pooling, Transazioni, ...)

Docente: Paolo Bellavista  
[paolo.bellavista@unibo.it](mailto:paolo.bellavista@unibo.it)

<http://lia.deis.unibo.it/Courses/sd1314-info/>  
<http://lia.deis.unibo.it/Staff/PaoloBellavista/>



## Da EJB2.1 a EJB3.0: Ease of Development!

- ❑ EJB2.1 è stato considerato dagli sviluppatori ***molto potente ma anche complesso da utilizzare***
  - Modello di programmazione non sempre naturale
  - Descrittori di deployment
  - Numerose classi e interfacce
  - Difficoltà di uso corretto (vedi qualche *antipattern* emerso, soprattutto per gli entity bean), lookup dei componenti sempre basato su JNDI, ...
- ❑ EJB3.0 risponde alla necessità di essere ***più semplice (sia apprendimento che utilizzo)***
  - Minor numero di classi e interfacce
  - ***Dependency injection***
  - Lookup di componenti più semplice
  - ***Non necessari interfacce per il container e descrittori di deployment***
  - ...



## Da EJB2.1 a EJB3.0: Ease of Development!

- ❑ EJB3.0 risponde alla necessità di essere **più semplice (sia apprendimento che utilizzo)**
  - **Gestione semplificata della persistenza:** entity bean come POJO e **persistenza come servizio esterno alla specifica EJB** (Java Persistence API, la vedremo più avanti nel corso)
  - Java Persistence API si occupa sostanzialmente del mapping oggetti/db relazionali
- ❑ **Compatibilità e migrazione**
  - ❑ **Tutte le applicazioni EJB esistenti continuano a funzionare**
  - ❑ Supporto a metodologie di integrazione per applicazioni e componenti pre-esistenti
  - ❑ Supporto **all'aggiornamento e alla sostituzione di componenti (attraverso API EJB3.0)** senza impatto sui clienti esistenti
  - ❑ Idea di facilitare l'adozione della tecnologia EJB3.0 tramite **migrazione incrementale**

Perché non se occupa direttamente il programmatore di applicazioni?



## L'Approccio di EJB3.0

- ❑ Semplificazione delle API EJB
  - Non più necessari **EJBHome** e **EJBObject**
  - Non più necessari i **descrittori di deployment**
  - Rimozione di API JNDI dalla vista cliente e sviluppatore
- ❑ Sfrutta i vantaggi dei **nuovi metadati (annotation) di linguaggio Java**
  - **Defaulting** per i casi più usuali
  - Tra l'altro, **annotation** progettate perché i casi più comuni siano i più semplici da esprimere
- ❑ **Più attività svolte dal container trasparente,** meno dallo sviluppatore
- ❑ **“Inversione di contratto e di controllo”**
  - I componenti specificano le loro necessità tramite metadati
  - **Interposizione dinamica e trasparente** del container per fornire i servizi richiesti

Ne parleremo ampiamente quando parleremo di Spring



# Semplificazione delle API per Componenti EJB

Quindi, scendendo in maggiore dettaglio dal punto di vista della programmazione EJB:

- ❑ Eliminazione di requisiti sulle interfacce dei componenti EJB
  - Interfacce di business sono **interfacce Java plain (POJI)**
- ❑ Eliminazione di requisiti sulle **interfacce Home**
  - Necessità solo di interfacce di business, non più di Home
- ❑ Eliminazione di requisiti sulle interfacce `javax.ejb.EnterpriseBean`
  - **Annotation per sfruttare callback** (anche addizionali e opzionali)
- ❑ Eliminazione della necessità di utilizzo di API JNDI
  - Uso di **dependency injection**, metodo di lookup semplificato



## Esempio EJB2.1

```
// EJB 2.1

public class PayrollBean
    implements javax.ejb.SessionBean {

    SessionContext ctx;
    DataSource empDB;

    public void setSessionContext(SessionContext ctx) {
        this.ctx = ctx;
    }

    public void ejbCreate() {
        Context initialContext = new InitialContext();
        empDB = (DataSource)initialContext.lookup(
            "java:comp/env/jdbc/empDB");
    }
}
```



## Esempio EJB2.1

```
// EJB 2.1 (cont.)

public void ejbActivate() {}
public void ejbPassivate() {}
public void ejbRemove() {}

public void setBenefitsDeduction (int empId, double
deduction) {
    ...
    Connection conn = empDB.getConnection();
    ...
}
    ...
}

// NOTA: il descrittore di deployment è NECESSARIO
```



## Esempio EJB2.1

### Esempio di Descrittore di deployment - NECESSARIO

```
<session>
  <ejb-name>PayrollBean</ejb-name>
  <local-home>PayrollHome</local-home>
  <local>Payroll</local>
  <ejb-class>com.example.PayrollBean</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Container</transaction-type>
  <resource-ref>
    <res-ref-name>jdbc/empDB</res-ref-name>
    <res-ref-type>javax.sql.DataSource</res-ref-type>
    <res-auth>Container</res-auth>
  </resource-ref>
</session>
...
<assembly-descriptor>
...
</assembly-descriptor>
```



```
@Stateless public class PayrollBean
    implements Payroll {

@Resource DataSource empDB;

public void setBenefitsDeduction (int empId, double
deduction) {
    ...
    Connection conn = empDB.getConnection();
    ...
}

    ...
}
```



- ❑ Le **risorse** di cui un bean ha necessità sono **“iniettate”** (*injected*) quando **l'istanza** del bean viene **effettivamente costruita**
- ❑ Ad esempio, riferimenti a:
  - **EJBContext** - **sorgenti di dati**
  - UserTransaction - entry di ambiente
  - **EntityManager** - TimerService
  - **Altri componenti EJB** - ...
- ❑ Dependency injection realizzata tramite **annotation** come via primaria (anche descrittori di deployment NON completi, se preferito)
  - **@EJB**
    - Riferimenti alle interfacce di business EJB
    - Utilizzato anche per riferimenti a EJBHome (per compatibilità con componenti EJB 2.1)
  - **@Resource**: qualsiasi altro riferimento



# Dependency Injection

- ❑ Le **risorse** di cui un bean ha necessità sono **“iniettate”** (*injected*) quando **l'istanza** del bean viene **effettivamente costruita** (vedremo più precisamente più avanti)
  - **Valutazione delle annotazioni anche a runtime**

## Discussione:

- Quali vantaggi/svantaggi rispetto a scelte **a tempo di scrittura del codice, a tempo di compilazione, a tempo di caricamento**?
- Quali vantaggi/svantaggi rispetto a descrittori di deployment?



# Lookup Componenti Semplificato

- ❑ Lo sviluppatore **NON ha più visibilità delle API JNDI** (*vista sviluppatore*)
- ❑ Le **dipendenze** corrispondenti sono espresse a livello di classe bean
- ❑ A runtime si utilizza il metodo `EJBContext.lookup`

```
@Resource(name="myDB", type=javax.sql.DataSource)
@Stateful public class ShoppingCartBean
    implements ShoppingCart {

    @Resource SessionContext cxt;

    public Collection startToShop (String productName) {
        ...
        DataSource productDB = (DataSource) cxt.lookup("myDB");
        Connection conn = myDB.getConnection();
        ...
    }
    ... }
```



## Esempio: Cliente in EJB2.x

```
// Vista cliente di ShoppingCart bean in EJB2.1

...
Context initialContext = new InitialContext();
ShoppingCartHome myCartHome = (ShoppingCartHome)
    initialContext.lookup("java:comp/env/ejb/cart");
ShoppingCart myCart= myCartHome.create();

//Utilizzo del bean

Collection widgets = myCart.startToShop("widgets")

...

// necessario anche il codice per gestire esplicitamente
// l'eccezione javax.ejb.CreateException
...
```



## Vista Cliente Semplificata in EJB3.0

```
// Vista cliente in EJB3.0

@EJB ShoppingCart myCart;

...

Collection widgets = myCart.startToShop("widgets");

...
```

Importantissimo per  
sviluppatori e industrie,  
meno dal punto di vista  
concettuale

## Interoperabilità e Migrazione fra EJB 3.0 e EJB 2.x

- ❑ Applicazioni scritte in modo conforme alla specifica EJB2.1 ***eseguono senza necessità di modifica anche nei container EJB3.0***
- ❑ ***Migrazione*** suggerita verso le API EJB3.0
  - ***Nuove applicazioni possono essere cliente di “vecchi” bean***
  - Vecchi clienti possono ***accedere anche ai nuovi componenti EJB3.0***
    - Anche senza modifiche alla vista cliente preesistente
- ❑ Molte funzionalità EJB3.0 sono disponibili anche per componenti conformi a EJB2.1 (offerte dal nuovo container)
  - Dependency injection, annotazioni per transazioni e callback, ...

```
// Vista cliente da EJB3.0 di un bean EJB2.1
```

```
...
```

```
@EJB ShoppingCartHome cartHome;  
Cart cart = cartHome.create();  
cart.addItem(...); ...  
cart.remove(); ...
```



## Interoperabilità e Migrazione fra EJB 3.0 e EJB 2.x

- ❑ “Vecchi” bean conformi a EJB2.1 possono utilizzare nuovi componenti
  - Questo consente a ***componenti server di essere aggiornati o sostituiti senza impatto sui clienti già distribuiti***
  - I nuovi bean supportano ***sia clienti EJB3.0 che di tipologie precedenti***
  - Le interfacce ***EJBHome e EJBObject vengono automaticamente mappate*** sulla classe del bean di tipo EJB3.0

```
// Vista cliente da EJB2.1 di un bean conforme a EJB3.0
```

```
...
```

```
Context initialContext = new InitialContext();  
ShoppingCartHome myCartHome = (ShoppingCartHome)  
    initialContext.lookup("java:comp/env/ejb/cart");  
ShoppingCart cart= myCartHome.create();  
cart.addItem(...); ...  
cart.remove();  
...
```





# Annotation vs. Descrittori di Deployment

- ❑ Annotation
  - Rendono i descrittori di deployment non necessari
  - I casi di default non necessitano di specifica
  - I casi più comuni possono essere specificati in modo molto semplice
- ❑ Descrittori di deployment continuano a poter essere utilizzati anche in EJB3.0 come **alternativa**
  - Alcuni sviluppatori preferiscono usare descrittori
  - I descrittori possono essere usati per **rendere “più esterni” i metadati**
  - Possono essere anche utilizzati per fare **override** di annotation (descrittori sono “prioritari” su annotation)
- ❑ A partire da EJB3.0, **i descrittori possono essere anche parziali e incompleti** (“sparse descriptor”)



## Tipologie di Bean EJB3.0

Dopo la panoramica appena fatta, ora scendiamo nel dettaglio tecnico di qualche aspetto...

- ❑ In EJB 3.0, **i bean di tipo sessione e message-driven sono classi Java ordinarie**
  - Rimossi i **requisiti di interfaccia**
  - **Tipo di bean specificato da annotation** (o da descrittore)
  - **Annotation principali**
    - **@Stateless, @Stateful, @MessageDriven**
    - **Specificati nella classe** del bean
- ❑ Gli entity bean di EJB2.x non sono stati modificati e possono continuare a essere utilizzati
  - **Ma Java Persistence API supporta nuove funzionalità**
  - **@Entity** si applica solo alle nuove entità relative a Java Persistence API, che sono oggetti Java utilizzabili anche al di fuori del container J2EE

```
// EJB2.1 Stateless Session Bean: Bean Class
```

```
public class PayrollBean implements javax.ejb.SessionBean {
    SessionContext cxt;
    public void setSessionContext(SessionContext cxt) {
        this.cxt = cxt;
    }
    public void ejbCreate() {...}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}

    public void setTaxDeductions(int empId, int deductions) {
    }
}
```

```
// EJB3.0 Stateless Session Bean: Bean Class
```

```
@Stateless
public class PayrollBean implements Payroll {

    public void setTaxDeductions(int empId,int deductions) {
        ...
    }
}
```

## Interfacce di Business

- ❑ **Interfacce in plain Java language** (POJI), nessun requisito aggiuntivo rispetto ad un normale oggetto
  - Vincoli sulle interfacce EJBObject, EJBHome rimossi
- ❑ **Accesso locale o remoto**
  - **Locale a default**
  - **Remoto tramite annotation** o descrittore di deployment
  - I metodi remoti non sono obbligati a lanciare esplicitamente RemoteException
- ❑ **La classe bean si occupa solo di implementare la sua interfaccia di business**
- ❑ **Annotation: @Remote, @Local, @WebService**
  - Si può specificare a livello di classe di bean o di interfaccia in modo equivalente (vedi regole generali delle annotation)



## Esempio

```
// EJB 2.1 Stateless Session Bean: Interfacce

public interface PayrollHome extends javax.ejb.EJBLocalHome {

    public Payroll create() throws CreateException;
}

public interface Payroll
    extends javax.ejb.EJBLocalObject {

    public void setTaxDeductions(int empId, int deductions);
}
```

```
// EJB 3.0 Stateless Session Bean: Interfaccia di business
// (solo logica applicativa)

public interface Payroll {

    public void setTaxDeductions(int empId, int deductions);
}
```



## Esempio

```
// EJB 3.0 Stateless Session Bean: Interfaccia remota

@Remote
public interface Payroll {

    public void setTaxDeductions(int empId, int deductions);
}

// EJB 3.0 Stateless Session Bean:
// Alternativa: @Remote annotation applicata direttamente alla
// classe bean

@Stateless @Remote
public class PayrollBean implements Payroll {

    public void setTaxDeductions(int empId, int deductions) {
        ...
    }
}
```



## Message Driven Bean in EJB3.0

- ❑ Deve implementare **l'interfaccia del listener di messaggi** `javax.jms.MessageListener` ma, a parte questo, ci si può focalizzare sulla **sola logica applicativa**
- ❑ Nessun requisito per l'implementazione di altre interfacce
- ❑ Annotation  
**@MessageDriven**

```
// EJB 3.0 Message-driven bean

@MessageDriven public class PayrollMDB
    implements javax.jms.MessageListener {

    public void onMessage(Message msg) {
        ...
    }
}
```



## Accesso all'Ambiente di Esecuzione e Dependency Injection

- ❑ Come già accennato, si accede all'ambiente di esecuzione tramite **dependency injection** o azioni di lookup semplificate
- ❑ **Dipendenze specificate tramite annotation** o descrittori (parziali) basati su XML
- ❑ Annotation per l'accesso all'ambiente:
  - **@Resource**  
Utilizzata per indicare **factory di connessioni**, semplici entry di ambiente, **topic/queue per JMS**, **EJBContext**, **UserTransaction**, ...
  - **@EJB**  
Utilizzata per indicare interfacce applicative che sono **EJB** o per **integrare interfacce EJBHome** di versioni precedenti
  - **@PersistenceContext**, **@PersistenceUnit**  
Utilizzate per **EntityManager** (vedremo in relazione a Java Persistence)



```
// EJB 3.0 Stateless Session Bean: classe bean
// Accesso ai dati usando injection e Java Persistence API

@Stateless public class PayrollBean implements Payroll {

    @PersistenceContext EntityManager payrollMgr;

    public void setTaxDeductions(int empId, int deductions) {
        payrollMgr.find(Employee.class, empId).setTaxDeductions
            (deductions);
    }
}
```



## Qualche Ulteriore Dettaglio su Dependency Injection...

- ❑ Annotazione **@Resource** serve a **dichiarare un riferimento** a una risorsa, come sorgenti dati o hook (“agganci”) all’ambiente di esecuzione  
Equivalente della **dichiarazione di un elemento resource-ref** nel deployment descriptor
- ❑ @Resource può essere specificata **a livello di classe, metodo o campo**
- ❑ **Il container è responsabile per gestione dependency injection** e per completare binding verso **risorsa JNDI appropriata**



## Qualche Ulteriore Dettaglio su Dependency Injection...

```
@Resource javax.sql.DataSource catalogDS;  
public getProductsByCategory() {  
    Connection conn = catalogDS.getConnection();  
    ... }  
}
```

- ❑ Il container si occupa “generalmente” dell’injection **all’atto dell’istanziamento del componente** (vedi dettagli precisi nel seguito...)
- ❑ Mapping con la risorsa JNDI è risolto **tramite inferenza a partire dal nome** `catalogDS` **e dal tipo** `javax.sql.DataSource`

Nel caso di risorse multiple:

```
@Resources {  
    @Resource (name="myDB" type=java.sql.DataSource),  
    @Resource (name="myMQ" type=javax.jms.ConnectionFactory) }  
}
```



## Elementi di @Resource

Più precisamente, il container si occupa dell’injection della risorsa nel componente **o a runtime o quando il componente è inizializzato**, in dipendenza dal fatto che la dependency injection sia relativa a campo/metodo o a classe

- ❑ Campo/metodo => all’inizializzazione del componente
- ❑ Classe => a runtime, by need (solo quando si ha necessità di accedere alla risorsa iniettata)

### **@Resource ha i seguenti elementi**

- **name**: nome JNDI della risorsa
- **type**: tipo (Java language type) per la risorsa
- **authenticationType**: tipo di autenticazione da usarsi
- **shareable**: possibilità di condividere la risorsa
- **mappedName**: nome non portabile e implementation-specific a cui associare la risorsa
- **description**



## Elementi di @Resource

### Name

L'elemento **name** è *opzionale per la injection a livello di campo o metodo*

- Livello di campo: **nome di default** è il nome del campo qualificato dal nome della classe
- Livello di metodo: **nome di default** è il nome della proprietà basato sul metodo indicato dal nome della classe

### Tipo di risorsa

Determinato da:

- Livello di campo: **tipo del campo** che l'annotazione @Resource sta decorando
- Livello di metodo: **tipo della proprietà** del componente che l'annotazione @Resource sta decorando
- Dall'elemento **type** di @Resource



## Elementi di @Resource

**authenticationType**: solo per risorse di tipo **connection factory**.  
Può avere valore CONTAINER (default) o APPLICATION

**shareable**: usato solo per risorse che sono **istanze di ORB o connection factory**

**mappedName**: nome non portabile (su diverse implementazioni di J2EE application server) e implementation-specific a cui la risorsa deve essere mappata. Usato tipicamente per riferire la **risorsa al di fuori dell'application server**

**description**: descrizione della risorsa, tipicamente nel linguaggio di default del sistema su cui si fa il deployment della risorsa. Usato per **aiutare nell'identificazione** della risorsa



## Injection a Livello di Campo

In questo caso, quindi, ***il container inferisce nome e tipo della risorsa se gli elementi name e type non sono stati specificati***

```
public class SomeClass {  
    @Resource  
    private javax.sql.DataSource myDB; ... }  
}
```

In questo snippet, l'inferenza è fatta sulla base del nome della classe e del campo: com.example.SomeClass/myDB per il nome, javax.sql.DataSource.class per il tipo

```
public class SomeClass {  
    @Resource(name="customerDB")  
    private javax.sql.DataSource myDB; ... }  
}
```

In questo snippet, il nome JNDI è customerDB e il tipo, determinato tramite inferenza, è javax.sql.DataSource.class



## Injection a Livello di Metodo

Il container inferisce nome e tipo della risorsa se non specificati. Il ***metodo di set deve seguire le convenzioni classiche*** dei bean per i nomi della proprietà: deve ***cominciare*** con set, restituire void ***come tipo di ritorno*** e avere ***un solo parametro***

```
public class SomeClass {  
    private javax.sql.DataSource myDB; ...  
    @Resource  
    private void setmyDB(javax.sql.DataSource ds) {  
        myDB = ds;  
    } ... }  
}
```

In questo codice il container inferisce il nome della risorsa sulla base dei nomi di classe e metodo: com.example.SomeClass/myDB per il nome, javax.sql.DataSource.class per il tipo





## Injection a Livello di Metodo

Il container inferisce nome e tipo della risorsa se non specificati. Il **metodo di set deve seguire le convenzioni classiche** dei bean per i nomi della proprietà: deve **cominciare** con **set**, restituire **void** **come tipo di ritorno** e avere **un solo parametro**

```
public class SomeClass {  
    private javax.sql.DataSource myDB; ...  
    @Resource(name="customerDB")  
    private void setmyDB(javax.sql.DataSource ds) {  
        myDB = ds;  
    } ... }
```

In questo codice il nome JNDI è customerDB  
e il tipo javax.sql.DataSource.class



## Injection a Livello di Classe

In questo caso è **obbligatorio utilizzare gli elementi name e type**

```
@Resource(name="myMessageQueue",  
           type="javax.jms.ConnectionFactory")  
public class SomeMessageBean { ... }
```

---

**@Resources** serve a raggruppare insieme diverse dichiarazioni @Resource a livello di classe

```
@Resources({  
    @Resource(name="myMessageQueue",  
               type="javax.jms.ConnectionFactory"),  
    @Resource(name="myMailSession",  
               type="javax.mail.Session")  
})  
public class SomeMessageBean { ... }
```



## Molto importante: Servizi di Sistema Container-based

Oltre ai dettagli di programmazione, ancora più rilevante è capire **quali servizi e come vengono supportati dall'ambiente** (principalmente tramite container):

- ❑ **Pooling e concorrenza**
- ❑ **Transazionalità**
- ❑ Gestione delle connessioni a risorse
- ❑ **Persistenza** (vedi Java Persistence API - JPA)
- ❑ **Messaggistica** (vedi Java Messaging System – JMS)
- ❑ Sicurezza



## 1) Gestione della Concorrenza

Migliaia fino a milioni di oggetti in uso simultaneo

Come gestire la relazione fra numero di clienti e numero di oggetti distribuiti richiesti per servire le richieste cliente?

### ❑ **Resource Pooling**

- Pooling dei componenti server-side da parte di EJB container (**instance pooling**). Idea base è di **evitare di mantenere una istanza separata di ogni EJB per ogni cliente**. Si applica a stateless session bean e message-driven bean (che cos'è un cliente per un MDB?)
- Anche pooling dei connector (vedremo in seguito)

Discussione: voi come realizzereste **instance pooling**?

### ❑ **Activation**

Utilizzata da stateful session bean per risparmiare risorse



# Stateless Session Bean

Ogni EJB container mantiene **un insieme di istanze del bean** pronte per servire richieste cliente

**Non esiste stato di sessione** da mantenere fra richieste successive; **ogni invocazione di metodo è indipendente** dalle precedenti

**Implementazione delle strategie di instance pooling demandate ai vendor di EJB container**, ma analoghi principi

Ciclo di vita di uno stateless session bean:

- ☐ **No state** (non istanziato; stato iniziale e terminale del ciclo di vita)
- ☐ **Pooled state** (istanziato ma non ancora associato ad alcuna richiesta cliente)
- ☐ **Ready state** (già associato con una richiesta EJB e pronto a rispondere ad una invocazione di metodo)



# Stateless Session Bean

Istanza del bean nel pool riceve un riferimento a **javax.ejb.EJBContext** (in caso di richiesta di injection nel codice tramite apposita annotation)

**EJBContext fornisce un'interfaccia per il bean per comunicare con ambiente EJB** (quindi necessità di conoscenza interfacce non è completamente scomparsa ☺)

Quando il bean passa in **stato ready**, **EJBContext** contiene anche informazioni sul **cliente che sta utilizzando il bean**. Inoltre contiene riferimento al proprio **EJB stub**, utile per passare riferimenti ad altri bean (senza usare JNDI)

Altrimenti che cosa succederebbe?

Ricordiamo che il fatto di essere **stateless** è indicato semplicemente tramite decoration con annotation **@javax.ejb.Stateless** (NOTA: **variabili di istanza non possono essere usate per mantenere stato della sessione**).



## Message-driven Bean

Come nel caso di stateless session bean, **anche questi bean non mantengono stato della sessione** e quindi il container può effettuare pooling in modo relativamente semplice

Strategie di pooling analoghe alle precedenti per SB. Unica differenza che **ogni EJB container contiene molti pool, ciascuno dei quali è composto di istanze con stessa destination JMS**

**Messaggi asincroni in JMS** e processamento messaggi da parte di message-driven bean prima del delivery verso destinatario (lo vedremo quando parleremo di JMS)



## Activation

Usata nel caso di **stateful session bean**

Gestione della coppia oggetto EJB + istanza di bean stateful

- ❑ **Passivation**: **disassociazione fra stateful bean instance e suo oggetto EJB**, con salvataggio dell'istanza su memoria (*serializzazione*). Processo del tutto trasparente per cliente
- ❑ **Activation**: recupero dalla memoria (*deserializzazione*) dello stato dell'istanza e riassociazione con oggetto EJB

Nella specifica J2EE, non richiesto che la classe di uno stateful SB sia **serializzabile**. Quindi?

Dipendenza dall'implementazione dello specifico vendor e attenzione al trattamento dei transient...



## Activation

La procedura di activation può essere associata anche ***all'invocazione di metodi di callback*** sui ***cambi di stato nel ciclo di vita*** di uno ***stateful session bean***

Ad esempio, l'annotation **`@javax.ejb.PostActivate`** associa l'invocazione del metodo a cui si applica ***immediatamente dopo l'attivazione*** di un'istanza

Similmente, **`@javax.ejb.PrePassivate`** (prima dell'azione di passivation)

Ad esempio, utilizzati spesso per la ***chiusura/apertura di connessioni a risorse per gestione più efficiente*** (a default vengono mantenuti e serializzati nello stato solo i riferimenti remoti ad altri bean, a SessionContext, al servizio EntityManager e all'oggetto UserTransaction, alcuni dei quali descritti in seguito)



## Gestione Concorrenza

Per definizione, ***session bean non possono essere concorrenti in senso logico***, nel senso che una ***singola istanza è associata a singolo cliente***

Vietato l'utilizzo di ***thread*** a livello applicativo e, ad esempio, della keyword ***synchronized***

Una singola istanza di ***entity bean*** invece può essere acceduta da più clienti: ***JPA e copia dell'istanza in dipendenza dalla transazione*** (vedi nel seguito)

***Message-driven bean: concorrenza come processamento concorrente di più messaggi*** verso la stessa destinazione: ovviamente, ***supportata***



## 2) Transazioni

Sappiamo tutti, vero ☺?, che cosa si intende per

**transazione** e per **proprietà di transazionalità**

Proprietà ACID (Atomicity, Consistency, Isolation e Durability)

**Transazione come unità indivisibile di processamento;**  
può terminare con un *commit* o un *rollback*

Session bean e message-driven bean possono sfruttare o

**Container-Managed Transaction** o **Bean-Managed Transaction**



## Demarcazione Automatica: Container-Managed Trans.

- ❑ Sono la tipologia di default
- ❑ **Transazione associata con l'intera esecuzione di un metodo** (**demarcazione automatica della transazione**: inizio immediatamente prima dell'inizio dell'esecuzione del metodo e commit immediatamente prima della terminazione del metodo)
- ❑ NON si possono utilizzare **metodi per gestione delle transazioni che interferiscano con gestione automatica** del container (ad esempio, proibito l'uso di `commit` o `rollback` di `java.sql.Connection`, di `rollback` di `javax.jms.Session` o dell'intera interfaccia `javax.Transaction.UserTransaction`)
- ❑ Annotation: **@TransactionManagement**
  - Valore uguale a **container (default) oppure a bean**

I cosiddetti **attributi di transazione** permettono di **controllare lo scope di una transazione**

Perchè sono necessari?



## Annotation: **@TransactionAttribute**

- Valori possibili: REQUIRED (implicito a default), REQUIRES\_NEW, MANDATORY, NOT\_SUPPORTED, SUPPORTS, NEVER

Attributo	Transazione lato cliente?	Transazione lato servitore?
Required	Nessuna	T2
	T1	T1
RequiresNew	Nessuna	T2
	T1	T2
Mandatory	Nessuna	Errore
	T1	T1
NotSupported	Nessuna	Nessuna
	T1	Nessuna
Supports	Nessuna	Nessuna
	T1	T1
Never	Nessuna	Nessuna
	T1	Errore





# CMT: esempi

```
import static TransactionAttribute.*;

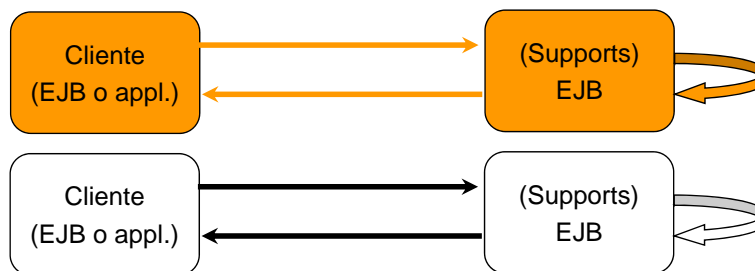
@Stateless
@TransactionAttribute(NOT_SUPPORTED)
public class TravelAgentBean implements TravelAgentRemote {
    public void setCustomer(Customer cust) { ... }
    @TransactionAttribute(REQUIRED)
    public TicketDO bookPassage(CreditCard card, double price) { ... }
}
```

## NotSupported

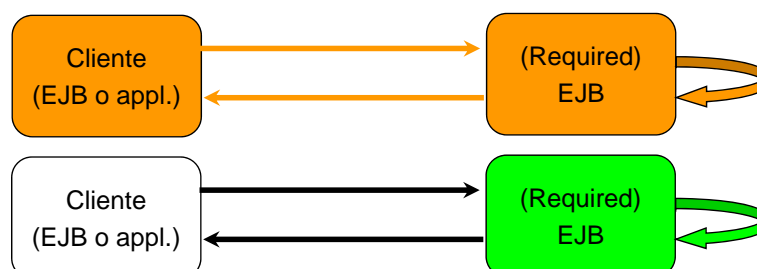


# CMT: esempi

## Supports



## Required

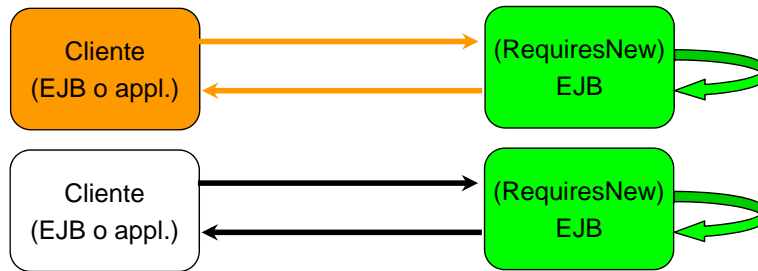




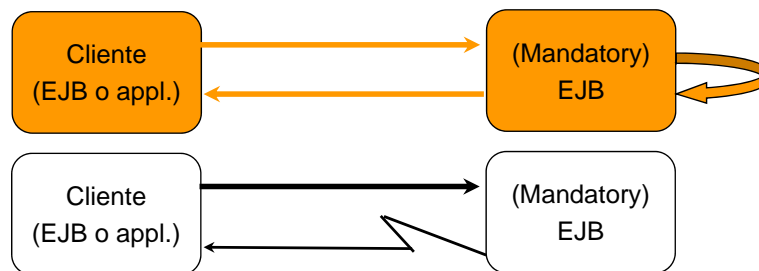


## CMT: esempi

### RequiresNew

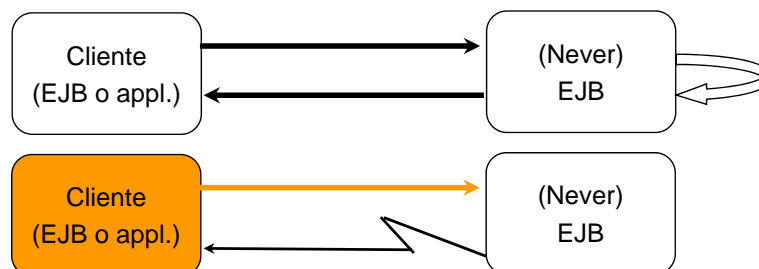


### Mandatory



## CMT: esempi

### Never





# Container-Managed Transaction

**Rollback di una transazione** di questo tipo può essere scatenato da due cause:

- ❑ Eccezione di sistema => container automaticamente lancia il rollback
- ❑ Invocando il metodo `setRollbackOnly` di **EJBContext**

È possibile **sincronizzare l'esecuzione** di un **session bean** a seconda di **eventi dovuti alla transazione** (interfaccia `SessionSynchronization`):

- ❑ metodo `afterBegin` invocato dal container immediatamente prima dell'invocazione del metodo di business all'interno della transazione
- ❑ metodo `beforeCompletion` invocato dal container immediatamente prima del commit della transazione
- ❑ metodo `afterCompletion` invocato dal container immediatamente dopo il completamento della transazione (con commit o rollback)



# Bean-Managed Transaction

**I limiti di demarcazione della transazione sono in questo caso decisi dal programmatore stesso**

**Maggiore complessità ma maggiore flessibilità**

Ad esempio (pseudocodice):

```
begin transaction
... update tableA
...
if (condition1)    commit transaction
    else if (condition2) { update tableB
                          commit transaction }
else rollback transaction ...
```

Possibilità di utilizzo di **Java Transactions API** per indipendenza dall'implementazione dello specifico transaction manager

Sia per Container- che Bean-Managed, **possibilità di configurare l'intervallo di timeout** per le transazioni (Admin Console di J2EE); valore di default 0, ovvero nessun timeout



## Esempio

```
// EJB 3.0: Container-managed transaction
@TransactionAttribute(MANDATORY)
@Stateless public class PayrollBean implements Payroll {

    public void setTaxDeductions(int empId,int deductions) {
        ...
    }
    @TransactionAttribute(REQUIRED)
    public int getTaxDeductions(int empId) {
        ...
    } }

// EJB 3.0: Bean-managed transaction

@TransactionManagement(BEAN)
@Stateless public class PayrollBean implements Payroll {

    @Resource UserTransaction utx;
    @PersistenceContext EntityManager payrollMgr;

    public void setTaxDeductions(int empId, int deductions) {
        utx.begin();
        payrollMgr.find(Employee.class, empId).setDeductions(deductions);
        utx.commit();
    } ... }
```



## 3) Gestione Connessioni a Risorse

Un componente può ***avere bisogno di utilizzare altri componenti e risorse***, come database e sistemi di messaging, ad esempio

In Java EE ***ritrovamento delle risorse desiderate è basato su un sistema di nomi ad alta portabilità come Java Naming and Directory Interface (JNDI)***; se un componente usa resource injection, sarà ***il container a utilizzare JNDI per ritrovare la risorsa desiderata, e non il componente stesso com'era usuale prima di EJB3.0***

In particolare, relativamente a risorse a database, ***connection pooling***: connessioni sono riutilizzabili per ridurre latenza e incrementare prestazioni nell'accesso a DB

Vedi quanto detto precedentemente per annotazione @Resource



## 4) Persistenza

- ❑ **Semplificare il modello di programmazione degli entity bean**
- ❑ Supporto ad una **modellazione lightweight** e semplificata del dominio, che consenta **ereditarietà e polimorfismo**
- ❑ Ricche capacità di query
- ❑ Supporto per **mapping mondo a oggetti/relazionale**
- ❑ Rendere le istanze delle entità utilizzabili **anche fuori da un container EJB**
  - Rimozione delle necessità di DTO e antipattern simili
- ❑ Evoluzione verso API di Java persistence comuni
  - Integrata l'esperienza e il feedback da Hibernate, Java Data Objects, TopLink, e precedenti versioni di tecnologia EJB
- ❑ **Supporto per provider di persistenza come terze parti**, anche determinabili dinamicamente



## Modello di Persistenza in EJB3.0

- ❑ **Entità sono semplici classi Java**
  - Classi “normali” – semplice utilizzo di `new`
  - **Metodi per effettuare get/set di proprietà o variabili di istanza persistenti**
  - **Interfacce bean e interfacce di callback NON necessarie**
- ❑ Utilizzabili anche come **oggetti “detached”** (li vedremo) in altri tier delle applicazioni
  - Nessuna necessità di DTO
- ❑ **EntityManager svolge il ruolo di “Home” non tipata** per le operazioni di entity
  - Metodi per le operazioni sul ciclo di vita (`persist`, `remove`, `merge`, `flush`, `refresh`, ...)
  - Funzionalità simili a Hibernate Session, JDO PersistenceManager, ...
  - Gestisce il **contesto della persistenza**, sia con lo scope della singola transazione che con scope più esteso



## Persistenza: Mapping O/R

- ❑ Facility di supporto di semplice utilizzo per effettuare il **mapping fra modello a oggetti e modello di database relazionale**
- ❑ Lo sviluppatore ha visibilità di tale mapping
  - Ne ha **pieno controllo** e ha consapevolezza della semantica associata
- ❑ **Mapping** può essere espresso **tramite annotation** o frammenti di descrittore XML
  - Supporto a mapping di default

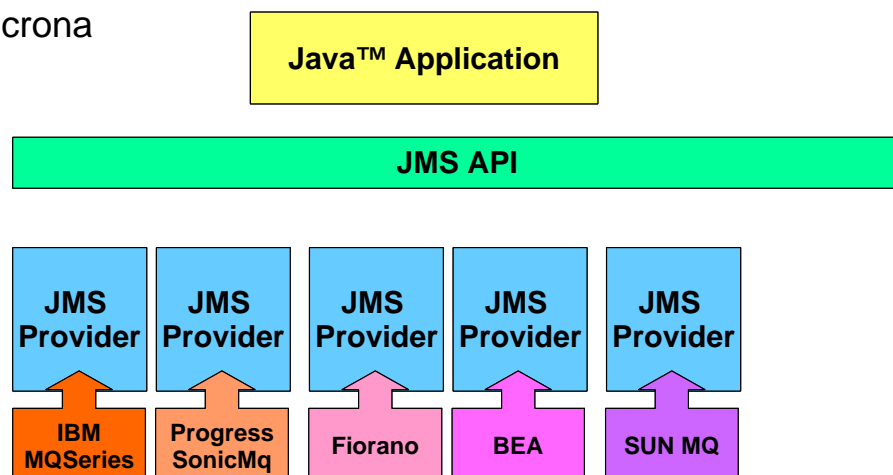
Per il resto dei dettagli su mapping O/R, posso confidare nei corsi di Sistemi Informativi T e Tecnologie Web T?



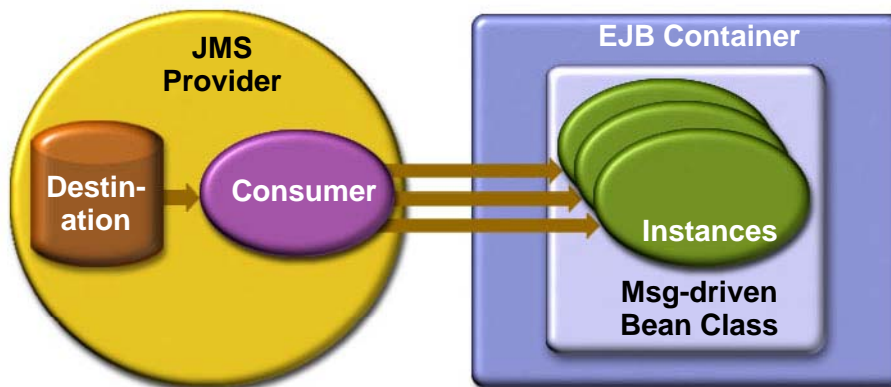
## 5) Messaging

**Java Messaging System (JMS)** come servizio di supporto JEE: **specifica** che definisce come un cliente JMS acceda alle **funzionalità supportate da un sistema di messaging di livello enterprise**. Solitamente:

- ❑ **Produzione, distribuzione e consegna** di messaggi
- ❑ **Semantiche** di consegna supportate:
  - sincrona/asincrona
  - transacted
  - garantita
  - durevole

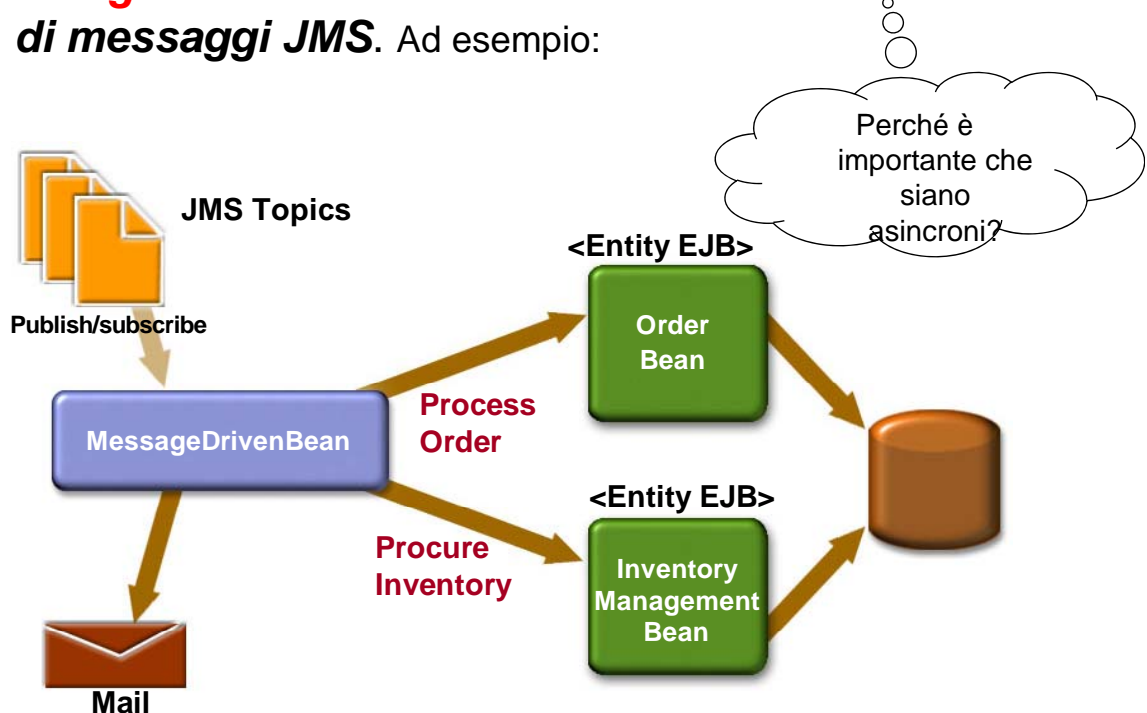


- ❑ Supporto ai **principali modelli** di messaging in uso
  - **Point-to-Point** (code affidabili)
  - **Publish/Subscribe**
- ❑ Selettori di messaggi (lato ricevente)
- ❑ Diverse tipologie di messaggi (le vedremo in seguito...)



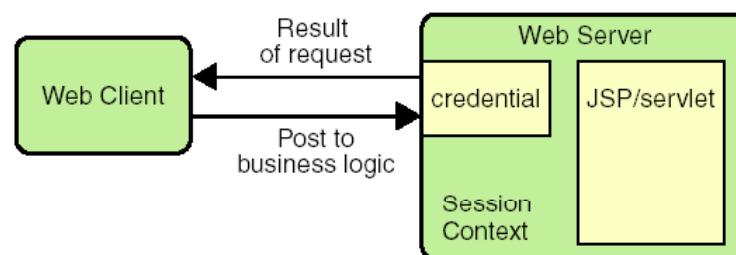
## MDB come consumatori di messaggi JMS

**Message-driven bean** come **consumatori asincroni** di messaggi **JMS**. Ad esempio:

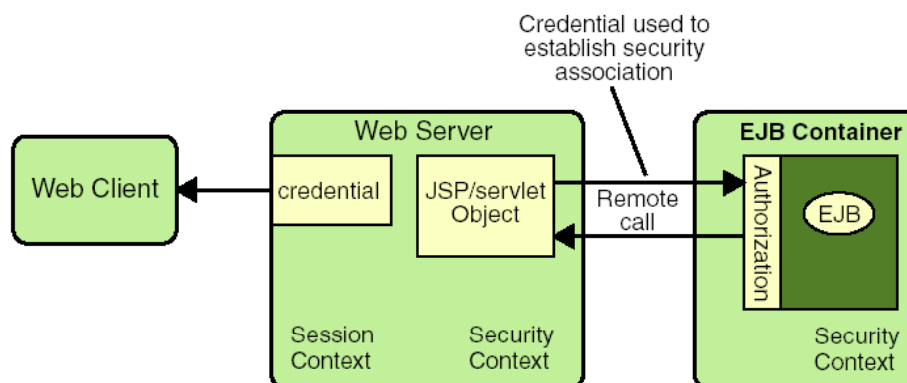


Il container EJB è anche **responsabile per svolgere azioni di controllo dell'accesso** sui metodi del bean

- ❑ Consulta **policy di sicurezza da applicare** (derivanti da deployment descriptor + annotation) per determinare i differenti ruoli di accesso
- ❑ Per ogni ruolo, il container EJB usa il **contesto di sicurezza associato** con l'invocazione per determinare permessi di chiamata
- ❑ "is authorized" quando il **container può mappare le credenziali del chiamante su ruolo autorizzato**: controllo passato a EJB
- ❑ "not authorized" => container lancia **un'eccezione** prima di cedere controllo



## Sicurezza e Meccanismi



Già **Java SE supporta numerose funzionalità e meccanismi di sicurezza**. Ad esempio:

- ❑ Java Authentication and Authorization Service (JAAS)
- ❑ JavaGeneric Security Services (Java GSS-API): token-based API per lo scambio sicuro di messaggi fra applicazioni (anche Kerberos-based)
- ❑ Java Cryptography Extension (JCE): meccanismi per crittografia simmetrica, asimmetrica, ciphering blocco/flusso, ...



e ancora:

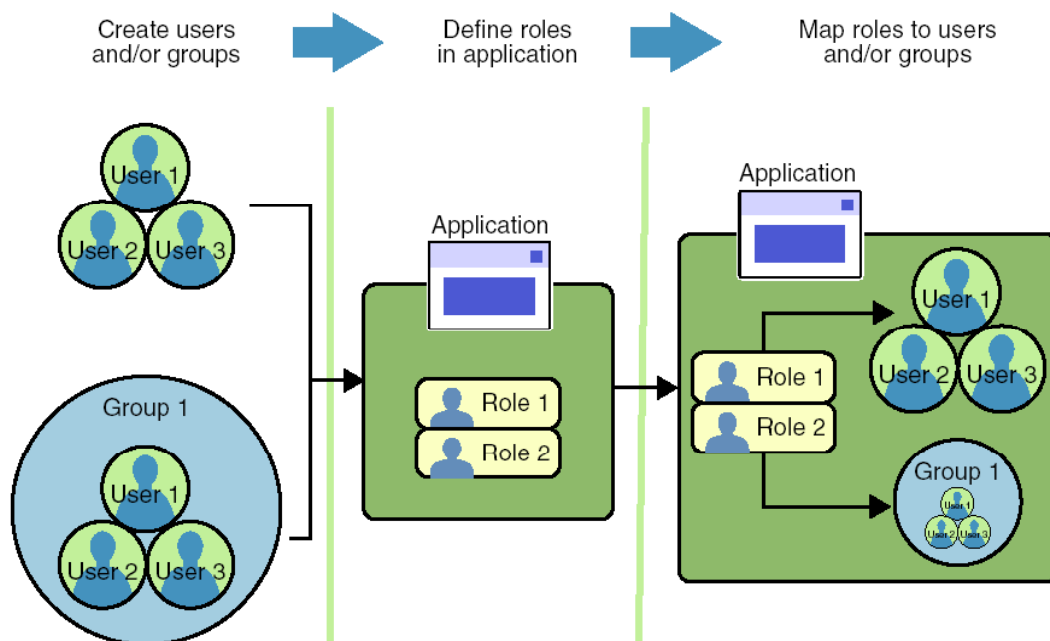
- ❑ Java Secure Sockets Extension (JSSE): protocolli SSL e TLS
- ❑ Simple Authentication and Security Layer (SASL): Internet standard (RFC 2222) per definire come i dati di autenticazione debbano essere scambiati

Il container EJB basa le sue decisioni di sicurezza sui concetti di **Realm, Utenti, Gruppi e Ruoli**

**Realm** come **collezione di utenti** di una singola applicazione (o di un loro insieme), controllati dalla **stessa policy di autenticazione**. Possono o meno essere parte dello stesso gruppo

Ad esempio, nel realm predefinito “file”, informazioni di autenticazione memorizzate nel file `keyfile`; nel realm predefinito “certificate”, informazioni mantenute in un db di certificati X.509

## Sicurezza: realm, gruppi, ruoli e utenti







- ❑ **La configurazione della sicurezza** viene tipicamente **svolta a deployment time** in ambienti enterprise
  - Sviluppatori possono però dare una guida direttam nel codice...
- ❑ Attributi di sicurezza (se non specificati => configurati al deployment o non controllati)
- ❑ Definizione di **permessi di esecuzione sui metodi**  
**Ruoli a cui si concede il diritto** di eseguire un dato insieme di metodi
- ❑ Annotation
  - @RolesAllowed** (valore è una lista di nomi di ruoli), **@PermitAll**, **@DenyAll** (applicabile solo a livello di singolo metodo)
- ❑ **Identità (principal) del chiamante**
  - @RunAs** per definire il principal di esecuzione
- ❑ **Determinazione dei ruoli di sicurezza svolta a runtime dal container**



```
// EJB 3.0: Sicurezza

@Stateless public PayrollBean implements Payroll {

    public void setBenefitsDeduction(int empId, double deduction) {...}

    public double getBenefitsDeduction(int empId) {...}

    public double getSalary(int empId) {...}

    // setting del salario ha un accesso più restrittivo
    @RolesAllowed("HR_PayrollAdministrator")
    public void setSalary(int empId, double salary) {...}

}
```



## Servizi aggiuntionali: gli Intercettori

- ❑ Sono una facility di semplice utilizzo per la **realizzazione di casi e applicazioni cosiddetti avanzati**. Sono **oggetti capaci di interpersi sulle chiamate di metodo** o su **eventi del ciclo di vita** di SB e MDB
- ❑ **Container si interpone** sempre sulle invocazioni dei metodi della logica applicativa
- ❑ **Gli intercettori (interceptor)**, quando presenti, **si interpongono dopo il container stesso** (e prima dell'esecuzione del metodo logica applicativa)

Esempio di quanto tempo impiega l'esecuzione di un metodo

```
long startTime=System.currentTimeMillis();  
try { ...  
} finally { long endTime=System.currentTimeMillis() - startTime;  
    ...  
}
```

Perché non ci piace troppo?



## Servizi aggiuntionali: gli Intercettori

- ❑ Modello di invocazione: **metodi "AroundInvoke"**
  - Fanno da **wrapper** per l'invocazione dei metodi di business
  - Controllano l'invocazione del metodo successivo (sia di altro intercettore che di business, vedi `InvocationContext.proceed()`)
  - **Possono manipolare argomenti e risultati**
  - **Invocati nello stesso stack di chiamate Java**, con stesso contesto di sicurezza e di transazionalità
  - **Dati di contesto possono anche essere mantenuti nella catena degli intercettori** (vedi `javax.interceptor.InvocationContext`)



# Intercettori

- ❑ Annotation: **@Interceptors** (per associare una classe/metodo di un componente di business alla classe intercettore correlata), **@AroundInvoke** (per definire quale metodo della classe intercettore eseguire all'atto dell'intercettazione)
- ❑ Intercettori di default
  - Si applicano a **tutti i metodi di business di ogni componente nel file ejb-jar**
  - Specificati nel **descrittore di deployment**
    - Non ci sono metadati di annotation a livello di applicazione
- ❑ Intercettori a livello di classe
  - Si applicano ai metodi di business della classe bean
- ❑ Intercettori a livello di metodo, per determinazioni più fine-grained e anche per fare overriding di associazioni precedenti



# Intercettori

Ad esempio:

```
//classe Profiler
public class Profiler {
    @AroundInvoke
    public Object profile() throws Exception {
        ...
    }
}

//classe intercettata
...
@Interceptors(Profiler.class)
public Object m1(...) throws ... {
    ...
}
```