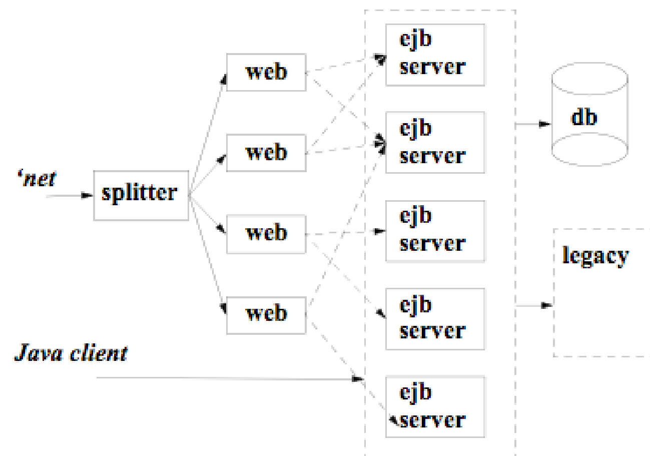
Enterprise JavaBeans

Gli EJB offrono vari vantaggi allo sviluppatore di una applicazione

- Un ambiente di esecuzione che gestisce
  - naming di oggetti, sicurezza, concorrenza, transazioni, persistenza, distribuzione oggetti (location transparency), messaggi asincroni
  - ciclo di vita oggetti (modello a pool), caching, sharing
- Il modello a componenti che permette di comprare/riusare parti e rende lo sviluppo più veloce.
- La chiara separazione del lavoro di sviluppo, deployment ed amministrazione di una applicazione EJB.

Scenario tipico per EJB



Enterprise JavaBeans

Un ***EJB*** (o bean) è un *componente* lato server con una business logic che è conforme a specifiche EJB ed esegue in un *container* EJB.

Componente: classe Java con un ben definito ruolo nel contesto dell'architettura EJB e conforme ad un set di interfacce. E' una unità che può essere installata da sola.

Container: offre funzionalità legate al ciclo di vita dei componenti e alla gestione di transazioni e sicurezza.

Il modello a container scherma i componenti dalla conoscenza dell'architettura sottostante.

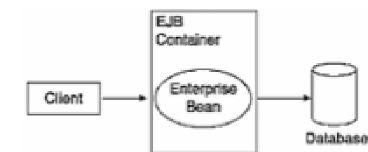
Non vi è una connessione diretta tra il chiamante (client) e l'EJB

- Il server genera il codice che viene inserito tra client ed EJB
- Questo codice permette di intervenire con i servizi richiesti

Le specifiche EJB definiscono come creare una implementazione di un framework per servizi Java lato server (ovviamente, gli sviluppatori di un'applicazione non dovranno preoccuparsi dell'implementazione del framework).

L'architettura EJB consiste di:

- server EJB
- container EJB
- home interface
- remote interface



Architettura EJB

Server

Il server EJB è il processo che gestisce i container EJB e che fornisce accesso ai servizi di sistema. Il server EJB può anche fornire interfacce per l'accesso ad un database, a servizi CORBA, etc.

Container

Il container EJB è un'astrazione che gestisce una o più classi e/o istanze EJB. Attualmente, il container è sempre fornito dal server EJB, poiché l'interfaccia tra server e container non è ancora stata standardizzata.

Home interface

La *home interface* definisce i metodi per gestire il ciclo di vita di un oggetto EJB. Questi sono invocati dal container per trovare, creare e rimuovere istanze. Lo sviluppatore deve definire la home interface estendendo l'interfaccia `javax.ejb.EJBHome`.

Al momento del deployment, a partire dall'home interface il server crea un oggetto detto *HomeObject*.

Permette di

- Creare una istanza di un bean
- Trovare un bean
- Rimuovere una istanza di un bean

Restituisce una remote interface

Architettura EJB

Remote interface

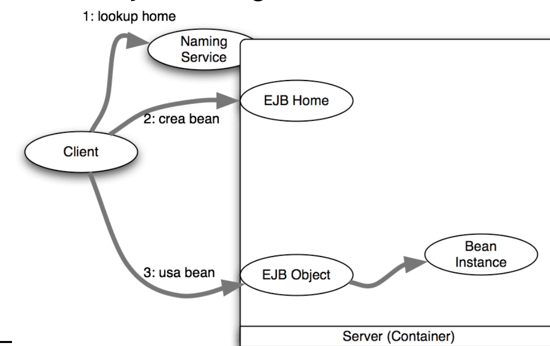
La *remote interface* definisce i metodi business di un EJB (quelli che il client usa). Lo sviluppatore crea la remote interface estendendo `javax.ejb.EJBObject`.

Al momento del deployment, un oggetto che implementa la remote interface, detto *EJBObject*, è creato dal server.

In un'applicazione con EJB, un *client* contatta un *server* per richiedere di eseguire delle elaborazioni.

1. Il client trova il bean tramite Java Naming and Directory Interface (JNDI).
2. Il client usa l'*EJBHome* per creare o distruggere istanze di un EJB. Il server crea l'oggetto (EJB instance) e ritorna un proxy object (*EJBObject*) che ha la stessa interfaccia dell'EJB.
3. Il client usa l'*EJBObject* per invocare i metodi di una istanza.

Il client conosce solo riferimenti ad istanze di *EJBObject* e quando invoca un metodo, l'*EJBObject* delega la richiesta ad un'istanza EJB.



Ruoli per lo sviluppo di EJB

Lo *sviluppatore EJB* scrive le classi usando le specifiche EJB. Le classi conterranno sia metodi per creare e rimuovere una istanza EJB, sia metodi (business) che implementano le funzionalità della classe.

L'*installatore (deployer) EJB* ha il compito di installare una classe EJB sul server EJB e di configurare le proprietà dell'ambiente del server come richiesto dalla classe EJB. L'installatore conosce le caratteristiche dell'ambiente server, come il tipo di database e la sua posizione.

Il fornitore del container EJB fornisce il software per installare una classe EJB sul server.

Per lo sviluppo occorre avere

- Le J2EE, dove le librerie per gli EJB sono disponibili

Per il deployment e l'esecuzione occorre

- Un ambiente per gli EJB, ad esempio jboss

Tipi di EJB

Ci sono tre tipi di EJB:

- Un **session** bean è un EJB che implementa una business logic per i client. Es. può elaborare ordini, codificare dati, ricercare dati in un database.
  - Ha vita pari a quella della sessione con il client che lo usa. E' creato quando il client richiede un servizio e *"distrutto"* quando il client si disconnette (il container gestisce un pool di bean che fornisce ai client rapidamente quando ne fanno richiesta).
  - Uno *stateful* session bean è associato con un singolo client e mantiene i dati conversazionali relativi a quel client.
  - Uno *stateless* session bean non mantiene nessun dato. L'invocazione di un metodo da parte di un client può essere delegata dal container a qualunque session bean del pool.
- Un **entity** bean rappresenta una informazione immagazzinata persistentemente in un database. Fornisce a vari utenti l'accesso condiviso ai dati del database.
  - Un entity bean ha un identificatore unico (primary key) che permette di ricercare il bean (e quindi il dato del database). Ha una vita lunga (come quella del database) e sopravvive a crash del server (viene aggiornato all'ultimo stato committed).
- Un **message-driven** bean esegue al ricevimento di un messaggio del client. E' come un session bean ma invocato in modo asincrono. Ha vita breve e non rappresenta direttamente dati del database.

Sviluppo di un Bean

Per sviluppare un bean è necessario:

- definire la home interface: `EJBHome`
- definire la remote interface: `EJBObject`
- implementare la classe EJB con i metodi business
- creare il deployment descriptor: un set di file XML che contiene informazioni su come assemblare parti dell'applicazione (è usato in fase di deployment)

Esempio: sviluppo di un session bean, chiamato `AdderBean` che somma due numeri

Home interface

```
// home interface AdderHome.java
import java.rmi.RemoteException;
import javax.ejb.*;
public interface AdderHome extends EJBHome {
   Adder create() throws RemoteException,
                                        CreateException;
}
```

Remote interface

```
// remote interface Adder.java
import java.rmi.RemoteException;
import javax.ejb.EJBObject;
public interface Adder extends EJBObject {
   public int add(int a, int b) throws RemoteException;
}
```


Sviluppo di un Bean

Classe EJB

```
// session bean AdderBean.java
import java.rmi.RemoteException;
import javax.ejb.*;
public class AdderBean implements SessionBean {
   public int add(int a, int b) {
      System.out.println("AdderBean esegue");
      return (a+b);
   }
   public void ejbCreate() {
   }
   public void ejbRemove() {
   }
   public void ejbActivate() {
   }
   public void ejbPassivate() {
   }
   public void setSessionContext(SessionContext sc) {
   }
}
```

Sviluppo di un Bean

Deployment Descriptor (file `ejb-jar.xml`)

```
<?xml version="1.0" ?>
<ejb-jar>
  <description>Primo Esempio di EJB</description>
  <display-name>Applicazione Adder</display-name>
  <enterprise-beans>
     <session>
        <ejb-name>Adder</ejb-name>
        <home>AdderHome</home>
        <remote>Adder</remote>
        <ejb-class>AdderBean</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Bean</transaction-type>
     </session>
  </enterprise-beans>
</ejb-jar>
```

Creazione del deployment file

1. Creare una directory `classes` ed inserirvi i file `Adder.class`, `AdderHome.class` e `AdderBean.class`
2. Creare una directory `META-INF` ed inserirvi il file `ejb-jar.xml`
3. Dalla directory padre delle 2 precedenti, eseguire:

```
jar cfv adder.jar * META-INF/ejb-jar.xml
```

In fase di deployment il file `adder.jar` è inserito in un posto conosciuto dal server.

Applicazione Client

Il client non invoca i metodi del bean direttamente, ma vede solo le interfacce home e remote del bean.

Esiste un naming service (JNDI) per cercare l'istanza del bean che si vuol usare.

Passi del client:

1. inizializzare JNDI
2. cercare (lookup) l'istanza del bean
3. creare il bean

Client

```
// client BeanClient.java
import java.util.Properties;
import javax.naming.*;
import Adder;
import AdderHome;

public class BeanClient {
  public static void main(String[] args) {
    // prepara le proprieta' per costruire il contest
    Properties prop = new Properties();
    prop.put(Context.INITIAL_CONTEXT_FACTORY,
            "org.jnp.interfaces.NamingContextFactory");
    prop.put(Context.PROVIDER_URL, "localhost:1099");
```

Applicazione Client

```
    try {
      // ottiene riferimento a JNDI
      InitialContext ctx = new InitialContext(prop);
      // ottiene riferimento alla home interface
      Object ref = ctx.lookup("Adder");
      // casting
      AdderHome home = (AdderHome) ref;
      // crea il bean
      Adder adder = home.create();
      // invoca il metodo del bean
      System.out.println("3 + 5 = " + adder.add(3, 5));
    } catch (Exception e) { }
  }
}
```

Note:

L'invocazione `home.create()` è disponibile attraverso l'HomeObject. Il container crea l'oggetto ed invoca il metodo `ejbCreate()` del bean.

Transazioni

Il Deployment Descriptor può specificare che il container fornisca il supporto per le transazioni al bean.

Il codice del bean non contiene costrutti per iniziare o finire una transazione, ma il container userà una modalità per gestire la transazione.

Esempio di Deployment Descriptor per gestire una transazione

```
<container-transaction>
   <method>
      <ejb-name>Book</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>*</method-name>
   </method>
   <trans-attribute>Required</trans-attribute>
</container-transaction>
```

Attributi per le Transazioni

- `Never` - nessuna transazione è permessa, se il client cerca di iniziare una transazione viene lanciata una `RemoteException`
- `NotSupported` - sospende la transazione se presente sul client
- `Supports` - la transazione è usata se presente, se non presente il metodo è chiamato senza transazione
- `Required` - crea una nuova transazione se non presente
- `RequiresNew` - crea sempre una nuova transazione, sospendendo quella esistente se presente
- `Mandatory` - lancia un'eccezione se nessuna transazione è presente